PLSA

Release 0.1

Sep 18, 2019

Contents:

CHAPTER 1

plsa.preprocessors module

Preprocessors for documents and words.

These preprocessors come in three flavours (functions, closures that return functions, and classes defining callable objects). The choice for the respective flavour is motivated by the complexity of the preprocessor. If it doesn't need any parameters, a simple function will do. If it is simple, does not need to be manipulated interactively, but needs some parameter(s), then a closure is fine. If it would be convenient to alter parameters of the preprocessor interactively, then a class is a good choice.

Preprocessors act either on an entire document string or, after splitting documents into individual words, on an iterable over the words contained in a single document. Therefore, they cannot be combined in arbitrary order but care must be taken to ensure that the return value of one matches the call signature of the next.

plsa.preprocessors.**remove_non_ascii**(*doc: str*) → str
Removes non-ASCII characters (i.e., with unicode > 127) from a string.

Parameters **doc** (*str*) – A document given as a single string.

Returns The document as a single string with all characters of unicode > 127 removed.

Return type str

plsa.preprocessors.**to_lower**(*doc: str*) → str
Converts a string to all-lowercase.

Parameters **doc** (*str*) – A document given as a single string.

Returns The document as a single string with all characters converted to lowercase.

Return type str

plsa.preprocessors.**remove_numbers**(*doc: str*) → str
Removes digit/number characters from a string.

Parameters **doc** (*str*) – A document given as a single string.

Returns The document as a single string with all number/digit characters removed.

Return type str

plsa.preprocessors.**remove_tags**(*exclude_regex: str*) → Callable[[str], str]
Returns callable that removes matches to the given regular expression.

Parameters **exclude_regex** (*str*) – A regular expression specifying specific patterns to remove from a document.

Returns A callable that removes patterns matching the given regular expression from a string.

Return type function

plsa.preprocessors.**remove_punctuation**(*punctuation: Iterable[str]*) → Callable[[str], str]
Returns callable that removes punctuation characters from a string."

Parameters **punctuation** (*iterable of str*) – An iterable over single-character strings specifying punctuation characters to remove from a document.

Returns A callable that removes the given punctuation characters from a string.

Return type function

plsa.preprocessors.**tokenize**(*doc: str*) → Tuple[str, ...]
Splits a string into individual words.

Parameters **doc** (*str*) – A document given as a single string.

Returns The document as tuple of individual words.

Return type tuple of str

class plsa.preprocessors.**RemoveStopwords**(*stopwords: Union[str, Iterable[str]]*)
Bases: object

Instantiate callable objects that remove stopwords from a document.

Parameters **stopwords** (*str or iterable of str*) – Stopword(s) to remove from a document given as an iterable over words.

Examples

```
>>> from plsa.preprocessors import RemoveStopwords
>>> remover = RemoveStopwords('is')
>>> remover.words
('is',)
```

```
>>> remover.words = 'the', 'are'
>>> remover.words
('the', 'are')
```

```
>>> remover += 'is', 'we'
>>> remover.words
('is', 'we', 'the', 'are')
```

```
>>> new_instance = remover + 'do'
>>> new_instance.words
('are', 'we', 'is', 'do', 'the')
```

words
The current stopwords.

class plsa.preprocessors.**LemmatizeWords**(**incl_pos*)

Bases: object

Instantiate callable objects that find the root form of words.

Parameters ***inc_pos** (*str*) – One or more positional tag(s) indicating the type(s) of words to retain and to find the root form of. Must be one of 'JJ' (adjectives), 'NN' (nouns), 'VB' (verbs), or 'RB' (adverbs).

Raises KeyError – If the given positional tags are not among the list of allowed ones.

