KeyphraseVectorizers

Release 0.0.11

Tim Schopf

Dec 23, 2022

USER GUIDE:

CHAPTER

ONE

KEYPHRASEVECTORIZERS

Set of vectorizers that extract keyphrases with part-of-speech patterns from a collection of text documents and convert them into a document-keyphrase matrix. A document-keyphrase matrix is a mathematical matrix that describes the frequency of keyphrases that occur in a collection of documents. The matrix rows indicate the text documents and columns indicate the unique keyphrases.

The package contains wrappers of the sklearn.feature_extraction.text.CountVectorizer and sklearn.feature_extraction.text.TfidfVectorizer classes. Instead of using n-gram tokens of a pre-defined range, these classes extract keyphrases from text documents using part-of-speech tags to compute document-keyphrase matrices.

1.1 Benefits

- Extract grammatically accurate keyphases based on their part-of-speech tags.
- No need to specify n-gram ranges.
- Get document-keyphrase matrices.
- Multiple language support.
- User-defined part-of-speech patterns for keyphrase extraction possible.

1.2 Table of Contents

1.3 How does it work?

First, the document texts are annotated with spaCy part-of-speech tags. A list of all possible spaCy part-of-speech tags for different languages is linked here. The annotation requires passing the spaCy pipeline of the corresponding language to the vectorizer with the `spacy_pipeline` parameter.

Second, words are extracted from the document texts whose part-of-speech tags match the regex pattern defined in the `pos_pattern` parameter. The keyphrases are a list of unique words extracted from text documents by this method.

Finally, the vectorizers calculate document-keyphrase matrices.

1.4 Installation

```
pip install keyphrase-vectorizers
```

1.5 Usage

For detailed information visit the API Guide.

1.5.1 KeyphraseCountVectorizer

Back to Table of Contents

English language

```
from keyphrase_vectorizers import KeyphraseCountVectorizer

docs = ["""Supervised learning is the machine learning task of learning a function that
         maps an input to an output based on example input-output pairs. It infers a
         function from labeled training data consisting of a set of training examples.
         In supervised learning, each example is a pair consisting of an input object
         (typically a vector) and a desired output value (also called the supervisory␣
↪signal).
         A supervised learning algorithm analyzes the training data and produces an␣
↪inferred function,
         which can be used for mapping new examples. An optimal scenario will allow for␣
↪the
         algorithm to correctly determine the class labels for unseen instances. This␣
↪requires
         the learning algorithm to generalize from the training data to unseen␣
↪situations in a
         'reasonable' way (see inductive bias).""",
```

(continues on next page)

(continued from previous page)

```
        """Keywords are defined as phrases that capture the main topics discussed in a␣
↪document.
        As they offer a brief yet precise summary of document content, they can be␣
↪utilized for various applications.
        In an information retrieval environment, they serve as an indication of document␣
↪relevance for users, as the list
        of keywords can quickly help to determine whether a given document is relevant␣
↪to their interest.
        As keywords reflect a document's main topics, they can be utilized to classify␣
↪documents into groups
        by measuring the overlap between the keywords assigned to them. Keywords are␣
↪also used proactively
        in information retrieval."""]

# Init default vectorizer.
vectorizer = KeyphraseCountVectorizer()

# Print parameters
print(vectorizer.get_params())
>>> {'binary': False, 'dtype': <class 'numpy.int64'>, 'lowercase': True, 'max_df': None,
↪'min_df': None, 'pos_pattern': '<J.*>*<N.*>+', 'spacy_pipeline': 'en_core_web_sm',
↪'stop_words': 'english', 'workers': 1}
```

By default, the vectorizer is initialized for the English language. That means, an English `spacy_pipeline` is specified, English `stop_words` are removed, and the `pos_pattern` extracts keywords that have 0 or more adjectives, followed by 1 or more nouns using the English spaCy part-of-speech tags.

```
# After initializing the vectorizer, it can be fitted
# to learn the keyphrases from the text documents.
vectorizer.fit(docs)
```

```
# After learning the keyphrases, they can be returned.
keyphrases = vectorizer.get_feature_names_out()

print(keyphrases)
>>> ['output' 'training data' 'task' 'way' 'input object' 'documents'
 'unseen instances' 'vector' 'interest' 'learning algorithm'
 'unseen situations' 'training examples' 'machine' 'given document'
 'document' 'document relevance' 'output pairs' 'document content'
 'class labels' 'new examples' 'pair' 'main topics' 'phrases' 'overlap'
 'algorithm' 'various applications' 'information retrieval' 'users' 'list'
 'example input' 'supervised learning' 'optimal scenario'
 'precise summary' 'keywords' 'input' 'supervised learning algorithm'
 'example' 'supervisory signal' 'indication' 'set'
 'information retrieval environment' 'output value' 'inductive bias'
 'groups' 'function']
```

```
# After fitting, the vectorizer can transform the documents
# to a document-keyphrase matrix.
# Matrix rows indicate the documents and columns indicate the unique keyphrases.
# Each cell represents the count.
```

