finalfusion

Release 0.2pre

Sebastian Pütz

Jun 05, 2020

CONTENTS

1 Contents **3**
- 1.1 Quickstart . . . 3
- 1.2 Install . . . 3
- 1.3 Top-level Exports . . . 4
- 1.4 API . . . 5

2 Indices and tables **37**

Python Module Index **39**

Index **41**

`ffp` is a Python package to interface with finalfusion embeddings. `ffp` supports all common embedding formats, including finalfusion, fastText, word2vec binary, text and textdims.

`ffp` integrates nicely with `numpy` since its *`ffp.storage.Storage`* types can be treated as ndarrays.

The `finalfusion` format revolves around *`ffp.io.Chunk`*s, these are specified in the finalfusion spec. Each component class in `ffp` implements the *`ffp.io.Chunk`* interface which specifies serialization and deserialization. Any unique combination of chunks can make up `ffp.Embeddings`.

CHAPTER

ONE

CONTENTS

1.1 Quickstart

You can *install* `ffp` through:

```
pip install ffp
```

And use embeddings by:

```
import ffp
# load finalfusion embeddings
embeddings = ffp.load_finalfusion("/path/to/embeddings.fifu")
# embedding lookup
embedding = embeddings["Test"]
# embedding lookup with default value
embedding = embeddings.embedding("Test", default=0)
# access storage and calculate dot product with an embedding
storage = embedding.dot(embeddings.storage)
# print 10 first vocab items
print(embeddings.vocab.words[:10])
# print embeddings metadata
print(embeddings.metadata)
```

`ffp` exports most common-use functions and types in the top level. See *Top-Level Exports* for an overview.

These re-exports are also available in their respective sub-packages and modules. The full API documentation can be foud *here*.

1.2 Install

`ffp` can be installed from GitHub via:

```
$ pip install git+https://github.com/sebpuetz/ffp
```

or directly from pypi:

```
$ pip install ffp
```

When building from source, `ffp` requires `Cython`.

1.3 Top-level Exports

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

1.3.1 Embeddings

ffp.embeddings.Embeddings(storage, vocab[, ...])	Embeddings class.
ffp.embeddings.load_finalfusion(file[, mmap])	Read embeddings from a file in finalfusion format.
ffp.embeddings.load_fastText(file)	Read embeddings from a file in fastText format.
ffp.embeddings.load_text(file)	Read embeddings in text format.
ffp.embeddings.load_textdims(file)	Read emebddings in textdims format.
ffp.embeddings.load_word2vec(file)	Read embeddings in word2vec binary format.

1.3.2 Metadata

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

1.3.3 Norms

ffp.norms.Norms	Embedding Norms.
ffp.norms.load_norms(file)	Load an Norms chunk from the given file.

1.3.4 Storage

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

1.3.5 Vocab

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

1.4 API

1.4.1 Embeddings

Finalfusion Embeddings

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

Bases: object

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*: either *NdArray* or *QuantizedArray* *(required)*
2. Vocab, one of *SimpleVocab*, *FinalfusionBucketVocab*, *FastTextVocab* and *ExplicitVocab*

   (required)
3. *Norms*
4. *Metadata*

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])
```

(continues on next page)

(continued from previous page)

```
True
>>> embeddings.metadata
{'Some': 'value', 'numerical': 0}
```

__init__(*storage: ffp.storage.storage.Storage*, *vocab:* ffp.vocab.vocab.Vocab, *norms: Optional[*ffp.norms.Norms*] = None*, *metadata: Optional[*ffp.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 embeddings.

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
>>> 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.

bucket_to_explicit() → *ffp.embeddings.Embeddings*
Convert bucket embeddings to embeddings with explicit lookup.

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.

A side effect of this method is the conversion from a quantized storage to an array storage.

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)

chunks() → List[*ffp.io.Chunk*]
Get the Embeddings Chunks as a list.

The Chunks are ordered in the expected serialization order: 1. Metadata 2. Vocabulary 3. Storage 4. Norms

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

Return type List[*Chunk*]

write (*file: str*)

Write the Embeddings to the given file.

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

Parameters **file** (*str*) – Path of the output file.

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

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

Read embeddings in word2vec binary format.

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 float32.

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*

ffp.embeddings.**load_textdims** (*file: Union[str, bytes, int, os.PathLike]*) → *ffp.embeddings.Embeddings*

Read emebddings in textdims format.

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*

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

Read embeddings in text format.

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*

ffp.embeddings.**load_fastText**(*file: Union[str, bytes, int, os.PathLike]*) → *ffp.embeddings.Embeddings*

Read embeddings from a file in fastText format.

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*

1.4.2 Storage

ffp.storage

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

NdArray

class ffp.storage.ndarray.**NdArray**(*array: numpy.ndarray*)

Bases: numpy.ndarray, *ffp.io.Chunk*, ffp.storage.storage.Storage

Array storage.

Essentially a numpy matrix, either in-memory or memory-mapped.

Examples