Examples

```
>>> from plsa.preprocessors import LemmatizeWords
>>> lemmatizer = LemmatizeWords('VB')
>>> lemmatizer.types
('VB',)
```

```
>>> lemmatizer.types = 'jj', 'nn'
>>> lemmatizer.types
('JJ', 'NN')
```

```
>>> lemmatizer += 'VB', 'NN'
>>> lemmatizer.types
('JJ', 'NN', 'VB')
```

```
>>> new_instance = lemmatizer + 'RB'
>>> new_instance.types
('JJ', 'RB', 'NN', 'VB')
```

types

The current type(s) of words to retain.

plsa.preprocessors.**remove_short_words**(*min_word_len: int*) → Callable[[Iterable[str]], Tuple[str, ...]]

Returns a callable that removes short words from an iterable of strings.

Parameters **min_word_len** (*int*) – Minimum number of characters in a word for it to be retained.

Returns A callable that removes words shorter than the given threshold from an iterable over strings.

Return type function

CHAPTER 2

plsa.pipeline module

class plsa.pipeline.**Pipeline**(**preprocessors*)

Bases: object

Encapsulates and applies multiple document preprocessors.

Each preprocessor is assumed to be a callable that takes a single document as input and produces a single document as output. Importantly, each document fed to the first preprocessor in the chain is delivered as a single string, while the last preprocessor is required to return it as an iterable over strings with each element representing one word of that document.

Other than that, preprocessors can be combined in any which way, provided that the return value of one matches the call signature of the next. The order in which they are applied is the order in which they are specified, *i.e.*, from left to right.

Parameters ***preprocessors** (*callable*) – Function(s) or other callable object(s) that each take a single document as input and produce a (processed) document as output.

See also:

plsa.preprocessors

process(*doc: str*) → Tuple[str, ...]

Applies a chain of one or more preprocessors to a document.

Parameters **doc** (*str*) – A text document given as a single string.

Returns Each element represents one word of the document.

Return type tuple of str

CHAPTER 3

plsa.corpus module

class plsa.corpus.**Corpus**(*corpus: Iterable[str], pipeline: plsa.pipeline.Pipeline*)

Bases: object

Processes raw document collections and provides numeric representations.

Parameters

- **corpus** (*iterable of str*) – An iterable over documents given as a single string each.
- **pipeline** (Pipeline) – The preprocessing pipeline.

See also:

plsa.pipeline

classmethod from_csv(*path: str*, *pipeline: plsa.pipeline.Pipeline*, *col: int = -1*, *encoding: str = 'latin_1'*, *max_docs: int = 1000*, ***kwargs*) → plsa.corpus.Corpus

Instantiate a corpus from documents in a column of a CSV file.

Parameters

- **path** (*str*) – Full path (incl. file name) to a CSV file with one column containing documents.
- **pipeline** – The preprocessing pipeline.
- **col** (*int*) – Which column contains the documents. Numbering starts with 0 for the first column. Negative numbers count back from the last column (*e.g.*, -1 for last, -2 just before the last, *etc.*).
- **encoding** (*str*) – A valid python encoding used to read the documents.
- **max_docs** (*int*) – The maximum number of documents to read from file.
- ****kwargs** – Keyword arguments are passed on to Python's own csv.reader function.

Raises StopIteration – If you do not have at least two lines in your CSV file.

Notes

If you set a `col` to a value outside the range present in the CSV file, it will be silently reset to the first or last column, depending on which side you exceed the permitted range.

A list of available encodings can be found at https://docs.python.org/3/library/codecs.html

Formatting parameters for the Python's `csv.reader` can be found at https://docs.python.org/3/library/csv.html#csv-fmt-params

classmethod from_xml(*directory: str*, *pipeline: plsa.pipeline.Pipeline*, *tag: str = 'post'*, *encoding: str = 'latin_1'*, *max_files: int = 100*) → plsa.corpus.Corpus

Instantiate a corpus from elements of XML files in a directory.