(continues on next page)

(continued from previous page)

```
document_keyphrase_matrix = vectorizer.transform(docs).toarray()

print(document_keyphrase_matrix)
>>> [[3 3 1 1 1 0 1 1 0 2 1 1 1 0 0 0 1 0 1 1 1 0 0 0 3 0 0 0 0 1 3 1 0 0 3 1
  2 1 0 1 0 1 1 0 3]
 [0 0 0 0 0 1 0 0 1 0 0 0 0 1 5 1 0 1 0 0 0 2 1 1 0 1 2 1 1 0 0 0 1 5 0 0
  0 0 1 0 1 0 0 1 0]]
```

```
# Fit and transform can also be executed in one step,
# which is more efficient.
document_keyphrase_matrix = vectorizer.fit_transform(docs).toarray()

print(document_keyphrase_matrix)
>>> [[3 3 1 1 1 0 1 1 0 2 1 1 1 0 0 0 1 0 1 1 1 0 0 0 3 0 0 0 0 1 3 1 0 0 3 1
  2 1 0 1 0 1 1 0 3]
 [0 0 0 0 0 1 0 0 1 0 0 0 0 1 5 1 0 1 0 0 0 2 1 1 0 1 2 1 1 0 0 0 1 5 0 0
  0 0 1 0 1 0 0 1 0]]
```

Other languages

Back to Table of Contents

```
german_docs = ["""Goethe stammte aus einer angesehenen bürgerlichen Familie.
                Sein Großvater mütterlicherseits war als Stadtschultheiß höchster␣
↪Justizbeamter der Stadt Frankfurt,
                sein Vater Doktor der Rechte und Kaiserlicher Rat. Er und seine␣
↪Schwester Cornelia erfuhren eine aufwendige
                Ausbildung durch Hauslehrer. Dem Wunsch seines Vaters folgend, studierte␣
↪Goethe in Leipzig und Straßburg
                Rechtswissenschaft und war danach als Advokat in Wetzlar und Frankfurt␣
↪tätig.
                Gleichzeitig folgte er seiner Neigung zur Dichtkunst.""",

               """Friedrich Schiller wurde als zweites Kind des Offiziers, Wundarztes␣
↪und Leiters der Hofgärtnerei in
               Marbach am Neckar Johann Kaspar Schiller und dessen Ehefrau Elisabetha␣
↪Dorothea Schiller, geb. Kodweiß,
               die Tochter eines Wirtes und Bäckers war, 1759 in Marbach am Neckar␣
↪geboren
               """]
# Init vectorizer for the german language
vectorizer = KeyphraseCountVectorizer(spacy_pipeline='de_core_news_sm', pos_pattern='
↪<ADJ.*>*<N.*>+', stop_words='german')
```

The German `spacy_pipeline` is specified and German `stop_words` are removed. Because the German spaCy part-of-speech tags differ from the English ones, the `pos_pattern` parameter is also customized. The regex pattern `<ADJ.*>*<N.*>+` extracts keywords that have 0 or more adjectives, followed by 1 or more nouns using the German spaCy part-of-speech tags.