```
>>> matrix = np.float32(np.random.rand(10, 50))
>>> ndarray_storage = NdArray(matrix)
>>> np.allclose(matrix, ndarray_storage)
True
>>> ndarray_storage.shape
(10, 50)
```

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

Construct a new NdArray storage.

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

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

property **shape**

The storage shape

Returns **(rows, cols)** – Tuple with storage dimensions

Return type Tuple[int, int]

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

Parameters

- **file** (*BinaryIO*) – File at the beginning of a finalfusion storage
- **mmap** (*bool*) – Toggles memory mapping the buffer.

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

Return type *Storage*

static read_chunk(*file: BinaryIO*) → *ffp.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*

static mmap_storage(*file: BinaryIO*) → *ffp.storage.ndarray.NdArray*
Memory map the storage.

Parallel method to *`ffp.io.Chunk.read_chunk()`*. Instead of storing the `Storage` in-memory, it memory maps the embeddings.

Parameters **file** (*BinaryIO*) – File at the beginning of a finalfusion storage

Returns **storage** – The memory mapped storage.

Return type *Storage*

static chunk_identifier()
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

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

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

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

Quantized

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

Bases: *ffp.io.Chunk*, ffp.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:* ffp.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

The storage shape

Returns **(rows, cols)** – Tuple with storage dimensions

Return type Tuple[int, int]

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

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*) → *ffp.storage.quantized.QuantizedArray*
Load Storage from the given finalfusion file.

Parameters

- **file** (*BinaryIO*) – File at the beginning of a finalfusion storage
- **mmap** (*bool*) – Toggles memory mapping the buffer.

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

Return type *Storage*

static read_chunk(*file: BinaryIO*) → *ffp.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_storage(*file: BinaryIO*) → *ffp.storage.quantized.QuantizedArray*
Memory map the storage.

Parallel method to *`ffp.io.Chunk.read_chunk()`*. Instead of storing the `Storage` in-memory, it memory maps the embeddings.

Parameters **file** (*BinaryIO*) – File at the beginning of a finalfusion storage

Returns **storage** – The memory mapped storage.

Return type *Storage*

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

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

static chunk_identifier() → *ffp.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** (*str, bytes, int, PathLike*) – Output file

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

class ffp.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*.

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

Common interface to finalfusion storage types.

abstract property shape

The storage shape

Returns **(rows, cols)** – Tuple with storage dimensions

Return type Tuple[int, int]

abstract classmethod load(*file: BinaryIO*, *mmap=False*) → ffp.storage.storage.Storage

Load Storage from the given finalfusion file.

Parameters

- **file** (*BinaryIO*) – File at the beginning of a finalfusion storage
- **mmap** (*bool*) – Toggles memory mapping the buffer.

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

Return type *Storage*

abstract static mmap_storage(*file: BinaryIO*) → ffp.storage.storage.Storage

Memory map the storage.

Parallel method to *ffp.io.Chunk.read_chunk()*. Instead of storing the *Storage* in-memory, it memory maps the embeddings.

Parameters **file** (*BinaryIO*) – File at the beginning of a finalfusion storage

Returns **storage** – The memory mapped storage.

Return type *Storage*

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

Load any storage from a finalfusion file.

Loads the first known storage from a finalfusion file.

Parameters

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

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

Return type Union[ffp.storage.NdArray, ffp.storage.QuantizedArray]

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

1.4.3 Vocabularies

ffp.vocab

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

SimpleVocab

class ffp.vocab.simple_vocab.**SimpleVocab**(*words: List[str]*, *index: Optional[Dict[str, int]] = None*)

Bases: *ffp.io.Chunk*, *ffp.vocab.vocab.Vocab*

Simple vocabulary.

SimpleVocabs provide a simple string to index mapping and index to string mapping. SimpleVocab is also the base type of other vocabulary types.

__init__(*words: List[str]*, *index: Optional[Dict[str, int]] = None*)

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