Parameters

- **directory** (*str*) – Path to the directory with the XML files.
- **pipeline** (*Pipeline*) – The preprocessing pipeline.
- **tag** – The XML tag that opens (<...>) and closes (</...>) the elements containing documents.
- **encoding** – A valid python encoding used to read the documents.
- **max_files** – The maximum number of XML files to read.

Notes

A list of available encodings can be found at https://docs.python.org/3/library/codecs.html

get_doc(*tf_idf: bool*) → numpy.ndarray

The marginal probability that any word comes from a given document.

This probability *p(d)* is obtained by summing the joint document- word probability *p(d, w)* over all words.

Parameters **tf_idf** (*bool*) – Whether to marginalize the term-frequency inverse-document-frequency or just the term-frequency matrix.

Returns The document probability *p(d)*.

Return type ndarray

get_doc_given_word(*tf_idf: bool*) → numpy.ndarray

The conditional probability of a particular word in a given document.

This probability *p(d|w)* is obtained by dividing the joint document- word probability *p(d, w)* by the marginal word probability *p(w)*.

Parameters **tf_idf** (*bool*) – Whether to base the conditional probability on the term-frequency inverse-document-frequency or just the term-frequency matrix.

Returns The conditional word probability *p(d|w)*.

Return type ndarray

get_doc_word(*tf_idf: bool*) → numpy.ndarray

The normalized document-word counts matrix.

Also referred to as the *term-frequency* matrix. Because words (or *terms*) that occur in the majority of documents are the least helpful in discriminating types of documents, each column of this matrix can be multiplied by the logarithm of the total number of documents divided by the number of documents containing the given word. The result is then referred to as the *term-frequency inverse-document-frequency* or *TF-IDF* matrix.

Either way, the returned matrix is always *normalized* such that it can be interpreted as the joint document-word probability *p(d, w)*.

Parameters **tf_idf** (*bool*) – Whether to return the term-frequency inverse-document-frequency or just the term-frequency matrix.

Returns The normalized document (rows) - word (columns) matrix, either as pure counts (if `tf_idf = False`) or weighted by the inverse document frequency (if `tf_idf` is `False`).

Return type ndarray

get_word(*tf_idf: bool*) → numpy.ndarray

The marginal probability of a particular word.

This probability *p(w)* is obtained by summing the joint document- word probability *p(d, w)* over all documents.

Parameters **tf_idf** (*bool*) – Whether to marginalize the term-frequency inverse-document-frequency or just the term-frequency matrix.

Returns The word probability *p(w)*.

Return type ndarray

idf

Logarithm of inverse fraction of documents each word occurs in.

index

Mapping from actual word to numeric word index.

n_docs

The number of non-empty documents.

n_occurrences

Total number of times any word occurred in any document.

n_words

The number of unique words retained after preprocessing.

pipeline

The pipeline of preprocessors for each document.

raw

The raw documents as they were read from the source.

vocabulary

Mapping from numeric word index to actual word.

CHAPTER 4

plsa.algorithms package

4.1 Submodules

4.1.1 plsa.algorithms.plsa module

class plsa.algorithms.plsa.**PLSA**(*corpus: plsa.corpus.Corpus*, *n_topics: int*, *tf_idf: bool = True*)

Implements probabilistic latent semantic analysis (PLSA).

At its core lies the assumption that the normalized document-word (or term-frequency) matrix *p(d, w)*, weighted with the inverse document frequency or not, can be factorized as:

$$p(d, w) \approx \sum_t \tilde{p}(d|t)\tilde{p}(w|t)\tilde{p}(t)$$

Parameters

- **corpus** (Corpus) – The corpus of preprocessed and numerically represented documents.
- **n_topics** (*int*) – The number of latent topics to identify.
- **tf_idf** (*bool*) – Whether to use the term-frequency inverse-document-frequency or just the term-frequency matrix as joint probability *p(d, w)* of documents and words.

Raises ValueError – If the number of topics is < 2 or the number of both, words and documents, in the corpus isn't greater than the number of topics.

