# Package 'tok'

July 6, 2024

**Title** Fast Text Tokenization

**Version** 0.1.3

**Description** Interfaces with the 'Hugging Face' tokenizers library to provide implementations of today's most used tokenizers such as the 'Byte-Pair Encoding' algorithm <https://huggingface.co/docs/tokenizers/index>. It's extremely fast for both training new vocabularies and tokenizing texts.

**License** MIT + file LICENSE

**SystemRequirements** Rust tool chain w/ cargo, libclang/llvm-config

**Encoding** UTF-8

**RoxygenNote** 7.3.1

**Depends** R (>= 4.2.0)

**Imports** R6, cli

**Suggests** rmarkdown, testthat (>= 3.0.0), hfhub (>= 0.1.1), withr

**Config/testthat/edition** 3

**URL** https://github.com/mlverse/tok

**BugReports** https://github.com/mlverse/tok/issues

**Config/rextendr/version** 0.3.1

**NeedsCompilation** yes

**Author** Daniel Falbel [aut, cre],
Posit [cph]

**Maintainer** Daniel Falbel <daniel@posit.co>

**Repository** CRAN

**Date/Publication** 2024-07-06 13:40:02 UTC

## Contents

---

`tok-package` *tok: Fast Text Tokenization*

---

## Description

Interfaces with the 'Hugging Face' tokenizers library to provide implementations of today's most used tokenizers such as the 'Byte-Pair Encoding' algorithm https://huggingface.co/docs/tokenizers/index. It's extremely fast for both training new vocabularies and tokenizing texts.

## Author(s)

**Maintainer**: Daniel Falbel <daniel@posit.co>

Other contributors:

- Posit [copyright holder]

## See Also

Useful links:

- https://github.com/mlverse/tok
- Report bugs at https://github.com/mlverse/tok/issues

`decoder_byte_level` *Byte level decoder*

## Description

Byte level decoder

Byte level decoder

## Details

This decoder is to be used with the pre_tokenizer_byte_level.

## Super class

`tok::tok_decoder` -> `tok_decoder_byte_level`

## Methods

### Public methods:

- `decoder_byte_level$new()`
- `decoder_byte_level$clone()`

**Method** `new()`**:** Initializes a byte level decoder

*Usage:*

```
decoder_byte_level$new()
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
decoder_byte_level$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other decoders: `tok_decoder`

`encoding` *Encoding*

## Description

Represents the output of a tokenizer.

## Value

An encoding object containing encoding information such as attention masks and token ids.

## Public fields

`.encoding` The underlying implementation pointer.

## Active bindings

`ids` The IDs are the main input to a Language Model. They are the token indices, the numerical representations that a LM understands.

`attention_mask` The attention mask used as input for transformers models.

## Methods

### Public methods:

- `encoding$new()`
- `encoding$clone()`

**Method** `new()`**:** Initializes an encoding object (Not to use directly)

*Usage:*

```
encoding$new(encoding)
```

*Arguments:*

`encoding` an encoding implementation object

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
encoding$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
withr::with_envvar(c(HUGGINGFACE_HUB_CACHE = tempdir()), {
try({
tok <- tokenizer$from_pretrained("gpt2")
encoding <- tok$encode("Hello world")
encoding
})
})
```

model_bpe *BPE model*

## Description

BPE model

BPE model

## Super class

tok::tok_model -> tok_model_bpe

## Methods

### Public methods:

- model_bpe$new()
- model_bpe$clone()

**Method** new(): Initializes a BPE model An implementation of the BPE (Byte-Pair Encoding) algorithm

*Usage:*

```
model_bpe$new(
  vocab = NULL,
  merges = NULL,
  cache_capacity = NULL,
  dropout = NULL,
  unk_token = NULL,
  continuing_subword_prefix = NULL,
  end_of_word_suffix = NULL,
  fuse_unk = NULL,
  byte_fallback = FALSE
)
```

*Arguments:*

vocab A named integer vector of string keys and their corresponding ids. Default: NULL

merges A list of pairs of tokens ([character, character]). Default: NULL.

cache_capacity The number of words that the BPE cache can contain. The cache speeds up the process by storing merge operation results. Default: NULL.