1.5.2 KeyphraseTfidfVectorizer

Back to Table of Contents

The `KeyphraseTfidfVectorizer` has the same function calls and features as the `KeyphraseCountVectorizer`. The only difference is, that document-keyphrase matrix cells represent tf or tf-idf values, depending on the parameter settings, instead of counts.

```
from keyphrase_vectorizers import KeyphraseTfidfVectorizer

docs = ["""Supervised learning is the machine learning task of learning a function that
         maps an input to an output based on example input-output pairs. It infers a
         function from labeled training data consisting of a set of training examples.
         In supervised learning, each example is a pair consisting of an input object
         (typically a vector) and a desired output value (also called the supervisory␣
↪signal).
         A supervised learning algorithm analyzes the training data and produces an␣
↪inferred function,
         which can be used for mapping new examples. An optimal scenario will allow for␣
↪the
         algorithm to correctly determine the class labels for unseen instances. This␣
↪requires
         the learning algorithm to generalize from the training data to unseen␣
↪situations in a
         'reasonable' way (see inductive bias).""",

        """Keywords are defined as phrases that capture the main topics discussed in a␣
↪document.
        As they offer a brief yet precise summary of document content, they can be␣
↪utilized for various applications.
        In an information retrieval environment, they serve as an indication of document␣
↪relevance for users, as the list
        of keywords can quickly help to determine whether a given document is relevant␣
↪to their interest.
        As keywords reflect a document's main topics, they can be utilized to classify␣
↪documents into groups
        by measuring the overlap between the keywords assigned to them. Keywords are␣
↪also used proactively
        in information retrieval."""]

# Init default vectorizer for the English language that computes tf-idf values
vectorizer = KeyphraseTfidfVectorizer()

# Print parameters
print(vectorizer.get_params())
>>> {'binary': False, 'dtype': <class 'numpy.float64'>, 'lowercase': True, 'max_df':␣
↪None, 'min_df': None, 'norm': 'l2', 'pos_pattern': '<J.*>*<N.*>+', 'smooth_idf': True,
↪'spacy_pipeline': 'en_core_web_sm', 'stop_words': 'english', 'sublinear_tf': False,
↪'use_idf': True, 'workers': 1}
```

To calculate tf values instead, set `use_idf=False`.

```
# Fit and transform to document-keyphrase matrix.
document_keyphrase_matrix = vectorizer.fit_transform(docs).toarray()
```

(continues on next page)

(continued from previous page)

```
print(document_keyphrase_matrix)
>>> [[0.11111111 0.22222222 0.11111111 0.         0.         0.
  0.11111111 0.         0.11111111 0.11111111 0.33333333 0.
  0.         0.         0.11111111 0.         0.         0.11111111
  0.         0.33333333 0.         0.22222222 0.         0.11111111
  0.11111111 0.11111111 0.11111111 0.11111111 0.33333333 0.11111111
  0.11111111 0.33333333 0.11111111 0.         0.33333333 0.
  0.         0.         0.11111111 0.         0.11111111 0.11111111
  0.         0.33333333 0.11111111]
 [0.         0.         0.         0.11785113 0.11785113 0.11785113
  0.         0.11785113 0.         0.         0.         0.11785113
  0.11785113 0.11785113 0.         0.11785113 0.23570226 0.
  0.23570226 0.         0.58925565 0.         0.11785113 0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.58925565 0.         0.11785113
  0.11785113 0.11785113 0.         0.11785113 0.         0.
  0.11785113 0.         0.        ]]
```

```
# Return keyphrases
keyphrases = vectorizer.get_feature_names_out()

print(keyphrases)
>>> ['optimal scenario' 'example' 'input object' 'groups' 'list'
 'precise summary' 'inductive bias' 'phrases' 'training examples'
 'output value' 'function' 'given document' 'documents'
 'information retrieval environment' 'new examples' 'interest'
 'main topics' 'unseen situations' 'information retrieval' 'input'
 'keywords' 'learning algorithm' 'indication' 'set' 'example input'
 'vector' 'machine' 'supervised learning algorithm' 'algorithm' 'pair'
 'task' 'training data' 'way' 'document' 'supervised learning' 'users'
 'document relevance' 'document content' 'supervisory signal' 'overlap'
 'class labels' 'unseen instances' 'various applications' 'output'
 'output pairs']
```

1.5.3 Keyphrase extraction with KeyBERT

Back to Table of Contents

The keyphrase vectorizers can be used together with KeyBERT to extract grammatically correct keyphrases that are most similar to a document. Thereby, the vectorizer first extracts candidate keyphrases from the text documents, which are subsequently ranked by KeyBERT based on their document similarity. The top-n most similar keyphrases can then be considered as document keywords.

The advantage of using KeyphraseVectorizers in addition to KeyBERT is that it allows users to get grammatically correct keyphrases instead of simple n-grams of pre-defined lengths. In KeyBERT, users can specify the `keyphrase_ngram_range` to define the length of the retrieved keyphrases. However, this raises two issues. First, users usually do not know the optimal n-gram range and therefore have to spend some time experimenting until they find a suitable n-gram range. Second, even after finding a good n-gram range, the returned keyphrases are sometimes still grammatically not quite correct or are slightly off-key. Unfortunately, this limits the quality of the returned keyphrases.

