6.2. Feature extraction

The `sklearn.feature_extraction` module can be used to extract features in a format supported b
machine learning algorithms from datasets consisting of formats such as text and image.

> **Note**
>
> Feature extraction is very different from Feature selection: the former consists in transforming arbitrary data, such as text or images, into numerical features usable for machine learning. The latter is a machine learning technique applied on these features.

6.2.1. Loading features from dicts

The class `DictVectorizer` can be used to convert feature arrays represented as lists of standard Python `dict` objects to the NumPy/SciPy representation used by scikit-learn estimators.

While not particularly fast to process, Python's `dict` has the advantages of being convenient to u
being sparse (absent features need not be stored) and storing feature names in addition to values

`DictVectorizer` implements what is called one-of-K or "one-hot" coding for categorical (aka nominal, discrete) features. Categorical features are "attribute-value" pairs where the value is restricted to a list of discrete possibilities without ordering (e.g. topic identifiers, types of objects, tags, names…).

In the following, "city" is a categorical attribute while "temperature" is a traditional numerical feature:

```
>>> measurements = [
...     {'city': 'Dubai', 'temperature': 33.},
...     {'city': 'London', 'temperature': 12.},
...     {'city': 'San Francisco', 'temperature': 18.},
... ]

>>> from sklearn.feature_extraction import DictVectorizer
>>> vec = DictVectorizer()

>>> vec.fit_transform(measurements).toarray()
array([[ 1.,  0.,  0., 33.],
       [ 0.,  1.,  0., 12.],
       [ 0.,  0.,  1., 18.]])

>>> vec.get_feature_names_out()
array(['city=Dubai', 'city=London', 'city=San Francisco', 'temperature'], ...)
```

`DictVectorizer` accepts multiple string values for one feature, like, e.g., multiple categories for a movie.

Assume a database classifies each movie using some categories (not mandatories) and its year of release.

```
>>> movie_entry = [{'category': ['thriller', 'drama'], 'year': 2003},
...                {'category': ['animation', 'family'], 'year': 2011},
...                {'year': 1974}]
>>> vec.fit_transform(movie_entry).toarray()
array([[0.000e+00, 1.000e+00, 0.000e+00, 1.000e+00, 2.003e+03],
       [1.000e+00, 0.000e+00, 1.000e+00, 0.000e+00, 2.011e+03],
       [0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 1.974e+03]])
>>> vec.get_feature_names_out()
array(['category=animation', 'category=drama', 'category=family',
       'category=thriller', 'year'], ...)
>>> vec.transform({'category': ['thriller'],
...                'unseen_feature': '3'}).toarray()
array([[0., 0., 0., 1., 0.]])
```

`DictVectorizer` is also a useful representation transformation for training sequence classifiers in Natural Language Processing models that typically work by extracting feature windows around a particular word of interest.

For example, suppose that we have a first algorithm that extracts Part of Speech (PoS) tags that we want to use as complementary tags for training a sequence classifier (e.g. a chunker). The followin dict could be such a window of features extracted around the word 'sat' in the sentence 'The cat s on the mat.':

```
>>> pos_window = [
...     {
...         'word-2': 'the',
...         'pos-2': 'DT',
...         'word-1': 'cat',
...         'pos-1': 'NN',
...         'word+1': 'on',
...         'pos+1': 'PP',
...     },
...     # in a real application one would extract many such dictionaries
... ]
```

This description can be vectorized into a sparse two-dimensional matrix suitable for feeding into a classifier (maybe after being piped into a `TfidfTransformer` for normalization):

```
>>> vec = DictVectorizer()
>>> pos_vectorized = vec.fit_transform(pos_window)
>>> pos_vectorized
<Compressed Sparse...dtype 'float64'
  with 6 stored elements and shape (1, 6)>
>>> pos_vectorized.toarray()
array([[1., 1., 1., 1., 1., 1.]])
>>> vec.get_feature_names_out()
array(['pos+1=PP', 'pos-1=NN', 'pos-2=DT', 'word+1=on', 'word-1=cat',
       'word-2=the'], ...)
```

As you can imagine, if one extracts such a context around each individual word of a corpus of documents the resulting matrix will be very wide (many one-hot-features) with most of them bein valued to zero most of the time. So as to make the resulting data structure able to fit in memory t `DictVectorizer` class uses a `scipy.sparse` matrix by default instead of a `numpy.ndarray`.

6.2.2. Feature hashing

The class `FeatureHasher` is a high-speed, low-memory vectorizer that uses a technique known as feature hashing, or the “hashing trick”. Instead of building a hash table of the features encountere in training, as the vectorizers do, instances of `FeatureHasher` apply a hash function to the feature to determine their column index in sample matrices directly. The result is increased speed and reduced memory usage, at the expense of inspectability; the hasher does not remember what the input features looked like and has no `inverse_transform` method.

Since the hash function might cause collisions between (unrelated) features, a signed hash functio is used and the sign of the hash value determines the sign of the value stored in the output matrix for a feature. This way, collisions are likely to cancel out rather than accumulate error, and the expected mean of any output feature's value is zero. This mechanism is enabled by default with `alternate_sign=True` and is particularly useful for small hash table sizes (`n_features < 10000`). Fo large hash table sizes, it can be disabled, to allow the output to be passed to estimators like **`MultinomialNB`** or **`chi2`** feature selectors that expect non-negative inputs.

`FeatureHasher` accepts either mappings (like Python's `dict` and its variants in the `collections` module), `(feature, value)` pairs, or strings, depending on the constructor parameter `input_type` Mapping are treated as lists of `(feature, value)` pairs, while single strings have an implicit value 1, so `['feat1', 'feat2', 'feat3']` is interpreted as `[('feat1', 1), ('feat2', 1), ('feat3', 1)]` . If a single feature occurs multiple times in a sample, the associated values will be summed (so `('feat', 2)` and `('feat', 3.5)` become `('feat', 5.5)`). The output from **`FeatureHasher`** is always a `scipy.sparse` matrix in the CSR format.

Feature hashing can be employed in document classification, but unlike **`CountVectorizer`** , **`FeatureHasher`** does not do word splitting or any other preprocessing except Unicode-to-UTF-8 encoding; see Vectorizing a large text corpus with the hashing trick, below, for a combined tokenizer/hasher.

As an example, consider a word-level natural language processing task that needs features extract from `(token, part_of_speech)` pairs. One could use a Python generator function to extract features:

```
def token_features(token, part_of_speech):
    if token.isdigit():
        yield "numeric"
    else:
        yield "token={}".format(token.lower())
        yield "token,pos={},{}".format(token, part_of_speech)
    if token[0].isupper():
        yield "uppercase_initial"
    if token.isupper():
        yield "all_uppercase"
    yield "pos={}".format(part_of_speech)
```

Then, the `raw_X` to be fed to `FeatureHasher.transform` can be constructed using:

```
raw_X = (token_features(tok, pos_tagger(tok)) for tok in corpus)
```

and fed to a hasher with:

```
hasher = FeatureHasher(input_type='string')
X = hasher.transform(raw_X)
```

to get a `scipy.sparse` matrix `X`.

Note the use of a generator comprehension, which introduces laziness into the feature extraction: tokens are only processed on demand from the hasher.

Implementation details

References

- Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola and Josh Attenberg (2009). Feature hashing for large scale multitask learning. Proc. ICML.

6.2.3. Text feature extraction

6.2.3.1. The Bag of Words representation

Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length

In order to address this, scikit-learn provides utilities for the most common ways to extract numerical features from text content, namely:

- **tokenizing** strings and giving an integer id for each possible token, for instance by using whi spaces and punctuation as token separators.
- **counting** the occurrences of tokens in each document.
- **normalizing** and weighting with diminishing importance tokens that occur in the majority of samples / documents.

In this scheme, features and samples are defined as follows:

- each **individual token occurrence frequency** (normalized or not) is treated as a **feature**.

- the vector of all the token frequencies for a given **document** is considered a multivariate **sample**.

A corpus of documents can thus be represented by a matrix with one row per document and one column per token (e.g. word) occurring in the corpus.

We call **vectorization** the general process of turning a collection of text documents into numerica feature vectors. This specific strategy (tokenization, counting and normalization) is called the **Bag Words** or "Bag of n-grams" representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

6.2.3.2. Sparsity

As most documents will typically use a very small subset of the words used in the corpus, the resulting matrix will have many feature values that are zeros (typically more than 99% of them).

For instance a collection of 10,000 short text documents (such as emails) will use a vocabulary wit size in the order of 100,000 unique words in total while each document will use 100 to 1000 uniqu words individually.

In order to be able to store such a matrix in memory but also to speed up algebraic operations matrix / vector, implementations will typically use a sparse representation such as the implementations available in the `scipy.sparse` package.

6.2.3.3. Common Vectorizer usage

`CountVectorizer` implements both tokenization and occurrence counting in a single class:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
```

This model has many parameters, however the default values are quite reasonable (please see the reference documentation for the details):

```
>>> vectorizer = CountVectorizer()
>>> vectorizer
CountVectorizer()
```

Let's use it to tokenize and count the word occurrences of a minimalistic corpus of text document

```
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third one.',
...     'Is this the first document?',
... ]
>>> X = vectorizer.fit_transform(corpus)
>>> X
<Compressed Sparse...dtype 'int64'
  with 19 stored elements and shape (4, 9)>
```

The default configuration tokenizes the string by extracting words of at least 2 letters. The specific function that does this step can be requested explicitly:

```
>>> analyze = vectorizer.build_analyzer()
>>> analyze("This is a text document to analyze.") == (
...     ['this', 'is', 'text', 'document', 'to', 'analyze'])
True
```

Each term found by the analyzer during the fit is assigned a unique integer index corresponding to column in the resulting matrix. This interpretation of the columns can be retrieved as follows:

```
>>> vectorizer.get_feature_names_out()
array(['and', 'document', 'first', 'is', 'one', 'second', 'the',
       'third', 'this'], ...)

>>> X.toarray()
array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 2, 1, 0, 1],
       [1, 0, 0, 0, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 0, 0, 1, 0, 1]]...)
```

The converse mapping from feature name to column index is stored in the `vocabulary_` attribute the vectorizer:

```
>>> vectorizer.vocabulary_.get('document')
1
```

Hence words that were not seen in the training corpus will be completely ignored in future calls to the transform method:

```
>>> vectorizer.transform(['Something completely new.']).toarray()
array([[0, 0, 0, 0, 0, 0, 0, 0, 0]]...)
```

Note that in the previous corpus, the first and the last documents have exactly the same words hence are encoded in equal vectors. In particular we lose the information that the last document is an interrogative form. To preserve some of the local ordering information we can extract 2-grams words in addition to the 1-grams (individual words):

```
>>> bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
...                                     token_pattern=r'\b\w+\b', min_df=1)
>>> analyze = bigram_vectorizer.build_analyzer()
>>> analyze('Bi-grams are cool!') == (
...     ['bi', 'grams', 'are', 'cool', 'bi grams', 'grams are', 'are cool'])
True
```

The vocabulary extracted by this vectorizer is hence much bigger and can now resolve ambiguities encoded in local positioning patterns:

```
>>> X_2 = bigram_vectorizer.fit_transform(corpus).toarray()
>>> X_2
array([[0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0],
       [0, 0, 1, 0, 0, 1, 1, 0, 0, 2, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0],
       [1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1]]...)
```

In particular the interrogative form “Is this” is only present in the last document:

```
>>> feature_index = bigram_vectorizer.vocabulary_.get('is this')
>>> X_2[:, feature_index]
array([0, 0, 0, 1]...)
```

6.2.3.4. Using stop words

Stop words are words like “and”, “the”, “him”, which are presumed to be uninformative in representing the content of a text, and which may be removed to avoid them being construed as signal for prediction. Sometimes, however, similar words are useful for prediction, such as in classifying writing style or personality.

There are several known issues in our provided 'english' stop word list. It does not aim to be a general, 'one-size-fits-all' solution as some tasks may require a more custom solution. See [NQY18 for more details.

Please take care in choosing a stop word list. Popular stop word lists may include words that are highly informative to some tasks, such as *computer*.

You should also make sure that the stop word list has had the same preprocessing and tokenizatic applied as the one used in the vectorizer. The word *we've* is split into *we* and *ve* by CountVectorize default tokenizer, so if *we've* is in `stop_words`, but *ve* is not, *ve* will be retained from *we've* in transformed text. Our vectorizers will try to identify and warn about some kinds of inconsistencies

References

[NQY18] J. Nothman, H. Qin and R. Yurchak (2018). "Stop Word Lists in Free Open-source Softwar Packages". In *Proc. Workshop for NLP Open Source Software*.

6.2.3.5. Tf–idf term weighting

In a large text corpus, some words will be very present (e.g. "the", "a", "is" in English) hence carryin very little meaningful information about the actual contents of the document. If we were to feed t direct count data directly to a classifier those very frequent terms would shadow the frequencies c rarer yet more interesting terms.

In order to re-weight the count features into floating point values suitable for usage by a classifier is very common to use the tf–idf transform.

Tf means **term-frequency** while tf–idf means term-frequency times **inverse document-frequenc** $\text{tf-idf(t,d)} = \text{tf(t,d)} \times \text{idf(t)}$.

Using the `TfidfTransformer`'s default settings, `TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)` the term frequency, the number of times a term occurs in given document, is multiplied with idf component, which is computed as

$$\text{idf}(t) = \log{\frac{1 + n}{1+\text{df}(t)}} + 1,$$

where n is the total number of documents in the document set, and $\text{df}(t)$ is the number of documents in the document set that contain term t. The resulting tf-idf vectors are then normalize by the Euclidean norm:

$$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{v_1{}^2 + v_2{}^2 + \dots + v_n{}^2}}.$$

This was originally a term weighting scheme developed for information retrieval (as a ranking function for search engines results) that has also found good use in document classification and clustering.

The following sections contain further explanations and examples that illustrate how the tf-idfs are computed exactly and how the tf-idfs computed in scikit-learn's `TfidfTransformer` and `TfidfVectorizer` differ slightly from the standard textbook notation that defines the idf as

$$\text{idf}(t) = \log \frac{n}{1+\text{df}(t)}.$$

In the `TfidfTransformer` and `TfidfVectorizer` with `smooth_idf=False`, the "1" count is added to the idf instead of the idf's denominator:

$$\text{idf}(t) = \log \frac{n}{\text{df}(t)} + 1$$

This normalization is implemented by the `TfidfTransformer` class:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> transformer = TfidfTransformer(smooth_idf=False)
>>> transformer
TfidfTransformer(smooth_idf=False)
```

Again please see the reference documentation for the details on all the parameters.

Numeric example of a tf-idf matrix

6.2.3.6. Decoding text files

Text is made of characters, but files are made of bytes. These bytes represent characters according to some *encoding*. To work with text files in Python, their bytes must be *decoded* to a character set called Unicode. Common encodings are ASCII, Latin-1 (Western Europe), KOI8-R (Russian) and the universal encodings UTF-8 and UTF-16. Many others exist.

Note

An encoding can also be called a 'character set', but this term is less accurate: several encodings can exist for a single character set.

The text feature extractors in scikit-learn know how to decode text files, but only if you tell them what encoding the files are in. The `CountVectorizer` takes an `encoding` parameter for this purpos For modern text files, the correct encoding is probably UTF-8, which is therefore the default (`encoding="utf-8"`).

If the text you are loading is not actually encoded with UTF-8, however, you will get a `UnicodeDecodeError` . The vectorizers can be told to be silent about decoding errors by setting the `decode_error` parameter to either `"ignore"` or `"replace"` . See the documentation for the Pytho function `bytes.decode` for more details (type `help(bytes.decode)` at the Python prompt).

Troubleshooting decoding text

6.2.3.7. Applications and examples

The bag of words representation is quite simplistic but surprisingly useful in practice.

In particular in a **supervised setting** it can be successfully combined with fast and scalable linear models to train **document classifiers**, for instance:

- Classification of text documents using sparse features

In an **unsupervised setting** it can be used to group similar documents together by applying clustering algorithms such as K-means:

- Clustering text documents using k-means

Finally it is possible to discover the main topics of a corpus by relaxing the hard assignment constraint of clustering, for instance by using Non-negative matrix factorization (NMF or NNMF):

- Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation

6.2.3.8. Limitations of the Bag of Words representation

A collection of unigrams (what bag of words is) cannot capture phrases and multi-word expressio effectively disregarding any word order dependence. Additionally, the bag of words model doesn' account for potential misspellings or word derivations.

N-grams to the rescue! Instead of building a simple collection of unigrams (n=1), one might prefe collection of bigrams (n=2), where occurrences of pairs of consecutive words are counted.

One might alternatively consider a collection of character n-grams, a representation resilient agai misspellings and derivations.

For example, let's say we're dealing with a corpus of two documents: `['words', 'wprds']`. The second document contains a misspelling of the word 'words'. A simple bag of words representati would consider these two as very distinct documents, differing in both of the two possible feature A character 2-gram representation, however, would find the documents matching in 4 out of 8 features, which may help the preferred classifier decide better:

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(2, 2))
>>> counts = ngram_vectorizer.fit_transform(['words', 'wprds'])
>>> ngram_vectorizer.get_feature_names_out()
array([' w', 'ds', 'or', 'pr', 'rd', 's ', 'wo', 'wp'], ...)
>>> counts.toarray().astype(int)
array([[1, 1, 1, 0, 1, 1, 1, 0],
       [1, 1, 0, 1, 1, 1, 0, 1]])
```

In the above example, `char_wb` analyzer is used, which creates n-grams only from characters insid word boundaries (padded with space on each side). The `char` analyzer, alternatively, creates n-grams that span across words:

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(5, 5))
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
<Compressed Sparse...dtype 'int64'
  with 4 stored elements and shape (1, 4)>

>>> ngram_vectorizer.get_feature_names_out()
array([' fox ', ' jump', 'jumpy', 'umpy '], ...)

>>> ngram_vectorizer = CountVectorizer(analyzer='char', ngram_range=(5, 5))
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
<Compressed Sparse...dtype 'int64'
  with 5 stored elements and shape (1, 5)>
>>> ngram_vectorizer.get_feature_names_out()
array(['jumpy', 'mpy f', 'py fo', 'umpy ', 'y fox'], ...)
```

The word boundaries-aware variant `char_wb` is especially interesting for languages that use white spaces for word separation as it generates significantly less noisy features than the raw `char` varia in that case. For such languages it can increase both the predictive accuracy and convergence spe of classifiers trained using such features while retaining the robustness with regards to misspelling and word derivations.

While some local positioning information can be preserved by extracting n-grams instead of individual words, bag of words and bag of n-grams destroy most of the inner structure of the document and hence most of the meaning carried by that internal structure.

In order to address the wider task of Natural Language Understanding, the local structure of sentences and paragraphs should thus be taken into account. Many such models will thus be caste as “Structured output” problems which are currently outside of the scope of scikit-learn.

6.2.3.9. Vectorizing a large text corpus with the hashing trick

The above vectorization scheme is simple but the fact that it holds an **in- memory mapping fron the string tokens to the integer feature indices** (the `vocabulary_` attribute) causes several **problems when dealing with large datasets**:

- the larger the corpus, the larger the vocabulary will grow and hence the memory use too,
- fitting requires the allocation of intermediate data structures of size proportional to that of th original dataset.
- building the word-mapping requires a full pass over the dataset hence it is not possible to fit text classifiers in a strictly online manner.
- pickling and un-pickling vectorizers with a large `vocabulary_` can be very slow (typically muc slower than pickling / un-pickling flat data structures such as a NumPy array of the same size)
- it is not easily possible to split the vectorization work into concurrent sub tasks as the `vocabulary_` attribute would have to be a shared state with a fine grained synchronization barrier: the mapping from token string to feature index is dependent on ordering of the first occurrence of each token hence would have to be shared, potentially harming the concurrent workers’ performance to the point of making them slower than the sequential variant.

It is possible to overcome those limitations by combining the “hashing trick” (Feature hashing) implemented by the `FeatureHasher` class and the text preprocessing and tokenization features of the `CountVectorizer`.

This combination is implementing in `HashingVectorizer` , a transformer class that is mostly API compatible with `CountVectorizer` . `HashingVectorizer` is stateless, meaning that you don't have t call `fit` on it:

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> hv = HashingVectorizer(n_features=10)
>>> hv.transform(corpus)
<Compressed Sparse...dtype 'float64'
  with 16 stored elements and shape (4, 10)>
```

You can see that 16 non-zero feature tokens were extracted in the vector output: this is less than t 19 non-zeros extracted previously by the `CountVectorizer` on the same toy corpus. The discrepar comes from hash function collisions because of the low value of the `n_features` parameter.

In a real world setting, the `n_features` parameter can be left to its default value of `2 ** 20` (roughly one million possible features). If memory or downstream models size is an issue selecting lower value such as `2 ** 18` might help without introducing too many additional collisions on typical text classification tasks.

Note that the dimensionality does not affect the CPU training time of algorithms which operate or CSR matrices (`LinearSVC(dual=True)` , `Perceptron` , `SGDClassifier` , `PassiveAggressive`) but it do for algorithms that work with CSC matrices (`LinearSVC(dual=False)` , `Lasso()` , etc.).

Let's try again with the default setting:

```
>>> hv = HashingVectorizer()
>>> hv.transform(corpus)
<Compressed Sparse...dtype 'float64'
  with 19 stored elements and shape (4, 1048576)>
```

We no longer get the collisions, but this comes at the expense of a much larger dimensionality of the output space. Of course, other terms than the 19 used here might still collide with each other.

The `HashingVectorizer` also comes with the following limitations:

- it is not possible to invert the model (no `inverse_transform` method), nor to access the origi string representation of the features, because of the one-way nature of the hash function that performs the mapping.
- it does not provide IDF weighting as that would introduce statefulness in the model. A `TfidfTransformer` can be appended to it in a pipeline if required.

Performing out-of-core scaling with HashingVectorizer

6.2.3.10. Customizing the vectorizer classes

It is possible to customize the behavior by passing a callable to the vectorizer constructor:

```
>>> def my_tokenizer(s):
...     return s.split()
...
>>> vectorizer = CountVectorizer(tokenizer=my_tokenizer)
>>> vectorizer.build_analyzer()(u"Some... punctuation!") == (
...     ['some...', 'punctuation!'])
True
```

In particular we name:

- `preprocessor` : a callable that takes an entire document as input (as a single string), and retur a possibly transformed version of the document, still as an entire string. This can be used to remove HTML tags, lowercase the entire document, etc.
- `tokenizer` : a callable that takes the output from the preprocessor and splits it into tokens, th returns a list of these.
- `analyzer` : a callable that replaces the preprocessor and tokenizer. The default analyzers all ca the preprocessor and tokenizer, but custom analyzers will skip this. N-gram extraction and sto word filtering take place at the analyzer level, so a custom analyzer may have to reproduce these steps.

(Lucene users might recognize these names, but be aware that scikit-learn concepts may not map one-to-one onto Lucene concepts.)

To make the preprocessor, tokenizer and analyzers aware of the model parameters it is possible tc derive from the class and override the `build_preprocessor` , `build_tokenizer` and `build_analyze` factory methods instead of passing custom functions.

Tips and tricks

6.2.4. Image feature extraction

6.2.4.1. Patch extraction

The `extract_patches_2d` function extracts patches from an image stored as a two-dimensional array, or three-dimensional with color information along the third axis. For rebuilding an image fro all its patches, use `reconstruct_from_patches_2d` . For example let us generate a 4x4 pixel picture with 3 color channels (e.g. in RGB format):

```
>>> import numpy as np
>>> from sklearn.feature_extraction import image

>>> one_image = np.arange(4 * 4 * 3).reshape((4, 4, 3))
>>> one_image[:, :, 0]  # R channel of a fake RGB picture
array([[ 0,  3,  6,  9],
       [12, 15, 18, 21],
       [24, 27, 30, 33],
       [36, 39, 42, 45]])

>>> patches = image.extract_patches_2d(one_image, (2, 2), max_patches=2,
...     random_state=0)
>>> patches.shape
(2, 2, 2, 3)
>>> patches[:, :, :, 0]
array([[[ 0,  3],
        [12, 15]],

       [[15, 18],
        [27, 30]]])
>>> patches = image.extract_patches_2d(one_image, (2, 2))
>>> patches.shape
(9, 2, 2, 3)
>>> patches[4, :, :, 0]
array([[15, 18],
       [27, 30]])
```

Let us now try to reconstruct the original image from the patches by averaging on overlapping areas:

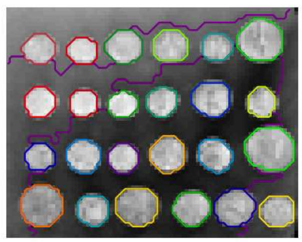
```
>>> reconstructed = image.reconstruct_from_patches_2d(patches, (4, 4, 3))
>>> np.testing.assert_array_equal(one_image, reconstructed)
```

The `PatchExtractor` class works in the same way as `extract_patches_2d` , only it supports multipl images as input. It is implemented as a scikit-learn transformer, so it can be used in pipelines. See

```
>>> five_images = np.arange(5 * 4 * 4 * 3).reshape(5, 4, 4, 3)
>>> patches = image.PatchExtractor(patch_size=(2, 2)).transform(five_images)
>>> patches.shape
(45, 2, 2, 3)
```

6.2.4.2. Connectivity graph of an image

Several estimators in the scikit-learn can use connectivity information between features or sample For instance Ward clustering (Hierarchical clustering) can cluster together only neighboring pixels an image, thus forming contiguous patches:



Previous
6.1. Pipelines and composite estimators

Next
6.3. Preprocessing data

For this purpose, the estimators use a 'connectivity' matrix, giving which samples are connected.