dropout A float between 0 and 1 representing the BPE dropout to use. Default: NULL

unk_token The unknown token to be used by the model. Default: 'NULL"'.

continuing_subword_prefix The prefix to attach to subword units that don't represent the beginning of a word. Default: NULL

end_of_word_suffix The suffix to attach to subword units that represent the end of a word. Default: NULL

fuse_unk Whether to fuse any subsequent unknown tokens into a single one. Default: NULL.

byte_fallback Whether to use the spm byte-fallback trick. Default: FALSE.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
model_bpe$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## See Also

Other model: model_unigram, model_wordpiece, tok_model

---

`model_unigram` *An implementation of the Unigram algorithm*

---

## Description

An implementation of the Unigram algorithm

An implementation of the Unigram algorithm

## Super class

tok::tok_model -> tok_model_unigram

## Methods

### Public methods:

- model_unigram$new()
- model_unigram$clone()

**Method** `new()`**:** Constructor for Unigram Model

*Usage:*

```
model_unigram$new(vocab = NULL, unk_id = NULL, byte_fallback = FALSE)
```

*Arguments:*

vocab A dictionary of string keys and their corresponding relative score. Default: NULL.

unk_id The unknown token id to be used by the model. Default: NULL.

byte_fallback Whether to use byte-fallback trick. Default: FALSE.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
model_unigram$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## See Also

Other model: model_bpe, model_wordpiece, tok_model

model_wordpiece *An implementation of the WordPiece algorithm*

## Description

An implementation of the WordPiece algorithm

An implementation of the WordPiece algorithm

## Super class

tok::tok_model -> tok_model_wordpiece

## Methods

### Public methods:

- model_wordpiece$new()
- model_wordpiece$clone()

**Method** new(): Constructor for the wordpiece tokenizer

*Usage:*

```
model_wordpiece$new(
  vocab = NULL,
  unk_token = NULL,
  max_input_chars_per_word = NULL
)
```

*Arguments:*

vocab A dictionary of string keys and their corresponding ids. Default: NULL.

unk_token The unknown token to be used by the model. Default: NULL.

max_input_chars_per_word The maximum number of characters to allow in a single word. Default: NULL.

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

```
model_wordpiece$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## See Also

Other model: model_bpe, model_unigram, tok_model

## normalizer_nfc *NFC normalizer*

### Description

NFC normalizer

NFC normalizer

### Super class

`tok::tok_normalizer` -> `tok_normalizer_nfc`

### Methods

#### Public methods:

- `normalizer_nfc$new()`
- `normalizer_nfc$clone()`

**Method** `new()`**:** Initializes the NFC normalizer

*Usage:*

```
normalizer_nfc$new()
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
normalizer_nfc$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

### See Also

Other normalizers: `normalizer_nfkc`, `tok_normalizer`

## normalizer_nfkc *NFKC normalizer*

### Description

NFKC normalizer

NFKC normalizer

### Super class

`tok::tok_normalizer` -> `tok_normalizer_nfc`

## Methods

### Public methods:

- normalizer_nfkc$new()
- normalizer_nfkc$clone()

**Method** `new()`**:** Initializes the NFKC normalizer

*Usage:*

```
normalizer_nfkc$new()
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
normalizer_nfkc$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other normalizers: normalizer_nfc, tok_normalizer

---

`pre_tokenizer` *Generic class for tokenizers*

---

## Description

Generic class for tokenizers

Generic class for tokenizers

## Public fields

`.pre_tokenizer` Internal pointer to tokenizer object

## Methods

### Public methods:

- pre_tokenizer$new()
- pre_tokenizer$clone()

**Method** `new()`**:** Initializes a tokenizer

*Usage:*

```
pre_tokenizer$new(pre_tokenizer)
```

*Arguments:*

`pre_tokenizer` a raw pointer to a tokenizer

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
pre_tokenizer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other pre_tokenizer: `pre_tokenizer_byte_level`, `pre_tokenizer_whitespace`

---

`pre_tokenizer_byte_level` *Byte level pre tokenizer*

---

## Description

Byte level pre tokenizer

Byte level pre tokenizer

## Details

This pre-tokenizer takes care of replacing all bytes of the given string with a corresponding representation, as well as splitting into words.

## Super class

`tok::tok_pre_tokenizer` -> `tok_pre_tokenizer_whitespace`

## Methods

### Public methods:

- `pre_tokenizer_byte_level$new()`
- `pre_tokenizer_byte_level$clone()`

**Method** `new()`**:** Initializes the bytelevel tokenizer

*Usage:*