To adress this issue, we can use the vectorizers of this package to first extract candidate keyphrases that consist of zero or more adjectives, followed by one or multiple nouns in a pre-processing step instead of simple n-grams. Wan and Xiao successfully used this noun phrase approach for keyphrase extraction during their research in 2008. The extracted candidate keyphrases are subsequently passed to KeyBERT for embedding generation and similarity calculation. To use both packages for keyphrase extraction, we need to pass KeyBERT a keyphrase vectorizer with the `vectorizer` parameter. Since the length of keyphrases now depends on part-of-speech tags, there is no need to define an n-gram length anymore.

Example:

KeyBERT can be installed via `pip install keybert`.

```
from keyphrase_vectorizers import KeyphraseCountVectorizer
from keybert import KeyBERT

docs = ["""Supervised learning is the machine learning task of learning a function that
         maps an input to an output based on example input-output pairs. It infers a
         function from labeled training data consisting of a set of training examples.
         In supervised learning, each example is a pair consisting of an input object
         (typically a vector) and a desired output value (also called the supervisory␣
↪signal).
         A supervised learning algorithm analyzes the training data and produces an␣
↪inferred function,
         which can be used for mapping new examples. An optimal scenario will allow for␣
↪the
         algorithm to correctly determine the class labels for unseen instances. This␣
↪requires
         the learning algorithm to generalize from the training data to unseen␣
↪situations in a
         'reasonable' way (see inductive bias).""",

        """Keywords are defined as phrases that capture the main topics discussed in a␣
↪document.
       As they offer a brief yet precise summary of document content, they can be␣
↪utilized for various applications.
       In an information retrieval environment, they serve as an indication of document␣
↪relevance for users, as the list
       of keywords can quickly help to determine whether a given document is relevant␣
↪to their interest.
       As keywords reflect a document's main topics, they can be utilized to classify␣
↪documents into groups
       by measuring the overlap between the keywords assigned to them. Keywords are␣
↪also used proactively
       in information retrieval."""]

kw_model = KeyBERT()
```

Instead of deciding on a suitable n-gram range which could be e.g.(1,2)...

```
>>> kw_model.extract_keywords(docs=docs, keyphrase_ngram_range=(1,2))
[[('labeled training', 0.6013),
  ('examples supervised', 0.6112),
  ('signal supervised', 0.6152),
```

(continues on next page)

(continued from previous page)

```
  ('supervised', 0.6676),
  ('supervised learning', 0.6779)],
 [('keywords assigned', 0.6354),
  ('keywords used', 0.6373),
  ('list keywords', 0.6375),
  ('keywords quickly', 0.6376),
  ('keywords defined', 0.6997)]]
```

we can now just let the keyphrase vectorizer decide on suitable keyphrases, without limitations to a maximum or minimum n-gram range. We only have to pass a keyphrase vectorizer as parameter to KeyBERT:

```
>>> kw_model.extract_keywords(docs=docs, vectorizer=KeyphraseCountVectorizer())
[[('training examples', 0.4668),
  ('training data', 0.5271),
  ('learning algorithm', 0.5632),
  ('supervised learning', 0.6779),
  ('supervised learning algorithm', 0.6992)],
 [('given document', 0.4143),
  ('information retrieval environment', 0.5166),
  ('information retrieval', 0.5792),
  ('keywords', 0.6046),
  ('document relevance', 0.633)]]
```

This allows us to make sure that we do not cut off important words caused by defining our n-gram range too short. For example, we would not have found the keyphrase "supervised learning algorithm" with keyphrase_ngram_range=(1,2). Furthermore, we avoid to get keyphrases that are slightly off-key like "labeled training", "signal supervised" or "keywords quickly".

1.5.4 Topic modeling with BERTopic and KeyphraseVectorizers

Back to Table of Contents

Similar to the application with KeyBERT, the keyphrase vectorizers can be used to obtain grammatically correct keyphrases as descriptions for topics instead of simple n-grams. This allows us to make sure that we do not cut off important topic description keyphrases by defining our n-gram range too short. Moreover, we don't need to clean stopwords upfront, can get more precise topic models and avoid to get topic description keyphrases that are slightly off-key.

