embfile

Release 0.1.0

Gianluca Gippetto

Jan 10, 2021

CONTENTS

docs	
tests	
package	

A package for working with files containing word embeddings (aka word vectors). Written for:

1. providing a common interface for different file formats;
2. providing a flexible function for building "embedding matrices" that you can use for initializing the *Embedding* layer of your deep learning model;
3. taking as less RAM as possible: no need to load 3M vectors like with *gensim.load_word2vec_format* when you only need 20K;
4. satisfying my (inexplicable) urge of writing a Python package.

CHAPTER

ONE

FEATURES

- Supports textual and Google's binary format plus a custom convenient format (.vvm) supporting constant-time access of word vectors (by word).
- Allows to easily implement, test and integrate new file formats.
- Supports virtually any text encoding and vector data type (though you should probably use only UTF-8 as encoding).
- Well-documented and type-annotated (meaning great IDE support).
- Extensively tested.
- Progress bars (by default) for every time-consuming operation.

CHAPTER

TWO

INSTALLATION

```
pip install embfile
```

CHAPTER

THREE

QUICK START

```
import embfile

with embfile.open("path/to/file.bin") as f:     # infer file format from file␣
↪extension

    print(f.vocab_size, f.vector_size)

    # Load some word vectors in a dictionary (raise KeyError if any word is missing)
    word2vec = f.load(['ciao', 'hello'])

    # Like f.load() but allows missing words (and returns them in a Set)
    word2vec, missing_words = f.find(['ciao', 'hello', 'someMissingWord'])

    # Build a matrix for initializing the Embedding layer either from
    # an iterable of words or a dictionary {word: index}. Handle the
    # initialization of eventual missing word vectors (see argument "oov_initializer")
    matrix, word2index, missing_words = embfile.build_matrix(f, words)
```

CHAPTER

FOUR

TABLE OF CONTENTS

4.1 Installation

At the terminal:

```
pip install embfile
```

4.2 Usage

Table of contents

4.2.1 Opening a file

The core class of the package is the abstract class *EmbFile*. Three subclasses are implemented, one per supported format. Each format is associated with a `format_id` (string) and one or multiple file extensions:

Class	format_id	Extensions	Description
TextEmbFile	txt	.txt, .vec	Glove/fastText format
BinaryEmbFile	bin	.bin	Google word2vec format
VVMEmbFile	vvm	.vvm	Custom format storing vocabulary vectors and metadata in separate files inside a TAR

You can open an embedding file either:

- using the constructor of any of the subclasses above:

```
from embfile import BinaryEmbFile

with BinaryEmbFile('GoogleNews-vectors-negative300.bin') as file:
    ...
```

- or using *embfile.open()*, which by default infers the file format from the file extension:

```
import embfile

with embfile.open('GoogleNews-vectors-negative300.bin') as file:
    print(file)

""" Will print:
BinaryEmbFile (
  path = GoogleNews-vectors-negative300.bin,
  vocab_size = 3000000,
  vector_size = 300
)
"""
```

  You can force a particular format passing the `format_id` argument.

All the `path` arguments can either be of type string or `pathlib.Path`. Object attributes storing paths are always `pathlib.Path`, not strings.

Shared arguments

All the *EmbFile* subclasses support two *optional* arguments (that you can safely pass to `embfile.open` as well):

- `out_dtype` (numpy.dtype) – if provided, all vectors read from the file are converted to this data type (if needed) before being returned;
- `verbose` (bool) – sets the *default value* of the `verbose` argument exposed by all time-consuming `EmbFile` methods; when `verbose is True`, progress bars are displayed by default; you can always pass `verbose=False` to a method to disable console output.

Format-specific arguments

For format-specific arguments, check out the specific class documentation:

BinaryEmbFile(path[, encoding, dtype, ...])	Format used by the Google word2vec tool.
TextEmbFile(path[, encoding, out_dtype, ...])	The format used by Glove and FastText files. Each vector pair is stored as a line of text made of space-separated fields::.
VVMEmbFile(path[, out_dtype, verbose])	(Custom format) A tar file storing vocabulary, vectors and metadata in 3 separate files.

You can pass format-specific arguments to `embfile.open` too.

Compressed files

How to handle compression is left to `EmbFile` subclasses. As a general rule, a concrete `EmbFile` requires non-compressed files unless the opposite is specified in its docstring. Anyway, in most cases, you want to work on non-compressed files because it's much faster (of course).

`embfile` provide utilities to work with compression in the submodule *compression*; the following functions can be used (or imported) directly from the root module:

embfile.extract(src_path[, member, ...])	Extracts a file compressed with gzip, bz2 or lzma or a member file inside a zip/tar archive.
embfile.extract_if_missing(src_path[, ...])	Extracts a file unless it already exists and returns its path.

Lazy (on-the-fly) decompression

Currently, `TextEmbFile` is the only format that allows you to open a compressed file directly and to decompress it "lazily" while reading it. Lazy decompression works for all compression formats but zip. For uniformity of behavior, you can still open zipped files directly but, under the hood, the file will be fully extracted to a temporary file before starting reading it.

Lazy decompression makes sense only if you only want to perform a single pass through the file (e.g. you are converting the file); indeed, every new operation (that requires to create a new *file reader*) requires to (lazily) decompress the file again.

Registering new formats or file extensions

Format ID and file extensions of each registered file format are stored in the global object `embfile.FORMATS`. To associate a file extension to a registered format you can use *associate_extension()*:

```
>>> import embfile
>>> embfile.associate_extension(ext='.w2v', format_id='bin')
>>> print(embfile.FORMATS)
Class          Format ID    Extensions
-------------  -----------  ------------
BinaryEmbFile  bin          .bin, .w2v
TextEmbFile    txt          .txt, .vec
VVMEmbFile     vvm          .vvm
```

To register a new format (see *Implementing a new format*), you can use the class decorator *register_format()*:

```
@embfile.register_format(format_id='hdf5', extensions=['.h5', '.hdf5'])
class HDF5EmbFile(EmbFile):
    # ...
```

4.2.2 Loading word vectors

Loading specific word-vectors

load(words[, verbose])	Loads the vectors for the input words in a `{word: vec}` dict, raising `KeyError` if any word is missing.
find(words[, verbose])	Looks for the input words in the file, return: 1) a dict `{word: vec}` containing the available words and 2) a set containing the words not found.
loader(words[, missing_ok, verbose])	Returns a *VectorsLoader*, an iterator that looks for the provided words in the file and yields available (word, vector) pairs one by one.

```
word2vec = f.load(['hello', 'world'])  # raises KeyError if any word is missing

word2vec, missing_words = f.find(['hello', 'world', 'missingWord'])
```

You should prefer `loader` to `find` when you want to store the vectors *directly* into some custom data structure without wasting time and memory for building an intermediate dictionary. For example, *build_matrix()* uses `loader` to load the vectors directly into a numpy array.

Here's how you use a loader:

```
data_structure = MyCustomStructure()
for word, vector in file.loader(many_words):
    data_structure[word] = vector
```

If you're interested in `missing_words`:

```
data_structure = MyCustomStructure()
loader = file.loader(many_words)
for word, vector in loader:
    data_structure[word] = vector
print('Missing words:', loader.missing_words)
```

Loading the entire file in memory

to_dict([verbose])	Returns the entire file content in a dictionary word -> vector.
to_list([verbose])	Returns the entire file content in a list of *WordVector*'s.

4.2.3 Building a matrix

The docstring of *embfile.build_matrix()* contains everything you need to know to use it. Here, we'll give some examples through an IPython session.

First, we'll generate a dummy file with only three vectors:

```
In [1]: import tempfile

In [2]: from pathlib import Path

In [3]: import numpy as np

In [4]: import embfile

In [5]: from embfile import VVMEmbFile

In [6]: word_vectors = [
   ...:     ('hello', np.array([1, 2, 3])),
   ...:     ('world', np.array([4, 5, 6])),
   ...:     ('!',     np.array([7, 8, 9]))
   ...: ]
   ...:

In [7]: path = Path(tempfile.gettempdir(), 'file.vvm')

In [8]: VVMEmbFile.create(path, word_vectors, overwrite=True, verbose=False)
```

Let's build a matrix out of a list of words. We'll use the default oov_initializer for initializing the vectors for out-of-file-vocabulary words:

```
In [9]: words = ['hello', 'ciao', 'world', 'mondo']

In [10]: with embfile.open(path, verbose=False) as f:
   ....:     result = embfile.build_matrix(
   ....:         f, words,
   ....:         start_index=1,   # map the first word to the row 1 (default is 0)
   ....:     )
   ....:

# result belongs to a class that extends NamedTuple
In [11]: print(result.pretty())
[ 0.000  0.000  0.000]  # 0:
[ 1.000  2.000  3.000]  # 1: hello
[ 1.802  2.867  5.709]  # 2: ciao [out of file vocabulary]
[ 4.000  5.000  6.000]  # 3: world
[ 4.375  4.634  6.280]  # 4: mondo [out of file vocabulary]

In [12]: result.matrix
Out[12]:
array([[0.        , 0.        , 0.        ],
       [1.        , 2.        , 3.        ],
       [1.80158469, 2.86658918, 5.70920508],
       [4.        , 5.        , 6.        ],
       [4.37541454, 4.63356608, 6.28049425]])

In [13]: result.word2index
Out[13]: {'hello': 1, 'ciao': 2, 'world': 3, 'mondo': 4}
```

(continues on next page)

(continued from previous page)

```
In [14]: result.missing_words
Out[14]: {'ciao', 'mondo'}
```

Now, we'll build a matrix from a dictionary `{word: index}`. We'll use a custom `oov_initializer` this time.

```
In [15]: word2index = {
   ....:     'hello': 1,
   ....:     'ciao': 3,
   ....:     'world': 4,
   ....:     'mondo': 5
   ....: }
   ....:

In [16]: with embfile.open(path, verbose=False) as f:
   ....:     def custom_initializer(shape):
   ....:         scale = 1 / np.sqrt(f.vector_size)
   ....:         return np.random.normal(loc=0, scale=scale, size=shape)
   ....:     result = embfile.build_matrix(f, word2index, oov_initializer=custom_
↪initializer)
   ....:

In [17]: print(result.pretty())
[ 0.000  0.000  0.000]  # 0:
[ 1.000  2.000  3.000]  # 1: hello
[ 0.000  0.000  0.000]  # 2:
[ 0.275  0.044  0.387]  # 3: ciao [out of file vocabulary]
[ 4.000  5.000  6.000]  # 4: world
[ 0.452  0.464 -0.488]  # 5: mondo [out of file vocabulary]

In [18]: result.matrix
Out[18]:
array([[ 0.        ,  0.        ,  0.        ],
       [ 1.        ,  2.        ,  3.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.27523144,  0.04421023,  0.38716581],
       [ 4.        ,  5.        ,  6.        ],
       [ 0.45208219,  0.46350951, -0.48754331]])

In [19]: result.word2index
Out[19]: {'hello': 1, 'ciao': 3, 'world': 4, 'mondo': 5}

In [20]: result.missing_words
Out[20]: {'ciao', 'mondo'}
```

See *embfile.initializers* for checking out the available initializers.

4.2.4 Iteration

File readers

Efficient iteration of the file is implemented by format-specific readers.

EmbFileReader(out_dtype)	*(Abstract class)* Iterator that yields a word at each step and read the corresponding vector only if the lazy property `current_vector` is accessed.

A new reader for a file can be created using the method *reader()*. Every method that requires to iterate the file entries sequentially uses this method to create a new reader.

You usually won't need to use a reader directly because `EmbFile` defines quicker-to-use methods that use a reader for you. If you are interested, the docstring is pretty detailed.

Dict-like methods

The following methods are wrappers of *reader()*. Keep in mind that every time you use these methods, you are creating a new file reader and items are read from disk (the vocabulary may be loaded in memory though, as in VVM files).

words()	Returns an iterable for all the words in the file.
vectors()	Returns an iterable for all the vectors in the file.
word_vectors()	Returns an iterable for all the (word, vector) word_vectors in the file.
filter(condition[, verbose])	Returns a generator that yields a word vector pair for each word in the file that satisfies a given condition. For example, to get all the words starting with "z"::.

Don't use `word_vectors()` if you want to filter the vectors based on a condition on words: it'll read vectors for all words you read, even those that don't meet the condition. Use `filter` instead.

4.2.5 Creating/converting a file

Each subclass of `EmbFile` implements the following two class methods:

create(out_path, word_vectors[, vocab_size, ...])	Creates a file on disk containing the provided word vectors.
create_from_file(source_file[, out_dir, ...])	Creates a new file on disk with the same content of another file.

Examples of file creation

You can create a new file either from:

- a dictionary `{word: vector}`
- an iterable of `(word, vector)` tuples; the iterable can also be an iterator/generator.

For example:

```
import numpy as np
from embfile import VVMEmbFile

word_vectors = {
    "hello": np.array([0.1, 0.2, 0.3]),
    "world": np.array([0.4, 0.5, 0.6])
    # ... a lot more word vectors
}

VVMEmbFile.create(
    '/tmp/dummy.vvm.gz',
    word_vectors,
    dtype='<2f',       # store numbers as little-endian 2-byte float
    compression='gz'   # compress with gzip
)
```

Example of file conversions

Let's convert a textual file to a vvm file. The following will generate a compressed vvm file in the same folder of the textual file (and with a proper file extension):

```
from embfile import VVMEmbFile

with embfile.open('path/to/source/file.txt') as src_file:
    dest_path = VVMEmbFile.create_from_file(src_file, compression='gz')

# dest_path  == Path('path/to/source/file.vvm.gz')
```

4.2.6 Implementing a new format

If you ever feel the need for implementing a new format, it's fairly easy to integrate your custom format in this library and to test it. My suggestion is:

1. grab the template below
2. read *EmbFile* docstring
3. look at existing implementations in the `embfile.formats` subpackage
4. for testing, see how they are tested in `tests/test_files.py`

You are highly suggested to use a IDE of course.

```
from pathlib import Path
from typing import Iterable, Optional, Tuple

import embfile
```

(continues on next page)

(continued from previous page)

```
from embfile.types import DType, PathType, VectorType
from embfile.core import EmbFile, EmbFileReader



# TODO: implement a reader
# Note: you could also extend AbstractEmbFileReader if it's convenient for you
class CustomEmbFileReader(EmbFileReader):
    """ Implements file sequential reading """
    def __init__(self, out_dtype: DType):  # TODO: add the needed arguments
        super().__init__(out_dtype)

    def _close(self) -> None:
        pass

    def reset(self) -> None:
        pass

    def next_word(self) -> str:
        pass

    def current_vector(self) -> VectorType:
        pass


@embfile.register_format('custom', extensions=['.cst', '.cust'])
class CustomEmbFile(EmbFile):

    def __init__(self, path: PathType, out_dtype: DType = None, verbose: int = 1):
        super().__init__(path, out_dtype, verbose)  # this is not optional
        # cls.vocab_size = ??
        # cls.vector_size = ??

    def _close(self) -> None:
        pass

    def _reader(self) -> EmbFileReader:
        return CustomEmbFileReader()  # TODO: pass the needed arguments

    # Optional:
    def _loader(self, words: Iterable[str], missing_ok: bool = True, verbose:␣
↪Optional[int] = None) -> 'VectorsLoader':
        """ By default, a SequentialLoader is returned. """
        return super()._loader(words, missing_ok, verbose)

    @classmethod
    def _create(cls, out_path: Path, word_vectors: Iterable[Tuple[str, VectorType]],
                vector_size: int, vocab_size: Optional[int], compression:␣
↪Optional[str] = None,
                verbose: bool = True, **format_kwargs) -> None:
        pass


if __name__ == '__main__':
    print(embfile.FORMATS)
```

This'll print:

```
"""
Class          Format ID    Extensions
-------------  -----------  ------------
BinaryEmbFile  bin          .bin
TextEmbFile    txt          .txt, .vec
VVMEmbFile     vvm          .vvm
CustomEmbFile  custom       .cst, .cust
"""
```

4.3 Formats benchmark

4.3.1 Description

This section is about a benchmark I did out of curiosity for comparing the performance of the formats supported by this library. The snippet under test is the following:

```
with ConcreteEmbFile(path, verbose=0) as f:
    f.find(query)
```

The benchmark was performed on generated files for increasing input sizes (number of words to load). For each input size, the test was repeated 5 times with the exact same input. The script used for running this tests is in the `benchmark` folder of the repository.

The inputs were obtained as following:

1. first, a list of `max(input_sizes)` words was (uniformly) sampled from the file vocabulary
2. the input for size `k` was obtained taking
    - the first `k` words of the sampled list
    - an additional out-of-file-vocabulary word

So, the input for the `i`-th size is a super-set of the previous ones.

Some notes

1. The additional out-of-file-vocabulary word forces txt and bin file objects to read the entire file. The number of missing words isn't an interesting parameter to consider, since missing words are simply added to a set in all the cases.
2. The input sizes reported below don't consider the additional word: the actual input size is `reported_size + 1`, but that's practically irrelevant.
3. The measured times (on each single try) include the time for opening the file; VVM files can take several seconds to open since the vocabulary is entirely read at the start; thus the actual time taken by only *`find()`* in VVM files is lower that those reported below.

System used for tests

Tests were performed on an old desktop computer upgraded with a SSD:

- **CPU:** Intel® Core™2 Quad Q6600
- **RAM:** 8GB DDR2 (4 x 2GB, 800Mhz)
- **SSD:** Samsung 850 EVO 256GB
- **OS:** Windows 10

Expect much better times on newer computers.

4.3.2 Results

Files with 1M vectors of size 100

	1K	50K	150K	300K
`BinaryEmbFile`	6.8	7.6	8.7	10.4
`TextEmbFile`	6.2	11.3	21.4	36.3
`VVMEmbFile`	1.7	3.4	5.4	8.1

Files with 1M vectors of size 300

	1K	50K	150K	300K
`BinaryEmbFile`	8.1	8.8	10.1	12.0
`TextEmbFile`	12.2	25.5	51.8	91.6
`VVMEmbFile`	1.8	4.0	7.4	11.1

Files with 3M vectors of size 100

	1K	50K	150K	300K
`BinaryEmbFile`	21.1	21.9	23.1	24.9
`TextEmbFile`	18.0	23.6	34.1	49.8
`VVMEmbFile`	5.8	7.8	10.9	14.3

Files with 3M vectors of size 300

	1K	50K	150K	300K
`BinaryEmbFile`	25.4	27.0	28.1	30.0
`TextEmbFile`	36.2	49.4	75.8	116.5
`VVMEmbFile`	5.7	8.3	12.7	18.2

4.4 embfile API

Substructure

4.4.1 embfile.core

Substructure

embfile.core.loaders

Classes

RandomAccessLoader(words, word2vec[, ...])	A loader for files that can randomly access word vectors.
SequentialLoader(file, words[, missing_ok, ...])	A Loader that just scans the file from beginning to the end and yields a word vector pair when it meets a requested word.
VectorsLoader(words[, missing_ok])	*(Abstract class)* Iterator that, given some input words, looks for the corresponding vectors into the file and yields a word vector pair for each vector found; once the iteration stops, the attribute `missing_words` contains the set of words not found.

Reference

class embfile.core.loaders.**VectorsLoader**(*words*, *missing_ok=True*)

Bases: abc.ABC, Iterator[WordVector]

(Abstract class) Iterator that, given some input words, looks for the corresponding vectors into the file and yields a word vector pair for each vector found; once the iteration stops, the attribute `missing_words` contains the set of words not found.

Subclasses can load the word vectors in any order.

Parameters

- **words** (Iterable[str]) – the words to load
- **missing_ok** (bool) – If False, raises a KeyError if any input word is not in the file

abstractmethod missing_words
The words that have still to be found; once the iteration stops, it's the set of the words that are in the input `words` but not in the file.

close()
Closes eventual open resources (e.g. a reader).

class embfile.core.loaders.**SequentialLoader**(*file*, *words*, *missing_ok=True*, *verbose=False*)
Bases: abc.ABC, Iterator[WordVector]

A Loader that just scans the file from beginning to the end and yields a word vector pair when it meets a requested word. Used by txt and bin files. It's unable to tell if a word is in the file or not before having read the entire file.

The progress bar shows the percentage of file that has been examined, not the number of yielded word vectors, so the iteration may stop before the bar reaches its 100% (in the case that all the input words are in the file).

(Abstract class) Iterator that, given some input words, looks for the corresponding vectors into the file and yields a word vector pair for each vector found; once the iteration stops, the attribute `missing_words` contains the set of words not found.

Subclasses can load the word vectors in any order.

Parameters

- **words** (Iterable[str]) – the words to load
- **missing_ok** (bool) – If `False`, raises a KeyError if any input word is not in the file

missing_words
The words that have still to be found; once the iteration stops, it's the set of the words that are in the input `words` but not in the file.

close()
Closes eventual open resources (e.g. a reader).

class embfile.core.loaders.**RandomAccessLoader**(*words*, *word2vec*, *word2index=None*, *missing_ok=True*, *verbose=False*, *close_hook=None*)
Bases: abc.ABC, Iterator[WordVector]

A loader for files that can randomly access word vectors. If word2index is provided, the words are sorted by their position and the corresponded vectors are loaded in this order; I observed that this significantly improves the performance (with VVMEmbFile) (presumably due to buffering).

Parameters

- **words** (Iterable[str]) –
- **word2vec** (Word2Vector) – object that implements `word2vec[word]` and `word in word2vec`
- **word2index** (Optional[Callable[[str], int]]) – function that returns the index (position) of a word inside the file; this enables an optimization for formats like VVM that store vectors sequentially in the same file.
- **missing_ok** (bool) –
- **verbose** (bool) –
- **close_hook** (Optional[Callable]) – function to call when closing this loader

missing_words
The words that have still to be found; once the iteration stops, it's the set of the words that are in the input `words` but not in the file.

close()
Closes eventual open resources (e.g. a reader).

embfile.core.reader

Reference

class embfile.core.reader.**EmbFileReader**(*out_dtype*)
Bases: abc.ABC

(Abstract class) Iterator that yields a word at each step and read the corresponding vector only if the lazy property `current_vector` is accessed.

Iteration model. The iteration model is not the most obvious: each iteration step doesn't return a word vector pair. Instead, for performance reasons, at each step a reader returns the next word. To read the vector for the current word, you must access the (lazy) property *current_vector()*:

```
with emb_file.reader() as reader:
    for word in reader:
        if word in my_vocab:
            word2vec[word] = reader.current_vector
```

When you access `current_vector()` for the first time, the vector data is read/parsed and a vector is created; the vector remains accessible until a new word is read.

Creation. Creating a reader usually implies the creation of a file object. That's why `EmbFileReader` implements the `ContextManager` interface so that you can use it inside a `with` clause. Nonetheless, a `EmbFile` keeps track of all its open readers and close them automatically when it is closed.

Parameters **out_dtype** (`Union`[`str`, `dtype`]) – all the vectors will be converted to this dtype before being returned

Variables **out_dtype** (*numpy.dtype*) – all the vectors will be converted to this data type before being returned

close()
Closes the reader

Return type `None`

abstractmethod reset()
(Abstract method) Brings back the reader to the first word vector pair

Return type `None`

abstractmethod next_word()
(Abstract method) Reads and returns the next word in the file.

Return type `str`

abstractmethod current_vector()
(Abstract method) The vector for the current word (i.e. the last word read). If accessed before any word has been read, it raises `IllegalOperation`. The dtype of the returned vector is cls.out_dtype.

Return type `ndarray`

class embfile.core.reader.**AbstractEmbFileReader**(*out_dtype*)

Bases: *embfile.core.reader.EmbFileReader*, abc.ABC

(Abstract class) Facilitates the implementation of a *EmbFileReader*, especially for a file that stores a word and its vector nearby in the file (txt and bin formats), though it can be used for other kind of formats as well if it looks convenient. It:

- keeps track of whether the reader is pointing to a word or a vector and skips the vector when it is not requested during an iteration
- caches the current vector once it is read

Sub-classes must implement:

_read_word()	*(Abstract method)* Reads a word assuming the next thing to read in the file is a word.
_read_vector()	*(Abstract method)* Reads the vector for the last word read.
_skip_vector()	*(Abstract method)* Called when we want to read the next word without loading the vector for the current word.
_close()	*(Abstract method)* Closes the reader

abstractmethod _read_word()

(Abstract method) Reads a word assuming the next thing to read in the file is a word. It must raise StopIteration if there's not another word to read.

Return type str

abstractmethod _read_vector()

(Abstract method) Reads the vector for the last word read. This method is never called if no word has been read or at the end of file. It is called at most time per word.

Return type ndarray

abstractmethod _skip_vector()

(Abstract method) Called when we want to read the next word without loading the vector for the current word. For some formats, it may be empty.

Return type None

abstractmethod _reset()

(Abstract method) Resets the reader

Return type None

abstractmethod _close()

(Abstract method) Closes the reader

Return type None

reset()

Brings back the reader to the beginning of the file

Return type None

next_word()

Reads and returns the next word in the file.

Return type str

current_vector
The vector associated to the current word (i.e. the last word read). If accessed before any word has been read, it raises `IllegalOperation`.

Return type ndarray

Classes

AbstractEmbFileReader(out_dtype)	*(Abstract class)* Facilitates the implementation of a *EmbFileReader*, especially for a file that stores a word and its vector nearby in the file (txt and bin formats), though it can be used for other kind of formats as well if it looks convenient.
EmbFile(path[, out_dtype, verbose])	*(Abstract class)* The base class of all the embedding files.
EmbFileReader(out_dtype)	*(Abstract class)* Iterator that yields a word at each step and read the corresponding vector only if the lazy property `current_vector` is accessed.
RandomAccessLoader(words, word2vec[, ...])	A loader for files that can randomly access word vectors.
SequentialLoader(file, words[, missing_ok, ...])	A Loader that just scans the file from beginning to the end and yields a word vector pair when it meets a requested word.
VectorsLoader(words[, missing_ok])	*(Abstract class)* Iterator that, given some input words, looks for the corresponding vectors into the file and yields a word vector pair for each vector found; once the iteration stops, the attribute `missing_words` contains the set of words not found.
WordVector(word, vector)	A (word, vector) NamedTuple

class embfile.core.**EmbFile**(*path*, *out_dtype=None*, *verbose=True*)
Bases: abc.ABC

(Abstract class) The base class of all the embedding files.

Sub-classes must:

1. ensure they set attributes *vocab_size* and `vector_size` when a file instance is created
2. implement a *EmbFileReader* for the format and implements the abstract method *_reader()*
3. implement the abstract method *_close()*
4. *(optionally)* implement a *VectorsLoader* (if they can improve upon the default loader) and override *loader()*
5. *(optionally)* implement a `EmbFileCreator` for the format and set the class constant `Creator`

Parameters

- **path** (*Path*) – path of the embedding file (eventually compressed)
- **out_dtype** (*numpy.dtype*) – all the vectors will be converted to this data type. The sub-class is responsible to set a suitable default value.
- **verbose** (*bool*) – whether to show a progress bar by default in all time-consuming operations

Variables

- **path** (*Path*) – path of the embedding file
- ***vocab_size*** (int or None) – number of words in the file (can be None for some TextEmbFile)
- **vector_size** (*int*) – length of the vectors
- **verbose** (*bool*) – whether to show a progress bar by default in all time-consuming operations
- **closed** (*bool*) – True if the file was closed

abstractmethod _reader()
(Abstract method) Returns a new reader for the file which allows to iterate efficiently the word-vectors inside it. Called by *reader()*.

Return type *EmbFileReader*

abstractmethod _close()
(Abstract method) Releases eventual resources used by the EmbFile.

Return type None

DEFAULT_EXTENSION: str

vocab_size: Optional[int]

close()
Releases all the open resources linked to this file, including the opened readers.

Return type None

reader()
Creates and returns a new file reader. When the file is closed, all the still opened readers are closed automatically.

Return type *EmbFileReader*

loader(*words*, *missing_ok=True*, *verbose=None*)
Returns a *VectorsLoader*, an iterator that looks for the provided words in the file and yields available (word, vector) pairs one by one. If missing_ok=True (default), provides the set of missing words in the property missing_words (once the iteration ends).

See *embfile.core.VectorsLoader* for more info.

Example

You should use a loader when you need to load many vectors in some custom data structure and you don't want to waste memory (e.g. build_matrix uses it to load the vectors directly into the matrix):

```
data_structure = MyCustomStructure()
with file.loader(many_words) as loader:
    for word, vector in loader:
        data_structure[word] = vector
print('Number of missing words:', len(loader.missing_words)
```

See also:

load() find()

Return type *VectorsLoader*

for ... in words()
Returns an iterable for all the words in the file.

Return type Iterable[str]

for ... in vectors()
Returns an iterable for all the vectors in the file.

Return type Iterable[ndarray]

for ... in word_vectors()
Returns an iterable for all the (word, vector) word_vectors in the file.

Return type Iterable[*WordVector*]

to_dict(*verbose=None*)
Returns the entire file content in a dictionary word -> vector.

Return type Dict[str, ndarray]

to_list(*verbose=None*)
Returns the entire file content in a list of *WordVector*'s.

Return type List[*WordVector*]

load(*words*, *verbose=None*)
Loads the vectors for the input words in a {word: vec} dict, raising KeyError if any word is missing.

Parameters

- **words** (Iterable[str]) – the words to get
- **verbose** (Optional[bool]) – if None, self.verbose is used

Return type Dict[str, ndarray]

Returns *(Dict[str, VectorType])* – a dictionary {word: vector}

See also:

find() - it returns the set of all missing words, instead of raising KeyError.

find(*words*, *verbose=None*)
Looks for the input words in the file, return: 1) a dict {word: vec} containing the available words and 2) a set containing the words not found.

Parameters

- **words** (*Iterable[str]*) – the words to look for
- **verbose** (Optional[bool]) – if None, self.verbose is used

Return type _FindOutput

Returns

namedtuple – a namedtuple with the following fields:

- **word2vec** (*Dict[str, VectorType]*): dictionary {word: vector}
- **missing_words** (*Set[str]*): set of words not found in the file

See also:

load() - which raises KeyError if any word is not found in the file.

for ... in filter(*condition*, *verbose=None*)

Returns a generator that yields a word vector pair for each word in the file that satisfies a given condition. For example, to get all the words starting with "z":

```
list(file.filter(lambda word: word.startswith('z')))
```

Parameters

- **condition** (Callable[[str], bool]) – a function that, given a word in input, outputs True if the word should be taken
- **verbose** (Optional[bool]) – if True, a progress bar is showed (the bar is updated each time a word is read, not each time a word vector pair is yielded).

Return type Iterator[Tuple[str, ndarray]]

save_vocab(*path=None*, *encoding='utf-8'*, *overwrite=False*, *verbose=None*)

Save the vocabulary of the embedding file on a text file. By default the file is saved in the same directory of the embedding file, e.g.:

```
/path/to/filename.txt.gz  ==> /path/to/filename_vocab.txt
```

Parameters

- **path** (Union[str, Path, None]) – where to save the file
- **encoding** (str) – text encoding
- **overwrite** (bool) – if the file exists and it is True, overwrite the file
- **verbose** (Optional[bool]) – if None, self.verbose is used

Return type Path

Returns *(Path)* – the path to the vocabulary file

classmethod create(*out_path*, *word_vectors*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, ***format_kwargs*)

Creates a file on disk containing the provided word vectors.

Parameters

- **out_path** (Union[str, Path]) – path to the created file
- **word_vectors** (*Dict[str, VectorType] or Iterable[Tuple[str, VectorType]]*) – it can be an iterable of word vector tuples or a dictionary `word -> vector`; the word vectors are written in the order determined by the iterable object.
- **vocab_size** (Optional[int]) – it must be provided if `word_vectors` has no `__len__` and the specific-format creator needs to know a priori the vocabulary size; in any case, the creator should check at the end that the provided `vocab_size` matches the actual length of `word_vectors`
- **compression** (Optional[str]) – valid values are: `"bz2"|"bz"`, `"gzip"|"gz"`, `"xz"|"lzma"`, `"zip"`
- **verbose** (bool) – if positive, show progress bars and information
- **overwrite** (bool) – overwrite the file if it already exists
- **format_kwargs** – format-specific arguments

Return type None

classmethod create_from_file(*source_file*, *out_dir=None*, *out_filename=None*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, ***format_kwargs*)

Creates a new file on disk with the same content of another file.

Parameters

- **source_file** (EmbFile) – the file to take data from
- **out_dir** (Union[str, Path, None]) – directory where the file will be stored; by default, it's the parent directory of the source file
- **out_filename** (Optional[str]) – filename of the produced name (inside out_dir); by default, it is obtained by replacing the extension of the source file with the proper one and appending the compression extension if compression is not None. **Note:** if you pass this argument, the compression extension is not automatically appended.
- **vocab_size** (Optional[int]) – if the source EmbFile has attribute vocab_size == None, then: if the specific creator requires it (bin and txt formats do), it *must* be provided; otherwise it *can* be provided for having ETA in progress bars.
- **compression** (Optional[str]) – valid values are: "bz2"|"bz", "gzip"|"gz", "xz"|"lzma", "zip"
- **verbose** (bool) – print info and progress bar
- **overwrite** (bool) – overwrite a file with the same name if it already exists
- **format_kwargs** – format-specific arguments (see above)

Return type Path

class embfile.core.**EmbFileReader**(*out_dtype*)

Bases: abc.ABC

(Abstract class) Iterator that yields a word at each step and read the corresponding vector only if the lazy property current_vector is accessed.

Iteration model. The iteration model is not the most obvious: each iteration step doesn't return a word vector pair. Instead, for performance reasons, at each step a reader returns the next word. To read the vector for the current word, you must access the (lazy) property *current_vector()*:

```
with emb_file.reader() as reader:
    for word in reader:
        if word in my_vocab:
            word2vec[word] = reader.current_vector
```

When you access current_vector() for the first time, the vector data is read/parsed and a vector is created; the vector remains accessible until a new word is read.

Creation. Creating a reader usually implies the creation of a file object. That's why EmbFileReader implements the ContextManager interface so that you can use it inside a with clause. Nonetheless, a EmbFile keeps track of all its open readers and close them automatically when it is closed.

Parameters **out_dtype** (Union[str, dtype]) – all the vectors will be converted to this dtype before being returned

Variables **out_dtype** (*numpy.dtype*) – all the vectors will be converted to this data type before being returned

close()

Closes the reader

Return type None

abstractmethod reset()
(Abstract method) Brings back the reader to the first word vector pair

Return type None

abstractmethod next_word()
(Abstract method) Reads and returns the next word in the file.

Return type str

abstractmethod current_vector()
(Abstract method) The vector for the current word (i.e. the last word read). If accessed before any word has been read, it raises IllegalOperation. The dtype of the returned vector is cls.out_dtype.

Return type ndarray

class embfile.core.**AbstractEmbFileReader**(*out_dtype*)
Bases: *embfile.core.reader.EmbFileReader*, abc.ABC

(Abstract class) Facilitates the implementation of a *EmbFileReader*, especially for a file that stores a word and its vector nearby in the file (txt and bin formats), though it can be used for other kind of formats as well if it looks convenient. It:

- keeps track of whether the reader is pointing to a word or a vector and skips the vector when it is not requested during an iteration
- caches the current vector once it is read

Sub-classes must implement:

_read_word()	*(Abstract method)* Reads a word assuming the next thing to read in the file is a word.
_read_vector()	*(Abstract method)* Reads the vector for the last word read.
_skip_vector()	*(Abstract method)* Called when we want to read the next word without loading the vector for the current word.
_close()	*(Abstract method)* Closes the reader

abstractmethod _read_word()
(Abstract method) Reads a word assuming the next thing to read in the file is a word. It must raise StopIteration if there's not another word to read.

Return type str

abstractmethod _read_vector()
(Abstract method) Reads the vector for the last word read. This method is never called if no word has been read or at the end of file. It is called at most time per word.

Return type ndarray

abstractmethod _skip_vector()
(Abstract method) Called when we want to read the next word without loading the vector for the current word. For some formats, it may be empty.

Return type None

abstractmethod _reset()
(Abstract method) Resets the reader

Return type None

abstractmethod _close()
(Abstract method) Closes the reader

Return type None

reset()
Brings back the reader to the beginning of the file

Return type None

next_word()
Reads and returns the next word in the file.

Return type str

current_vector
The vector associated to the current word (i.e. the last word read). If accessed before any word has been read, it raises IllegalOperation.

Return type ndarray

class embfile.core.**VectorsLoader**(*words*, *missing_ok=True*)
Bases: abc.ABC, Iterator[*WordVector*]

(Abstract class) Iterator that, given some input words, looks for the corresponding vectors into the file and yields a word vector pair for each vector found; once the iteration stops, the attribute missing_words contains the set of words not found.

Subclasses can load the word vectors in any order.

Parameters

- **words** (Iterable[str]) – the words to load
- **missing_ok** (bool) – If False, raises a KeyError if any input word is not in the file

abstractmethod missing_words
The words that have still to be found; once the iteration stops, it's the set of the words that are in the input words but not in the file.

close()
Closes eventual open resources (e.g. a reader).

class embfile.core.**SequentialLoader**(*file*, *words*, *missing_ok=True*, *verbose=False*)
Bases: abc.ABC, Iterator[*WordVector*]

A Loader that just scans the file from beginning to the end and yields a word vector pair when it meets a requested word. Used by txt and bin files. It's unable to tell if a word is in the file or not before having read the entire file.

The progress bar shows the percentage of file that has been examined, not the number of yielded word vectors, so the iteration may stop before the bar reaches its 100% (in the case that all the input words are in the file).

(Abstract class) Iterator that, given some input words, looks for the corresponding vectors into the file and yields a word vector pair for each vector found; once the iteration stops, the attribute missing_words contains the set of words not found.

Subclasses can load the word vectors in any order.

Parameters

- **words** (Iterable[str]) – the words to load
- **missing_ok** (bool) – If False, raises a KeyError if any input word is not in the file

missing_words
The words that have still to be found; once the iteration stops, it's the set of the words that are in the input `words` but not in the file.

close()
Closes eventual open resources (e.g. a reader).

class embfile.core.**RandomAccessLoader**(*words*, *word2vec*, *word2index=None*, *missing_ok=True*, *verbose=False*, *close_hook=None*)
Bases: abc.ABC, Iterator[*WordVector*]

A loader for files that can randomly access word vectors. If word2index is provided, the words are sorted by their position and the corresponded vectors are loaded in this order; I observed that this significantly improves the performance (with VVMEmbFile) (presumably due to buffering).

Parameters

- **words** (Iterable[str]) –
- **word2vec** (Word2Vector) – object that implements `word2vec[word]` and `word in word2vec`
- **word2index** (Optional[Callable[[str], int]]) – function that returns the index (position) of a word inside the file; this enables an optimization for formats like VVM that store vectors sequentially in the same file.
- **missing_ok** (bool) –
- **verbose** (bool) –
- **close_hook** (Optional[Callable]) – function to call when closing this loader

missing_words
The words that have still to be found; once the iteration stops, it's the set of the words that are in the input `words` but not in the file.

close()
Closes eventual open resources (e.g. a reader).

class embfile.core.**WordVector**(*word: str*, *vector: numpy.ndarray*)
Bases: tuple

A (word, vector) NamedTuple

Create new instance of WordVector(word, vector)

word
Alias for field number 0

vector
Alias for field number 1

staticmethod format_vector(*arr*)
Used by __repr__ to convert a numpy vector to string. Feel free to monkey-patch it.

4.4.2 embfile.formats

Substructure

embfile.formats.bin

Classes

BinaryEmbFile(path[, encoding, dtype, ...])	Format used by the Google word2vec tool.
BinaryEmbFileReader(file_obj[, encoding, ...])	*EmbFileReader* for the binary format.

Reference

class embfile.formats.bin.**BinaryEmbFile**(*path*, *encoding='utf-8'*, *dtype=dtype('float32')*, *out_dtype=None*, *verbose=True*)

Bases: embfile.core._file.EmbFile

Format used by the Google word2vec tool. You can use it to read the file GoogleNews-vectors-negative300.bin.

It begins with a text header line of space-separated fields:

```
<vocab_size> <vector_size>
```

Each word vector pair is encoded as following:

- encoded word + space
- followed by the binary representation of the vector.

Variables

- **path** –
- **encoding** –
- **dtype** –
- **out_dtype** –
- **verbose** –

Parameters

- **path** (Union[str, Path]) – path to the (eventually compressed) file
- **encoding** (str) – text encoding; **note:** if you provide an utf encoding (e.g. *utf-16*) that uses a BOM (Byte Order Mark) without specifying the byte-endianness (e.g. *utf-16-le* or *utf-16-be*), the little-endian version is used (*utf-16-le*).
- **dtype** (Union[str, dtype]) – a valid numpy data type (or whatever you can pass to numpy.dtype()) (default: '<f4'; little-endian float, 4 bytes)
- **out_dtype** (Union[str, dtype, None]) – all the vectors returned will be (eventually) converted to this data type; by default, it is equal to the original data type of the vectors in the file, i.e. no conversion takes place.

DEFAULT_EXTENSION: str = '.bin'

vocab_size: Optional[int]

classmethod create(*out_path*, *word_vectors*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *dtype=None*)

Format-specific arguments are `encoding` and `dtype`.

Note: all the text is encoded without BOM (Byte Order Mark). If you pass "utf-16" or "utf-18", the little-endian version is used (e.g. "utf-16-le")

See `create()` for more.

Return type None

classmethod create_from_file(*source_file*, *out_dir=None*, *out_filename=None*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *dtype=None*)

Format-specific arguments are `encoding` and `dtype`.

Note: all the text is encoded without BOM (Byte Order Mark). If you pass "utf-16" or "utf-18", the little-endian version is used (e.g. "utf-16-le")

See `create_from_file()` for more.

Return type Path

class embfile.formats.bin.**BinaryEmbFileReader**(*file_obj*, *encoding='utf-8'*, *dtype=dtype('float32')*, *out_dtype=None*)

Bases: *embfile.core.reader.AbstractEmbFileReader*

EmbFileReader for the binary format.

classmethod from_path(*path*, *encoding='utf-8'*, *dtype=dtype('float32')*, *out_dtype=None*)

embfile.formats.txt

Classes

TextEmbFile(path[, encoding, out_dtype, ...])	The format used by Glove and FastText files. Each vector pair is stored as a line of text made of space-separated fields::.
TextEmbFileReader(file_obj[, out_dtype, ...])	*EmbFileReader* for the textual format.

Reference

class embfile.formats.txt.**TextEmbFile**(*path*, *encoding='utf-8'*, *out_dtype='float32'*, *vocab_size=None*, *verbose=True*)

Bases: embfile.core._file.EmbFile

The format used by Glove and FastText files. Each vector pair is stored as a line of text made of space-separated fields:

```
word vec[0] vec[1] ... vec[vector_size-1]
```

It may have or not an (automatically detected) "header", containing `vocab_size` and `vector_size` (in this order).

If the file doesn't have a header, `vector_size` is set to the length of the first vector. If you know `vocab_size` (even an approximate value), you may want to provide it to have ETA in progress bars.

If the file has a header and you provide `vocab_size`, the provided value is ignored.

Compressed files are decompressed while you proceed reeding. Note that each file reader will decompress the file independently, so if you need to read the file multiple times it's better you decompress the entire file first and then open it.

Variables

- **path** –
- **encoding** –
- **out_dtype** –
- **verbose** –

Parameters

- **path** (Union[str, Path]) – path to the embedding file
- **encoding** (str) – encoding of the text file; default is utf-8
- **out_dtype** (Union[str, dtype]) – the dtype of the vectors that will be returned; default is single-precision float
- **vocab_size** (Optional[int]) – useful when the file has no header but you know vocab_size; if the file has a header, this argument is ignored.
- **verbose** (int) – default level of verbosity for all methods

DEFAULT_EXTENSION: str = '.txt'

vocab_size: Optional[int]

classmethod create(*out_path*, *word_vectors*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *precision=5*)

Creates a file on disk containing the provided word vectors.

Parameters

- **out_path** (Union[str, Path]) – path to the created file
- **word_vectors** (*Dict[str, VectorType] or Iterable[Tuple[str, VectorType]]*) – it can be an iterable of word vector tuples or a dictionary `word -> vector`; the word vectors are written in the order determined by the iterable object.
- **vocab_size** (Optional[int]) – it must be provided if `word_vectors` has no `__len__` and the specific-format creator needs to know a priori the vocabulary size; in any case, the creator should check at the end that the provided `vocab_size` matches the actual length of `word_vectors`
- **compression** (Optional[str]) – valid values are: `"bz2"|"bz"`, `"gzip"|"gz"`, `"xz"|"lzma"`, `"zip"`
- **verbose** (bool) – if positive, show progress bars and information
- **overwrite** (bool) – overwrite the file if it already exists
- **format_kwargs** – format-specific arguments

Return type None

classmethod create_from_file(*source_file*, *out_dir=None*, *out_filename=None*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *precision=5*)

Creates a new file on disk with the same content of another file.

Parameters

- **source_file** (EmbFile) – the file to take data from
- **out_dir** (Union[str, Path, None]) – directory where the file will be stored; by default, it's the parent directory of the source file
- **out_filename** (Optional[str]) – filename of the produced name (inside out_dir); by default, it is obtained by replacing the extension of the source file with the proper one and appending the compression extension if compression is not None. **Note:** if you pass this argument, the compression extension is not automatically appended.
- **vocab_size** (Optional[int]) – if the source EmbFile has attribute vocab_size == None, then: if the specific creator requires it (bin and txt formats do), it *must* be provided; otherwise it *can* be provided for having ETA in progress bars.
- **compression** (Optional[str]) – valid values are: "bz2"|"bz", "gzip"|"gz", "xz"|"lzma", "zip"
- **verbose** (bool) – print info and progress bar
- **overwrite** (bool) – overwrite a file with the same name if it already exists
- **format_kwargs** – format-specific arguments (see above)

Return type Path

class embfile.formats.txt.**TextEmbFileReader**(*file_obj*, *out_dtype=dtype('float32')*, *vocab_size=None*)

Bases: *embfile.core.reader.AbstractEmbFileReader*

EmbFileReader for the textual format.

classmethod from_path(*path*, *encoding='utf-8'*, *out_dtype=dtype('float32')*, *vocab_size=None*)

Returns a *TextEmbFileReader* from the path of a (eventually compressed) text file.

Return type *TextEmbFileReader*

classmethod parse_header(*line*)

Return type Dict[str, Any]

embfile.formats.vvm

Classes

VVMEmbFile(path[, out_dtype, verbose])	(Custom format) A tar file storing vocabulary, vectors and metadata in 3 separate files.
VVMEmbFileReader(file, vectors_file)	*EmbFileReader* for the vvm format.

Reference

class embfile.formats.vvm.**VVMEmbFile**(*path*, *out_dtype=None*, *verbose=True*)

Bases: embfile.core._file.EmbFile, embfile.core.loaders.Word2Vector

(Custom format) A tar file storing vocabulary, vectors and metadata in 3 separate files.

Features:

1. the vocabulary can be loaded very quickly (with no need for an external vocab file) and it is loaded in memory when the file is opened;
2. direct access to vectors
   - by word using *__getitem__()* (e.g. `file['hello']`)
   - by index using *vector_at()*
3. implements *__contains__()* (e.g. `'hello' in file`)
4. all the information needed to open the file are stored in the file itself

Specifics. The files contained in a VVM file are:

- *vocab.txt*: contains each word on a separate line
- *vectors.bin*: contains the vectors in binary format (concatenated)
- *meta.json*: must contain (at least) the following fields:
  - *vocab_size*: number of word vectors in the file
  - *vector_size*: length of a word vector
  - *encoding*: text encoding used for vocab.txt
  - *dtype*: vector data type string (notation used by numpy)

Variables

- **path** –
- **encoding** –
- **dtype** –
- **out_dtype** –
- **verbose** –
- **vocab** (*OrderedDict[str, int]*) – map each word to its index in the file

Parameters

- **path** (Union[str, Path]) –
- **out_dtype** (Union[str, dtype, None]) –
- **verbose** (int) –

DEFAULT_EXTENSION: str = '.vvm'

vocab_size: Optional[int]

words()

Returns an iterable for all the words in the file.

Return type Iterable[str]

__contains__(*word*)
Returns True if the file contains a vector for `word`

Return type `bool`

vector_at(*index*)
Returns a vector by its index in the file (random access).

Return type `ndarray`

__getitem__(*word*)
Returns the vector associated to a word (random access to file).

Return type `ndarray`

classmethod create(*out_path*, *word_vectors*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *dtype=None*)
Format-specific arguments are encoding and dtype.

Being VVM a tar file, you should use a compression supported by the tarfile package (avoid zip): gz, bz2 or xz.

See `create()` for more doc.

Return type `None`

classmethod create_from_file(*source_file*, *out_dir=None*, *out_filename=None*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *dtype=None*)
Format-specific arguments are encoding and dtype. Being VVM a tar file, you should use a compression supported by the tarfile package (avoid zip): gz, bz2 or xz.

See `create_from_file()` for more doc.

Return type `Path`

class embfile.formats.vvm.**VVMEmbFileReader**(*file*, *vectors_file*)
Bases: *embfile.core.reader.AbstractEmbFileReader*

EmbFileReader for the vvm format.

4.4.3 embfile.compression

Functions

extract_file(src_path[, member, dest_dir, ...])	Extracts a file compressed with gzip, bz2 or lzma or a member file inside a zip/tar archive.
extract_if_missing(src_path[, member, ...])	Extracts a file unless it already exists and returns its path.
open_file(path[, mode, encoding, compression])	Open a file, eventually with (de)compression.

Data

COMPRESSION_TO_EXTENSIONS	Maps each compression format to its associated extensions
EXTENSION_TO_COMPRESSION	Maps a compression extensions to the corresponding compression format name

Reference

embfile.compression.**open_file**(*path*, *mode='rt'*, *encoding=None*, *compression=None*)

Open a file, eventually with (de)compression.

If `compression` is not given, it is inferred from the file extension. If the file has not the extension of a supported compression format, the file is opened without compression, unless the argument `compression` is given.

embfile.compression.**extract_file**(*src_path*, *member=None*, *dest_dir='.'*, *dest_filename=None*, *overwrite=False*)

Extracts a file compressed with gzip, bz2 or lzma or a member file inside a zip/tar archive. The compression format is inferred from the extension or from the magic number of the file (in the case of zip and tar).

The file is first extracted to a .part file that is renamed when the extraction is completed.

Parameters

- **src_path** (Union[str, Path]) – source file path
- **member** (Optional[str]) – must be provided if src_path points to an archive that contains multiple files;
- **dest_dir** (Union[str, Path]) – destination directory; by default, it's the current working directory
- **dest_filename** (Optional[str]) – destination filename; by default, it's equal to `member` (if provided)
- **overwrite** (bool) – overwrite existing file at `dest_path` if it already exists

Return type Path

Returns *Path* – the path to the extracted file

embfile.compression.**extract_if_missing**(*src_path*, *member=None*, *dest_dir='.'*, *dest_filename=None*)

Extracts a file unless it already exists and returns its path.

Note: during extraction, a .part file is used, so there's no risk of using a partially extracted file.

Parameters

- **src_path** (Union[str, Path]) –
- **member** (Optional[str]) –
- **dest_dir** (Union[str, Path]) –
- **dest_filename** (Optional[str]) –

Return type Path

Returns The path of the decompressed file is returned.

embfile.compression.**EXTENSION_TO_COMPRESSION = {'.bz2': 'bz2', '.gz': 'gz', '.gzip': 'gz**
Maps a compression extensions to the corresponding compression format name

embfile.compression.**COMPRESSION_TO_EXTENSIONS = {'bz2': ['.bz2'], 'gz': ['.gz', '.gzip'],**
Maps each compression format to its associated extensions

4.4.4 embfile.errors

Reference

exception embfile.errors.**Error**
Bases: Exception

Base class of all errors raised by embfile

exception embfile.errors.**IllegalOperation**
Bases: *embfile.errors.Error*

Raised when the user attempts to perform an operation that is illegal in the current state (e.g. using a closed file)

exception embfile.errors.**BadEmbFile**
Bases: *embfile.errors.Error*

Raised when the file is malformed.

4.4.5 embfile.initializers

Embedding initializers.

Classes

Initializer()	A random number generator meant to be used with *build_matrix()*.
NormalInitializer()	Generates vectors using a normal distribution with the same mean and standard deviation of the set of vectors passed to the fit method.

Functions

normal([mean, deviation])	Returns a normal sampler.

Reference

class embfile.initializers.**Initializer**
Bases: abc.ABC

A random number generator meant to be used with *build_matrix()*. It can be fit to a sequence of other vectors in order to compute stats to be used for generation. When passed to build_matrix, the initializer is fit to the found vectors.

abstractmethod __call__(*shape*)
(Abstract method) Generate an array of shape shape

Return type ndarray

abstractmethod fit(*vectors*)
(Abstract method) Computes stats that will be use for generating new vectors.

Parameters **vectors** (ndarray) –

class embfile.initializers.**NormalInitializer**
Bases: *embfile.initializers.Initializer*

Generates vectors using a normal distribution with the same mean and standard deviation of the set of vectors passed to the fit method. When used with *build_matrix()*, it initializes out-of-file-vocabulary vectors so that they have the same mean and deviation of the vectors found in the file.

If not fit before to generate vectors, it raises IllegalOperation

fit(*vectors*)
Computes mean and standard deviation of the input vectors

embfile.initializers.**normal**(*mean=0.0*, *deviation=None*)
Returns a normal sampler. If deviation is not given, it is set dynamically to

1.0 / sqrt(shape[-1])

where shape[-1] is the vector size.

4.4.6 embfile.registry

Reference

class embfile.registry.**FormatsRegistry**
Bases: object

Maps each EmbFile subclass to a format_id and one or multiple file extensions.

Variables

- **id_to_class** –
- **extension_to_id** –
- **id_to_extensions** –

register_format(*embfile_class*, *format_id*, *extensions*, *overwrite=False*)
Registers a new embedding file format with a given id and associates the provided file extensions to it.

Parameters

- **embfile_class** (Type[EmbFile]) –
- **format_id** (str) –

- **extensions** (Iterable[str]) –
- **overwrite** (bool) –

associate_extension(*ext*, *format_id*, *overwrite=False*)
Associates a file extension to a registered embedding file format.

Parameters

- **ext** (str) –
- **format_id** (str) –
- **overwrite** (bool) –

extensions()

format_ids()

format_classes()

extension_to_class(*ext*)

4.4.7 embfile.types

Type aliases used in the library

Classes

VectorType	alias of numpy.ndarray

Data

DType	The central part of internal API.
PairsType	The central part of internal API.
PathType	The central part of internal API.

embfile.types.**VectorType**
alias of numpy.ndarray

4.4.8 embfile.word_vector

Classes

WordVector(word, vector)	A (word, vector) NamedTuple

Reference

class embfile.word_vector.**WordVector**(*word: str*, *vector: numpy.ndarray*)
Bases: tuple

A (word, vector) NamedTuple

Create new instance of WordVector(word, vector)

word
Alias for field number 0

vector
Alias for field number 1

staticmethod format_vector(*arr*)
Used by __repr__ to convert a numpy vector to string. Feel free to monkey-patch it.

Classes

BinaryEmbFile(path[, encoding, dtype, ...])	Format used by the Google word2vec tool.
BuildMatrixOutput(matrix, word2index, int], ...)	NamedTuple returned by *build_matrix()*
EmbFile(path[, out_dtype, verbose])	*(Abstract class)* The base class of all the embedding files.
TextEmbFile(path[, encoding, out_dtype, ...])	The format used by Glove and FastText files. Each vector pair is stored as a line of text made of space-separated fields::.
VVMEmbFile(path[, out_dtype, verbose])	(Custom format) A tar file storing vocabulary, vectors and metadata in 3 separate files.

Functions

associate_extension(ext, format_id[, overwrite])	Associates a file extension to a registered embedding file format.
build_matrix(f, words[, start_index, dtype, ...])	Creates an embedding matrix for the provided words.
extract(src_path[, member, dest_dir, ...])	Extracts a file compressed with gzip, bz2 or lzma or a member file inside a zip/tar archive.
extract_if_missing(src_path[, member, ...])	Extracts a file unless it already exists and returns its path.
open(path[, format_id])	Opens an embedding file inferring the file format from the file extension (if not explicitly provided in format_id).
register_format(format_id, extensions[, ...])	Class decorator that associates a new EmbFile subclass with a format_id and one or multiple extensions.

Data

FORMATS	Maps each EmbFile subclass to a format_id and one or multiple file extensions.

embfile.**open**(*path*, *format_id=None*, ***format_kwargs*)

Opens an embedding file inferring the file format from the file extension (if not explicitly provided in format_id). Note that you can always open a file using the specific EmbFile subclass; it can be more convenient since you get auto-completion and quick doc for format-specific arguments.

Example:

```
with embfile.open('path/to/embfile.txt') as f:
    # do something with f
```

Supported formats:

Class	format_id	Extensions	Description
TextEmbFile	txt	.txt, .vec	Glove/fastText format
BinaryEmbFile	bin	.bin	Google word2vec format
VVMEmbFile	vvm	.vvm	A tarball containing three files: vocab.txt, vectors.bin, meta.json

You can **register new formats or extensions** using the functions *embfile.register_format()* and *embfile.associate_extension()*.

Parameters

- **path** (Union[str, Path]) – path to the file
- **format_id** (Optional[str]) – string ID of the embedding file format. If not provided, it is inferred from the file name. Valid choices are: 'txt', 'bin', 'vvm'.
- **format_kwargs** – additional format-specific arguments (see doc for specific file formats)

Return type EmbFile

Returns An instance of a concrete subclass of *EmbFile* .

See also:

***embfile.register_format()*:** registers your custom EmbFile implementation so it is recognized by this function

***embfile.associate_extension()*:** associates an extension to a registered format

embfile.**build_matrix**(*f*, *words*, *start_index=0*, *dtype=None*, *oov_initializer=<embfile.initializers.NormalInitializer object>*, *verbose=None*)

Creates an embedding matrix for the provided words. words can be:

1. an **iterable of strings** – in this case, the words in the iterable are mapped to consecutive rows of the matrix starting from the row of index start_index (by default, 0); the rows with index i < start_index are left to zeros.
2. a **dictionary** {word -> index} that maps each word to a row – in this case, the matrix has shape:

```
[max_index + 1, vector_size]
```

where `max_index = max(word_to_index.values())`. The rows that are not associated with any word are left to zeros. If multiple words are mapped to the same row, the function raises `ValueError`.

In both cases, all the word vectors that are not found in the file are initialized using `oov_initializer`, which can be:

1. `None` – leave missing vectors to zeros
2. a function that takes the shape of the array to generate (a tuple) as first argument:

```
oov_initializer=lambda shape: numpy.random.normal(scale=0.01, size=shape)
oov_initializer=numpy.ones  # don't use this for word vectors :|
```

3. an instance of *Initializer*, which is a "fittable" initializer; in this case, the initializer is fit on the found vectors (the vectors that are both in `vocab` and in the file).

By default, *oov_initializer* is an instance of *NormalInitializer* which generates vectors using a normal distribution with the same mean and standard deviation of the vectors found.

Parameters

- **f** (`EmbFile`) – the file containing the word vectors
- **words** (*Iterable[str] or Dict[str, int]*) – iterable of words or dictionary that maps each word to a row index
- **start_index** (*int*) – ignored if `vocab` is a dict; if `vocab` is a collection, determines the index associated to the first word (and so, the number of rows left to zeros at the beginning of the matrix)
- **dtype** (*optional, DType*) – matrix data type; if `None`, `cls.out_dtype` is used
- **oov_initializer** (optional, Callable or *Initializer*) – initializer for out-of-(file)-vocabulary word vectors. See the class docstring for more information.
- **verbose** (*bool*) – if None, f.verbose is used

Return type `BuildMatrixOutput`

class embfile.**BuildMatrixOutput**(*matrix: numpy.ndarray*, *word2index: Dict[str, int]*, *missing_words: Set[str]*)

Bases: `tuple`

NamedTuple returned by *build_matrix()*

Create new instance of BuildMatrixOutput(matrix, word2index, missing_words)

matrix
Alias for field number 0

word2index
Alias for field number 1

missing_words
Alias for field number 2

found_words()

word_indexes(*words*)

Return type `List[int]`

vector(*word*)

pretty(*precision=3*, *threshold=5*)
Pretty string method for documentation purposes.

class embfile.**EmbFile**(*path*, *out_dtype=None*, *verbose=True*)
Bases: abc.ABC

(Abstract class) The base class of all the embedding files.

Sub-classes must:

1. ensure they set attributes *vocab_size* and vector_size when a file instance is created
2. implement a *EmbFileReader* for the format and implements the abstract method *_reader()*
3. implement the abstract method *_close()*
4. *(optionally)* implement a *VectorsLoader* (if they can improve upon the default loader) and override *loader()*
5. *(optionally)* implement a EmbFileCreator for the format and set the class constant Creator

Parameters

- **path** (*Path*) – path of the embedding file (eventually compressed)
- **out_dtype** (*numpy.dtype*) – all the vectors will be converted to this data type. The sub-class is responsible to set a suitable default value.
- **verbose** (*bool*) – whether to show a progress bar by default in all time-consuming operations

Variables

- **path** (*Path*) – path of the embedding file
- ***vocab_size*** (int or None) – number of words in the file (can be None for some TextEmbFile)
- **vector_size** (*int*) – length of the vectors
- **verbose** (*bool*) – whether to show a progress bar by default in all time-consuming operations
- **closed** (*bool*) – True if the file was closed

abstractmethod _reader()
(Abstract method) Returns a new reader for the file which allows to iterate efficiently the word-vectors inside it. Called by *reader()*.

Return type *EmbFileReader*

abstractmethod _close()
(Abstract method) Releases eventual resources used by the EmbFile.

Return type None

DEFAULT_EXTENSION: str

close()
Releases all the open resources linked to this file, including the opened readers.

Return type None

reader()
Creates and returns a new file reader. When the file is closed, all the still opened readers are closed automatically.

Return type *EmbFileReader*

loader(*words*, *missing_ok=True*, *verbose=None*)
Returns a *VectorsLoader*, an iterator that looks for the provided words in the file and yields available (word, vector) pairs one by one. If `missing_ok=True` (default), provides the set of missing words in the property `missing_words` (once the iteration ends).

See *embfile.core.VectorsLoader* for more info.

Example

You should use a loader when you need to load many vectors in some custom data structure and you don't want to waste memory (e.g. build_matrix uses it to load the vectors directly into the matrix):

```
data_structure = MyCustomStructure()
with file.loader(many_words) as loader:
    for word, vector in loader:
        data_structure[word] = vector
print('Number of missing words:', len(loader.missing_words)
```

See also:

load() find()

Return type *VectorsLoader*

for ... in words()
Returns an iterable for all the words in the file.

Return type Iterable[str]

for ... in vectors()
Returns an iterable for all the vectors in the file.

Return type Iterable[ndarray]

for ... in word_vectors()
Returns an iterable for all the (word, vector) word_vectors in the file.

Return type Iterable[*WordVector*]

to_dict(*verbose=None*)
Returns the entire file content in a dictionary word -> vector.

Return type Dict[str, ndarray]

to_list(*verbose=None*)
Returns the entire file content in a list of `WordVector`'s.

Return type List[*WordVector*]

load(*words*, *verbose=None*)
Loads the vectors for the input words in a `{word: vec}` dict, raising `KeyError` if any word is missing.

Parameters

- **words** (Iterable[str]) – the words to get

- **verbose** (Optional[bool]) – if None, self.verbose is used

Return type Dict[str, ndarray]

Returns *(Dict[str, VectorType])* – a dictionary {word: vector}

See also:

find() - it returns the set of all missing words, instead of raising KeyError.

find(*words*, *verbose=None*)

Looks for the input words in the file, return: 1) a dict {word: vec} containing the available words and 2) a set containing the words not found.

Parameters

- **words** (*Iterable[str]*) – the words to look for
- **verbose** (Optional[bool]) – if None, self.verbose is used

Return type _FindOutput

Returns

namedtuple – a namedtuple with the following fields:

- **word2vec** (*Dict[str, VectorType]*): dictionary {word: vector}
- **missing_words** (*Set[str]*): set of words not found in the file

See also:

load() - which raises KeyError if any word is not found in the file.

for ... in filter(*condition*, *verbose=None*)

Returns a generator that yields a word vector pair for each word in the file that satisfies a given condition. For example, to get all the words starting with "z":

```
list(file.filter(lambda word: word.startswith('z')))
```

Parameters

- **condition** (Callable[[str], bool]) – a function that, given a word in input, outputs True if the word should be taken
- **verbose** (Optional[bool]) – if True, a progress bar is showed (the bar is updated each time a word is read, not each time a word vector pair is yielded).

Return type Iterator[Tuple[str, ndarray]]

save_vocab(*path=None*, *encoding='utf-8'*, *overwrite=False*, *verbose=None*)

Save the vocabulary of the embedding file on a text file. By default the file is saved in the same directory of the embedding file, e.g.:

```
/path/to/filename.txt.gz  ==> /path/to/filename_vocab.txt
```

Parameters

- **path** (Union[str, Path, None]) – where to save the file
- **encoding** (str) – text encoding
- **overwrite** (bool) – if the file exists and it is True, overwrite the file
- **verbose** (Optional[bool]) – if None, self.verbose is used

Return type Path

Returns *(Path)* – the path to the vocabulary file

classmethod create(*out_path*, *word_vectors*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, ***format_kwargs*)

Creates a file on disk containing the provided word vectors.

Parameters

- **out_path** (Union[str, Path]) – path to the created file
- **word_vectors** (*Dict[str, VectorType] or Iterable[Tuple[str, VectorType]]*) – it can be an iterable of word vector tuples or a dictionary `word -> vector`; the word vectors are written in the order determined by the iterable object.
- **vocab_size** (Optional[int]) – it must be provided if `word_vectors` has no `__len__` and the specific-format creator needs to know a priori the vocabulary size; in any case, the creator should check at the end that the provided `vocab_size` matches the actual length of `word_vectors`
- **compression** (Optional[str]) – valid values are: `"bz2"|"bz"`, `"gzip"|"gz"`, `"xz"|"lzma"`, `"zip"`
- **verbose** (bool) – if positive, show progress bars and information
- **overwrite** (bool) – overwrite the file if it already exists
- **format_kwargs** – format-specific arguments

Return type None

classmethod create_from_file(*source_file*, *out_dir=None*, *out_filename=None*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, ***format_kwargs*)

Creates a new file on disk with the same content of another file.

Parameters

- **source_file** (EmbFile) – the file to take data from
- **out_dir** (Union[str, Path, None]) – directory where the file will be stored; by default, it's the parent directory of the source file
- **out_filename** (Optional[str]) – filename of the produced name (inside `out_dir`); by default, it is obtained by replacing the extension of the source file with the proper one and appending the compression extension if `compression is not None`. **Note:** if you pass this argument, the compression extension is not automatically appended.
- **vocab_size** (Optional[int]) – if the source EmbFile has attribute `vocab_size == None`, then: if the specific creator requires it (bin and txt formats do), it *must* be provided; otherwise it *can* be provided for having ETA in progress bars.
- **compression** (Optional[str]) – valid values are: `"bz2"|"bz"`, `"gzip"|"gz"`, `"xz"|"lzma"`, `"zip"`
- **verbose** (bool) – print info and progress bar
- **overwrite** (bool) – overwrite a file with the same name if it already exists
- **format_kwargs** – format-specific arguments (see above)

Return type Path

class embfile.**BinaryEmbFile**(*path*, *encoding='utf-8'*, *dtype=dtype('float32')*, *out_dtype=None*, *verbose=True*)

Bases: embfile.core._file.EmbFile

Format used by the Google word2vec tool. You can use it to read the file GoogleNews-vectors-negative300.bin.

It begins with a text header line of space-separated fields:

```
<vocab_size> <vector_size>
```

Each word vector pair is encoded as following:

- encoded word + space
- followed by the binary representation of the vector.

Variables

- **path** –
- **encoding** –
- **dtype** –
- **out_dtype** –
- **verbose** –

Parameters

- **path** (Union[str, Path]) – path to the (eventually compressed) file
- **encoding** (str) – text encoding; **note:** if you provide an utf encoding (e.g. *utf-16*) that uses a BOM (Byte Order Mark) without specifying the byte-endianness (e.g. *utf-16-le* or *utf-16-be*), the little-endian version is used (*utf-16-le*).
- **dtype** (Union[str, dtype]) – a valid numpy data type (or whatever you can pass to numpy.dtype()) (default: '<f4'; little-endian float, 4 bytes)
- **out_dtype** (Union[str, dtype, None]) – all the vectors returned will be (eventually) converted to this data type; by default, it is equal to the original data type of the vectors in the file, i.e. no conversion takes place.

DEFAULT_EXTENSION: str = '.bin'

vocab_size: Optional[int]

classmethod create(*out_path*, *word_vectors*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *dtype=None*)

Format-specific arguments are encoding and dtype.

Note: all the text is encoded without BOM (Byte Order Mark). If you pass "utf-16" or "utf-18", the little-endian version is used (e.g. "utf-16-le")

See create() for more.

Return type None

classmethod create_from_file(*source_file*, *out_dir=None*, *out_filename=None*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *dtype=None*)

Format-specific arguments are encoding and dtype.

Note: all the text is encoded without BOM (Byte Order Mark). If you pass "utf-16" or "utf-18", the little-endian version is used (e.g. "utf-16-le")

See `create_from_file()` for more.

Return type `Path`

class `embfile.`**TextEmbFile**(*path*, *encoding='utf-8'*, *out_dtype='float32'*, *vocab_size=None*, *verbose=True*)

Bases: `embfile.core._file.EmbFile`

The format used by Glove and FastText files. Each vector pair is stored as a line of text made of space-separated fields:

```
word vec[0] vec[1] ... vec[vector_size-1]
```

It may have or not an (automatically detected) "header", containing `vocab_size` and `vector_size` (in this order).

If the file doesn't have a header, `vector_size` is set to the length of the first vector. If you know `vocab_size` (even an approximate value), you may want to provide it to have ETA in progress bars.

If the file has a header and you provide `vocab_size`, the provided value is ignored.

Compressed files are decompressed while you proceed reeding. Note that each file reader will decompress the file independently, so if you need to read the file multiple times it's better you decompress the entire file first and then open it.

Variables

- **path** –
- **encoding** –
- **out_dtype** –
- **verbose** –

Parameters

- **path** (`Union[str, Path]`) – path to the embedding file
- **encoding** (`str`) – encoding of the text file; default is utf-8
- **out_dtype** (`Union[str, dtype]`) – the dtype of the vectors that will be returned; default is single-precision float
- **vocab_size** (`Optional[int]`) – useful when the file has no header but you know vocab_size; if the file has a header, this argument is ignored.
- **verbose** (`int`) – default level of verbosity for all methods

DEFAULT_EXTENSION: str = '.txt'

vocab_size: Optional[int]

classmethod create(*out_path*, *word_vectors*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *precision=5*)

Creates a file on disk containing the provided word vectors.

Parameters

- **out_path** (`Union[str, Path]`) – path to the created file
- **word_vectors** (*`Dict[str, VectorType]` or `Iterable[Tuple[str, VectorType]]`*) – it can be an iterable of word vector tuples or a dictionary `word -> vector`; the word vectors are written in the order determined by the iterable object.

- **vocab_size** (Optional[int]) – it must be provided if `word_vectors` has no `__len__` and the specific-format creator needs to know a priori the vocabulary size; in any case, the creator should check at the end that the provided `vocab_size` matches the actual length of `word_vectors`
- **compression** (Optional[str]) – valid values are: `"bz2"|"bz"`, `"gzip"|"gz"`, `"xz"|"lzma"`, `"zip"`
- **verbose** (bool) – if positive, show progress bars and information
- **overwrite** (bool) – overwrite the file if it already exists
- **format_kwargs** – format-specific arguments

Return type None

classmethod create_from_file(*source_file*, *out_dir=None*, *out_filename=None*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *precision=5*)

Creates a new file on disk with the same content of another file.

Parameters

- **source_file** (EmbFile) – the file to take data from
- **out_dir** (Union[str, Path, None]) – directory where the file will be stored; by default, it's the parent directory of the source file
- **out_filename** (Optional[str]) – filename of the produced name (inside `out_dir`); by default, it is obtained by replacing the extension of the source file with the proper one and appending the compression extension if `compression is not None`. **Note:** if you pass this argument, the compression extension is not automatically appended.
- **vocab_size** (Optional[int]) – if the source EmbFile has attribute `vocab_size == None`, then: if the specific creator requires it (bin and txt formats do), it *must* be provided; otherwise it *can* be provided for having ETA in progress bars.
- **compression** (Optional[str]) – valid values are: `"bz2"|"bz"`, `"gzip"|"gz"`, `"xz"|"lzma"`, `"zip"`
- **verbose** (bool) – print info and progress bar
- **overwrite** (bool) – overwrite a file with the same name if it already exists
- **format_kwargs** – format-specific arguments (see above)

Return type Path

class embfile.**VVMEmbFile**(*path*, *out_dtype=None*, *verbose=True*)

Bases: embfile.core._file.EmbFile, embfile.core.loaders.Word2Vector

(Custom format) A tar file storing vocabulary, vectors and metadata in 3 separate files.

Features:

1. the vocabulary can be loaded very quickly (with no need for an external vocab file) and it is loaded in memory when the file is opened;
2. direct access to vectors
   - by word using *__getitem__()* (e.g. `file['hello']`)
   - by index using *vector_at()*
3. implements *__contains__()* (e.g. `'hello' in file`)

4. all the information needed to open the file are stored in the file itself

Specifics. The files contained in a VVM file are:

- *vocab.txt*: contains each word on a separate line
- *vectors.bin*: contains the vectors in binary format (concatenated)
- *meta.json*: must contain (at least) the following fields:
  - *vocab_size*: number of word vectors in the file
  - *vector_size*: length of a word vector
  - *encoding*: text encoding used for vocab.txt
  - *dtype*: vector data type string (notation used by numpy)

Variables

- **path** –
- **encoding** –
- **dtype** –
- **out_dtype** –
- **verbose** –
- **vocab** (*OrderedDict[str, int]*) – map each word to its index in the file

Parameters

- **path** (Union[str, Path]) –
- **out_dtype** (Union[str, dtype, None]) –
- **verbose** (int) –

DEFAULT_EXTENSION: str = '.vvm'

vocab_size: Optional[int]

words()
Returns an iterable for all the words in the file.

Return type Iterable[str]

__contains__(*word*)
Returns True if the file contains a vector for word

Return type bool

vector_at(*index*)
Returns a vector by its index in the file (random access).

Return type ndarray

__getitem__(*word*)
Returns the vector associated to a word (random access to file).

Return type ndarray

classmethod create(*out_path*, *word_vectors*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *dtype=None*)
Format-specific arguments are encoding and dtype.

Being VVM a tar file, you should use a compression supported by the tarfile package (avoid zip): gz, bz2 or xz.

See `create()` for more doc.

Return type None

classmethod create_from_file(*source_file*, *out_dir=None*, *out_filename=None*, *vocab_size=None*, *compression=None*, *verbose=True*, *overwrite=False*, *encoding='utf-8'*, *dtype=None*)

Format-specific arguments are encoding and dtype. Being VVM a tar file, you should use a compression supported by the tarfile package (avoid zip): gz, bz2 or xz.

See `create_from_file()` for more doc.

Return type Path

embfile.**register_format**(*format_id*, *extensions*, *overwrite=False*)

Class decorator that associates a new `EmbFile` sub-class with a format_id and one or multiple extensions. Once you register a format, you can use *open()* to open files of that format.

embfile.**associate_extension**(*ext*, *format_id*, *overwrite=False*)

Associates a file extension to a registered embedding file format.

embfile.**extract**(*src_path*, *member=None*, *dest_dir='.'*, *dest_filename=None*, *overwrite=False*)

Extracts a file compressed with gzip, bz2 or lzma or a member file inside a zip/tar archive. The compression format is inferred from the extension or from the magic number of the file (in the case of zip and tar).

The file is first extracted to a .part file that is renamed when the extraction is completed.

Parameters

- **src_path** (Union[str, Path]) – source file path
- **member** (Optional[str]) – must be provided if src_path points to an archive that contains multiple files;
- **dest_dir** (Union[str, Path]) – destination directory; by default, it's the current working directory
- **dest_filename** (Optional[str]) – destination filename; by default, it's equal to `member` (if provided)
- **overwrite** (bool) – overwrite existing file at `dest_path` if it already exists

Return type Path

Returns *Path* – the path to the extracted file

embfile.**extract_if_missing**(*src_path*, *member=None*, *dest_dir='.'*, *dest_filename=None*)

Extracts a file unless it already exists and returns its path.

Note: during extraction, a .part file is used, so there's no risk of using a partially extracted file.

Parameters

- **src_path** (Union[str, Path]) –
- **member** (Optional[str]) –
- **dest_dir** (Union[str, Path]) –
- **dest_filename** (Optional[str]) –

Return type Path

Returns The path of the decompressed file is returned.

4.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.5.2 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/janLuke/embfile/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.5.3 Development

To set up *embfile* for local development:

1. Fork embfile (look for the "Fork" button).
2. Clone your fork locally:

   ```
   git clone git@github.com:your_name_here/embfile.git
   ```

3. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

   ```
   tox
   ```

5. Commit your changes and push your branch to GitHub:

   ```
   git add .
   git commit -m "Your detailed description of your changes."
   git push origin name-of-your-bugfix-or-feature
   ```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

Testing tips

To run all the tests run:

```
tox
```

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

4.6 Authors

- Gianluca Gippetto - gianluca.gippetto@gmail.com

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

4.7 Changelog

4.7.1 v0.1.0 (2020-01-24)

- First release on PyPI.

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

I

L

M

N

O

P

R

S

T

V

W