finalfusion

Release 0.7pre

Sebastian Pütz

Jun 05, 2020

CONTENTS

`finalfusion` is a Python package for reading, writing and using finalfusion embeddings, but also supports other commonly used embeddings like fastText, GloVe and word2vec.

The Python package supports the same types of embeddings as the finalfusion-rust crate:

- Vocabulary
  - No subwords
  - Subwords
- Embedding matrix
  - Array
  - Memory-mapped
  - Quantized
- Norms
- Metadata

This package extends (de-)serialization capabilities of `finalfusion` *`Chunk`*s by allowing loading and writing single chunks. E.g. a *`Vocab`* can be loaded from a finalfusion spec file without loading the *`Storage`*. Single chunks can also be serialized to their own files through *`write()`*. This is different from the functionality of `finalfusion-rust`, loading stand-alone components is only supported by the Python package. Reading will fail with other tools from the `finalfusion` ecosystem.

It integrates nicely with `numpy` since its *`Storage`* types can be treated as numpy arrays.

`finalfusion` comes with some *scripts* to convert between embedding formats, do analogy and similarity queries and turn bucket subword embeddings into explicit subword embeddings.

The package is implemented in Python with some `Cython` extensions, it is not based on bindings to the finalfusion-rust crate.

CHAPTER

ONE

CONTENTS

1.1 Quickstart

1.1.1 Install

You can *install* `finalfusion` through:

```
pip install finalfusion
```

1.1.2 Package

And use embeddings by:

```
import finalfusion
# loading from different formats
w2v_embeds = finalfusion.load_word2vec("/path/to/w2v.bin")
text_embeds = finalfusion.load_text("/path/to/embeds.txt")
text_dims_embeds = finalfusion.load_text_dims("/path/to/embeds.dims.txt")
fasttext_embeds = finalfusion.load_fasttext("/path/to/fasttext.bin")
fifu_embeds = finalfusion.load_finalfusion("/path/to/embeddings.fifu")

# serialization to formats works similarly
finalfusion.compat.write_word2vec("to_word2vec.bin", fifu_embeds)

# embedding lookup
embedding = fifu_embeds["Test"]

# reading an embedding into a buffer
import numpy as np
buffer = np.zeros(fifu_embeds.storage.shape[1], dtype=np.float32)
fifu_embeds.embedding("Test", out=buffer)

# similarity and analogy query
sim_query = fifu_embeds.word_similarity("Test")
analogy_query = fifu_embeds.analogy("A", "B", "C")

# accessing the vocab and printing the first 10 words
vocab = fifu_embeds.vocab
print(vocab.words[:10])

# SubwordVocabs give access to the subword indexer:
```

(continues on next page)

(continued from previous page)

```
subword_indexer = vocab.subword_indexer
print(subword_indexer.subword_indices("Test", with_ngrams=True))

# accessing the storage and calculate its dot product with an embedding
res = embedding.dot(fifu_embeds.storage)

# printing metadata
print(fifu_embeds.metadata)
```

`finalfusion` exports most commonly used functions and types in the top level. See *Top-Level Exports* for an overview.

The full API documentation can be found *here*.

1.1.3 Conversion

`finalfusion` also comes with a conversion tool to convert between supported file formats and from bucket subword embeddings to explicit subword embeddings:

```
$ ffp-convert -f fasttext from_fasttext.bin -t finalfusion to_finalfusion.fifu
$ ffp-bucket-to-explicit buckets.fifu explicit.fifu
```

See *Scripts*

1.1.4 Similarity and Analogy

```
$ echo Tübingen | ffp-similar embeddings.fifu
$ echo Tübingen Stuttgart Heidelberg | ffp-analogy embeddings.fifu
```

See *Scripts*

1.2 Install

`finalfusion` is compatible with Python `3.6` and more recent versions. Direct dependencies are numpy and toml. Installing for `3.6` additionally depends on dataclasses.

1.2.1 Pip

From Pypi:

```
$ pip install finalfusion
```

From GitHub:

```
$ pip install git+https://github.com/finalfusion/finalfusion-python
```

1.2.2 Source

Installing from source requires Cython. When `finalfusion` is built with `pip`, you don't need to install `Cython` manually since the dependency is specified in `pyproject.toml`.

```
$ git clone https://github.com/finalfusion/finalfusion-python
$ cd finalfusion-python
$ pip install . # or python setup.py install
```

Building a wheel from source is possible if wheel is installed by:

```
$ git clone https://github.com/finalfusion/finalfusion-python
$ cd finalfusion-python
$ python setup.py bdist_wheel
```

1.3 Top-level Exports

`finalfusion` re-exports some common types at the top-level. These types cover the typical use-cases.

1.3.1 Embeddings

finalfusion.embeddings.Embeddings(storage, vocab)	Embeddings class.
finalfusion.embeddings.load_finalfusion(file)	Read embeddings from a file in finalfusion format.

1.3.2 Compat

finalfusion.compat.fasttext.load_fasttext(file)	Read embeddings from a file in fastText format.
finalfusion.compat.text.load_text(file)	Read embeddings in text format.
finalfusion.compat.text.load_text_dims(file)	Read emebddings in text-dims format.
finalfusion.compat.word2vec.load_word2vec(file)	Read embeddings in word2vec binary format.

1.3.3 Metadata

finalfusion.metadata.Metadata	Embeddings metadata
finalfusion.metadata.load_metadata(file)	Load a Metadata chunk from the given file.

1.3.4 Norms

finalfusion.norms.Norms	Norms Chunk.
finalfusion.norms.load_norms(file)	Load Norms from a finalfusion file.

1.3.5 Storage

finalfusion.storage.storage.Storage	Common interface to finalfusion storage types.
finalfusion.storage.load_storage(file[, mmap])	Load any vocabulary from a finalfusion file.

1.3.6 Vocab

finalfusion.vocab.vocab.Vocab	Finalfusion vocabulary interface.
finalfusion.vocab.load_vocab(file)	Load any vocabulary from a finalfusion file.

1.4 API

1.4.1 Embeddings

Finalfusion Embeddings

class finalfusion.embeddings.**Embeddings**(*storage:* finalfusion.storage.storage.Storage, *vocab:* finalfusion.vocab.vocab.Vocab, *norms: Optional[*finalfusion.norms.Norms*] = None*, *metadata: Optional[*finalfusion.metadata.Metadata*] = None*)

Embeddings class.

Embeddings always contain a *Storage* and *Vocab*. Optional chunks are *Norms* corresponding to the embeddings of the in-vocab tokens and *Metadata*.

Embeddings can be retrieved through three methods:

1. *Embeddings.embedding()* allows to provide a default value and returns this value if no embedding could be found.
2. *Embeddings.__getitem__()* retrieves an embedding for the query but raises an exception if it cannot retrieve an embedding.
3. *Embeddings.embedding_with_norm()* requires a *Norms* chunk and returns an embedding together with the corresponding L2 norm.

Embeddings are composed of the 4 chunk types:

1. *Storage* *(required)*:
   - *NdArray*
   - QuantizedArray
2. *Vocab* *(required)*:

- *SimpleVocab*
- *FinalfusionBucketVocab*
- *FastTextVocab*
- *ExplicitVocab*

3. *Metadata*
4. *Norms*

Examples

```
>>> storage = NdArray(np.float32(np.random.rand(2, 10)))
>>> vocab = SimpleVocab(["Some", "words"])
>>> metadata = Metadata({"Some": "value", "numerical": 0})
>>> norms = Norms(np.float32(np.random.rand(2)))
>>> embeddings = Embeddings(storage=storage, vocab=vocab, metadata=metadata,␣
↪norms=norms)
>>> embeddings.vocab.words
['Some', 'words']
>>> np.allclose(embeddings["Some"], storage[0])
True
>>> try:
...     embeddings["oov"]
... except KeyError:
...     True
True
>>> _, n = embeddings.embedding_with_norm("Some")
>>> np.isclose(n, norms[0])
True
>>> embeddings.metadata
{'Some': 'value', 'numerical': 0}
```

__init__(*storage:* finalfusion.storage.storage.Storage, *vocab:* finalfusion.vocab.vocab.Vocab, *norms:* *Optional[*finalfusion.norms.Norms*]* = *None*, *metadata:* *Optional[*finalfusion.metadata.Metadata*]* = *None*)

Initialize Embeddings.

Initializes Embeddings with the given chunks.

Conditions The following conditions need to hold if the respective chunks are passed:

- Chunks need to have the expected type.
- `vocab.idx_bound == storage.shape[0]`
- `len(vocab) == len(norms)`
- `len(norms) == len(vocab) and len(norms) >= storage.shape[0]`

Parameters

- **storage** (*Storage*) – Embeddings Storage.
- **vocab** (*Vocab*) – Embeddings Vocabulary.
- **norms** (*Norms, optional*) – Embeddings Norms.
- **metadata** (*Metadata, optional*) – Embeddings Metadata.

Raises **AssertionError** – If any of the conditions don't hold.

__getitem__(*item: str*) → numpy.ndarray
Returns an embedding.

Parameters **item** (*str*) – The query item.

Returns **embedding** – The embedding.

Return type numpy.ndarray

Raises **KeyError** – If no embedding could be retrieved.