Example:

BERTopic can be installed via `pip install bertopic`.

```
from keyphrase_vectorizers import KeyphraseCountVectorizer
from bertopic import BERTopic
from sklearn.datasets import fetch_20newsgroups

# load text documents
docs = fetch_20newsgroups(subset='all',  remove=('headers', 'footers', 'quotes'))['data']
# only use subset of the data
docs = docs[:5000]
```

(continues on next page)

(continued from previous page)

```
# train topic model with KeyphraseCountVectorizer
keyphrase_topic_model = BERTopic(vectorizer_model=KeyphraseCountVectorizer())
keyphrase_topics, keyphrase_probs = keyphrase_topic_model.fit_transform(docs)

# get topics
>>> keyphrase_topic_model.topics
{-1: [('file', 0.007265527630674131),
  ('one', 0.007055454904474792),
  ('use', 0.00633563957153475),
  ('program', 0.006053271092949018),
  ('get', 0.006011060091056076),
  ('people', 0.005729309058970368),
  ('know', 0.005635951168273583),
  ('like', 0.005569244980291 6015),
  ('time', 0.00527028825803415),
  ('us', 0.00525564504880084)],
 0: [('game', 0.024134589719090525),
  ('team', 0.021852806383170772),
  ('players', 0.01749406934044139),
  ('games', 0.014397938026886745),
  ('hockey', 0.013932342023677305),
  ('win', 0.013706115572901401),
  ('year', 0.013297593024390321),
  ('play', 0.012533185558169046),
  ('baseball', 0.012412743802062559),
  ('season', 0.011602725885164318)],
 1: [('patients', 0.022600352291162015),
  ('msg', 0.02023877371575874),
  ('doctor', 0.018816282737587457),
  ('medical', 0.018614407917995103),
  ('treatment', 0.0165028251400717),
  ('food', 0.01604980195180696),
  ('candida', 0.015255961242066143),
  ('disease', 0.015115496310099693),
  ('pain', 0.014129703072484495),
  ('hiv', 0.012884503220341102)],
 2: [('key', 0.028851633177510126),
  ('encryption', 0.024375137861044675),
  ('clipper', 0.023565947302544528),
  ('privacy', 0.019258719348097385),
  ('security', 0.018983682856076434),
  ('chip', 0.018822199098878365),
  ('keys', 0.016060139239615384),
  ('internet', 0.01450486904722165),
  ('encrypted', 0.013194373119964168),
  ('government', 0.01303978311708837)],
  ...
```

The same topics look a bit different when no keyphrase vectorizer is used:

```
from bertopic import BERTopic
```

(continues on next page)

(continued from previous page)

```
from sklearn.datasets import fetch_20newsgroups

# load text documents
docs = fetch_20newsgroups(subset='all',  remove=('headers', 'footers', 'quotes'))['data']
# only use subset of the data
docs = docs[:5000]

# train topic model without KeyphraseCountVectorizer
topic_model = BERTopic()
topics, probs = topic_model.fit_transform(docs)

# get topics
>>> topic_model.topics
{-1: [('the', 0.012864641020408933),
  ('to', 0.01187920529994724),
  ('and', 0.011431498631699856),
  ('of', 0.01099851927541331),
  ('is', 0.010995478673036962),
  ('in', 0.009908233622158523),
  ('for', 0.009903667215879675),
  ('that', 0.009619596716087699),
  ('it', 0.009578499681829809),
  ('you', 0.009532846440753)],
 0: [('game', 0.013949166096523719),
  ('team', 0.012458483177116456),
  ('he', 0.012354733462693834),
  ('the', 0.01119583508278812),
  ('10', 0.010190243555226108),
  ('in', 0.0101436249231417),
  ('players', 0.009682212470082758),
  ('to', 0.00933700544705287),
  ('was', 0.009172402203816335),
  ('and', 0.008653375901739337)],
 1: [('of', 0.012771267188340924),
  ('to', 0.012581337590513296),
  ('is', 0.012554884458779008),
  ('patients', 0.011983273578628046),
  ('and', 0.011863499662237566),
  ('that', 0.011616113472989725),
  ('it', 0.011581944987387165),
  ('the', 0.011475148304229873),
  ('in', 0.011395485985801054),
  ('msg', 0.010715000656335596)],
 2: [('key', 0.01725282988290282),
  ('the', 0.014634841495851404),
  ('be', 0.014429762197907552),
  ('encryption', 0.013530733999898166),
  ('to', 0.013443159534369817),
  ('clipper', 0.01296614319927958),
  ('of', 0.012164734232650158),
  ('is', 0.012128295958613464),
  ('and', 0.011972763728732667),
```

(continues on next page)