Notes

The implementation follows algorithm 15.2 in Barber's book[1] to the letter. What is not said there is that, in order to update the conditional probability *p(t|d, w)* of a certain topic given a certain word in a certain document, one first needs to find the joint probability of all random variables as

$$\tilde{p}(t, d, w) = \tilde{p}(d|t)\tilde{p}(w|t)\tilde{p}(t)$$

and then divide by the marginal $\tilde{p}(d, w)$.

[1] "Bayesian Reasoning and Machine Learning", David Barber (Cambridge Press, 2012).

References

best_of (*n_runs: int = 3*, ***kwargs*) → plsa.algorithms.result.PlsaResult
Finds the best PLSA model among the specified number of runs.

As with any iterative algorithm, also the probabilities in PSLA need to be (randomly) initialized prior to the first iteration step. Therefore, calling the `fit` method of two different instances operating on the *same* corpus with the *same* number of topics potentially leads to (slightly) different results, corresponding to different local minima of the Kullback-Leibler divergence between the true document-word probability and its approximate factorization. To mitigate this effect, perform multiple runs and pick the best model.

Parameters

- **n_runs** (*int, optional*) – Number of runs to pick the best model of. Defaults to 3.
- ****kwargs** – Keyword-only arguments are passed on to the `fit` method.

Returns Container class for the best result.

Return type *PlsaResult*

fit (*eps: float = 1e-05*, *max_iter: int = 200*, *warmup: int = 5*) → plsa.algorithms.result.PlsaResult
Run EM-style training to find latent topics in documents.

Expectation-maximization (EM) iterates until either the maximum number of iterations is reached or if relative changes of the Kullback- Leibler divergence between the actual document-word probability and its approximate fall below a certain threshold, whichever occurs first.

Since all quantities are update in-place, calling the `fit` method again after a successful run (possibly with changed convergence criteria) will continue to add more iterations on top of the status quo rather than starting all over again from scratch.

Because a few EM iterations are needed to get things going, you can specify an initial *warm-up* period, during which progress in the Kullback-Leibler divergence is not tracked, and which does not count towards the maximum number of iterations.

Parameters

- **eps** (*float, optional*) – The convergence cutoff for relative changes in the Kullback- Leibler divergence between the actual document-word probability and its approximate. Defaults to 1e-5.
- **max_iter** (*int, optional*) – The maximum number of iterations to perform. Defaults to 200.
- **warmup** (*int, optional*) – The number of iterations to perform before changes in the Kullback-Leibler divergence are tracked for convergence.

Returns Container class for the results of the latent semantic analysis.

Return type *PlsaResult*

n_topics
The number of topics to find.

tf_idf
Use inverse document frequency to weigh the document-word counts?

4.1.2 plsa.algorithms.conditional_plsa module

class plsa.algorithms.conditional_plsa.**ConditionalPLSA**(*corpus: plsa.corpus.Corpus*, *n_topics: int*, *tf_idf: bool = True*)

Implements conditional probabilistic latent semantic analysis (PLSA).

Given that the normalized document-word (or term-frequency) matrix *p(d, w)*, weighted with the inverse document frequency or not, can always be written as,

$$p(d, w) = p(d|w)p(w)$$

the core of conditional PLSA is the assumption that the conditional *p(d|w)* can be factorized as:

$$p(d|w) \approx \sum_t \tilde{p}(d|t)\tilde{p}(t|w)$$

Parameters

- **corpus** (Corpus) – The corpus of preprocessed and numerically represented documents.
- **n_topics** (*int*) – The number of latent topics to identify.
- **tf_idf** (*bool*) – Whether to use the term-frequency inverse-document-frequency or just the term-frequency matrix as joint probability *p(d, w)* of documents and words.

Raises ValueError – If the number of topics is < 2 or the number of both, words and documents, in the corpus isn't greater than the number of topics.

Notes

Importantly, the present implementation does *not* follow algorithm 15.3 in Barber's book[1]. The update equations there appear non-sensical. Following through the derivation that gives (non-conditional) PLSA, one arrives at the following updates:

$$\tilde{p}(d|t) = \sum_w p(d, w)q(t|d, w)$$
$$\tilde{p}(t|w) = \sum_d p(d, w)q(t|d, w)$$
$$\tilde{p}(t, d, w) = p(w)\sum_t \tilde{p}(d|t)\tilde{p}(t|w)$$
$$q(t|d, w) = \tilde{p}(t, d, w)/\tilde{p}(d, w)$$

References

best_of(*n_runs: int = 3*, ***kwargs*) → plsa.algorithms.result.PlsaResult

Finds the best PLSA model among the specified number of runs.

As with any iterative algorithm, also the probabilities in PSLA need to be (randomly) initialized prior to the first iteration step. Therefore, calling the fit method of two different instances operating on the *same* corpus with the *same* number of topics potentially leads to (slightly) different results, corresponding to different local minima of the Kullback-Leibler divergence between the true document-word probability and its approximate factorization. To mitigate this effect, perform multiple runs and pick the best model.

Parameters

[1] "Bayesian Reasoning and Machine Learning", David Barber (Cambridge Press, 2012).

- **n_runs** (*int, optional*) – Number of runs to pick the best model of. Defaults to 3.
- ****kwargs** – Keyword-only arguments are passed on to the `fit` method.

Returns Container class for the best result.

Return type *PlsaResult*

fit(*eps: float = 1e-05*, *max_iter: int = 200*, *warmup: int = 5*) → plsa.algorithms.result.PlsaResult
Run EM-style training to find latent topics in documents.

Expectation-maximization (EM) iterates until either the maximum number of iterations is reached or if relative changes of the Kullback- Leibler divergence between the actual document-word probability and its approximate fall below a certain threshold, whichever occurs first.

Since all quantities are update in-place, calling the `fit` method again after a successful run (possibly with changed convergence criteria) will continue to add more iterations on top of the status quo rather than starting all over again from scratch.

Because a few EM iterations are needed to get things going, you can specify an initial *warm-up* period, during which progress in the Kullback-Leibler divergence is not tracked, and which does not count towards the maximum number of iterations.

Parameters

- **eps** (*float, optional*) – The convergence cutoff for relative changes in the Kullback- Leibler divergence between the actual document-word probability and its approximate. Defaults to 1e-5.
- **max_iter** (*int, optional*) – The maximum number of iterations to perform. Defaults to 200.
- **warmup** (*int, optional*) – The number of iterations to perform before changes in the Kullback-Leibler divergence are tracked for convergence.

Returns Container class for the results of the latent semantic analysis.

Return type *PlsaResult*

n_topics
The number of topics to find.

tf_idf
Use inverse document frequency to weigh the document-word counts?

4.1.3 plsa.algorithms.result module

class plsa.algorithms.result.**PlsaResult**(*topic_given_doc: numpy.ndarray, word_given_topic: numpy.ndarray, topic_given_word: numpy.ndarray, topic: numpy.ndarray, kl_divergences: List[float], corpus: plsa.corpus.Corpus, tf_idf: bool*)

Bases: `object`

Container for the results generated by a (conditional) PLSA run.

Parameters

- **topic_given_doc** (*ndarray*) – The conditional probability *p(t|d)* as $n_{topics} \times n_{docs}$ array.
- **word_given_topic** (*ndarray*) – The conditional probability *p(w|t)* as $n_{words} \times n_{topics}$ array.

- **topic_given_word** (*ndarray*) – The conditional probability *p(t|w)* as $n_{topics} \times n_{words}$ array.
- **topic** (*ndarray*) – The marginal probability *p(w)*.
- **kl_divergences** (*list of float*) – The Kullback-Leibler divergences between the original document-word probability *p(d, w)* and its approximate for each iteration.
- **corpus** (Corpus) – The original corpus the PLSA model was trained on.
- **tf_idf** (*bool*) – Whether to weigh the document.word matrix with the inverse document frequencies or not.

convergence
The convergence of the Kullback-Leibler divergence.

kl_divergence
KL-divergence of approximate and true document-word probability.

n_topics
The number of latent topics identified.

predict (*doc: str*) → Tuple[numpy.ndarray, int, Tuple[str, ...]]
Predict the relative importance of latent topics in a new document.

Parameters **doc** (*str*) – A new document given as a single string.

Returns

- *ndarray* – A 1-D array with the relative importance of latent topics.
- *int* – The number of words in the new document that were not present in the corpus the PLSA model was trained on.
- *tuple of str* – Those words in the new document that were not present in the corpus the PLSA model was trained on.

Raises ValueError – If the document to predict on is an empty string, if there are no words left after preprocessing the document, or if there are no known words in the document.

tf_idf
Used inverse document frequency to weigh the document-word counts?

topic
The relative importance of latent topics.

topic_given_doc
The relative importance of latent topics in each document.

Dimensions are $n_{docs} \times n_{topics}$.

word_given_topic
The words in each latent topic and their relative importance.

Results are presented as a tuple of 2-tuples (word, word importance).

CHAPTER 5

plsa.visualize module

class plsa.visualize.**Visualize**(*result: plsa.algorithms.result.PlsaResult*)

Bases: object

Visualize the results of probabilistic latent semantic analysis.

Parameters **result** (PlsaResult) – The results object returned by the fit method of a PLSA model object.

convergence(*axis: matplotlib.axes._subplots.AxesSubplot*) → List[matplotlib.lines.Line2D]

Plot the convergence of the PLSA run.

The quantity to be minimized is the Kullback-Leibler divergence between the original document-word matrix and its approximation given by the (conditional) PLSA factorization.

Parameters **axis** (*Subplot*) – The matplotlib axis to plot into.

Returns The line object plotted into the given axis.

Return type list of Line2D

prediction(*doc: str*, *axis: matplotlib.axes._subplots.AxesSubplot*) → matplotlib.container.BarContainer

Plot the predicted relative weights of topics in a new document.

Parameters

- **doc** (*str*) – A new document given as a single string.
- **axis** (*Subplot*) – The matplotlib axis to plot into.

Returns The container for the bars plotted into the given axis.

Return type BarContainer

topics(*axis: matplotlib.axes._subplots.AxesSubplot*) → matplotlib.container.BarContainer

Plot the relative importance of the individual topics.

Parameters **axis** (*Subplot*) – The matplotlib axis to plot into.

Returns The container for the bars plotted into the given axis.

Return type BarContainer

topics_in_doc(*i_doc: int*, *axis: matplotlib.axes._subplots.AxesSubplot*) → matplotlib.container.BarContainer

Plot the relative weights of topics in a given document.

Parameters

- **i_doc** (*int*) – Index of the document to plot. Numbering starts at 0.
- **axis** (*Subplot*) – The matplotlib axis to plot into.

Returns The container for the bars plotted into the given axis.

Return type BarContainer

wordclouds(*figure: matplotlib.figure.Figure*) → List[matplotlib.image.AxesImage]

Plot the relative importance of words in all topics.

Parameters **figure** (*Figure*) – An empty matplotlib figure to plot into.

Returns List of images with the created word clouds.

Return type list of AxisImage

words_in_topic(*i_topic: int*, *axis: matplotlib.axes._subplots.AxesSubplot*) → matplotlib.image.AxesImage

Plot the relative importance of words in a given topic.

Parameters

- **i_topic** (*int*) – Index of the topic to plot. Numbering starts at 0.
- **axis** (*Subplot*) – The matplotlib axis to plot into.

Returns The image with the produced word cloud.

Return type AxesImage

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

T

V

W