```
pre_tokenizer_byte_level$new(add_prefix_space = TRUE, use_regex = TRUE)
```

*Arguments:*

`add_prefix_space` Whether to add a space to the first word

`use_regex` Set this to False to prevent this pre_tokenizer from using the GPT2 specific regexp for spliting on whitespace.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
pre_tokenizer_byte_level$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other pre_tokenizer: pre_tokenizer, pre_tokenizer_whitespace

---

`pre_tokenizer_whitespace` *This pre-tokenizer simply splits using the following regex:* `\w+|[^\w\s]+`

---

## Description

This pre-tokenizer simply splits using the following regex: `\w+|[^\w\s]+`

This pre-tokenizer simply splits using the following regex: `\w+|[^\w\s]+`

## Super class

`tok::tok_pre_tokenizer` -> `tok_pre_tokenizer_whitespace`

## Methods

### Public methods:

- `pre_tokenizer_whitespace$new()`
- `pre_tokenizer_whitespace$clone()`

**Method** `new()`**:** Initializes the whistespace tokenizer

*Usage:*

```
pre_tokenizer_whitespace$new()
```

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
pre_tokenizer_whitespace$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other pre_tokenizer: pre_tokenizer, pre_tokenizer_byte_level

`processor_byte_level` *Byte Level post processor*

## Description

Byte Level post processor

Byte Level post processor

## Details

This post-processor takes care of trimming the offsets. By default, the ByteLevel BPE might include whitespaces in the produced tokens. If you don't want the offsets to include these whitespaces, then this PostProcessor must be used.

## Super class

`tok::tok_processor` -> `tok_processor_byte_level`

## Methods

### Public methods:

- `processor_byte_level$new()`
- `processor_byte_level$clone()`

**Method** `new()`**:** Initializes the byte level post processor

*Usage:*

```
processor_byte_level$new(trim_offsets = TRUE)
```

*Arguments:*

`trim_offsets` Whether to trim the whitespaces from the produced offsets.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
processor_byte_level$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other processors: `tok_processor`

`tokenizer` *Tokenizer*

## Description

A Tokenizer works as a pipeline. It processes some raw text as input and outputs an encoding.

## Value

A tokenizer that can be used for encoding character strings or decoding integers.

## Public fields

`.tokenizer` (unsafe usage) Lower level pointer to tokenizer

## Active bindings

`pre_tokenizer` instance of the pre-tokenizer

`normalizer` Gets the normalizer instance

`post_processor` Gets the post processor used by tokenizer

`decoder` Gets and sets the decoder

`padding` Gets padding configuration

`truncation` Gets truncation configuration

## Methods

### Public methods:

- `tokenizer$new()`
- `tokenizer$encode()`
- `tokenizer$decode()`
- `tokenizer$encode_batch()`
- `tokenizer$decode_batch()`
- `tokenizer$from_file()`
- `tokenizer$from_pretrained()`
- `tokenizer$train()`
- `tokenizer$train_from_memory()`
- `tokenizer$save()`
- `tokenizer$enable_padding()`
- `tokenizer$no_padding()`
- `tokenizer$enable_truncation()`
- `tokenizer$no_truncation()`
- `tokenizer$get_vocab_size()`
- `tokenizer$clone()`

**Method** `new()`**:** Initializes a tokenizer

*Usage:*

```
tokenizer$new(tokenizer)
```

*Arguments:*

`tokenizer` Will be cloned to initialize a new tokenizer

**Method** `encode()`**:** Encode the given sequence and pair. This method can process raw text sequences as well as already pre-tokenized sequences.

*Usage:*