(continued from previous page)

```
 ('chip', 0.010785744492767285)],
...
```

CHAPTER

TWO

KEYPHRASECOUNTVECTORIZER

class keyphrase_vectorizers.keyphrase_count_vectorizer.**KeyphraseCountVectorizer**(*spacy_pipeline: ~typing.Union[str, ~spacy.language.Language] = 'en_core_web_sm', pos_pattern: str = '<J.*>*<N.*>+', stop_words: ~typing.Union[str, ~typing.List[str]] = 'english', lowercase: bool = True, workers: int = 1, spacy_exclude: ~typing.Optional[~typing.List[str]] = None, custom_pos_tagger: ~typing.Optional[callable] = None, max_df: ~typing.Optional[int] = None, min_df: ~typing.Optional[int] = None, binary: bool = False, dtype: ~numpy.dtype = <class 'numpy.int64'>*)

KeyphraseCountVectorizer converts a collection of text documents to a matrix of document-token counts. The tokens are keyphrases that are extracted from the text documents based on their part-of-speech tags. The matrix rows indicate the documents and columns indicate the unique keyphrases. Each cell represents the count. The part-of-speech pattern of keyphrases can be defined by the `pos_pattern` parameter. By default, keyphrases are extracted, that have 0 or more adjectives, followed by 1 or more nouns. A list of extracted keyphrases matching the defined part-of-speech pattern can be returned after fitting via *get_feature_names_out()*.

Attention: If the vectorizer is used for languages other than English, the `spacy_pipeline` and `stop_words` parameters must be customized accordingly. Additionally, the `pos_pattern` parameter has to be customized

as the spaCy part-of-speech tags differ between languages. Without customizing, the words will be tagged with wrong part-of-speech tags and no stopwords will be considered.

Parameters

- **spacy_pipeline** (*Union[str, spacy.Language], default='en_core_web_sm'*) – A spacy.Language object or the name of the spaCy pipeline, used to tag the parts-of-speech in the text. Standard is the 'en' pipeline.
- **pos_pattern** (*str, default='<J.*>*<N.*>+'*) – The regex pattern of POS-tags used to extract a sequence of POS-tagged tokens from the text. Standard is to only select keyphrases that have 0 or more adjectives, followed by 1 or more nouns.
- **stop_words** (*Union[str, List[str]], default='english'*) – Language of stopwords to remove from the document, e.g. 'english'. Supported options are stopwords available in NLTK. Removes unwanted stopwords from keyphrases if 'stop_words' is not None. If given a list of custom stopwords, removes them instead.
- **lowercase** (*bool, default=True*) – Whether the returned keyphrases should be converted to lowercase.
- **workers** (*int, default=1*) – How many workers to use for spaCy part-of-speech tagging. If set to -1, use all available worker threads of the machine. SpaCy uses the specified number of cores to tag documents with part-of-speech. Depending on the platform, starting many processes with multiprocessing can add a lot of overhead. In particular, the default start method spawn used in macOS/OS X (as of Python 3.8) and in Windows can be slow. Therefore, carefully consider whether this option is really necessary.
- **spacy_exclude** (*List[str], default=None*) – A list of spaCy pipeline components that should be excluded during the POS-tagging. Removing not needed pipeline components can sometimes make a big difference and improve loading and inference speed.
- **custom_pos_tagger** (*callable, default=None*) – A callable function which expects a list of strings in a 'raw_documents' parameter and returns a list of (word token, POS-tag) tuples. If this parameter is not None, the custom tagger function is used to tag words with parts-of-speech, while the spaCy pipeline is ignored.
- **max_df** (*int, default=None*) – During fitting ignore keyphrases that have a document frequency strictly higher than the given threshold.
- **min_df** (*int, default=None*) – During fitting ignore keyphrases that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature.
- **binary** (*bool, default=False*) – If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.
- **dtype** (*type, default=np.int64*) – Type of the matrix returned by fit_transform() or transform().

fit(*raw_documents: List[str]*) → object

Learn the keyphrases that match the defined part-of-speech pattern from the list of raw documents.

Parameters

raw_documents (*iterable*) – An iterable of strings.

Returns

self – Fitted vectorizer.

Return type
object

`fit_transform`(*raw_documents: List[str]*) → List[List[int]]

Learn the keyphrases that match the defined part-of-speech pattern from the list of raw documents and return the document-keyphrase matrix. This is equivalent to fit followed by transform, but more efficiently implemented.

Parameters
raw_documents (*iterable*) – An iterable of strings.

Returns
X – Document-keyphrase matrix.

Return type
array of shape (n_samples, n_features)

`get_feature_names`() → List[str]

DEPRECATED: get_feature_names() is deprecated in scikit-learn 1.0 and will be removed with scikit-learn 1.2. Please use get_feature_names_out() instead.