- **index** (*Optional[Dict[str, int]]*) – Dictionary providing an entry -> index mapping.

Raises `ValueError` – if the length of *index* and *word* doesn't match.

static from_corpus (*file: Union[str, bytes, int, os.PathLike]*, *cutoff: finalfusion.vocab.cutoff.Cutoff = Cutoff(30, 'min_freq')*)

Construct a simple vocabulary from the given corpus.

Parameters

- **file** (*str, bytes, int, PathLike*) – Path to corpus file
- **cutoff** (*Cutoff*) – Frequency cutoff or target size to restrict vocabulary size.

Returns (vocab, counts) – Tuple containing the Vocabulary as first item and counts of in-vocabulary items as the second item.

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

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]

property idx_bound

The exclusive upper bound of indices in this vocabulary.

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

Return type 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 ()

Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

idx(*item*, *default=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** – `int` if there is a single index for a known item `list` of indices if the vocab can provide subword indices for a unknown item. The `default` item if the vocab can't provide indices.

Return type int, List[int], optional

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

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

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

Return type *SimpleVocab*

FinalfusionBucketVocab

class ffp.vocab.subword.**FinalfusionBucketVocab**(*words: List[str]*, *indexer: ffp.subwords.hash_indexers.FinalfusionHashIndexer = None*, *index: Optional[Dict[str, int]] = None*)
Bases: *ffp.io.Chunk*, *ffp.vocab.subword.SubwordVocab*

Finalfusion Bucket Vocabulary.

__init__(*words: List[str]*, *indexer: ffp.subwords.hash_indexers.FinalfusionHashIndexer = None*, *index: Optional[Dict[str, int]] = None*)
Initialize a FinalfusionBucketVocab.

Initializes the vocabulary with the given words and optional index and indexer.

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

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
- **indexer** (*FinalfusionHashIndexer, optional*) – Subword indexer to use for the vocabulary. Defaults to an indexer with 2^21 buckets with range 3-6.
- **index** (*Dict[str, int], optional*) – Dictionary providing an entry -> index mapping.

Raises

- **ValueError** – if the length of *index* and *word* doesn't match.

- **AssertionError** – If the indexer is not a FinalfusionHashIndexer.

static from_corpus(*file: Union[str, bytes, int, os.PathLike]*, *cutoff: Optional[ffp.vocab.cutoff.Cutoff] = None*, *indexer: Optional[*ffp.subwords.hash_indexers.FinalfusionHashIndexer*] = None*) → Tuple[*ffp.vocab.subword.FinalfusionBucketVocab*, List[int]]

Build a Finalfusion Bucket Vocabulary from a corpus.

Parameters

- **file** (*str, bytes, int, PathLike*) – File with white-space separated tokens.
- **cutoff** (*Cutoff*) – Frequency cutoff or target size to restrict vocabulary size. Defaults to minimum frequency cutoff of 30.
- **indexer** (*FinalfusionHashIndexer*) – Subword indexer to use for the vocabulary. Defaults to an indexer with 2^21 buckets with range 3-6.

Returns (vocab, counts) – Tuple containing the Vocabulary as first item and counts of in-vocabulary items as the second item.

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

Raises **AssertionError** – If the indexer is not a FinalfusionHashIndexer.

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

Returns a Vocabulary with explicit storage built from this vocab.

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 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]

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*) → *ffp.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()
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

__getitem__(*item: str*) → Union[int, List[int]]
Lookup the query item.

This method raises an exception if the vocab can't provide indices.

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

Raises **KeyError** – If no indices can be provided.

idx(*item: str*, *default=None*) → Union[List[int], 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** – `int` if there is a single index for a known item `list` of indices if the vocab can provide subword indices for a unknown item. The `default` item if the vocab can't provide indices.

Return type int, List[int], optional

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

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

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

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*) → List[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.

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