See also:

embedding(), *embedding_with_norm()*

embedding(*word: str*, *out: Optional[numpy.ndarray] = None*, *default: Optional[numpy.ndarray] = None*) → Optional[numpy.ndarray]
Embedding lookup.

Looks up the embedding for the input word.

If an *out* array is specified, the embedding is written into the array.

If it is not possible to retrieve an embedding for the input word, the *default* value is returned. This defaults to *None*. An embedding can not be retrieved if the vocabulary cannot provide an index for *word*.

This method never fails. If you do not provide a default value, check the return value for None. `out` is left untouched if no embedding can be found and `default` is None.

Parameters

- **word** (*str*) – The query word.
- **out** (*numpy.ndarray, optional*) – Optional output array to write the embedding into.
- **default** (*numpy.ndarray, optional*) – Optional default value to return if no embedding can be retrieved. Defaults to None.

Returns **embedding** – The retrieved embedding or the default value.

Return type numpy.ndarray, optional

Examples

```
>>> matrix = np.float32(np.random.rand(2, 10))
>>> storage = NdArray(matrix)
>>> vocab = SimpleVocab(["Some", "words"])
>>> embeddings = Embeddings(storage=storage, vocab=vocab)
>>> np.allclose(embeddings.embedding("Some"), matrix[0])
True
>>> # default value is None
>>> embeddings.embedding("oov") is None
True
>>> # It's possible to specify a default value
>>> default = embeddings.embedding("oov", default=storage[0])
>>> np.allclose(default, storage[0])
True
>>> # Embeddings can be written to an output buffer.
>>> out = np.zeros(10, dtype=np.float32)
>>> out2 = embeddings.embedding("Some", out=out)
>>> out is out2
True
```

(continues on next page)

(continued from previous page)

```
>>> np.allclose(out, matrix[0])
True
```

See also:

embedding_with_norm(), *__getitem__()*

embedding_with_norm(*word: str*, *out: Optional[numpy.ndarray] = None*, *default: Optional[Tuple[numpy.ndarray, float]] = None*) → Optional[Tuple[numpy.ndarray, float]]

Embedding lookup with norm.

Looks up the embedding for the input word together with its norm.

If an *out* array is specified, the embedding is written into the array.

If it is not possible to retrieve an embedding for the input word, the *default* value is returned. This defaults to *None*. An embedding can not be retrieved if the vocabulary cannot provide an index for *word*.

This method raises a TypeError if norms are not set.

Parameters

- **word** (*str*) – The query word.
- **out** (*numpy.ndarray, optional*) – Optional output array to write the embedding into.
- **default** (*Tuple[numpy.ndarray, float], optional*) – Optional default value to return if no embedding can be retrieved. Defaults to None.

Returns **(embedding, norm)** – Tuple with the retrieved embedding or the default value at the first index and the norm at the second index.

Return type EmbeddingWithNorm, optional

See also:

embedding(), *__getitem__()*

property storage

Get the *Storage*.

Returns **storage** – The embeddings storage.

Return type *Storage*

property vocab

The *Vocab*.

Returns **vocab** – The vocabulary

Return type *Vocab*

property norms

The Norms.

Getter Returns None or the Norms.

Setter Set the Norms.

Returns **norms** – The Norms or None.

Return type *Norms*, optional

Raises

- **AssertionError** – if `embeddings.storage.shape[0] < len(embeddings.norms)` or `len(embeddings.norms) != len(embeddings.vocab)`
- **TypeError** – If `norms` is neither Norms nor None.

property metadata

The `Metadata`.

Getter Returns None or the Metadata.

Setter Set the Metadata.

Returns **metadata** – The Metadata or None.

Return type *Metadata*, optional

Raises **TypeError** – If `metadata` is neither Metadata nor None.

chunks() → List[*finalfusion.io.Chunk*]

Get the Embeddings Chunks as a list.

The Chunks are ordered in the expected serialization order:

1. Metadata (optional)
2. Vocabulary
3. Storage
4. Norms (optional)

Returns **chunks** – List of embeddings chunks.

Return type List[*Chunk*]

write(*file: Union[str, bytes, int, os.PathLike]*)

Write the Embeddings to the given file.

Writes the Embeddings to a finalfusion file at the given file.

Parameters **file** (*str, bytes, int, PathLike*) – Path of the output file.

bucket_to_explicit() → *finalfusion.embeddings.Embeddings*

Bucket to explicit Embeddings conversion.

Multiple embeddings can still map to the same bucket, but all buckets that are not indexed by in-vocabulary n-grams are eliminated. This can have a big impact on the size of the embedding matrix.

Metadata is **not** copied to the new embeddings since it doesn't reflect the changes. You can manually set the metadata and update the values accordingly.

Returns **embeddings** – Embeddings with an ExplicitVocab instead of a hash-based vocabulary.

Return type *Embeddings*

Raises **TypeError** – If the current vocabulary is not a hash-based vocabulary (FinalfusionBucketVocab or FastTextVocab)

analogy(*word1: str, word2: str, word3: str, k: int = 1, skip: Set[str] = None*) → Optional[List[*finalfusion.embeddings.SimilarityResult*]]

Perform an analogy query.

This method returns words that are close in vector space the analogy query *word1* is to *word2* as *word3* is to *?*. More concretely, it searches embeddings that are similar to:

```
embedding(word2) - embedding(word1) + embedding(word3)
```

Words specified in `skip` are not considered as answers. If `skip` is None, the query words `word1`, `word2` and `word3` are excluded.

At most, `k` results are returned. `None` is returned when no embedding could be computed for any of the tokens.

Parameters

- **word1** (*str*) – Word1 is to...
- **word2** (*str*) – word2 like...
- **word3** (*str*) – word3 is to the return value
- **skip** (*Set[str]*) – Set of strings which should not be considered as answers. Defaults to `None` which excludes the query strings. To allow the query strings as answers, pass an empty set.
- **k** (*int*) – Number of answers to return, defaults to 1.

Returns **answers** – List of answers.

Return type List[*SimilarityResult*]

word_similarity(*query: str*, *k: int = 10*) → Optional[List[*finalfusion.embeddings.SimilarityResult*]]

Retrieves the nearest neighbors of the query string.

The similarity between the embedding of the query and other embeddings is defined by the dot product of the embeddings. If the vectors are unit vectors, this is the cosine similarity.

At most, `k` results are returned.

Parameters

- **query** (*str*) – The query string
- **k** (*int*) – The number of neighbors to return, defaults to 10.

Returns **neighbours** – List of tuples with neighbour and similarity measure. None if no embedding can be found for `query`.

Return type List[Tuple[str, float], optional

embedding_similarity(*query: numpy.ndarray*, *k: int = 10*, *skip: Optional[Set[str]] = None*) → Optional[List[*finalfusion.embeddings.SimilarityResult*]]

Retrieves the nearest neighbors of the query embedding.

The similarity between the query embedding and other embeddings is defined by the dot product of the embeddings. If the vectors are unit vectors, this is the cosine similarity.

At most, `k` results are returned.

Parameters

- **query** (*str*) – The query array.
- **k** (*int*) – The number of neighbors to return, defaults to 10.
- **skip** (*Set[str], optional*) – Set of strings that should not be considered as neighbours.

Returns **neighbours** – List of tuples with neighbour and similarity measure. None if no embedding can be found for `query`.

Return type List[Tuple[str, float], optional

finalfusion.embeddings.**load_finalfusion**(*file: Union[str, bytes, int, os.PathLike]*, *mmap: bool = False*) → *finalfusion.embeddings.Embeddings*

Read embeddings from a file in finalfusion format.

Parameters

- **file** (*str, bytes, int, PathLike*) – Path to a file with embeddings in finalfusoin format.
- **mmap** (*bool*) – Toggles memory mapping the storage buffer.

Returns **embeddings** – The embeddings from the input file.

Return type *Embeddings*

class finalfusion.embeddings.**SimilarityResult**(*word: str*, *similarity: float*)

Container for a Similarity result.

The word can be accessed through `result.word`, the similarity through `result.similarity`.

1.4.2 Storage

finalfusion.storage

finalfusion.storage.load_storage(file[, mmap])	Load any vocabulary from a finalfusion file.
finalfusion.storage.ndarray.load_ndarray(file)	Load an array chunk from the given file.
finalfusion.storage.ndarray.NdArray	Array storage.
finalfusion.storage.quantized.load_quantized_array(file)	Load a quantized array chunk from the given file.
finalfusion.storage.quantized.QuantizedArray(pq, ...)	QuantizedArray storage.

NdArray

class finalfusion.storage.ndarray.**NdArray**

Bases: numpy.ndarray, *finalfusion.storage.storage.Storage*

Array storage.

Wraps an numpy matrix, either in-memory or memory-mapped.

Examples

```
>>> storage = NdArray(np.array([[1., 0.5], [0.5, 1.], [0.3, 0.4]],
...                           dtype=np.float32))
>>> # slicing an NdArray returns a storage backed by the same array
>>> storage[:2]
NdArray([[1. , 0.5],
         [0.5, 1. ]], dtype=float32)
>>> # NdArray storage can be treated as numpy arrays
>>> storage * 2
NdArray([[2. , 1. ],
         [1. , 2. ],
```

(continues on next page)

(continued from previous page)

```
        [0.6, 0.8]], dtype=float32)
>>> # Indexing with arrays, lists or ints returns numpy.ndarray
>>> storage[0]
array([1. , 0.5], dtype=float32)
```

static __new__ (*cls*, *array: numpy.ndarray*)
Construct a new NdArray storage.

Parameters **array** (*np.ndarray*) – The storage buffer.

Raises **TypeError** – If the array is not a 2-dimensional float32 array.

classmethod load (*file: BinaryIO*, *mmap: bool = False*) → *finalfusion.storage.ndarray.NdArray*
Load Storage from the given finalfusion file.

Parameters

- **file** (*IO[bytes]*) – Finalfusion file with a storage chunk
- **mmap** (*bool*)

Returns

- **storage** (*Storage*) – The first storage in the input file
- **mmap** (*bool*) – Toggles memory mapping the storage buffer as read-only.

Raises **ValueError** – If the file did not contain a storage.

static chunk_identifier() → *finalfusion.io.ChunkIdentifier*
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

static read_chunk (*file: BinaryIO*) → *finalfusion.storage.ndarray.NdArray*
Read the Chunk and return it.

The file must be positioned before the contents of the `Chunk` but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

property shape
Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[::2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: incompatible shape for a non-contiguous array
```

See also:

numpy.reshape similar function

ndarray.reshape similar method

static mmap_chunk(*file: BinaryIO*) → *finalfusion.storage.ndarray.NdArray*

Memory maps the storage as a read-only buffer.

Parameters **file** (*IO[bytes]*) – Finalfusion file with a storage chunk

Returns **storage** – The first storage in the input file

Return type *Storage*

Raises **ValueError** – If the file did not contain a storage.

write_chunk(*file: BinaryIO*)

Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

finalfusion.storage.ndarray.**load_ndarray**(*file: Union[str, bytes, int, os.PathLike]*, *mmap: bool = False*) → *finalfusion.storage.ndarray.NdArray*

Load an array chunk from the given file.

Parameters

- **file** (*str, bytes, int, PathLike*) – Finalfusion file with a ndarray chunk.
- **mmap** (*bool*) – Toggles memory mapping the array buffer as read only.

Returns **storage** – The NdArray storage from the file.

Return type *NdArray*

Raises **ValueError** – If the file did not contain and NdArray chunk.

Quantized

class finalfusion.storage.quantized.**QuantizedArray**(*pq:* finalfusion.storage.quantized.PQ, *quantized_embeddings:* *numpy.ndarray*, *norms:* *Optional[numpy.ndarray]*)

Bases: *finalfusion.storage.storage.Storage*

QuantizedArray storage.

QuantizedArrays support slicing, indexing with integers, lists of integers and arbitrary dimensional integer arrays. Slicing a QuantizedArray returns a new QuantizedArray but does not copy any buffers.

QuantizedArrays offer two ways of indexing:

1. **`QuantizedArray.__getitem__()`:**
   - passing a slice returns a new view of the QuantizedArray.
   - passing an integer returns a single embedding, lists and arrays return ndims + 1 dimensional embeddings.
2. ***`QuantizedArray.embedding()`*:**
   - embeddings can be written to an output buffer.
   - passing a slice returns a matrix holding **reconstructed** embeddings.
   - otherwise, this method behaves like `__getitem__()`

A QuantizedArray can be treated as `numpy.ndarray` through `numpy.asarray()`. This restores the original matrix and copies into a **new** buffer.

Using common numpy functions on a QuantizedArray will produce a regular `ndarray` in the process and is therefore an expensive operation.

__init__(*pq:* finalfusion.storage.quantized.PQ, *quantized_embeddings:* *numpy.ndarray*, *norms:* *Optional[numpy.ndarray]*)

Initialize a QuantizedArray.

Parameters

- **pq** (*PQ*) – A product quantizer
- **quantized_embeddings** (*numpy.ndarray*) – The quantized embeddings
- **norms** (*numpy.ndarray, optional*) – Optional norms corresponding to the quantized embeddings. Reconstructed embeddings are scaled by their norm.

property shape

Get the shape of the storage.

Returns **shape** – Tuple containing *(rows, columns)*

Return type Tuple[int, int]

embedding(*key*, *out:* *numpy.ndarray* = *None*) → numpy.ndarray

Get embeddings.

- if `key` is an integer, a single reconstructed embedding is returned.
- if `key` is a list of integers or a slice, a matrix of reconstructed embeddings is returned.
- if `key` is an n-dimensional array, a tensor with reconstructed embeddings is returned. This tensor has one new axis in the last dimension containing the embeddings.

If `out` is passed, the reconstruction is written to this buffer. `out.shape` needs to match the dimensions described above.

Parameters

- **key** (*int, list, numpy.ndarray, slice*) – Key specifying which embeddings to retrieve.
- **out** (*numpy.ndarray*) – Array to reconstruct the embeddings into.

Returns **reconstruction** – The reconstructed embedding or embeddings.

Return type numpy.ndarray

property quantized_len
Length of the quantized embeddings.

Returns **quantized_len** – Length of quantized embeddings.

Return type int

property quantizer
Get the quantizer.

Returns **pq** – The Product Quantizer.

Return type *PQ*

classmethod load(*file: BinaryIO*, *mmap=False*) → *finalfusion.storage.quantized.QuantizedArray*
Load Storage from the given finalfusion file.

Parameters

- **file** (*IO[bytes]*) – Finalfusion file with a storage chunk
- **mmap** (*bool*)

Returns

- **storage** (*Storage*) – The first storage in the input file
- **mmap** (*bool*) – Toggles memory mapping the storage buffer as read-only.

Raises **ValueError** – If the file did not contain a storage.

static read_chunk(*file: BinaryIO*) → *finalfusion.storage.quantized.QuantizedArray*
Read the Chunk and return it.

The file must be positioned before the contents of the `Chunk` but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

static mmap_chunk(*file: BinaryIO*) → *finalfusion.storage.quantized.QuantizedArray*
Memory maps the storage as a read-only buffer.

Parameters **file** (*IO[bytes]*) – Finalfusion file with a storage chunk

Returns **storage** – The first storage in the input file

Return type *Storage*

Raises **ValueError** – If the file did not contain a storage.

write_chunk(*file: BinaryIO*)
Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

static chunk_identifier() → *finalfusion.io.ChunkIdentifier*
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

write(*file: Union[str, bytes, int, os.PathLike]*)
Write the Chunk as a standalone finalfusion file.

Parameters **file** (*Union[str, bytes, int, PathLike]*) – Output path

Raises **TypeError** – If the Chunk is a `Header`.

class finalfusion.storage.quantized.**PQ**(*quantizers: numpy.ndarray*, *projection: Optional[numpy.ndarray]*)
Product Quantizer

Product Quantizers are vector quantizers which decompose high dimensional vector spaces into subspaces. Each of these subspaces is a slice of the the original vector space. Embeddings are quantized by assigning their ith slice to the closest centroid.

Product Quantizers can reconstruct vectors by concatenating the slices of the quantized vector.

__init__(*quantizers: numpy.ndarray*, *projection: Optional[numpy.ndarray]*)
Initializes a Product Quantizer.

Parameters

- **quantizers** (*np.ndarray*) – 3-d ndarray with dtype uint8
- **projection** (*np.ndarray, optional*) – Projection matrix, must be a square matrix with shape *[reconstructed_len, reconstructed_len]*

Raises **AssertionError** – If the projection shape does not match the *reconstructed_len*

property n_centroids
Number of centroids per quantizer.

Returns **n_centroids** – The number of centroids per quantizer.

Return type int

property projection
Projection matrix.

Returns **projection** – Projection Matrix (2-d numpy array with datatype float32) or None.

Return type np.ndarray, optional

property reconstructed_len
Reconstructed length.

Returns **reconstructed_len** – Length of the reconstructed vectors.

Return type int

property subquantizers
Get the quantizers.

Returns a 3-d array with shape *quantizers * n_centroids * reconstructed_len / quantizers*

Returns

- **quantizers** (*np.ndarray*) – 3-d np.ndarray with dtype=np.uint8

- **@return** (*3d tensor of quantizers*)

reconstruct (*quantized: numpy.ndarray*, *out: numpy.ndarray = None*) → numpy.ndarray
Reconstruct vectors.

Input

Parameters

- **quantized** (*np.ndarray*) – Batch of quantized vectors. 2-d np.ndarray with integers required.
- **out** (*np.ndarray, optional*) – 2-d np.ndarray to write the output into.

Returns **out** – Batch of reconstructed vectors.

Return type np.ndarray

Raises **AssertionError** – If *out* is passed and its last dimension does not match *reconstructed_len* or its first *n-1* dimensions do not match the first *n-1* dimensions of *quantized*.

finalfusion.storage.quantized.**load_quantized_array** (*file: Union[str, bytes, int, os.PathLike]*, *mmap: bool = False*) → *finalfusion.storage.quantized.QuantizedArray*
Load a quantized array chunk from the given file.

Parameters

- **file** (*str, bytes, int, PathLike*) – Finalfusion file with a quantized array chunk.
- **mmap** (*bool*) – Toggles memory mapping the array buffer as read only.

Returns **storage** – The QuantizedArray storage from the file.

Return type *QuantizedArray*

Raises **ValueError** – If the file did not contain a QuantizedArray chunk.

Storage Interface

class finalfusion.storage.storage.**Storage**
Common interface to finalfusion storage types.

abstract property shape
Get the shape of the storage.

Returns **shape** – Tuple containing *(rows, columns)*

Return type Tuple[int, int]

classmethod load (*file: BinaryIO*, *mmap: bool = False*) → *finalfusion.storage.storage.Storage*
Load Storage from the given finalfusion file.

Parameters

- **file** (*IO[bytes]*) – Finalfusion file with a storage chunk
- **mmap** (*bool*)

Returns

- **storage** (*Storage*) – The first storage in the input file
- **mmap** (*bool*) – Toggles memory mapping the storage buffer as read-only.

Raises `ValueError` – If the file did not contain a storage.

abstract static mmap_chunk(*file: BinaryIO*) → *finalfusion.storage.storage.Storage*

Memory maps the storage as a read-only buffer.

Parameters **file** (*IO[bytes]*) – Finalfusion file with a storage chunk

Returns **storage** – The first storage in the input file

Return type *Storage*

Raises `ValueError` – If the file did not contain a storage.

finalfusion.storage.**load_storage**(*file: Union[str, bytes, int, os.PathLike]*, *mmap: bool = False*) → *finalfusion.storage.storage.Storage*

Load any vocabulary from a finalfusion file.

Loads the first known vocabulary from a finalfusion file.

Parameters

- **file** (*str*) – Path to finalfusion file containing a storage chunk.
- **mmap** (*bool*) – Toggles memory mapping the storage buffer as read-only.

Returns **storage** – First finalfusion Storage in the file.

Return type *Storage*

Raises `ValueError` – If the file did not contain a vocabulary.

1.4.3 Vocabularies

finalfusion.vocab

finalfusion.vocab.load_vocab(file)	Load any vocabulary from a finalfusion file.
finalfusion.vocab.subword.load_finalfusion_bucket_vocab(file)	Load a FinalfusionBucketVocab from the given finalfusion file.
finalfusion.vocab.subword.load_fasttext_vocab(file)	Load a FastTextVocab from the given finalfusion file.
finalfusion.vocab.subword.load_explicit_vocab(file)	Load a ExplicitVocab from the given finalfusion file.
finalfusion.vocab.simple_vocab.load_simple_vocab(file)	Load a SimpleVocab from the given finalfusion file.
finalfusion.vocab.vocab.Vocab	Finalfusion vocabulary interface.
finalfusion.vocab.simple_vocab.SimpleVocab(words)	Simple vocabulary.
finalfusion.vocab.subword.SubwordVocab	Interface for vocabularies with subword lookups.
finalfusion.vocab.subword.FinalfusionBucketVocab(words)	Finalfusion Bucket Vocabulary.
finalfusion.vocab.subword.FastTextVocab(words)	FastText vocabulary
finalfusion.vocab.subword.ExplicitVocab(...)	A vocabulary with explicitly stored n-grams.

SimpleVocab

class finalfusion.vocab.simple_vocab.**SimpleVocab**(*words: List[str]*)
Bases: *finalfusion.vocab.vocab.Vocab*

Simple vocabulary.

SimpleVocabs provide a simple string to index mapping and index to string mapping.

__init__(*words: List[str]*)
Initialize a SimpleVocab.

Initializes the vocabulary with the given words and optional index. If no index is given, the nth word in the *words* list is assigned index *n*. The word list cannot contain duplicate entries and it needs to be of same length as the index.

Parameters **words** (*List[str]*) – List of unique words

Raises **AssertionError** – if `words` contains duplicate entries.

property words
Get the list of known words

Returns **words** – list of known words

Return type List[str]

property word_index
Get the index of known words

Returns **dict** – index of known words

Return type Dict[str, int]

property upper_bound
The exclusive upper bound of indices in this vocabulary.

Returns **upper_bound** – Exclusive upper bound of indices covered by the vocabulary.

Return type int

idx(*item: str*, *default: Union[list, int, None] = None*) → Union[list, int, None]
Lookup the given query item.

This lookup does not raise an exception if the vocab can't produce indices.

Parameters

- **item** (*str*) – The query item.
- **default** (*Optional[Union[int, List[int]]]*) – Fall-back value to return if the vocab can't provide indices.

Returns

index –

- An integer if there is a single index for a known item.
- A list if the vocab can provide subword indices for a unknown item.
- The provided *default* item if the vocab can't provide indices.

Return type Optional[Union[int, List[int]]]

static read_chunk(*file: BinaryIO*) → *finalfusion.vocab.simple_vocab.SimpleVocab*
Read the Chunk and return it.

The file must be positioned before the contents of the `Chunk` but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

write_chunk(*file: BinaryIO*)
Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

static chunk_identifier() → *finalfusion.io.ChunkIdentifier*
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

write(*file: Union[str, bytes, int, os.PathLike]*)
Write the Chunk as a standalone finalfusion file.

Parameters **file** (*Union[str, bytes, int, PathLike]*) – Output path

Raises **TypeError** – If the Chunk is a `Header`.

finalfusion.vocab.simple_vocab.**load_simple_vocab**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.vocab.simple_vocab.SimpleVocab*
Load a SimpleVocab from the given finalfusion file.

Parameters **file** (*str*) – Path to file containing a SimpleVocab chunk.

Returns **vocab** – Returns the first SimpleVocab in the file.

Return type *SimpleVocab*

FinalfusionBucketVocab

class finalfusion.vocab.subword.**FinalfusionBucketVocab**(*words: List[str]*, *indexer: Optional[finalfusion.subword.hash_indexers.FinalfusionHashIndexer] = None*)
Bases: *finalfusion.vocab.subword.SubwordVocab*

Finalfusion Bucket Vocabulary.

__init__(*words: List[str]*, *indexer: Optional[finalfusion.subword.hash_indexers.FinalfusionHashIndexer] = None*)
Initialize a FinalfusionBucketVocab.

Initializes the vocabulary with the given words.

If no indexer is passed, a FinalfusionHashIndexer with bucket exponent 21 is used.

The word list cannot contain duplicate entries.

Parameters

- **words** (*List[str]*) – List of unique words

- **indexer** (*FinalfusionHashIndexer, optional*) – Subword indexer to use for the vocabulary. Defaults to an indexer with 2^21 buckets with range 3-6.

Raises **AssertionError** – If the indexer is not a FinalfusionHashIndexer or `words` contains duplicate entries.

to_explicit() → *finalfusion.vocab.subword.ExplicitVocab*
Return an ExplicitVocab built from this vocab.

This method iterates over the known words and extracts all ngrams within this vocab's bounds. Each of the ngrams is hashed and mapped to an index. This index is not necessarily unique for each ngram, if hashes collide, multiple ngrams will be mapped to the same index.

The returned vocab will be unable to produce indices for unknown ngrams.

The indices of the new vocabs known indices will be cover *[0, vocab.upper_bound)*

Returns **explicit_vocab** – The converted vocabulary.

Return type *ExplicitVocab*

write_chunk(*file: BinaryIO*)
Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

property words
Get the list of known words

Returns **words** – list of known words

Return type List[str]

property subword_indexer
Get this vocab's subword Indexer.

The subword indexer produces indices for n-grams.

In case of bucket vocabularies, this is a hash-based indexer (*FinalfusionHashIndexer*, *FastTextIndexer*). For explicit subword vocabularies, this is an *ExplicitIndexer*.

Returns **subword_indexer** – The subword indexer of the vocabulary.

Return type *ExplicitIndexer*, *FinalfusionHashIndexer*, *FastTextIndexer*

property word_index
Get the index of known words

Returns **dict** – index of known words

Return type Dict[str, int]

static read_chunk(*file: BinaryIO*) → *finalfusion.vocab.subword.FinalfusionBucketVocab*
Read the Chunk and return it.

The file must be positioned before the contents of the `Chunk` but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

static chunk_identifier() → *finalfusion.io.ChunkIdentifier*
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

idx (*item: str*, *default: Union[int, List[int], None] = None*) → Union[int, List[int], None]
Lookup the given query item.

This lookup does not raise an exception if the vocab can't produce indices.

Parameters

- **item** (*str*) – The query item.
- **default** (*Optional[Union[int, List[int]]]*) – Fall-back value to return if the vocab can't provide indices.

Returns

index –

- An integer if there is a single index for a known item.
- A list if the vocab can provide subword indices for a unknown item.
- The provided *default* item if the vocab can't provide indices.

Return type Optional[Union[int, List[int]]]

property max_n
Get the upper bound of the range of extracted n-grams.

Returns **max_n** – upper bound of n-gram range.

Return type int

property min_n
Get the lower bound of the range of extracted n-grams.

Returns **min_n** – lower bound of n-gram range.

Return type int

subword_indices (*item: str*, *bracket: bool = True*, *with_ngrams: bool = False*) → List[Union[int, Tuple[str, int]]]
Get the subword indices for the given item.

This list does not contain the index for known items.

Parameters

- **item** (*str*) – The query item.
- **bracket** (*bool*) – Toggles bracketing the item with '<' and '>' before extraction.
- **with_ngrams** (*bool*) – Toggles returning ngrams together with indices.

Returns **indices** – The list of subword indices.

Return type List[Union[int, Tuple[str, int]]]

subwords (*item: str*, *bracket: bool = True*) → List[str]
Get the n-grams of the given item as a list.

The n-gram range is determined by the *min_n* and *max_n* values.

Parameters

- **item** (*str*) – The query item to extract n-grams from.
- **bracket** (*bool*) – Toggles bracketing the item with '<' and '>' before extraction.

Returns **ngrams** – List of n-grams.

Return type List[str]

property upper_bound
The exclusive upper bound of indices in this vocabulary.

Returns **upper_bound** – Exclusive upper bound of indices covered by the vocabulary.

Return type int

write(*file: Union[str, bytes, int, os.PathLike]*)
Write the Chunk as a standalone finalfusion file.

Parameters **file** (*Union[str, bytes, int, PathLike]*) – Output path

Raises **TypeError** – If the Chunk is a `Header`.

finalfusion.vocab.subword.**load_finalfusion_bucket_vocab**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.vocab.subword.FinalfusionBucketVocab*
Load a FinalfusionBucketVocab from the given finalfusion file.

Parameters **file** (*str, bytes, int, PathLike*) – Path to file containing a FinalfusionBucketVocab chunk.

Returns **vocab** – Returns the first FinalfusionBucketVocab in the file.

Return type *FinalfusionBucketVocab*

ExplicitVocab

class finalfusion.vocab.subword.**ExplicitVocab**(*words: List[str]*, *indexer:* finalfusion.subword.explicit_indexer.ExplicitIndexer)
Bases: *finalfusion.vocab.subword.SubwordVocab*

A vocabulary with explicitly stored n-grams.

__init__(*words: List[str]*, *indexer:* finalfusion.subword.explicit_indexer.ExplicitIndexer)
Initialize an ExplicitVocab.

Initializes the vocabulary with the given words and ExplicitIndexer.

The word list cannot contain duplicate entries.

Parameters

- **words** (*List[str]*) – List of unique words
- **indexer** (*ExplicitIndexer*) – Subword indexer to use for the vocabulary.

Raises **AssertionError** – If the indexer is not an ExplicitIndexer.

See also:

ExplicitIndexer

property word_index
Get the index of known words

Returns **dict** – index of known words

Return type Dict[str, int]

property subword_indexer
Get this vocab's subword Indexer.

The subword indexer produces indices for n-grams.

In case of bucket vocabularies, this is a hash-based indexer (`FinalfusionHashIndexer`, `FastTextIndexer`). For explicit subword vocabularies, this is an `ExplicitIndexer`.

Returns **subword_indexer** – The subword indexer of the vocabulary.

Return type *ExplicitIndexer*, *FinalfusionHashIndexer*, *FastTextIndexer*

property words
Get the list of known words

Returns **words** – list of known words

Return type List[str]

static chunk_identifier() → *finalfusion.io.ChunkIdentifier*
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

static read_chunk(*file: BinaryIO*) → *finalfusion.vocab.subword.ExplicitVocab*
Read the Chunk and return it.

The file must be positioned before the contents of the `Chunk` but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

write_chunk(*file*) → None
Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

idx(*item: str*, *default: Union[int, List[int], None] = None*) → Union[int, List[int], None]
Lookup the given query item.

This lookup does not raise an exception if the vocab can't produce indices.

Parameters

- **item** (*str*) – The query item.
- **default** (*Optional[Union[int, List[int]]]*) – Fall-back value to return if the vocab can't provide indices.

Returns

index –

- An integer if there is a single index for a known item.
- A list if the vocab can provide subword indices for a unknown item.
- The provided *default* item if the vocab can't provide indices.

Return type Optional[Union[int, List[int]]]

property max_n
Get the upper bound of the range of extracted n-grams.

Returns **max_n** – upper bound of n-gram range.

Return type int

property min_n
Get the lower bound of the range of extracted n-grams.

Returns **min_n** – lower bound of n-gram range.

Return type int

subword_indices(*item: str*, *bracket: bool = True*, *with_ngrams: bool = False*) → List[Union[int, Tuple[str, int]]]
Get the subword indices for the given item.

This list does not contain the index for known items.

Parameters

- **item** (*str*) – The query item.
- **bracket** (*bool*) – Toggles bracketing the item with '<' and '>' before extraction.
- **with_ngrams** (*bool*) – Toggles returning ngrams together with indices.

Returns **indices** – The list of subword indices.

Return type List[Union[int, Tuple[str, int]]]

subwords(*item: str*, *bracket: bool = True*) → List[str]
Get the n-grams of the given item as a list.

The n-gram range is determined by the *min_n* and *max_n* values.

Parameters

- **item** (*str*) – The query item to extract n-grams from.
- **bracket** (*bool*) – Toggles bracketing the item with '<' and '>' before extraction.

Returns **ngrams** – List of n-grams.

Return type List[str]

property upper_bound
The exclusive upper bound of indices in this vocabulary.

Returns **upper_bound** – Exclusive upper bound of indices covered by the vocabulary.

Return type int

write(*file: Union[str, bytes, int, os.PathLike]*)
Write the Chunk as a standalone finalfusion file.

Parameters **file** (*Union[str, bytes, int, PathLike]*) – Output path

Raises **TypeError** – If the Chunk is a `Header`.

finalfusion.vocab.subword.**load_explicit_vocab**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.vocab.subword.ExplicitVocab*
Load a ExplicitVocab from the given finalfusion file.

Parameters **file** (*str, bytes, int, PathLike*) – Path to file containing a ExplicitVocab chunk.

Returns **vocab** – Returns the first ExplicitVocab in the file.

Return type *ExplicitVocab*

FastTextVocab

class finalfusion.vocab.subword.**FastTextVocab**(*words: List[str]*, *indexer: Optional[*finalfusion.subword.hash_indexers.FastTextIndexer*] = None*)

Bases: *finalfusion.vocab.subword.SubwordVocab*

FastText vocabulary

__init__(*words: List[str]*, *indexer: Optional[*finalfusion.subword.hash_indexers.FastTextIndexer*] = None*)

Initialize a FastTextVocab.

Initializes the vocabulary with the given words.

If no indexer is passed, a FastTextIndexer with 2_000_000 buckets is used.

The word list cannot contain duplicate entries.

Parameters

- **words** (*List[str]*) – List of unique words
- **indexer** (*FastTextIndexer, optional*) – Subword indexer to use for the vocabulary. Defaults to an indexer with 2_000_000 buckets and range 3-6.

Raises **AssertionError** – If the indexer is not a FastTextIndexer or `words` contains duplicate entries.

to_explicit() → *finalfusion.vocab.subword.ExplicitVocab*

Return an ExplicitVocab built from this vocab.

This method iterates over the known words and extracts all ngrams within this vocab's bounds. Each of the ngrams is hashed and mapped to an index. This index is not necessarily unique for each ngram, if hashes collide, multiple ngrams will be mapped to the same index.

The returned vocab will be unable to produce indices for unknown ngrams.

The indices of the new vocabs known indices will be cover *[0, vocab.upper_bound)*

Returns **explicit_vocab** – The converted vocabulary.

Return type *ExplicitVocab*

property subword_indexer

Get this vocab's subword Indexer.

The subword indexer produces indices for n-grams.

In case of bucket vocabularies, this is a hash-based indexer (*FinalfusionHashIndexer*, *FastTextIndexer*). For explicit subword vocabularies, this is an *ExplicitIndexer*.

Returns **subword_indexer** – The subword indexer of the vocabulary.

Return type *ExplicitIndexer*, *FinalfusionHashIndexer*, *FastTextIndexer*

property word_index

Get the index of known words

Returns **dict** – index of known words

Return type Dict[str, int]

property words

Get the list of known words

Returns **words** – list of known words

Return type List[str]

static read_chunk(*file: BinaryIO*) → *finalfusion.vocab.subword.FastTextVocab*
Read the Chunk and return it.

The file must be positioned before the contents of the `Chunk` but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

write_chunk(*file: BinaryIO*)
Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

static chunk_identifier()
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

idx(*item: str*, *default: Union[int, List[int], None] = None*) → Union[int, List[int], None]
Lookup the given query item.

This lookup does not raise an exception if the vocab can't produce indices.

Parameters

- **item** (*str*) – The query item.
- **default** (*Optional[Union[int, List[int]]]*) – Fall-back value to return if the vocab can't provide indices.

Returns

index –

- An integer if there is a single index for a known item.
- A list if the vocab can provide subword indices for a unknown item.
- The provided *default* item if the vocab can't provide indices.

Return type Optional[Union[int, List[int]]]

property max_n
Get the upper bound of the range of extracted n-grams.

Returns **max_n** – upper bound of n-gram range.

Return type int

property min_n
Get the lower bound of the range of extracted n-grams.

Returns **min_n** – lower bound of n-gram range.

Return type int

subword_indices(*item: str*, *bracket: bool = True*, *with_ngrams: bool = False*) → List[Union[int, Tuple[str, int]]]
Get the subword indices for the given item.

This list does not contain the index for known items.

Parameters

- **item** (*str*) – The query item.
- **bracket** (*bool*) – Toggles bracketing the item with '<' and '>' before extraction.
- **with_ngrams** (*bool*) – Toggles returning ngrams together with indices.

Returns **indices** – The list of subword indices.

Return type List[Union[int, Tuple[str, int]]]

subwords (*item: str*, *bracket: bool = True*) → List[str]

Get the n-grams of the given item as a list.

The n-gram range is determined by the *min_n* and *max_n* values.

Parameters

- **item** (*str*) – The query item to extract n-grams from.
- **bracket** (*bool*) – Toggles bracketing the item with '<' and '>' before extraction.

Returns **ngrams** – List of n-grams.

Return type List[str]

property upper_bound

The exclusive upper bound of indices in this vocabulary.

Returns **upper_bound** – Exclusive upper bound of indices covered by the vocabulary.

Return type int

write (*file: Union[str, bytes, int, os.PathLike]*)

Write the Chunk as a standalone finalfusion file.

Parameters **file** (*Union[str, bytes, int, PathLike]*) – Output path

Raises **TypeError** – If the Chunk is a `Header`.

finalfusion.vocab.subword.**load_fasttext_vocab** (*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.vocab.subword.FastTextVocab*

Load a FastTextVocab from the given finalfusion file.

Parameters **file** (*str, bytes, int, PathLike*) – Path to file containing a FastTextVocab chunk.

Returns **vocab** – Returns the first FastTextVocab in the file.

Return type *FastTextVocab*

Interfaces

class finalfusion.vocab.vocab.**Vocab**

Bases: *finalfusion.io.Chunk*, collections.abc.Collection, typing.Generic

Finalfusion vocabulary interface.

Vocabs provide at least a simple string to index mapping and index to string mapping. Vocab is the base type of all vocabulary types.

abstract property words

Get the list of known words

Returns **words** – list of known words

Return type List[str]

abstract property word_index
Get the index of known words

Returns **dict** – index of known words

Return type Dict[str, int]

abstract property upper_bound
The exclusive upper bound of indices in this vocabulary.

Returns **upper_bound** – Exclusive upper bound of indices covered by the vocabulary.

Return type int

abstract idx(*item: str*, *default: Union[int, List[int], None] = None*) → Union[int, List[int], None]
Lookup the given query item.

This lookup does not raise an exception if the vocab can't produce indices.

Parameters

- **item** (*str*) – The query item.
- **default** (*Optional[Union[int, List[int]]]*) – Fall-back value to return if the vocab can't provide indices.

Returns

index –

- An integer if there is a single index for a known item.
- A list if the vocab can provide subword indices for a unknown item.
- The provided *default* item if the vocab can't provide indices.

Return type Optional[Union[int, List[int]]]

class finalfusion.vocab.subword.**SubwordVocab**
Bases: *finalfusion.vocab.vocab.Vocab*

Interface for vocabularies with subword lookups.

idx(*item: str*, *default: Union[int, List[int], None] = None*) → Union[int, List[int], None]
Lookup the given query item.

This lookup does not raise an exception if the vocab can't produce indices.

Parameters

- **item** (*str*) – The query item.
- **default** (*Optional[Union[int, List[int]]]*) – Fall-back value to return if the vocab can't provide indices.

Returns

index –

- An integer if there is a single index for a known item.
- A list if the vocab can provide subword indices for a unknown item.
- The provided *default* item if the vocab can't provide indices.

Return type Optional[Union[int, List[int]]]

property upper_bound
The exclusive upper bound of indices in this vocabulary.

Returns **upper_bound** – Exclusive upper bound of indices covered by the vocabulary.

Return type int

property min_n
Get the lower bound of the range of extracted n-grams.

Returns **min_n** – lower bound of n-gram range.

Return type int

property max_n
Get the upper bound of the range of extracted n-grams.

Returns **max_n** – upper bound of n-gram range.

Return type int

abstract property subword_indexer
Get this vocab's subword Indexer.

The subword indexer produces indices for n-grams.

In case of bucket vocabularies, this is a hash-based indexer (*FinalfusionHashIndexer*, *FastTextIndexer*). For explicit subword vocabularies, this is an *ExplicitIndexer*.

Returns **subword_indexer** – The subword indexer of the vocabulary.

Return type *ExplicitIndexer*, *FinalfusionHashIndexer*, *FastTextIndexer*

subwords(*item: str*, *bracket: bool = True*) → List[str]
Get the n-grams of the given item as a list.

The n-gram range is determined by the *min_n* and *max_n* values.

Parameters

- **item** (*str*) – The query item to extract n-grams from.
- **bracket** (*bool*) – Toggles bracketing the item with '<' and '>' before extraction.

Returns **ngrams** – List of n-grams.

Return type List[str]

subword_indices(*item: str*, *bracket: bool = True*, *with_ngrams: bool = False*) → List[Union[int, Tuple[str, int]]]
Get the subword indices for the given item.

This list does not contain the index for known items.

Parameters

- **item** (*str*) – The query item.
- **bracket** (*bool*) – Toggles bracketing the item with '<' and '>' before extraction.
- **with_ngrams** (*bool*) – Toggles returning ngrams together with indices.

Returns **indices** – The list of subword indices.

Return type List[Union[int, Tuple[str, int]]]

finalfusion.vocab.**load_vocab**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.vocab.vocab.Vocab*

Load any vocabulary from a finalfusion file.

Loads the first known vocabulary from a finalfusion file.

One of:

- *SimpleVocab*,
- *FinalfusionBucketVocab*
- *FastTextVocab*
- *ExplicitVocab*

Parameters **file** (*str, bytes, int, PathLike*) – Path to file containing a finalfusion vocab chunk.

Returns **vocab** – First vocabulary in the file.

Return type *Vocab*

Raises **ValueError** – If the file did not contain a vocabulary.

1.4.4 Subwords

finalfusion.subword

finalfusion.subword.hash_indexers.FastTextIndexer([...])	File: src/finalfusion/subword/hash_indexers.pyx (starting at line 155)
finalfusion.subword.hash_indexers.FinalfusionHashIndexer([...])	File: src/finalfusion/subword/hash_indexers.pyx (starting at line 17)
finalfusion.subword.explicit_indexer.ExplicitIndexer(...)	File: src/finalfusion/subword/explicit_indexer.pyx (starting at line 9)
finalfusion.subword.ngrams.ngrams(... [, bracket])	File: src/finalfusion/subword/ngrams.pyx (starting at line 7)

FinalfusionHashIndexer

class finalfusion.subword.hash_indexers.**FinalfusionHashIndexer**(*bucket_exp=21*, *min_n=3*, *max_n=6*)

File: src/finalfusion/subword/hash_indexers.pyx (starting at line 17)

FinalfusionHashIndexer

FinalfusionHashIndexer is a hash-based subword indexer. It hashes n-grams with the FNV-1a algorithm and maps the hash to a predetermined bucket space.

N-grams can be indexed directly through the *__call__* method or all n-grams in a string can be indexed in bulk through the *subword_indices* method.

buckets_exp

The bucket exponent.

The indexer has 2**buckets_exp buckets.

Returns **buckets_exp** – The buckets exponent

Return type int

max_n
The upper bound of the n-gram range.

Returns **max_n** – Upper bound of n-gram range

Return type int

min_n
The lower bound of the n-gram range.

Returns **min_n** – Lower bound of n-gram range

Return type int

subword_indices (*self*, *unicode word*, *uint64_t offset=0*, *bracket=True*, *with_ngrams=False*)
File: src/finalfusion/subword/hash_indexers.pyx (starting at line 97)

Get the subword indices for a word.

Parameters

- **word** (*str*) – The string to extract n-grams from
- **offset** (*int*) – The offset to add to the index, e.g. the length of the word-vocabulary.
- **bracket** (*bool*) – Toggles bracketing the input string with < and >
- **with_ngrams** (*bool*) – Toggles returning tuples of (ngram, idx)

Returns **indices** – List of n-gram indices, optionally as *(str, int)* tuples.

Return type list

Raises **TypeError** – If *word* is None.

upper_bound
Get the **exclusive** upper bound

This is the number of distinct indices.

Returns **upper_bound** – Exclusive upper bound of the indexer.

Return type int

FastTextIndexer

class finalfusion.subword.hash_indexers.**FastTextIndexer**(*n_buckets=2000000*, *min_n=3*, *max_n=6*)
File: src/finalfusion/subword/hash_indexers.pyx (starting at line 155)

FastTextIndexer

FastTextIndexer is a hash-based subword indexer. It hashes n-grams with (a slightly faulty) FNV-1a variant and maps the hash to a predetermined bucket space.

N-grams can be indexed directly through the *__call__* method or all n-grams in a string can be indexed in bulk through the *subword_indices* method.

max_n
The upper bound of the n-gram range.

Returns **max_n** – Upper bound of n-gram range

Return type int

min_n
The lower bound of the n-gram range.

Returns **min_n** – Lower bound of n-gram range

Return type int

n_buckets
Get the number of buckets.

Returns **n_buckets** – Number of buckets

Return type int

subword_indices (*self*, *unicode word*, *uint64_t offset=0*, *bracket=True*, *with_ngrams=False*)
File: src/finalfusion/subword/hash_indexers.pyx (starting at line 219)

Get the subword indices for a word.

Parameters

- **word** (*str*) – The string to extract n-grams from
- **offset** (*int*) – The offset to add to the index, e.g. the length of the word-vocabulary.
- **bracket** (*bool*) – Toggles bracketing the input string with < and >
- **with_ngrams** (*bool*) – Toggles returning tuples of (ngram, idx)

Returns **indices** – List of n-gram indices, optionally as *(str, int)* tuples.

Return type list

Raises **TypeError** – If *word* is None.

ExplicitIndexer

class finalfusion.subword.explicit_indexer.**ExplicitIndexer**(*ngrams: List[str]*, *min_n: int = 3*, *max_n: int = 6*, *ngram_index: Optional[Dict[str, int]] = None*)
File: src/finalfusion/subword/explicit_indexer.pyx (starting at line 9)

ExplicitIndexer

Explicit Indexers do not index n-grams through hashing but define an actual lookup table.

It can be constructed from a list of **unique** ngrams. In that case, the ith ngram in the list will be mapped to index i. It is also possible to pass a mapping via *ngram_index* which allows mapping multiple ngrams to the same value.

N-grams can be indexed directly through the *__call__* method or all n-grams in a string can be indexed in bulk through the *subword_indices* method.

subword_indices optionally returns tuples of form *(ngram, idx)*, otherwise a list of indices belonging to the input string is returned.

max_n
The upper bound of the n-gram range.

Returns **max_n** – Upper bound of n-gram range

Return type int

min_n
The lower bound of the n-gram range.

Returns **min_n** – Lower bound of n-gram range

Return type int

ngram_index
Get the ngram-index mapping.

Note: If you mutate this mapping you can make the indexer invalid.

Returns **ngram_index** – The ngram -> index mapping.

Return type Dict[str, int]

ngrams
Get the list of n-grams.

Note: If you mutate this list you can make the indexer invalid.

Returns **ngrams** – The list of in-vocabulary n-grams.

Return type List[str]

subword_indices (*self*, *unicode word*, *uint64_t offset=0*, *bracket=True*, *with_ngrams=False*)
File: src/finalfusion/subword/explicit_indexer.pyx (starting at line 129)

Get the subword indices for a word.

Parameters

- **word** (*str*) – The string to extract n-grams from
- **offset** (*int*) – The offset to add to the index, e.g. the length of the word-vocabulary.
- **bracket** (*bool*) – Toggles bracketing the input string with < and >
- **with_ngrams** (*bool*) – Toggles returning tuples of (ngram, idx)

Returns **indices** – List of n-gram indices, optionally as *(str, int)* tuples.

Return type list

Raises **TypeError** – If *word* is None.

upper_bound
Get the **exclusive** upper bound

This is the number of distinct indices.

This number can become invalid if *ngram_index* or *ngrams* is mutated.

Returns **idx_bound** – Exclusive upper bound of the indexer.

Return type int

NGrams

finalfusion.subword.ngrams.**ngrams**(*unicode word*, *uint32_t min_n=3*, *uint32_t max_n=6*, *bracket=True*)

File: src/finalfusion/subword/ngrams.pyx (starting at line 7)

Get the ngrams for the given word.

Parameters

- **word** (*str*) – The string to extract n-grams from
- **min_n** (*int*) – Inclusive lower bound of n-gram range. Must be greater than zero and smaller or equal to *max_n*
- **max_n** (*int*) – Inclusive upper bound of n-gram range. Must be greater than zero and greater or equal to *min_n*
- **bracket** (*bool*) – Toggles bracketing the input string with < and >

Returns **ngrams** – List of n-grams.

Return type list

Raises

- **AssertionError** – If *max_n* < *min_n* or *min_n* <= *0*.
- **TypeError** – If *word* is None.

1.4.5 Metadata

finalfusion metadata

class finalfusion.metadata.**Metadata**

Bases: dict, *finalfusion.io.Chunk*

Embeddings metadata

Metadata can be used as a regular Python dict. For serialization, the contents need to be serializable through *toml.dumps*. Finalfusion assumes metadata to be a TOML formatted string.

Examples

```
>>> metadata = Metadata({'Some': 'value', 'number': 1})
>>> metadata
{'Some': 'value', 'number': 1}
>>> metadata['Some']
'value'
>>> metadata['Some'] = 'other value'
>>> metadata['Some']
'other value'
```

static chunk_identifier() → *finalfusion.io.ChunkIdentifier*

Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

static read_chunk(*file: BinaryIO*) → *finalfusion.metadata.Metadata*
Read the Chunk and return it.

The file must be positioned before the contents of the `Chunk` but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

write_chunk(*file: BinaryIO*)
Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

finalfusion.metadata.**load_metadata**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.metadata.Metadata*
Load a Metadata chunk from the given file.

Parameters **file** (*str, bytes, int, PathLike*) – Finalfusion file with a metadata chunk.

Returns **metadata** – The Metadata from the file.

Return type *Metadata*

Raises **ValueError** – If the file did not contain an Metadata chunk.

1.4.6 Norms

Norms module.

class finalfusion.norms.**Norms**
Bases: `numpy.ndarray`, *finalfusion.io.Chunk*, `collections.abc.Collection`, `typing.Generic`

Norms Chunk.

Norms subclass *numpy.ndarray*, all typical numpy operations are available.

static __new__(*cls*, *array: numpy.ndarray*)
Construct new Norms.

Parameters **array** (*numpy.ndarray*) – Norms array.

Returns **norms** – The norms.

Return type *Norms*

Raises **AssertionError** – If array is not a 1-d array of float32 values.

static chunk_identifier() → *finalfusion.io.ChunkIdentifier*
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

static read_chunk(*file: BinaryIO*) → *finalfusion.norms.Norms*
Read the Chunk and return it.

The file must be positioned before the contents of the `Chunk` but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

write_chunk (*file: BinaryIO*)
Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

finalfusion.norms.**load_norms** (*file: Union[str, bytes, int, os.PathLike]*)
Load Norms from a finalfusion file.

Loads the first Norms chunk from a finalfusion file.

Parameters **file** (*str, bytes, int, PathLike*) – Path to finalfusion file containing a Norms chunk.

Returns **norms** – First finalfusion Norms in the file.

Return type *Norms*

Raises **ValueError** – If the file did not contain norms.

1.4.7 IO

class finalfusion.io.**Chunk**
Basic building blocks of finalfusion files.

write (*file: Union[str, bytes, int, os.PathLike]*)
Write the Chunk as a standalone finalfusion file.

Parameters **file** (*Union[str, bytes, int, PathLike]*) – Output path

Raises **TypeError** – If the Chunk is a `Header`.

abstract static chunk_identifier () → *finalfusion.io.ChunkIdentifier*
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

abstract static read_chunk (*file: BinaryIO*) → *finalfusion.io.Chunk*
Read the Chunk and return it.

The file must be positioned before the contents of the *Chunk* but after its header.

Parameters **file** (*BinaryIO*) – a finalfusion file containing the given Chunk

Returns **chunk** – The chunk read from the file.

Return type *Chunk*

abstract write_chunk (*file: BinaryIO*)
Write the Chunk to a file.

Parameters **file** (*BinaryIO*) – Output file for the Chunk

class finalfusion.io.**ChunkIdentifier**
Bases: enum.IntEnum

Known finalfusion Chunk types.

Header = 0

SimpleVocab = 1

NdArray = 2

BucketSubwordVocab = 3

QuantizedArray = 4

Metadata = 5

NdNorms = 6

FastTextSubwordVocab = 7

ExplicitSubwordVocab = 8

class finalfusion.io.**FinalfusionFormatError**
Bases: Exception

Exception to specify that the format of a finalfusion file was incorrect.

class finalfusion.io.**TypeId**
Bases: enum.IntEnum

Known finalfusion data types.

u8 = 1

f32 = 10

1.4.8 Compat

finalfusion.compat

finalfusion.compat.load_fasttext(file)	Read embeddings from a file in fastText format.
finalfusion.compat.write_fasttext(file, embeds)	Write embeddings in fastText format.
finalfusion.compat.load_text(file)	Read embeddings in text format.
finalfusion.compat.write_text(file, embeddings)	Write embeddings in text format.
finalfusion.compat.load_text_dims(file)	Read emebddings in text-dims format.
finalfusion.compat.write_text_dims(file, ...)	Write embeddings in text-dims format.
finalfusion.compat.load_word2vec(file)	Read embeddings in word2vec binary format.
finalfusion.compat.write_word2vec(file, ...)	Write embeddings in word2vec binary format.

Word2Vec

Word2vec binary format.

finalfusion.compat.word2vec.**load_word2vec**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.embeddings.Embeddings*

Read embeddings in word2vec binary format.

The returned embeddings have a SimpleVocab, NdArray storage and a Norms chunk. The storage is l2-normalized per default and the corresponding norms are stored in the Norms.

Files are expected to start with a line containing rows and cols in utf-8. Words are encoded in utf-8 followed by a single whitespace. After the whitespace, the embedding components are expected as little-endian single-precision floats.

Parameters **file** (*str, bytes, int, PathLike*) – Path to a file with embeddings in word2vec binary format.

Returns **embeddings** – The embeddings from the input file.

Return type *Embeddings*

finalfusion.compat.word2vec.**write_word2vec**(*file: Union[str, bytes, int, os.PathLike]*, *embeddings:* finalfusion.embeddings.Embeddings)

Write embeddings in word2vec binary format.

If the embeddings are not compatible with the w2v format (e.g. include a SubwordVocab), only the known words and embeddings are serialized. I.e. the subword matrix is discarded.

Embeddings are un-normalized before serialization, if norms are present, each embedding is scaled by the associated norm.

The output file will contain the shape encoded in utf-8 on the first line as *rows columns*. This is followed by the embeddings.

Each embedding consists of:

- utf-8 encoded word
- single space `' '` following the word
- `cols` single-precision floating point numbers
- `'\n'` newline at the end of each line.

Parameters

- **file** (*str, bytes, int, PathLike*) – Output file
- **embeddings** (*Embeddings*) – The embeddings to serialize.

Text

Text based embedding formats.

finalfusion.compat.text.**load_text_dims**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.embeddings.Embeddings*

Read emebddings in text-dims format.

The returned embeddings have a SimpleVocab, NdArray storage and a Norms chunk. The storage is l2-normalized per default and the corresponding norms are stored in the Norms.

The first line contains whitespace separated rows and cols, the rest of the file contains whitespace separated word and vector components.

Parameters **file** (*str, bytes, int, PathLike*) – Path to a file with embeddings in word2vec binary format.

Returns **embeddings** – The embeddings from the input file.

Return type *Embeddings*

finalfusion.compat.text.**load_text**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.embeddings.Embeddings*

Read embeddings in text format.

The returned embeddings have a SimpleVocab, NdArray storage and a Norms chunk. The storage is l2-normalized per default and the corresponding norms are stored in the Norms.

Expects a file with utf-8 encoded lines with:

- word at the start of the line

- followed by whitespace
- followed by whitespace separated vector components

Parameters **file** (*str, bytes, int, PathLike*) – Path to a file with embeddings in word2vec binary format.

Returns **embeddings** – Embeddings from the input file. The resulting Embeddings will have a SimpleVocab, NdArray and Norms.

Return type *Embeddings*

finalfusion.compat.text.**write_text**(*file: Union[str, bytes, int, os.PathLike]*, *embeddings:* finalfusion.embeddings.Embeddings, *sep=' '*)

Write embeddings in text format.

Embeddings are un-normalized before serialization, if norms are present, each embedding is scaled by the associated norm.

The output consists of utf-8 encoded lines with:

- word at the start of the line
- followed by whitespace
- followed by whitespace separated vector components

Parameters

- **file** (*str, bytes, int, PathLike*) – Output file
- **embeddings** (*Embeddings*) – Embeddings to write
- **sep** (*str*) – Separator of word and embeddings.

finalfusion.compat.text.**write_text_dims**(*file: Union[str, bytes, int, os.PathLike]*, *embeddings:* finalfusion.embeddings.Embeddings, *sep=' '*)

Write embeddings in text-dims format.

Embeddings are un-normalized before serialization, if norms are present, each embedding is scaled by the associated norm.

The output consists of utf-8 encoded lines with:

- *rows cols* on the **first** line
- word at the start of the line
- followed by whitespace
- followed by whitespace separated vector components

Parameters

- **file** (*str, bytes, int, PathLike*) – Output file
- **embeddings** (*Embeddings*) – Embeddings to write
- **sep** (*str*) – Separator of word and embeddings.

FastText

Fasttext IO compat module.

finalfusion.compat.fasttext.**load_fasttext**(*file: Union[str, bytes, int, os.PathLike]*) → *finalfusion.embeddings.Embeddings*

Read embeddings from a file in fastText format.

The returned embeddings have a FastTextVocab, NdArray storage and a Norms chunk.

Loading embeddings with this method will precompute embeddings for each word by averaging all of its subword embeddings together with the distinct word vector. Additionally, all precomputed vectors are l2-normalized and the corresponding norms are stored in the Norms. The subword embeddings are **not** l2-normalized.

Parameters **file** (*str, bytes, int, PathLike*) – Path to a file with embeddings in word2vec binary format.

Returns **embeddings** – The embeddings from the input file.

Return type *Embeddings*

finalfusion.compat.fasttext.**write_fasttext**(*file: Union[str, bytes, int, os.PathLike]*, *embeds:* finalfusion.embeddings.Embeddings)

Write embeddings in fastText format.

fastText requires Metadata with all expected keys for fastText configs:

- dims: int (inferred from model)
- window_size: int (default -1)
- min_count: int (default -1)
- ns: int (default -1)
- word_ngrams: int (default 1)
- loss: one of `['HierarchicalSoftmax', 'NegativeSampling', 'Softmax']` (default Softmax)
- model: one of `['CBOW', 'SkipGram', 'Supervised']` (default SkipGram)
- buckets: int (inferred from model)
- min_n: int (inferred from model)
- max_n: int (inferred from model)
- lr_update_rate: int (default -1)
- sampling_threshold: float (default -1)

`dims`, `buckets`, `min_n` and `max_n` are inferred from the model. If other values are unspecified, a default value of `-1` is used for all numerical fields. Loss defaults to `Softmax`, model to `SkipGram`. Unknown values for `loss` and `model` are overwritten with defaults since the models are incompatible with fastText otherwise.

Some information from original fastText models gets lost e.g.:

- word frequencies
- n_tokens

Embeddings are un-normalized before serialization: if norms are present, each embedding is scaled by the associated norm. Additionally, the original state of the embedding matrix is restored, precomputation and l2-normalization of word embeddings is undone.

Only embeddings with a FastTextVocab or SimpleVocab can be serialized to this format.

Parameters

- **file** (*str, bytes, int, PathLike*) – Output file
- **embeds** (*Embeddings*) – Embeddings to write

1.5 Scripts

Installing `finalfusion` adds some exectuables:

- `ffp-convert` for converting embeddings
- `ffp-similar` for similarity queries
- `ffp-analogy` for analogy queries
- `ffp-bucket-to-explicit` to convert bucket subword to explicit subword embeddings

1.5.1 Convert

`ffp-convert` makes conversion between all supported embedding formats possible:

```
$ ffp-convert --help
usage: ffp-convert [-h] [-f INPUT_FORMAT] [-t OUTPUT_FORMAT] INPUT OUTPUT

Convert embeddings.

positional arguments:
  INPUT                 Input embeddings
  OUTPUT                Output path

optional arguments:
  -h, --help            show this help message and exit
  -f INPUT_FORMAT, --from INPUT_FORMAT
                        Valid choices: ['word2vec', 'finalfusion', 'fasttext',
                        'text', 'textdims'] Default: 'word2vec'
  -t OUTPUT_FORMAT, --to OUTPUT_FORMAT
                        Valid choices: ['word2vec', 'finalfusion', 'fasttext',
                        'text', 'textdims'] Default: 'finalfusion'
```

1.5.2 Similar

`ffp-similar` supports similarity queries:

```
$ ffp-similar --help
usage: ffp-similar [-h] [-f INPUT_FORMAT] [-k K] EMBEDDINGS [input]

Similarity queries.

positional arguments:
  EMBEDDINGS            Input embeddings
  input                 Optional input file with one word per line. If
                        unspecified reads from stdin
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -f INPUT_FORMAT, --format INPUT_FORMAT
                        Valid choices: ['word2vec', 'finalfusion', 'fasttext',
                        'text', 'textdims'] Default: 'finalfusion'
  -k K                  Number of neighbours. Default: 10
```

1.5.3 Analogy

`ffp-analogy` answers analogy queries:

```
$ ffp-analogy --help
usage: ffp-analogy [-h] [-f INPUT_FORMAT] [-i {a,b,c} [{a,b,c} ...]] [-k K]
                   EMBEDDINGS [input]

Analogy queries.

positional arguments:
  EMBEDDINGS            Input embeddings
  input                 Optional input file with 3 words per line. If
                        unspecified reads from stdin

optional arguments:
  -h, --help            show this help message and exit
  -f INPUT_FORMAT, --format INPUT_FORMAT
                        Valid choices: ['word2vec', 'finalfusion', 'fasttext',
                        'text', 'textdims'] Default: 'finalfusion'
  -i {a,b,c} [{a,b,c} ...], --include {a,b,c} [{a,b,c} ...]
                        Specify query parts that should be allowed as answers.
                        Valid choices: ['a', 'b', 'c']
  -k K                  Number of neighbours. Default: 10
```

1.5.4 Bucket to Explicit

`ffp-bucket-to-explicit` converts bucket subword embeddings to explicit subword embeddings:

```
$ ffp-bucket-to-explicit
usage: ffp-bucket-to-explicit [-h] [-f INPUT_FORMAT] INPUT OUTPUT

Convert bucket embeddings to explicit lookups.

positional arguments:
  INPUT                 Input bucket embeddings
  OUTPUT                Output path

optional arguments:
  -h, --help            show this help message and exit
  -f INPUT_FORMAT, --from INPUT_FORMAT
                        Valid choices: ['finalfusion', 'fasttext'] Default:
                        'finalfusion'
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

E

F

H

I

L

M

N

P

Q

R

S

T

U

V

W