Array mapping from feature integer indices to feature name.

Returns
feature_names – A list of fitted keyphrases.

Return type
list

`get_feature_names_out`() -> *array(<class 'str'>, dtype=object)*

Get fitted keyphrases for transformation.

Returns
feature_names_out – Transformed keyphrases.

Return type
ndarray of str objects

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

`inverse_transform`(*X: List[List[int]]*) → List[List[str]]

Return keyphrases per document with nonzero entries in X.

Parameters
X (*{array-like, sparse matrix} of shape (n_samples, n_features)*) – Document-keyphrase matrix.

Returns
X_inv – List of arrays of keyphrase.

Return type
list of arrays of shape (n_samples,)

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

`transform`(*raw_documents: List[str]*) → List[List[int]]

Transform documents to document-keyphrase matrix. Extract token counts out of raw text documents using the keyphrases fitted with fit.

Parameters
raw_documents (*iterable*) – An iterable of strings.

Returns
X – Document-keyphrase matrix.

Return type
sparse matrix of shape (n_samples, n_features)

CHAPTER

THREE

KEYPHRASETFIDFVECTORIZER

class keyphrase_vectorizers.keyphrase_tfidf_vectorizer.**KeyphraseTfidfVectorizer**(*spacy_pipeline: ~typing.Union[str, ~spacy.language.Language] = 'en_core_web_sm', pos_pattern: str = '<J.*>*<N.*>+', stop_words: ~typing.Union[str, ~typing.List[str]] = 'english', lowercase: bool = True, workers: int = 1, spacy_exclude: ~typing.Optional[~typing.List[str]] = None, custom_pos_tagger: ~typing.Optional[callable] = None, max_df: ~typing.Optional[int] = None, min_df: ~typing.Optional[int] = None, binary: bool = False, dtype: ~numpy.dtype = <class 'numpy.float64'>, norm: str = 'l2', use_idf: bool = True, smooth_idf: bool = True, sublinear_tf: bool = False*)

KeyphraseTfidfVectorizer converts a collection of text documents to a normalized tf or tf-idf document-token matrix. The tokens are keyphrases that are extracted from the text documents based on their part-of-speech tags. The matrix rows indicate the documents and columns indicate the unique keyphrases. Each cell represents the tf

or tf-idf value, depending on the parameter settings. The part-of-speech pattern of keyphrases can be defined by the `pos_pattern` parameter. By default, keyphrases are extracted, that have 0 or more adjectives, followed by 1 or more nouns. A list of extracted keyphrases matching the defined part-of-speech pattern can be returned after fitting via *get_feature_names_out()*.

Attention: If the vectorizer is used for languages other than English, the `spacy_pipeline` and `stop_words` parameters must be customized accordingly. Additionally, the `pos_pattern` parameter has to be customized as the spaCy part-of-speech tags differ between languages. Without customizing, the words will be tagged with wrong part-of-speech tags and no stopwords will be considered.

Tf means term-frequency while tf-idf means term-frequency times inverse document-frequency. This is a common term weighting scheme in information retrieval, that has also found good use in document classification.

The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus.

The formula that is used to compute the tf-idf for a term t of a document d in a document set is tf-idf(t, d) = tf(t, d) * idf(t), and the idf is computed as idf(t) = log [n / df(t)] + 1 (if `smooth_idf=False`), where n is the total number of documents in the document set and df(t) is the document frequency of t; the document frequency is the number of documents in the document set that contain the term t. The effect of adding "1" to the idf in the equation above is that terms with zero idf, i.e., terms that occur in all documents in a training set, will not be entirely ignored. (Note that the idf formula above differs from the standard textbook notation that defines the idf as idf(t) = log [n / (df(t) + 1)]).

If `smooth_idf=True` (the default), the constant "1" is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions: idf(t) = log [(1 + n) / (1 + df(t))] + 1.

Furthermore, the formulas used to compute tf and idf depend on parameter settings that correspond to the SMART notation used in IR as follows:

Tf is "n" (natural) by default, "l" (logarithmic) when `sublinear_tf=True`. Idf is "t" when use_idf is given, "n" (none) otherwise. Normalization is "c" (cosine) when `norm='l2'`, "n" (none) when `norm=None`.