Return type List[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]

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

Parameters **file** (*str, bytes, int, PathLike*) – Output file

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

ffp.vocab.subword.**load_finalfusion_bucket_vocab** (*file: Union[str, bytes, int, os.PathLike]*) → *ffp.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 ffp.vocab.subword.**ExplicitVocab** (*words: List[str]*, *indexer: ffp.subwords.explicit_indexer.ExplicitIndexer*, *index: Dict[str, int] = None*)
Bases: *ffp.io.Chunk*, *ffp.vocab.subword.SubwordVocab*

A vocabulary with explicitly stored n-grams.

__init__ (*words: List[str]*, *indexer: ffp.subwords.explicit_indexer.ExplicitIndexer*, *index: Dict[str, int] = None*)
Initialize an ExplicitVocab.

Initializes the vocabulary with the given words, subword indexer and an optional word 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
- **indexer** (*ExplicitIndexer*) – Subword indexer to use for the vocabulary.
- **index** (*Dict[str, int], optional*) – Dictionary providing a word -> index mapping.

Raises

- **ValueError** – if the length of `index` and `word` doesn't match.
- **AssertionError** – If the indexer is not an ExplicitIndexer.

See also:

ExplicitIndexer

static from_corpus(*file: Union[str, bytes, int, os.PathLike]*, *ngram_range=3, 6*, *token_cutoff: Optional[ffp.vocab.cutoff.Cutoff] = None*, *ngram_cutoff: Optional[ffp.vocab.cutoff.Cutoff] = None*)

Build an ExplicitVocab from a corpus.

Parameters

- **file** (*str, bytes, int, PathLike*) – File with white-space separated tokens.
- **ngram_range** (*Tuple[int, int]*) – Specifies the n-gram range for the indexer.
- **token_cutoff** (*Cutoff, optional*) – Frequency cutoff or target size to restrict token vocabulary size. Defaults to minimum frequency cutoff of 30.
- **ngram_cutoff** (*Cutoff, optional*) – Frequency cutoff or target size to restrict ngram vocabulary size. Defaults to minimum frequency cutoff of 30.

Returns (vocab, counts) – Tuple containing the Vocabulary as first item, counts of in-vocabulary tokens as the second item and in-vocabulary ngram counts as the last item.

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

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 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*

static chunk_identifier()

Get the ChunkIdentifier for this Chunk.

Returns chunk_identifier

Return type *ChunkIdentifier*

static read_chunk (*file: BinaryIO*) → *ffp.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=None*) → Union[List[int], 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** – `int` if there is a single index for a known item `list` of indices if the vocab can provide subword indices for a unknown item. The `default` item if the vocab can't provide indices.

Return type int, List[int], optional

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

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

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

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*) → List[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.

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

Return type List[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]

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

Parameters **file** (*str, bytes, int, PathLike*) – Output file

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

ffp.vocab.subword.**load_explicit_vocab**(*file: Union[str, bytes, int, os.PathLike]*) → *ffp.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 ffp.vocab.subword.**FastTextVocab**(*words: List[str]*, *indexer: ffp.subwords.hash_indexers.FastTextIndexer = None*, *index: Optional[Dict[str, int]] = None*)
Bases: *ffp.io.Chunk*, *ffp.vocab.subword.SubwordVocab*

FastText vocabulary

__init__(*words: List[str]*, *indexer: ffp.subwords.hash_indexers.FastTextIndexer = None*, *index: Optional[Dict[str, int]] = None*)
Initialize a FastTextVocab.

Initializes the vocabulary with the given words and optional index and indexer.

If no indexer is passed, a FastTextIndexer with 2,000,000 buckets is used.

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
- **indexer** (*FastTextIndexer, optional*) – Subword indexer to use for the vocabulary. Defaults to an indexer with 2,000,000 buckets with range 3-6.
- **index** (*Dict[str, int], optional*) – Dictionary providing an entry -> index mapping.

Raises

- **ValueError** – if the length of *index* and *word* doesn't match.

- **AssertionError** – If the indexer is not a FastTextIndexer.

static from_corpus(*file: Union[str, bytes, int, os.PathLike]*, *cutoff: Optional[ffp.vocab.cutoff.Cutoff] = None*, *indexer: Optional[*ffp.subwords.hash_indexers.FastTextIndexer*] = None*) → Tuple[*ffp.vocab.subword.FastTextVocab*, List[int]]

Build a fastText vocabulary from a corpus.

Parameters

- **file** (*str, bytes, int, PathLike*) – File with white-space separated tokens.
- **cutoff** (*Cutoff, optional*) – Frequency cutoff or target size to restrict vocabulary size. Defaults to minimum frequency cutoff of 30.
- **indexer** (*FastTextIndexer, optional*) – Subword indexer to use for the vocabulary. Defaults to an indexer with 2,000,000 buckets with range 3-6.

Returns (vocab, counts) – Tuple containing the Vocabulary as first item and counts of in-vocabulary items as the second item.

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

Raises **AssertionError** – If the indexer is not a FastTextIndexer.

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

Returns a Vocabulary with explicit storage built from this vocab.

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 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]

static read_chunk(*file: BinaryIO*) → *ffp.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*

ffp.vocab.subword.**load_fasttext_vocab**(*file: Union[str, bytes, int, os.PathLike]*) → *ffp.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 ffp.vocab.vocab.**Vocab**
Bases: abc.ABC

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 idx_bound
The exclusive upper bound of indices in this vocabulary.

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

Return type int

abstract idx(*item: str*, *default: Union[List[int], int, None] = None*) → Union[List[int], 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** – `int` if there is a single index for a known item `list` of indices if the vocab can provide subword indices for a unknown item. The `default` item if the vocab can't provide indices.

Return type int, List[int], optional

__getitem__(*item: str*) → Union[list, int]
Lookup the query item.

This method raises an exception if the vocab can't provide indices.

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

Raises **KeyError** – If no indices can be provided.

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

Interface for vocabularies with subword lookups.

idx(*item: str*, *default=None*) → Union[List[int], 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** – `int` if there is a single index for a known item `list` of indices if the vocab can provide subword indices for a unknown item. The `default` item if the vocab can't provide indices.

Return type int, List[int], optional

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

Returns **idx_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*) → List[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.

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

Return type List[int]

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

Loads the first known vocabulary from a finalfusion file.

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

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

Return type *SimpleVocab*, *FastTextVocab*, *FinalfusionBucketVocab*, *ExplicitVocab*

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

1.4.4 Subwords

ffp.subwords

ffp.subwords.hash_indexers.FastTextIndexer([...])	FastTextIndexer
ffp.subwords.hash_indexers.FinalfusionHashIndexer([...])	FinalfusionHashIndexer
ffp.subwords.explicit_indexer.ExplicitIndexer(...)	ExplicitIndexer
ffp.subwords.ngrams.word_ngrams(...)	Get the ngrams for the given word.

FinalfusionHashIndexer

class ffp.subwords.hash_indexers.**FinalfusionHashIndexer**(*bucket_exp=21*, *min_n=3*, *max_n=6*)

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
'uint64_t'

Type buckets_exp

idx_bound
Get the **exclusive** upper bound

This is the number of distinct indices.

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

Return type int

max_n
'uint32_t'

Type max_n

min_n
'uint32_t'

Type min_n

subword_indices(*self*, *unicode word*, *uint64_t offset=0*, *bool bracket=True*, *bool with_ngrams=False*)
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.

FastTextIndexer

class ffp.subwords.hash_indexers.**FastTextIndexer**(*n_buckets=2000000*, *min_n=3*, *max_n=6*)

FastTextIndexer

FastTextIndexer is a hash-based subword indexer. It hashes n-grams with (a slightly) 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
'uint32_t'

Type max_n

min_n
'uint32_t'

Type min_n

n_buckets
'uint64_t'

Type n_buckets

subword_indices(*self*, *unicode word*, *uint64_t offset=0*, *bool bracket=True*, *bool with_ngrams=False*)
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 ffp.subwords.explicit_indexer.**ExplicitIndexer**(*ngrams: List[str]*, *ngram_range: Tuple[int, int] = 3, 6*, *ngram_index: Optional[Dict[str, int]] = None*)

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.

idx_bound
Get the **exclusive** upper bound

This is the number of distinct indices.

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

Return type int

max_n
'uint32_t'

Type max_n

min_n
'uint32_t'

Type min_n

ngram_index
Get the ngram-index mapping.

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