```
tokenizer$encode(
  sequence,
  pair = NULL,
  is_pretokenized = FALSE,
  add_special_tokens = TRUE
)
```

*Arguments:*

`sequence` The main input sequence we want to encode. This sequence can be either raw text or pre-tokenized, according to the is_pretokenized argument

`pair` An optional input sequence. The expected format is the same that for sequence.

`is_pretokenized` Whether the input is already pre-tokenized

`add_special_tokens` Whether to add the special tokens

**Method** `decode()`**:** Decode the given list of ids back to a string

*Usage:*

```
tokenizer$decode(ids, skip_special_tokens = TRUE)
```

*Arguments:*

`ids` The list of ids that we want to decode

`skip_special_tokens` Whether the special tokens should be removed from the decoded string

**Method** `encode_batch()`**:** Encodes a batch of sequences. Returns a list of encodings.

*Usage:*

```
tokenizer$encode_batch(
  input,
  is_pretokenized = FALSE,
  add_special_tokens = TRUE
)
```

*Arguments:*

`input` A list of single sequences or pair sequences to encode. Each sequence can be either raw text or pre-tokenized, according to the is_pretokenized argument.

`is_pretokenized` Whether the input is already pre-tokenized

`add_special_tokens` Whether to add the special tokens

**Method** `decode_batch()`**:** Decode a batch of ids back to their corresponding string

*Usage:*

```
tokenizer$decode_batch(sequences, skip_special_tokens = TRUE)
```

*Arguments:*

`sequences` The batch of sequences we want to decode

`skip_special_tokens` Whether the special tokens should be removed from the decoded strings

**Method** `from_file()`**:** Creates a tokenizer from the path of a serialized tokenizer. This is a static method and should be called instead of `$new` when initializing the tokenizer.

*Usage:*

```
tokenizer$from_file(path)
```

*Arguments:*

`path` Path to tokenizer.json file

**Method** `from_pretrained()`**:** Instantiate a new Tokenizer from an existing file on the Hugging Face Hub.

*Usage:*

```
tokenizer$from_pretrained(identifier, revision = "main", auth_token = NULL)
```

*Arguments:*

`identifier` The identifier of a Model on the Hugging Face Hub, that contains a tokenizer.json file

`revision` A branch or commit id

`auth_token` An optional auth token used to access private repositories on the Hugging Face Hub

**Method** `train()`**:** Train the Tokenizer using the given files. Reads the files line by line, while keeping all the whitespace, even new lines.

*Usage:*

```
tokenizer$train(files, trainer)
```

*Arguments:*

`files` character vector of file paths.

`trainer` an instance of a trainer object, specific to that tokenizer type.

**Method** `train_from_memory()`**:** Train the tokenizer on a chracter vector of texts

*Usage:*

```
tokenizer$train_from_memory(texts, trainer)
```

*Arguments:*

`texts` a character vector of texts.

`trainer` an instance of a trainer object, specific to that tokenizer type.

**Method** `save()`**:** Saves the tokenizer to a json file

*Usage:*

```
tokenizer$save(path, pretty = TRUE)
```

*Arguments:*

`path` A path to a file in which to save the serialized tokenizer.

`pretty` Whether the JSON file should be pretty formatted.

**Method** `enable_padding()`**:** Enables padding for the tokenizer

*Usage:*

```
tokenizer$enable_padding(
  direction = "right",
  pad_id = 0L,
  pad_type_id = 0L,
  pad_token = "[PAD]",
  length = NULL,
  pad_to_multiple_of = NULL
)
```

*Arguments:*

`direction` (str, optional, defaults to right) — The direction in which to pad. Can be either `'right'` or `'left'`

`pad_id` (int, defaults to 0) — The id to be used when padding

`pad_type_id` (int, defaults to 0) — The type id to be used when padding

`pad_token` (str, defaults to `'[PAD]'`) — The pad token to be used when padding

`length` (int, optional) — If specified, the length at which to pad. If not specified we pad using the size of the longest sequence in a batch.

`pad_to_multiple_of` (int, optional) — If specified, the padding length should always snap to the next multiple of the given value. For example if we were going to pad with a length of 250 but `pad_to_multiple_of=8` then we will pad to 256.

**Method** `no_padding()`**:** Disables padding

*Usage:*

```
tokenizer$no_padding()
```

**Method** `enable_truncation()`**:** Enables truncation on the tokenizer

*Usage:*

```
tokenizer$enable_truncation(
  max_length,
  stride = 0,
  strategy = "longest_first",
  direction = "right"
)
```

*Arguments:*

`max_length` The maximum length at which to truncate.

`stride` The length of the previous first sequence to be included in the overflowing sequence. Default: `0`.

`strategy` The strategy used for truncation. Can be one of: "longest_first", "only_first", or "only_second". Default: "longest_first".

`direction` The truncation direction. Default: "right".

**Method** `no_truncation()`**:** Disables truncation

*Usage:*

```
tokenizer$no_truncation()
```

**Method** `get_vocab_size()`**:** Gets the vocabulary size

*Usage:*

```
tokenizer$get_vocab_size(with_added_tokens = TRUE)
```

*Arguments:*

`with_added_tokens` Wether to count added tokens

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
tokenizer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
withr::with_envvar(c(HUGGINGFACE_HUB_CACHE = tempdir()), {
try({
tok <- tokenizer$from_pretrained("gpt2")
tok$encode("Hello world")$ids
})
})
```

---

`tok_decoder` *Generic class for decoders*

---

## Description

Generic class for decoders

Generic class for decoders

## Public fields

`.decoder` The raw pointer to the decoder

## Methods

### Public methods:

- `tok_decoder$new()`
- `tok_decoder$clone()`

**Method** `new()`**:** Initializes a decoder

*Usage:*

```
tok_decoder$new(decoder)
```

*Arguments:*

`decoder` a raw decoder pointer

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
tok_decoder$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other decoders: `decoder_byte_level`

---

`tok_model` *Generic class for tokenization models*

---

## Description

Generic class for tokenization models

Generic class for tokenization models

## Public fields

`.model` stores the pointer to the model. internal

## Methods

### Public methods:

- `tok_model$new()`
- `tok_model$clone()`

**Method** `new()`**:** Initializes a genric abstract tokenizer model

*Usage:*

```
tok_model$new(model)
```

*Arguments:*

`model` Pointer to a tokenization model

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
tok_model$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other model: `model_bpe`, `model_unigram`, `model_wordpiece`

tok_normalizer *Generic class for normalizers*

## Description

Generic class for normalizers

Generic class for normalizers

## Public fields

`.normalizer` Internal pointer to normalizer object

## Methods

### Public methods:

- `tok_normalizer$new()`
- `tok_normalizer$clone()`

**Method** `new()`**:** Initializes a tokenizer

*Usage:*

```
tok_normalizer$new(normalizer)
```

*Arguments:*

`normalizer` a raw pointer to a tokenizer

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
tok_normalizer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other normalizers: `normalizer_nfc`, `normalizer_nfkc`

tok_processor *Generic class for processors*

## Description

Generic class for processors

Generic class for processors

## Public fields

`.processor` Internal pointer to processor object

## Methods

### Public methods:

- `tok_processor$new()`
- `tok_processor$clone()`

**Method** `new()`**:** Initializes a tokenizer

*Usage:*

```
tok_processor$new(processor)
```

*Arguments:*

`processor` a raw pointer to a processor

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
tok_processor$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other processors: `processor_byte_level`

tok_trainer *Generic training class*

## Description

Generic training class

Generic training class

## Public fields

`.trainer` a pointer to a raw trainer

## Methods

### Public methods:

- `tok_trainer$new()`
- `tok_trainer$clone()`

**Method** `new()`**:** Initializes a generic trainer from a raw trainer

*Usage:*

```
tok_trainer$new(trainer)
```

*Arguments:*

`trainer` raw trainer (internal)

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
tok_trainer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other trainer: `trainer_bpe`, `trainer_unigram`, `trainer_wordpiece`

trainer_bpe *BPE trainer*

## Description

BPE trainer

BPE trainer

## Super class

tok::tok_trainer -> tok_trainer_bpe

## Methods

### Public methods:

- trainer_bpe$new()
- trainer_bpe$clone()

**Method** new(): Constrcutor for the BPE trainer

*Usage:*

```
trainer_bpe$new(
  vocab_size = NULL,
  min_frequency = NULL,
  show_progress = NULL,
  special_tokens = NULL,
  limit_alphabet = NULL,
  initial_alphabet = NULL,
  continuing_subword_prefix = NULL,
  end_of_word_suffix = NULL,
  max_token_length = NULL
)
```

*Arguments:*

vocab_size The size of the final vocabulary, including all tokens and alphabet. Default: NULL.

min_frequency The minimum frequency a pair should have in order to be merged. Default: NULL.

show_progress Whether to show progress bars while training. Default: TRUE.

special_tokens A list of special tokens the model should be aware of. Default: NULL.

limit_alphabet The maximum number of different characters to keep in the alphabet. Default: NULL.

initial_alphabet A list of characters to include in the initial alphabet, even if not seen in the training dataset. Default: NULL.

continuing_subword_prefix A prefix to be used for every subword that is not a beginning-of-word. Default: NULL.

end_of_word_suffix A suffix to be used for every subword that is an end-of-word. Default: NULL.

`max_token_length` Prevents creating tokens longer than the specified size. Default: `NULL`.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
trainer_bpe$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other trainer: `tok_trainer`, `trainer_unigram`, `trainer_wordpiece`

---

`trainer_unigram` *Unigram tokenizer trainer*

---

## Description

Unigram tokenizer trainer

Unigram tokenizer trainer

## Super class

`tok::tok_trainer` -> `tok_trainer_unigram`

## Methods

### Public methods:

- `trainer_unigram$new()`
- `trainer_unigram$clone()`

**Method** `new()`**:** Constructor for the Unigram tokenizer

*Usage:*

```
trainer_unigram$new(
  vocab_size = 8000,
  show_progress = TRUE,
  special_tokens = NULL,
  shrinking_factor = 0.75,
  unk_token = NULL,
  max_piece_length = 16,
  n_sub_iterations = 2
)
```

*Arguments:*

`vocab_size` The size of the final vocabulary, including all tokens and alphabet.

`show_progress` Whether to show progress bars while training.

`special_tokens` A list of special tokens the model should be aware of.

`shrinking_factor` The shrinking factor used at each step of training to prune the vocabulary.

`unk_token` The token used for out-of-vocabulary tokens.

`max_piece_length` The maximum length of a given token.

`n_sub_iterations` The number of iterations of the EM algorithm to perform before pruning the vocabulary.

`initial_alphabet` A list of characters to include in the initial alphabet, even if not seen in the training dataset. If the strings contain more than one character, only the first one is kept.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
trainer_unigram$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other trainer: `tok_trainer`, `trainer_bpe`, `trainer_wordpiece`

---

`trainer_wordpiece` *WordPiece tokenizer trainer*

---

## Description

WordPiece tokenizer trainer

WordPiece tokenizer trainer

## Super class

`tok::tok_trainer` -> `tok_trainer_wordpiece`

## Methods

### Public methods:

- `trainer_wordpiece$new()`
- `trainer_wordpiece$clone()`

**Method** `new()`**:** Constructor for the WordPiece tokenizer trainer

*Usage:*

```
trainer_wordpiece$new(
  vocab_size = 30000,
  min_frequency = 0,
  show_progress = FALSE,
  special_tokens = NULL,
  limit_alphabet = NULL,
  initial_alphabet = NULL,
```

```
  continuing_subword_prefix = "##",
  end_of_word_suffix = NULL
)
```

*Arguments:*

`vocab_size` The size of the final vocabulary, including all tokens and alphabet. Default: `NULL`.

`min_frequency` The minimum frequency a pair should have in order to be merged. Default: `NULL`.

`show_progress` Whether to show progress bars while training. Default: `TRUE`.

`special_tokens` A list of special tokens the model should be aware of. Default: `NULL`.

`limit_alphabet` The maximum number of different characters to keep in the alphabet. Default: `NULL`.

`initial_alphabet` A list of characters to include in the initial alphabet, even if not seen in the training dataset. If the strings contain more than one character, only the first one is kept. Default: `NULL`.

`continuing_subword_prefix` A prefix to be used for every subword that is not a beginning-of-word. Default: `NULL`.

`end_of_word_suffix` A suffix to be used for every subword that is an end-of-word. Default: `NULL`.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
trainer_wordpiece$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other trainer: `tok_trainer`, `trainer_bpe`, `trainer_unigram`

# Index