Parameters

- **spacy_pipeline** (*Union[str, spacy.Language], default='en_core_web_sm'*) – A spacy.Language object or the name of the spaCy pipeline, used to tag the parts-of-speech in the text. Standard is the 'en' pipeline.
- **pos_pattern** (*str, default='<J.*>*<N.*>+'*) – The regex pattern of POS-tags used to extract a sequence of POS-tagged tokens from the text. Standard is to only select keyphrases that have 0 or more adjectives, followed by 1 or more nouns.
- **stop_words** (*Union[str, List[str]], default='english'*) – Language of stopwords to remove from the document, e.g. 'english'. Supported options are stopwords available in NLTK. Removes unwanted stopwords from keyphrases if 'stop_words' is not None. If given a list of custom stopwords, removes them instead.
- **lowercase** (*bool, default=True*) – Whether the returned keyphrases should be converted to lowercase.
- **workers** (*int, default=1*) – How many workers to use for spaCy part-of-speech tagging. If set to -1, use all available worker threads of the machine. SpaCy uses the specified number of cores to tag documents with part-of-speech. Depending on the platform, starting many processes with multiprocessing can add a lot of overhead. In particular, the default

start method spawn used in macOS/OS X (as of Python 3.8) and in Windows can be slow. Therefore, carefully consider whether this option is really necessary.

- **spacy_exclude** (*List[str], default=None*) – A list of spaCy pipeline components that should be excluded during the POS-tagging. Removing not needed pipeline components can sometimes make a big difference and improve loading and inference speed.
- **custom_pos_tagger** (*callable, default=None*) – A callable function which expects a list of strings in a 'raw_documents' parameter and returns a list of (word token, POS-tag) tuples. If this parameter is not None, the custom tagger function is used to tag words with parts-of-speech, while the spaCy pipeline is ignored.
- **max_df** (*int, default=None*) – During fitting ignore keyphrases that have a document frequency strictly higher than the given threshold.
- **min_df** (*int, default=None*) – During fitting ignore keyphrases that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature.
- **binary** (*bool, default=False*) – If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.
- **dtype** (*type, default=np.int64*) – Type of the matrix returned by fit_transform() or transform().
- **norm** (*{'l1', 'l2'}, default='l2'*) – Each output row will have unit norm, either: - 'l2': Sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied. - 'l1': Sum of absolute values of vector elements is 1.
- **use_idf** (*bool, default=True*) – Enable inverse-document-frequency reweighting. If False, idf(t) = 1.
- **smooth_idf** (*bool, default=True*) – Smooth idf weights by adding one to document frequencies, as if an extra document was seen containing every term in the collection exactly once. Prevents zero divisions.
- **sublinear_tf** (*bool, default=False*) – Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

fit(*raw_documents: List[str]*) → object

Learn the keyphrases that match the defined part-of-speech pattern and idf from the list of raw documents.

Parameters
raw_documents (*iterable*) – An iterable of strings.

Returns
self – Fitted vectorizer.

Return type
object

fit_transform(*raw_documents: List[str]*) → List[List[float]]

Learn the keyphrases that match the defined part-of-speech pattern and idf from the list of raw documents. Then return document-keyphrase matrix. This is equivalent to fit followed by transform, but more efficiently implemented.

Parameters
raw_documents (*iterable*) – An iterable of strings.

Returns
X – Tf-idf-weighted document-keyphrase matrix.

Return type
sparse matrix of (n_samples, n_features)

`get_feature_names`() → List[str]

DEPRECATED: get_feature_names() is deprecated in scikit-learn 1.0 and will be removed with scikit-learn 1.2. Please use get_feature_names_out() instead.

Array mapping from feature integer indices to feature name.

Returns
feature_names – A list of fitted keyphrases.

Return type
list

`get_feature_names_out`(*) -> *array(<class 'str'>, dtype=object)*

Get fitted keyphrases for transformation.

Returns
feature_names_out – Transformed keyphrases.

Return type
ndarray of str objects

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

`inverse_transform`(*X: List[List[int]]*) → List[List[str]]

Return keyphrases per document with nonzero entries in X.

Parameters
X ({*array-like, sparse matrix*} *of shape* (*n_samples, n_features*)) – Document-keyphrase matrix.

Returns
X_inv – List of arrays of keyphrase.

Return type
list of arrays of shape (n_samples,)

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

`transform`(*raw_documents: List[str]*) → List[List[float]]

Transform documents to document-keyphrase matrix. Uses the keyphrases and document frequencies (df) learned by fit (or fit_transform).

Parameters
raw_documents (*iterable*) – An iterable of strings.

Returns
X – Tf-idf-weighted document-keyphrase matrix.

Return type
sparse matrix of (n_samples, n_features)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX