# word-vectors

*Release 4.0.0*

**Brian Lester**

**Jun 27, 2020**

# CONTENTS:

CHAPTER

# ONE

# USING PIP

PyPI install with `pip`:

```
pip install word-vectors
```

CHAPTER

TWO

# FROM SOURCE

To install from the source, clone the github repository and install with pip.

```
git clone https://github.com/blester125/word-vectors.get
cd word-vectors
pip install .
```

## 2.1 Local Development

If you want to install the package and run tests install the optional testing dependencies.

```
pip install .[test]
```

Run the tests with `pytest`.

```
pytest
```

Set up `pre-commit` hooks to autoformat your changes with black.

```
pip install pre-commit
pre-commit install
```

## 2.2 Building the Docs

To build the documentation locally install the documentation requirements and run make.

```
pip install -r requirements-docs.txt
cd docs
make html
open build/html/index.html
```

CHAPTER

# THREE

# API

## 3.1 word_vectors

Read, Write, and Convert between different word vector serialization formats.

word_vectors.**Vocab = typing.Dict[str, int]**
A mapping of word to integer index. This index is used pull the this words vector from the matrix of word vectors.

word_vectors.**Vectors = <class 'numpy.ndarray'>**
The actual word vectors. These are always of rank 2 and have the shape `[vocab size, vector size]`

**class** word_vectors.**FileType**(*value*)
An Enumeration of the Word Vector file types supported.

**GLOVE = 'glove'**
The format used by Glove. See *read_glove()* for a description of file format and common pre-trained embeddings that use this format.

**W2V_TEXT = 'w2v-text'**
The text format introduced by Word2Vec. See *read_w2v_text()* for a description of the file format and common pre-trained embeddings that use this format.

**W2V = 'w2v'**
The binary format used by Word2Vec and pre-trained GoogleNews vectors. See *read_w2v()* for a description of the file format and common pre-trained embeddings that use this format.

**LEADER = 'leader'**
Our new Leader file format. See *read_leader()* for a description of the file format.

**FASTTEXT = 'w2v-text'**
The file format used to distribute FastText vectors, it is just the word2vec text format. See *read_w2v_text()* for a description of the file format.

**NUMBERBATCH = 'w2v-text'**
The file format used to distribute Numberbatch vectors, it is just the word2vec text format. See *read_w2v_text()* for a description of the file format.

**classmethod from_string**(*value*)
Convert a string into the Enum value.

**Parameters** **value** (*str*) – The string specifying the file type.

**Returns** The Enum value parsed from the string.

**Raises** **ValueError** – If the string wasn't able to be parsed into an Enum value.

**Return type** *word_vectors.FileType*

word_vectors.**INT_SIZE = 4**
The size of an int32 in bytes used when reading binary files.

word_vectors.**FLOAT_SIZE = 4**
The size of a float32 in bytes when reading a binary file.

word_vectors.**LONG_SIZE = 8**
The size of an int64 in bytes when reading binary files.

word_vectors.**LEADER_HEADER = 3**
The number of elements in the Leader format header.

word_vectors.**LEADER_MAGIC_NUMBER = 38941**
A magic number used to identify a Leader format file.

## 3.2 word_vectors.read

Read word vectors from a file.

We provide a main *read()* function for reading vectors from a file. The serialization format can be explicitly provided with by passing a *FileType* or automatically inferred using *sniff()*. There are also several provided convenience functions for reading from specific formats.

word_vectors.read.**read**(*f*, *file_type=None*)
Read vectors from a file.

This function can dispatch to one of the following word vector format readers:

- *read_glove()*
- *read_w2v_text()*
- *read_w2v()*
- *read_leader()*

Check the documentation of a specific reader to see a description of the file format as well as common pre-trained vectors that ship with this format.

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

---

**Note:** Without a specified file type this function uses *word_vectors.read.sniff()* to determine the word vector format and dispatches to the appropriate reader.

I haven't seen a sniffing failure but if your file type can't be determined you can pass the `file_type` explicitly or call the specific reading function yourself.

---

**Parameters**

- **f** (*Union[str, TextIO, BinaryIO]*) – The file to read from.
- **file_type** (*Optional[word_vectors.FileType]*) – The vector file format. If `None` the file is sniffed to determine format.

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

word_vectors.read.**read_with_vocab**(*f*, *user_vocab*, *initializer=<function uniform_initializer.<locals>._unif_initializer>*, *keep_extra=False*, *file_type=None*)

Read vectors from a file subject user provided vocabulary constraints.

This function can dispatch to one of the following word vector format readers:

- *read_glove_with_vocab()*
- *read_w2v_text_with_vocab()*
- *read_w2v_with_vocab()*
- *read_leader_with_vocab()*

Check the documentation of a specific reader to see a description of the file format as well as common pre-trained vectors that ship with this format.

When provided a vocabulary this function will not reorder it. If you pass in that the word `dog` is index `12` then in the resulting vocabulary it will still be index `12`.

When collecting extra vocabulary (words that are in the pre-trained embeddings but not in the user vocab) these will all be at the end of the vocabulary. Again the indices of user provided words will not change.

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

---

**Note:** Without a specified file type this function uses *word_vectors.read.sniff()* to determine the word vector format and dispatches to the appropriate reader.

I haven't seen a sniffing failure but if your file type can't be determined you can pass the `file_type` explicitly or call the specific reading function yourself.

---

**Parameters**

- **f** (*Union[str, IO]*) – The file to read from.
- **user_vocab** (*Dict[str, int]*) – A specific vocabulary the user wants to extract form the pre-trained embeddings.
- **initializer** (*Callable[[int], numpy.ndarray]*) – A function that takes the vector size and generates a new vector. this is used to generate a representation for a word in the user vocab that is not in the pre-train embeddings.
- **keep_extra** (*bool*) – Should you also include vectors that are in the pre-trained embedding but not in the user provided vocab?
- **file_type** (*Optional[word_vectors.FileType]*) – The vector file format. If `None` the file is sniffed to determine format.

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

word_vectors.read.**read_glove**(*f*)

Read vectors from a glove file.

The GloVe format is a pure text format. Each (word, vector) pair is represented by a single line in the file. The line starts with the word, a space, and then the float32 text representations of the elements in the vector associated with that word. Each of these vector elements are also separated with a space.

The main vectors distributed in this format are the GloVe vectors (Pennington, et. al., 2014)

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

**Parameters** **f** (*Union[str, TextIO]*) – The file to read from

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

word_vectors.read.**read_glove_with_vocab**(*f*, *user_vocab*, *initializer=<function uniform_initializer.<locals>._unif_initializer>*, *keep_extra=False*)

Read vectors from a glove file subject to user vocabulary constraints.

See *read_glove()* for a description of the file format and common pre-train embeddings that use this format.

When provided a vocabulary this function will not reorder it. If you pass in that the word `dog` is index `12` then in the resulting vocabulary it will still be index `12`.

When collecting extra vocabulary (words that are in the pre-trained embeddings but not in the user vocab) these will all be at the end of the vocabulary. Again the indices of user provided words will not change.

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

**Parameters**

- **f** (*Union[str, TextIO]*) – The file to read from.
- **user_vocab** (*Dict[str, int]*) – A specific vocabulary the user wants to extract form the pre-trained embeddings.
- **initializer** (*Callable[[int], numpy.ndarray]*) – A function that takes the vector size and generates a new vector. this is used to generate a representation for a word in the user vocab that is not in the pre-train embeddings.
- **keep_extra** (*bool*) – Should you also include vectors that are in the pre-trained embedding but not in the user provided vocab?

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

word_vectors.read.**read_w2v_text**(*f*)

Read vectors from a text based w2v file.

One of two different vector serialization formats introduced in the word2vec software (Mikolov, et. al., 2013).

The word2vec text format is a pure text format. The first line is two integers, represented as text and separated by a space, that specify the number of types in the vocabulary and the size of the word vectors respectively. Each following line represents a (word, vector) pair. The line stars with the word, a space, and then the float 32 text representations of the elements in the vector associated with that word. Each of these vector elements are also separated with a space.

One can see that that this is actually the same as the *GloVe* format except that in GloVe they removed the header line.

The main embeddings distributed in this format are FastText (Bojanowski, et. al., 2017) and NumberBatch (Speer, et. al., 2017)

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

**Parameters** **f** (*Union[str, TextIO]*) – The file to read from

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

word_vectors.read.**read_w2v_text_with_vocab**(*f*, *user_vocab*, *initializer=<function uniform_initializer.<locals>._unif_initializer>*, *keep_extra=False*)

Read vectors from a Word2Vec text file subject to user vocabulary constraints.

See *read_w2v_text()* for a description of the file format and common pre-train embeddings that use this format.

When provided a vocabulary this function will not reorder it. If you pass in that the word `dog` is index `12` then in the resulting vocabulary it will still be index `12`.

When collecting extra vocabulary (words that are in the pre-trained embeddings but not in the user vocab) these will all be at the end of the vocabulary. Again the indices of user provided words will not change.

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

**Parameters**

- **f** (*Union[str, TextIO]*) – The file to read from.
- **user_vocab** (*Dict[str, int]*) – A specific vocabulary the user wants to extract form the pre-trained embeddings.
- **initializer** (*Callable[[int], numpy.ndarray]*) – A function that takes the vector size and generates a new vector. this is used to generate a representation for a word in the user vocab that is not in the pre-train embeddings.
- **keep_extra** (*bool*) – Should you also include vectors that are in the pre-trained embedding but not in the user provided vocab?

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

`word_vectors.read.`**`read_w2v`**(*f*)

Read vectors from a word2vec file.

One of two different vector serialization formats introduced in the word2vec software (Mikolov, et. al., 2013).

The word2vec binary format is a mix of textual an binary representations. The first line is two integers (as text, separated be a space) representing the number of types in the vocabulary and the size of the word vectors respectively. (word, vector) pairs follow. The word is represented as text and a space. After the space each element of a vector is represented as a binary float32.

The most well-known pre-trained embeddings distributed in this format are the GoogleNews vectors.

---

**Note:** There is no formal definition of this file format, the only definitive reference on it is the original implementation in the word2vec software

Due to the lack of a definition (and no special handling of it in the code) there is no explicit statements about the endianness of the binary representations. Most code just uses the `numpy.from_buffer` and that seems to work now that most people have little-endian machines. However due to the lack of explicit direction on this encoding I would advise caution when loading vectors that were trained on big-endian hardware.

---

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

**Parameters** **`f`** (*`Union[str, BinaryIO]`*) – The file to read from

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

`word_vectors.read.`**`read_w2v_with_vocab`**(*f*, *user_vocab*, *initializer=<function uniform_initializer.<locals>._unif_initializer>*, *keep_extra=False*)

Read vectors from a Word2Vec file subject to user vocabulary constraints.

See *`read_w2v()`* for a description of the file format and common pre-train embeddings that use this format.

When provided a vocabulary this function will not reorder it. If you pass in that the word `dog` is index `12` then in the resulting vocabulary it will still be index `12`.

When collecting extra vocabulary (words that are in the pre-trained embeddings but not in the user vocab) these will all be at the end of the vocabulary. Again the indices of user provided words will not change.

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file to read from.
- **user_vocab** (*Dict[str, int]*) – A specific vocabulary the user wants to extract form the pre-trained embeddings.
- **initializer** (*Callable[[int], numpy.ndarray]*) – A function that takes the vector size and generates a new vector. this is used to generate a representation for a word in the user vocab that is not in the pre-train embeddings.
- **keep_extra** (*bool*) – Should you also include vectors that are in the pre-trained embedding but not in the user provided vocab?

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

word_vectors.read.**read_leader**(*f*)

Read vectors from a leader file.

This is our fully binary vector format.

The first line is a header for the leader format and it is a 3-tuple. The elements of this tuple are: A magic number, the size of the vocabulary, and the size of the vectors. These numbers are represented as little-endian unsigned long longs that have a size of 8 bytes.

Following the header there are (length, word, vector) tuples. The length is the length of this particular word encoded as a little-endian unsigned integer. The word is stored as `utf-8` bytes. After the word the vector is stored where each element is a little-endian float32 (4 bytes).

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

**Parameters** **f** (*Union[str, BinaryIO]*) – The file to read from

**Returns** The vocab and vectors.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

word_vectors.read.**read_leader_with_vocab**(*f*, *user_vocab*, *initializer=<function uniform_initializer.<locals>._unif_initializer>*, *keep_extra=False*)

Read vectors from a Leader file subject to user vocabulary constraints.

See *read_leader()* for a description of the file format and common pre-train embeddings that use this format.

When provided a vocabulary this function will not reorder it. If you pass in that the word `dog` is index `12` then in the resulting vocabulary it will still be index `12`.

When collecting extra vocabulary (words that are in the pre-trained embeddings but not in the user vocab) these will all be at the end of the vocabulary. Again the indices of user provided words will not change.

---

**Note:** In the case of duplicated words in the saved vectors we use the index and associated vector from the first occurrence of the word.

---

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file to read from.
- **user_vocab** (*Dict[str, int]*) – A specific vocabulary the user wants to extract form the pre-trained embeddings.
- **initializer** (*Callable[[int], numpy.ndarray]*) – A function that takes the vector size and generates a new vector. this is used to generate a representation for a word in the user vocab that is not in the pre-train embeddings.
- **keep_extra** (*bool*) – Should you also include vectors that are in the pre-trained embedding but not in the user provided vocab?

**Returns** The vocab and vectors. The vocab is a mapping from word to integer and vectors are a numpy array of shape `[vocab size, vector size]`. The vocab gives the index offset into the vector matrix for some word.

**Return type** Tuple[Dict[str, int], numpy.ndarray]

word_vectors.read.**sniff**(*f*, *buf_size=1024*)

Figure out what kind of vector file it is.

**Parameters**

- **f** (*Union[str, TextIO]*) – The file we are sniffing.
- **buf_size** (*int*) – How many bytes to read in when sniffing the file.

**Returns** The guessed file type.

**Return type** *word_vectors.FileType*

word_vectors.read.**read_leader_header**(*buf*)

Read the header from the leader file.

The header for the leader format is a 3-tuple. The elements of this tuple are: A magic number, the size of the vocabulary, and the size of the vectors. These numbers are represented as little-endian unsigned long longs that have a size of 8 bytes.

---

**Note:** The magic number if used to make sure this is can actual file and not just trying to extract word vectors from a random binary file. The Magic Number is `38941`.

---

**Parameters** **buf** (*bytes*) – The beginning of the file we are reading the header from.

**Returns** The vocab size, the vector size, and the maximum length of any of the words

**Raises** **ValueError** – If the magic number doesn't match.

**Return type** Tuple[int, int]

word_vectors.read.**verify_leader**(*buf*)

Check if a file is in the leader format by comparing the magic number.

**Parameters**

- **buf** (*bytes*) – The beginning of the file we are trying to determine if the it
- **a Leader formatted file.** (*is*) –

**Returns** True if the magic number matched, False otherwise.

**Return type** bool

## 3.3 word_vectors.write

Write Word Vectors to a file.

We provide the main *write()* function that can write to various vector serialization formats based on the passed FileType. There are also several convenience functions for writing specific formats.

word_vectors.write.**write**(*wf*, *vocab*, *vectors*, *file_type*, *max_len=None*)

Write word vectors to a file.

This function dispatches to on of the following word vector format writers based on the file of file_type.

- *write_glove()*
- *write_w2v_text()*
- *write_w2v()*
- *write_leader()*

**Parameters**

- **wf** (*Union[str, IO]*) – The file we are writing to.
- **vocab** (*Union[Dict[str, int], Iterable[str]]*) – The vocab mapping words -> ints.
- **vectors** (*numpy.ndarray*) – The vectors as a np.ndarray.
- **file_type** (word_vectors.FileType) – The format to use when writing the vectors to disk.
- **max_len** (*Optional[int]*) – The maximum length of a word in vocab. Only used when writing Leader vectors.

**Raises** **ValueError** – If the an unsupported file type is passed

word_vectors.write.**write_glove**(*wf*, *vocab*, *vectors*)

Write vectors to a glove file.

See *word_vectors.read.read_glove()* for a description of the file format and examples of common pre-trained embeddings that use this format.

**Parameters**

- **wf** (*Union[str, TextIO]*) – The file we are writing to
- **vocab** (*Union[Dict[str, int], Iterable[str]]*) – The vocab of words -> ints.
- **vectors** (*numpy.ndarray*) – The vectors as a np.ndarray.

word_vectors.write.**write_w2v_text**(*wf*, *vocab*, *vectors*)

Write vectors in the word2vec format in a text file.

See *word_vectors.read.read_w2v_text()* for a description of the file format and examples of common pre-trained embeddings that use this format.

**Parameters**

- **wf** (*Union[str, TextIO]*) – The file we are writing to
- **vocab** (*Union[Dict[str, int], Iterable[str]]*) – The vocab of words -> ints

- **vectors** (*numpy.ndarray*) – The vectors we are writing

word_vectors.write.**write_w2v**(*wf*, *vocab*, *vectors*)

Write vectors to the word2vec format as a binary file.

See *word_vectors.read.read_w2v()* for a description of the file format and examples of common pre-trained embeddings that use this format.

**Parameters**

- **wf** (*Union[str, BinaryIO]*) – The file we are writing to
- **vocab** (*Union[Dict[str, int], Iterable[str]]*) – The vocab of words -> ints.
- **vectors** (*numpy.ndarray*) – The vectors as a np.ndarray.

word_vectors.write.**write_leader**(*wf*, *vocab*, *vectors*)

Write vectors to a leader file.

See *word_vectors.read.read_leader()* for a description of the file format.

**Parameters**

- **wf** (*Union[str, BinaryIO]*) – The file we are writing to.
- **vocab** (*Union[Dict[str, int], Iterable[str]]*) – The vocab of words -> ints.
- **vectors** (*numpy.ndarray*) – The vectors as a np.ndarray.
- **max_len** – The longest length of the words as (utf-8) bytes.

## 3.4 word_vectors.convert

Convert between word vector formats.

We provide the main *convert()* function for converting between arbitrary formats based on the passed *FileType* (or by sniffing the input file with *sniff()* when not provided) as well as several convenience function for converting between different pairs of formats.

word_vectors.convert.**convert**(*f*, *output=None*, *output_file_type=<FileType.LEADER: 'leader'>*, *input_file_type=None*)

Convert vectors from one format to another.

**Parameters**

- **f** (*Union[str, TextIO, BinaryIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.
- **output_file_type** (word_vectors.FileType) – The vector serialization format to use when writing out the vectors.
- **input_file_type** (*Optional[word_vectors.FileType]*) – An explicit vector format to use when reading.

word_vectors.convert.**w2v_to_leader**(*f*, *output=None*)

Convert binary Word2Vec formatted vectors to the Leader format.

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file to read from.

- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**glove_to_leader**(*f*, *output=None*)
Convert GloVe formatted vectors to the Leader format.

**Parameters**

- **f** (*Union[str, TextIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**w2v_text_to_leader**(*f*, *output=None*)
Convert text Word2Vec formatted vectors to the Leader format.

**Parameters**

- **f** (*Union[str, TextIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**w2v_to_w2v_text**(*f*, *output=None*)
Convert binary Word2Vec formatted vectors to the Binary Word2Vec format.

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**w2v_to_glove**(*f*, *output=None*)
Convert binary Word2Vec formatted vectors to the GloVe format.

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**w2v_text_to_glove**(*f*, *output=None*)
Convert text Word2Vec formatted vectors to the Glove format.

**Parameters**

- **f** (*Union[str, TextIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**w2v_text_to_w2v**(*f*, *output=None*)
Convert text Word2Vec formatted vectors to the binary Word2Vec format.

**Parameters**

- **f** (*Union[str, TextIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**glove_to_w2v**(*f*, *output=None*)
Convert GloVe formatted vectors to the binary Word2Vec format.

**Parameters**

- **f** (*Union[str, TextIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**glove_to_w2v_text**(*f*, *output=None*)
Convert GloVe formatted vectors to the text Word2Vec format.

**Parameters**

- **f** (*Union[str, TextIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**leader_to_w2v**(*f*, *output=None*)
Convert Leader formatted vectors to the binary Word2Vec format.

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**leader_to_w2v_text**(*f*, *output=None*)
Convert Leader formatted vectors to the text Word2Vec format.

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

word_vectors.convert.**leader_to_glove**(*f*, *output=None*)
Convert Leader formatted vectors to the GloVe format.

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file to read from.
- **output** (*Optional[str]*) – The name for the output file. If not provided we use the input file name with a modified extension.

## 3.5 word_vectors.utils

Utilities for working with word vector I/O.

word_vectors.utils.**find_space**(*buf*, *offset*)
Find the first space starting from offset and return word that spans the spaces and the new offset.

**Parameters**

- **buf** (*bytes*) – The bytes buffer we are looking for a space in.
- **offset** (*int*) – Where in the buffer we start looking.

**Returns** A (word, offset) tuple where word is the text (decoded from `utf-8`) starting at the original offset until the first space. Offset is index of the location just after the space we just found.

**Return type** Tuple[str, int]

word_vectors.utils.**is_binary**(*f*, *block_size=512*, *ratio=0.3*, *text_characters=b' !"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqr*

Guess if a file is binary or not.

This is based on the implementation from here

**Parameters**

- **f** (*Union[str, BinaryIO]*) – The file we are testing.
- **block_size** (*int*) – The amount of the file to read in for checking.
- **ratio** (*float*) – How many non-ascii characters before we assume it is binary.
- **text_characters** (*bytes*) – Characters that we define as text characters, the ratio of these characters to others is used to determine if the file was binary or not.

**Returns** True if the file is binary, False otherwise

**Return type** bool

word_vectors.utils.**bookmark**(*f*)

Bookmark where we are in a file so we can return.

This is a context manager that lets us save our spot in an open file, to some operations on that file, and then return to the original stop.

This is very useful for things like sniffing a file. If the file is already open and you read in some bytes to estimate the format you need to remember to reset to the start or else you will get wrong results. This context manager automates this.

```
f.tell()
>>> 120
with bookmark(f):
    _ = f.read(1024)
    print(f.tell())
>>> 1144
f.tell()
>>> 120
```

**Parameters** **f** (*IO*) – The file we are bookmarking.

word_vectors.utils.**to_vocab**(*words*)

Convert a series of words to a vocab mapping strings to ints.

**Parameters** **words** (*Iterable[str]*) – The words in the vocab

**Returns** The Vocabulary

**Return type** Dict[str, int]

word_vectors.utils.**create_output_path**(*path*, *file_type*)

Create the output path by stripping the extension and added a new one based on the vector format.

**Parameters**

- **path** (*Union[str, IO, pathlib.PurePath]*) – The path to the input file.
- **file_type** (word_vectors.FileType) – The vector format we are converting to.

**Returns** The new output path with an extension determined by the file type.

**Return type** str

word_vectors.utils.**uniform_initializer**(*unif*)

Create a vector initialization function that takes a vector size as input.

**Parameters** **unif** (*float*) – The bounds that the new vector will be initialized within

**Returns** A function that returns a uniformly random vector between `-unif` and `unif`,

**Return type** Callable[[int], numpy.ndarray]

CHAPTER

FOUR

# GETTING HELP

If you run into trouble be sure to check the issues on github. Please check if someone else was having the same problem as you but if none of the fixes apply to you feel free to open a new issue.

CHAPTER

FIVE

# WORD VECTORS

A fast, light-weight library for reading, writing, and converting between various word vector serialization formats.



## 5.1 What are Word Vectors?

Word vectors are low-dimensional, dense representations of words. This sounds very complicated but then you boil it down is becomes a lot clearer. The it really means that each word is associated with a list of numbers (a vector) that are used to represent the semantic meaning of that word. There vectors normal range in size from as little as 100 elements to around 300. It might seem like a stretch to call that "low-dimensional" but these vectors are very small compared to older methods of vector representations of words. Words used to be encoded as "one-hot" vectors where each word was given a unique index and the vector was full or zeros except for a one at that index. This results in massive vectors (each vector is the size of the vocabulary and the vector size scales linearly as the vocabulary grows). The other problem with this method is that vectors are orthogonal. All none word index elements are zero so when you do something like a dot product between two vectors you will always get zero. Dense vectors, on the other hand, have a fixed size (as you add more terms to your vocabulary the vectors stay the same size) and when you take the dot product of two vectors you get non-zero values. This can be used for tasks like semantic similarity between different words. For a more complete introduction to word vectors and the algorithms used to crate them check out these lectures from Stanford.

- Word Vectors
- Word Vectors and Word Senses

## 5.2 Supported File Formats

This library supports reading and writing several formats of vector serialization. These formats are often under-specified and only truly defined by the implementations of the original software than wrote out the vectors. In the next section we quickly summarize some of the most common file formats.

### 5.2.1 GloVe

The GloVe format is a pure text format. Each (word, vector) pair is represented by a single line in the file. The line starts with the word, a space, and then the float32 text representations of the elements in the vector associated with that word. Each of these vector elements are also separated with a space.

The main vectors distributed in this format are the GloVe vectors (Pennington, et. al., 2014)

### 5.2.2 Word2Vec

There are two different vector serialization file formats introduced by the word2vec software (Mikolov, et. al., 2013). One is a pure text format and the other a binary one.

#### Text

The word2vec text format is a pure text format. The first line is two integers, represented as text and separated by a space, that specify the number of types in the vocabulary and the size of the word vectors respectively. Each following line represents a (word, vector) pair. The line stars with the word, a space, and then the float 32 text representations of the elements in the vector associated with that word. Each of these vector elements are also separated with a space.

One can see that that this is actually the same as the GloVe format except that in GloVe they removed the header line.

The main embeddings distributed in this format are FastText (Bojanowski, et. al., 2017) and NumberBatch (Speer, et. al., 2017)

#### Binary

The word2vec binary format is a mix of textual an binary representations. The first line is two integers (as text, separated be a space) representing the number of types in the vocabulary and the size of the word vectors respectively. (word, vector) pairs follow. The word is represented as text and a space. After the space each element of a vector is represented as a binary float32.

The most well-known pre-trained embeddings distributed in this format are the GoogleNews vectors.

> **Danger:** There is no formal definition of this file format, the only definitive reference on it is the original implementation in the word2vec software
>
> Due to the lack of a definition (and no special handling of it in the code) there is no explicit statements about the endianess of the binary representations. Most code just uses the `numpy.from_buffer` and that seems to work now that most people have little-endian machines. However due to the lack of explicit direction on this encoding I would advise caution when loading vectors that were trained on big-endian hardware.

### 5.2.3 Leader

This is our fully binary vector format.

The first line is a header for the leader format and it is a 3-tuple. The elements of this tuple are: A magic number, the size of the vocabulary, and the size of the vectors. These numbers are represented as little-endian unsigned long longs that have a size of 8 bytes.

Following the header there are (length, word, vector) tuples. The length is the length of this particular word encoded as a little-endian unsigned integer. The word is stored as `utf-8` bytes. After the word the vector is stored where each element is a little-endian float32 (4 bytes).

By tracking the length of the word we can jump directly to the start of them vector instead of having to iterate through the word like we do in the word2vec binary format.

---

**Note:** The magic number if used to make sure this is can actual file and not just trying to extract word vectors from a random binary file. The Magic Number is `38941`.

---

---

**Note:** One of the downsides of this format is that it is harder to inspect the file to see information like the vocabulary size or the vector size. Unlink the Word2Vec format the header is not text so a simple `head -n 1 embedding-file` will **NOT** work. Instead you can use `od -l --endian=little -N 24 embedding-file` and you should see the magic number, the vocabulary size, the vector size, and the max length of the tokens (as `utf-8` bytes).

---

*A note on the Senna format*: There is an older format of embeddings called Senna embeddings (Collobert, et. al., 2011). The format actually uses two files. There is a vocabulary file where each line has a single word and an vector file where each line has the text representations of the float32 elements in a vector separated by a space. These files are aligned so that the word on line `i` of the word file is represented by the vector on line `i` of the vector file. Due to the mismatch in API supporting this format would cause (requiring two file rather than just one) we have decided not to provide reading utilities for this format. Luckily the conversion of this format into the GloVe format is a single `paste` command.

```
paste -d" " /path/to/word/file.senna /path/to/vector/file.senna > word_vectors.glove
```

## 5.3 Usage

While these vector formats are not very complex it is annoying to have to write code to read them in for each project. This causes a lot of people to pull in pretty large libraries just to use the vector reading functionality. The problem with this (beside the heavy dependency) is that these libraries tend to return the vocabulary and vectors within some complex, library specific class. There is often a lot of utility to be gained from these classes when you are actually using the rest of the library but when all you care about is reading in the vectors this is a hindrance.

We designed this library to fix both of these at once. The library is small and focused. You won't be pulling in a lot of code that does (really cool) things you will never touch. We also return results using the simplest formats possible for maximum flexibility.

The main data structure that people conceptually think about when working with word vectors is a mapping for word to vector. This is natural to represent as a python dictionary. This isn't the format that people actually use however. Having many single vectors inside of a dictionary is less space efficient and harder to work with than a single large matrix the vectors stacked on one another. When using this format the data structure that comes to mind is an pair of associated arrays. The word at index `i` in one array is associated with the vector at index `i` in the other. The main use

case is a look up from word to vector however so instead of storing an actual list of words we use a dictionary mapping words to integers. These integers can then be used to look up the vector in the dense matrix.

Our vocabulary is simply `Dict[str, int]` and our vectors type is just a `np.ndarray` of size `[number of words in vocab, size of vector]`.

These simple datatypes give us a lot of flexibility downstream. First we read in the vocabulary and vectors from a file.

```
>>> from word_vectors import read
>>> v, wv = read("/home/blester/embeddings/glove-6B.100d")
>>> len(v)
400000
>>> wv.shape
(400000, 50)
```

Then we can lookup a single word by getting its index in the vocabulary and pulling the vector from the matrix.

```
>>> wv[v['the']]
array([ 4.1800e-01,  2.4968e-01, -4.1242e-01,  1.2170e-01,  3.4527e-01,
       -4.4457e-02, -4.9688e-01, -1.7862e-01, -6.6023e-04, -6.5660e-01,
        2.7843e-01, -1.4767e-01, -5.5677e-01,  1.4658e-01, -9.5095e-03,
        1.1658e-02,  1.0204e-01, -1.2792e-01, -8.4430e-01, -1.2181e-01,
       -1.6801e-02, -3.3279e-01, -1.5520e-01, -2.3131e-01, -1.9181e-01,
       -1.8823e+00, -7.6746e-01,  9.9051e-02, -4.2125e-01, -1.9526e-01,
        4.0071e+00, -1.8594e-01, -5.2287e-01, -3.1681e-01,  5.9213e-04,
        7.4449e-03,  1.7778e-01, -1.5897e-01,  1.2041e-02, -5.4223e-02,
       -2.9871e-01, -1.5749e-01, -3.4758e-01, -4.5637e-02, -4.4251e-01,
        1.8785e-01,  2.7849e-03, -1.8411e-01, -1.1514e-01, -7.8581e-01],
       dtype=float32)
>>> wv[v['the']].shape
(50,)
```

We can also lookup an entire sentence in a single go getting back a dense matrix of `[tokens, embeddings]` which is perfect for downstream machine leaning applications like the input to neural networks.

```
>>> wv[[v[t] for t in "the quick brown fox".split()]]
array([[ 4.1800e-01,  2.4968e-01, -4.1242e-01,  1.2170e-01,  3.4527e-01,
        -4.4457e-02, -4.9688e-01, -1.7862e-01, -6.6023e-04, -6.5660e-01,
         2.7843e-01, -1.4767e-01, -5.5677e-01,  1.4658e-01, -9.5095e-03,
         1.1658e-02,  1.0204e-01, -1.2792e-01, -8.4430e-01, -1.2181e-01,
        -1.6801e-02, -3.3279e-01, -1.5520e-01, -2.3131e-01, -1.9181e-01,
        -1.8823e+00, -7.6746e-01,  9.9051e-02, -4.2125e-01, -1.9526e-01,
         4.0071e+00, -1.8594e-01, -5.2287e-01, -3.1681e-01,  5.9213e-04,
         7.4449e-03,  1.7778e-01, -1.5897e-01,  1.2041e-02, -5.4223e-02,
        -2.9871e-01, -1.5749e-01, -3.4758e-01, -4.5637e-02, -4.4251e-01,
         1.8785e-01,  2.7849e-03, -1.8411e-01, -1.1514e-01, -7.8581e-01],
       [ 1.3967e-01, -5.3798e-01, -1.8047e-01, -2.5142e-01,  1.6203e-01,
        -1.3868e-01, -2.4637e-01,  7.5111e-01,  2.7264e-01,  6.1035e-01,
        -8.2548e-01,  3.8647e-02, -3.2361e-01,  3.0373e-01, -1.4598e-01,
        -2.3551e-01,  3.9267e-01, -1.1287e+00, -2.3636e-01, -1.0629e+00,
         4.6277e-02,  2.9143e-01, -2.5819e-01, -9.4902e-02,  7.9478e-01,
        -1.2095e+00, -1.0390e-02, -9.2086e-02,  8.4322e-01, -1.1061e-01,
         3.0096e+00,  5.1652e-01, -7.6986e-01,  5.1074e-01,  3.7508e-01,
         1.2156e-01,  8.2794e-02,  4.3605e-01, -1.5840e-01, -6.1048e-01,
         3.5006e-01,  5.2465e-01, -5.1747e-01,  3.4705e-03,  7.3625e-01,
         1.6252e-01,  8.5279e-01,  8.5268e-01,  5.7892e-01,  6.4483e-01],
       [-8.8497e-01,  7.1685e-01, -4.0379e-01, -1.0698e-01,  8.1457e-01,
         1.0258e+00, -1.2698e+00, -4.9382e-01, -2.7839e-01, -9.2251e-01,
```

(continues on next page)

(continued from previous page)

```
        -4.9409e-01,  7.8942e-01, -2.0066e-01, -5.7371e-02,  6.0682e-02,
         3.0746e-01,  1.3441e-01, -4.9376e-01, -5.4788e-01, -8.1912e-01,
        -4.5394e-01,  5.2098e-01,  1.0325e+00, -8.5840e-01, -6.5848e-01,
        -1.2736e+00,  2.3616e-01,  1.0486e+00,  1.8442e-01, -3.9010e-01,
         2.1385e+00, -4.5301e-01, -1.6911e-01, -4.6737e-01,  1.5938e-01,
        -9.5071e-02, -2.6512e-01, -5.6479e-02,  6.3849e-01, -1.0494e+00,
         3.7507e-02,  7.6434e-01, -6.4120e-01, -5.9594e-01,  4.6589e-01,
         3.1494e-01, -3.4072e-01, -5.9167e-01, -3.1057e-01,  7.3274e-01],
       [ 4.4206e-01,  5.9552e-02,  1.5861e-01,  9.2777e-01,  1.8760e-01,
         2.4256e-01, -1.5930e+00, -7.9847e-01, -3.4099e-01, -2.4021e-01,
        -3.2756e-01,  4.3639e-01, -1.1057e-01,  5.0472e-01,  4.3853e-01,
         1.9738e-01, -1.4980e-01, -4.6979e-02, -8.3286e-01,  3.9878e-01,
         6.2174e-02,  2.8803e-01,  7.9134e-01,  3.1798e-01, -2.1933e-01,
        -1.1015e+00, -8.0309e-02,  3.9122e-01,  1.9503e-01, -5.9360e-01,
         1.7921e+00,  3.8260e-01, -3.0509e-01, -5.8686e-01, -7.6935e-01,
        -6.1914e-01, -6.1771e-01, -6.8484e-01, -6.7919e-01, -7.4626e-01,
        -3.6646e-02,  7.8251e-01, -1.0072e+00, -5.9057e-01, -7.8490e-01,
        -3.9113e-01, -4.9727e-01, -4.2830e-01, -1.5204e-01,  1.5064e+00]],
       dtype=float32)
>>> wv[[v[t] for t in "the quick brown fox".split()]].shape
(4, 50)
```

### 5.3.1 Reading

Reading is most often done with the `word_vectors.read.read` function. We can use the `word_vectors.FileType` argument to specify a specific format to read the file as or we can let the code infer the format for itself (you can also use one of the format specific readers to read a certain file format. The read API is very simply just pass in the file name.

```
>>> from word_vectors.read import read
>>> # Read where the format is determined by sniffing
... w, wv = read("/path/to/vector-file")
>>> from word_vectors import FileType
>>> # Read using the binary Word2Vec format
... v, wv = read("/path/to/vector-file", FileType.W2V)
>>> from word_vectors.read import read_leader
>>> # Read leader formatted vectors
... v, wv = read_leader("/path/to/leader-vector-file")
```

You can also use the `_with_vocab` version of all the reader function to only read a subsection of the vocabulary. Below we can see an example. First we read the full vocabulary from the file. We can see that is has the string representations of numbers from zero for fourteen. We can see the vectors for several tokens. Then we create a user vocabulary that only has the even numbers, and we re-read the vectors with this vocab. We see that we have now only read in a subset of the word and that our vocab is in the same order that we passed in. We can also see the vectors for a word haven't changed. Finally we re-read the vectors again but this time we ask for it to keep the vectors in the pre-train vocabulary that are not present in our vocab using `keep_extra=True`. We can see the indices from our user vocabulary have not changed but we get the full vocabulary back with the extra words appearing at the end.

```
>>> from word_vectors import read, read_with_vocab
>>> v, wv = read("leader.bin")
>>> v
{'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10
↪': 10, '11': 11, '12': 12, '13': 13, '14': 14}
>>> wv[v["4"]]
```

(continues on next page)

(continued from previous page)

```
array([4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4.,
       4., 4., 4.], dtype=float32)
>>> wv[v["13"]]
array([13., 13., 13., 13., 13., 13., 13., 13., 13., 13., 13., 13., 13.,
       13., 13., 13., 13., 13., 13.], dtype=float32)
>>> wv.shape
(15, 20)
>>> user_vocab = {k: i for i, k in enumerate(k for k, x in v.items() if x % 2 == 0)}
>>> user_vocab
{'0': 0, '2': 1, '4': 2, '6': 3, '8': 4, '10': 5, '12': 6, '14': 7}
>>> v, wv = read_with_vocab("leader.bin", user_vocab)
>>> v
{'0': 0, '2': 1, '4': 2, '6': 3, '8': 4, '10': 5, '12': 6, '14': 7}
>>> wv[v["4"]]
array([4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4.,
       4., 4., 4.], dtype=float32)
>>> wv.shape
(8, 20)
>>> v, wv = read_with_vocab("leader.bin", user_vocab, keep_extra=True)
>>> v
{'0': 0, '2': 1, '4': 2, '6': 3, '8': 4, '10': 5, '12': 6, '14': 7, '1': 8, '3': 9, '5
↪': 10, '7': 11, '9': 12, '11': 13, '13': 14}
>>> wv[v["4"]]
array([4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4., 4.,
       4., 4., 4.], dtype=float32)
>>> wv[v["13"]]
array([13., 13., 13., 13., 13., 13., 13., 13., 13., 13., 13., 13., 13.,
       13., 13., 13., 13., 13., 13.], dtype=float32)
>>> wv.shape
(15, 20)
```

### 5.3.2 Writing

Writing similarly has a main `word_vectors.write.write` function that dispatches on the `word_vectors.FileType` argument and there are format specific writers if you want to use those instead.

```
>>> from word_vectors.read import read
>>> v, wv = read("/path/to/vectors")
>>> from word_vectors import FileType
>>> from word_vectors.write import write
>>> write("/path/to/vectors.leader", v, wv, FileType.LEADER)
>>> write("/path/to/vectors.w2v", v, wv, FileType.W2V)
>>> write_glove("/path/to/vectors.glove", v, wv)
```

### 5.3.3 Converting

Conversions also have a general function (`word_vectors.convert.convert`) dispatching on `word_vectors.FileType` and specific functions for converting between certain pairs.

```
>>> from word_vectors import FileType
>>> from word_vectors.convert import convert
>>> # Conversion to w2v via sniffing the original file
... convert("/path/to/vectors", output="/path/to/vectors.w2v", output_file_
↪type=FileType.W2V)
>>> # Conversion to w2v with an explicit input type
... convert(
...     "/path/to/vectors.glove",
...     output="/path/to/vectors.w2v",
...     output_file_type=FileType.w2v,
...     input_file_type=FileType.GLOVE
... )
>>> # Converting between specific formats
>>> from word_vectors.convert import w2v_text_to_w2v
... w2v_text_to_w2v("/path/to/vectors.w2v-text", output="/path/to/vectors.w2v")
```

## 5.4 Indices and tables

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## W

# INDEX

## W