Return type dict

ngrams
Get the list of n-grams.

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

Return type list

subword_indices (*self*, *unicode word*, *offset=0*, *bool bracket=True*, *bool with_ngrams=False*)
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.

NGrams

ffp.subwords.ngrams.**word_ngrams**(*unicode word*, *uint32_t min_n=3*, *uint32_t max_n=6*, *bool bracket=True*)

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 ffp.metadata.**Metadata**

Bases: dict, *ffp.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()

Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

static read_chunk(*file: BinaryIO*) → *ffp.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

ffp.metadata.**load_metadata**(*file: Union[str, bytes, int, os.PathLike]*) → *ffp.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 ffp.norms.**Norms**
Bases: numpy.ndarray, *ffp.io.Chunk*

Embedding Norms.

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

The ith norm is expected to correspond to the l2 norm of the ith row in the storage before normalizing it. Therefore, Norms should have at most the same length as a given Storage and are expected to match the length of the Vocabulary.

static chunk_identifier()
Get the ChunkIdentifier for this Chunk.

Returns **chunk_identifier**

Return type *ChunkIdentifier*

static read_chunk(*file: BinaryIO*) → *ffp.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

ffp.norms.**load_norms**(*file: Union[str, bytes, int, os.PathLike]*) → *ffp.norms.Norms*
Load an Norms chunk from the given file.

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

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

Return type *Norms*

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

1.4.7 IO

This module defines some common IO operations and types.

Chunk is the building block of finalfusion embeddings, each component is serialized as its own, non-overlapping, chunk in finalfusion files.

ChunkIdentifier is a unique integer identifiers for *Chunk*.

TypeId is used to uniquely identify numerical types.

The *Header* handles the preamble of finalfusion files.

FinalfusionFormatError is raised upon reading from malformed finalfusion files.

class ffp.io.**Chunk**
Bases: abc.ABC

Basic building blocks of finalfusion files.

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

Parameters **file** (*str, bytes, int, PathLike*) – Output file

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

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

Returns **chunk_identifier**

Return type *ChunkIdentifier*

abstract static read_chunk(*file: BinaryIO*) → *ffp.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 ffp.io.**Header**(*chunk_ids*)
Bases: *ffp.io.Chunk*

Header Chunk

The header chunk handles the preamble.

property chunk_ids
Get the chunk IDs from the header

Returns **chunk_ids** – List of ChunkIdentifiers in the Header.

Return type List[*ChunkIdentifier*]

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

Returns **chunk_identifier**

Return type *ChunkIdentifier*

static read_chunk(*file: BinaryIO*) → *ffp.io.Header*
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

ffp.io.**find_chunk**(*file: BinaryIO*, *chunks: List[ChunkIdentifier]*) → Optional[*ffp.io.ChunkIdentifier*]
Find a `Chunk` in a file.

Looks for one of the specified *chunks* in the input file and seeks the file to the beginning of the first chunk found from *chunks*. I.e. the file is positioned before the content but after the header of a chunk.

The `Chunk.read_chunk()` method can be invoked on the Chunk corresponding to the returned `ChunkIdentifier`.

This method seeks the input file to the beginning before searching.

Parameters

- **file** (*BinaryIO*) – finalfusion file
- **chunks** (*List[ChunkIdentifier]*) – List of Chunks to look for in the input file.

Returns **chunk_id** – The first ChunkIdentifier found in the file. None if none of the chunks could be found.

Return type Optional[*ChunkIdentifier*]

class ffp.io.**ChunkIdentifier**
Bases: `enum.IntEnum`

Known finalfusion Chunk types.

is_storage() → bool
Return if this Identifier belongs to a storage.

Returns **is_storage**

Return type bool

is_vocab() → bool
Return if this Identifier belongs to a vocab.

Returns **is_vocab**

Return type bool

class ffp.io.**TypeId**
Bases: `enum.IntEnum`

Known finalfusion data types.

exception ffp.io.**FinalfusionFormatError**
Bases: Exception

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

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

ffp.embeddings, 5
ffp.io, 34
ffp.metadata, 32
ffp.norms, 33

INDEX

Symbols

__getitem__() (*ffp.embeddings.Embeddings method*), 6
__getitem__() (*ffp.vocab.subword.FinalfusionBucketVocab method*), 20
__getitem__() (*ffp.vocab.vocab.Vocab method*), 27
__init__() (*ffp.embeddings.Embeddings method*), 6
__init__() (*ffp.storage.quantized.PQ method*), 14
__init__() (*ffp.storage.quantized.QuantizedArray method*), 12
__init__() (*ffp.vocab.simple_vocab.SimpleVocab method*), 16
__init__() (*ffp.vocab.subword.ExplicitVocab method*), 21
__init__() (*ffp.vocab.subword.FastTextVocab method*), 24
__init__() (*ffp.vocab.subword.FinalfusionBucketVocab method*), 18
__new__() (*ffp.storage.ndarray.NdArray static method*), 10

B

bucket_to_explicit() (*ffp.embeddings.Embeddings method*), 8
buckets_exp (*ffp.subwords.hash_indexers.FinalfusionHashIndexer attribute*), 29

C

Chunk (*class in ffp.io*), 34
chunk_identifier() (*ffp.io.Chunk static method*), 34
chunk_identifier() (*ffp.io.Header static method*), 35
chunk_identifier() (*ffp.metadata.Metadata static method*), 32
chunk_identifier() (*ffp.norms.Norms static method*), 33
chunk_identifier() (*ffp.storage.ndarray.NdArray static method*), 11
chunk_identifier() (*ffp.storage.quantized.QuantizedArray static method*), 13
chunk_identifier() (*ffp.vocab.simple_vocab.SimpleVocab static method*), 17
chunk_identifier() (*ffp.vocab.subword.ExplicitVocab static method*), 22
chunk_identifier() (*ffp.vocab.subword.FastTextVocab static method*), 26
chunk_identifier() (*ffp.vocab.subword.FinalfusionBucketVocab static method*), 20
chunk_ids() (*ffp.io.Header property*), 34
ChunkIdentifier (*class in ffp.io*), 35
chunks() (*ffp.embeddings.Embeddings method*), 8

E

embedding() (*ffp.embeddings.Embeddings method*), 6
embedding() (*ffp.storage.quantized.QuantizedArray method*), 12
embedding_with_norm() (*ffp.embeddings.Embeddings method*), 7
Embeddings (*class in ffp.embeddings*), 5
ExplicitIndexer (*class in ffp.subwords.explicit_indexer*), 30
ExplicitVocab (*class in ffp.vocab.subword*), 21

F

FastTextIndexer (*class in ffp.subwords.hash_indexers*), 30
FastTextVocab (*class in ffp.vocab.subword*), 24
ffp.embeddings
    module, 5
ffp.io
    module, 34
ffp.metadata
    module, 32
ffp.norms
    module, 33
FinalfusionBucketVocab (*class in ffp.vocab.subword*), 18
FinalfusionFormatError, 35

FinalfusionHashIndexer (*class in ffp.subwords.hash_indexers*), 29
find_chunk() (*in module ffp.io*), 35
from_corpus() (*ffp.vocab.simple_vocab.SimpleVocab static method*), 17
from_corpus() (*ffp.vocab.subword.ExplicitVocab static method*), 22
from_corpus() (*ffp.vocab.subword.FastTextVocab static method*), 25
from_corpus() (*ffp.vocab.subword.FinalfusionBucketVocab static method*), 19

H

Header (*class in ffp.io*), 34

I

idx() (*ffp.vocab.simple_vocab.SimpleVocab method*), 18
idx() (*ffp.vocab.subword.ExplicitVocab method*), 23
idx() (*ffp.vocab.subword.FinalfusionBucketVocab method*), 20
idx() (*ffp.vocab.subword.SubwordVocab method*), 27
idx() (*ffp.vocab.vocab.Vocab method*), 26
idx_bound (*ffp.subwords.explicit_indexer.ExplicitIndexer attribute*), 31
idx_bound (*ffp.subwords.hash_indexers.FinalfusionHashIndexer attribute*), 29
idx_bound() (*ffp.vocab.simple_vocab.SimpleVocab property*), 17
idx_bound() (*ffp.vocab.subword.ExplicitVocab property*), 23
idx_bound() (*ffp.vocab.subword.FinalfusionBucketVocab property*), 20
idx_bound() (*ffp.vocab.subword.SubwordVocab property*), 27
idx_bound() (*ffp.vocab.vocab.Vocab property*), 26
is_storage() (*ffp.io.ChunkIdentifier method*), 35
is_vocab() (*ffp.io.ChunkIdentifier method*), 35

L

load() (*ffp.storage.ndarray.NdArray class method*), 11
load() (*ffp.storage.quantized.QuantizedArray class method*), 13
load() (*ffp.storage.Storage class method*), 15
load_explicit_vocab() (*in module ffp.vocab.subword*), 24
load_fastText() (*in module ffp.embeddings*), 9
load_fasttext_vocab() (*in module ffp.vocab.subword*), 26
load_finalfusion() (*in module ffp.embeddings*), 9
load_finalfusion_bucket_vocab() (*in module ffp.vocab.subword*), 21
load_metadata() (*in module ffp.metadata*), 33
load_ndarray() (*in module ffp.storage.ndarray*), 11
load_norms() (*in module ffp.norms*), 33
load_quantized_array() (*in module ffp.storage.quantized*), 15
load_simple_vocab() (*in module ffp.vocab.simple_vocab*), 18
load_storage() (*in module ffp.storage*), 16
load_text() (*in module ffp.embeddings*), 9
load_textdims() (*in module ffp.embeddings*), 9
load_vocab() (*in module ffp.vocab*), 28
load_word2vec() (*in module ffp.embeddings*), 9

M

max_n (*ffp.subwords.explicit_indexer.ExplicitIndexer attribute*), 31
max_n (*ffp.subwords.hash_indexers.FastTextIndexer attribute*), 30
max_n (*ffp.subwords.hash_indexers.FinalfusionHashIndexer attribute*), 29
max_n() (*ffp.vocab.subword.ExplicitVocab property*), 23
max_n() (*ffp.vocab.subword.FinalfusionBucketVocab property*), 20
max_n() (*ffp.vocab.subword.SubwordVocab property*), 27
Metadata (*class in ffp.metadata*), 32
metadata() (*ffp.embeddings.Embeddings property*), 8
min_n (*ffp.subwords.explicit_indexer.ExplicitIndexer attribute*), 31
min_n (*ffp.subwords.hash_indexers.FastTextIndexer attribute*), 30
min_n (*ffp.subwords.hash_indexers.FinalfusionHashIndexer attribute*), 29
min_n() (*ffp.vocab.subword.ExplicitVocab property*), 23
min_n() (*ffp.vocab.subword.FinalfusionBucketVocab property*), 20
min_n() (*ffp.vocab.subword.SubwordVocab property*), 27
mmap_storage() (*ffp.storage.ndarray.NdArray static method*), 11
mmap_storage() (*ffp.storage.quantized.QuantizedArray static method*), 13
mmap_storage() (*ffp.storage.Storage static method*), 15
module
    ffp.embeddings, 5
    ffp.io, 34
    ffp.metadata, 32
    ffp.norms, 33

N

n_buckets (*ffp.subwords.hash_indexers.FastTextIndexer attribute*), 30

n_centroids() (*ffp.storage.quantized.PQ property*), 14
NdArray (*class in ffp.storage.ndarray*), 10
ngram_index (*ffp.subwords.explicit_indexer.ExplicitIndexer attribute*), 31
ngrams (*ffp.subwords.explicit_indexer.ExplicitIndexer attribute*), 31
Norms (*class in ffp.norms*), 33
norms() (*ffp.embeddings.Embeddings property*), 8

P

PQ (*class in ffp.storage.quantized*), 14
projection() (*ffp.storage.quantized.PQ property*), 14

Q

quantized_len() (*ffp.storage.quantized.QuantizedArray property*), 13
QuantizedArray (*class in ffp.storage.quantized*), 12
quantizer() (*ffp.storage.quantized.QuantizedArray property*), 13

R

read_chunk() (*ffp.io.Chunk static method*), 34
read_chunk() (*ffp.io.Header static method*), 35
read_chunk() (*ffp.metadata.Metadata static method*), 32
read_chunk() (*ffp.norms.Norms static method*), 33
read_chunk() (*ffp.storage.ndarray.NdArray static method*), 11
read_chunk() (*ffp.storage.quantized.QuantizedArray static method*), 13
read_chunk() (*ffp.vocab.simple_vocab.SimpleVocab static method*), 17
read_chunk() (*ffp.vocab.subword.ExplicitVocab static method*), 22
read_chunk() (*ffp.vocab.subword.FastTextVocab static method*), 25
read_chunk() (*ffp.vocab.subword.FinalfusionBucketVocab static method*), 19
reconstruct() (*ffp.storage.quantized.PQ method*), 14
reconstructed_len() (*ffp.storage.quantized.PQ property*), 14

S

shape() (*ffp.storage.ndarray.NdArray property*), 10
shape() (*ffp.storage.quantized.QuantizedArray property*), 12
shape() (*ffp.storage.Storage property*), 15
SimpleVocab (*class in ffp.vocab.simple_vocab*), 16
Storage (*class in ffp.storage*), 15
storage() (*ffp.embeddings.Embeddings property*), 8
subquantizers() (*ffp.storage.quantized.PQ property*), 14
subword_indexer() (*ffp.vocab.subword.ExplicitVocab property*), 22
subword_indexer() (*ffp.vocab.subword.FastTextVocab property*), 25
subword_indexer() (*ffp.vocab.subword.FinalfusionBucketVocab property*), 19
subword_indexer() (*ffp.vocab.subword.SubwordVocab property*), 27
subword_indices() (*ffp.subwords.explicit_indexer.ExplicitIndexer method*), 31
subword_indices() (*ffp.subwords.hash_indexers.FastTextIndexer method*), 30
subword_indices() (*ffp.subwords.hash_indexers.FinalfusionHashIndexer method*), 29
subword_indices() (*ffp.vocab.subword.ExplicitVocab method*), 23
subword_indices() (*ffp.vocab.subword.FinalfusionBucketVocab method*), 20
subword_indices() (*ffp.vocab.subword.SubwordVocab method*), 28
subwords() (*ffp.vocab.subword.ExplicitVocab method*), 24
subwords() (*ffp.vocab.subword.FinalfusionBucketVocab method*), 21
subwords() (*ffp.vocab.subword.SubwordVocab method*), 28
SubwordVocab (*class in ffp.vocab.subword*), 27

T

to_explicit() (*ffp.vocab.subword.FastTextVocab method*), 25
to_explicit() (*ffp.vocab.subword.FinalfusionBucketVocab method*), 19
TypeId (*class in ffp.io*), 35

V

Vocab (*class in ffp.vocab.vocab*), 26
vocab() (*ffp.embeddings.Embeddings property*), 8

W

word_index() (*ffp.vocab.simple_vocab.SimpleVocab property*), 17
word_index() (*ffp.vocab.subword.ExplicitVocab property*), 22

word_index() (*ffp.vocab.subword.FastTextVocab property*), 25
word_index() (*ffp.vocab.subword.FinalfusionBucketVocab property*), 19
word_index() (*ffp.vocab.vocab.Vocab property*), 26
word_ngrams() (*in module ffp.subwords.ngrams*), 32
words() (*ffp.vocab.simple_vocab.SimpleVocab property*), 17
words() (*ffp.vocab.subword.ExplicitVocab property*), 22
words() (*ffp.vocab.subword.FastTextVocab property*), 25
words() (*ffp.vocab.subword.FinalfusionBucketVocab property*), 19
words() (*ffp.vocab.vocab.Vocab property*), 26
write() (*ffp.embeddings.Embeddings method*), 9
write() (*ffp.io.Chunk method*), 34
write() (*ffp.storage.quantized.QuantizedArray method*), 14
write() (*ffp.vocab.subword.ExplicitVocab method*), 24
write() (*ffp.vocab.subword.FinalfusionBucketVocab method*), 21
write_chunk() (*ffp.io.Chunk method*), 34
write_chunk() (*ffp.io.Header method*), 35
write_chunk() (*ffp.metadata.Metadata method*), 33
write_chunk() (*ffp.norms.Norms method*), 33
write_chunk() (*ffp.storage.ndarray.NdArray method*), 11
write_chunk() (*ffp.storage.quantized.QuantizedArray method*), 13
write_chunk() (*ffp.vocab.simple_vocab.SimpleVocab method*), 17
write_chunk() (*ffp.vocab.subword.ExplicitVocab method*), 23
write_chunk() (*ffp.vocab.subword.FastTextVocab method*), 26
write_chunk() (*ffp.vocab.subword.FinalfusionBucketVocab method*), 19