6.1. Pipelines and composite estimators

Transformers are usually combined with classifiers, regressors or other estimators to build a composite estimator. The most common tool is a Pipeline. Pipeline is often used in combination with FeatureUnion which concatenates the output of transformers into a composite feature space. TransformedTargetRegressor deals with transforming the target (i.e. log-transform y). In contrast, Pipelines only transform the observed data (X).

6.1.1. Pipeline: chaining estimators

`Pipeline` can be used to chain multiple estimators into one. This is useful as there is often a fixed sequence of steps in processing the data, for example feature selection, normalization and classification. `Pipeline` serves multiple purposes here:

Convenience and encapsulation
You only have to call fit and predict once on your data to fit a whole sequence of estimators.

Joint parameter selection
You can grid search over parameters of all estimators in the pipeline at once.

Safety
Pipelines help avoid leaking statistics from your test data into the trained model in cross-validation, by ensuring that the same samples are used to train the transformers and predictors.

All estimators in a pipeline, except the last one, must be transformers (i.e. must have a transform method). The last estimator may be any type (transformer, classifier, etc.).

6.1.1.1. Usage

Construction

The `Pipeline` is built using a list of `(key, value)` pairs, where the `key` is a string containing the name you want to give this step and `value` is an estimator object:

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.svm import SVC
>>> from sklearn.decomposition import PCA
>>> estimators = [('reduce_dim', PCA()), ('clf', SVC())]
>>> pipe = Pipeline(estimators)
>>> pipe
Pipeline(steps=[('reduce_dim', PCA()), ('clf', SVC())])
```

The utility function `make_pipeline` is a shorthand for constructing pipelines; it takes a variable number of estimators and returns a pipeline, filling in the names automatically:

```
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.naive_bayes import MultinomialNB
>>> from sklearn.preprocessing import Binarizer
>>> make_pipeline(Binarizer(), MultinomialNB())
Pipeline(steps=[('binarizer', Binarizer()), ('multinomialnb', MultinomialNB())])
```

Accessing steps

The estimators of a pipeline are stored as a list in the `steps` attribute, but can be accessed by index or name by indexing (with `[idx]`) the Pipeline:

```
>>> pipe.steps[0]
('reduce_dim', PCA())
>>> pipe[0]
PCA()
>>> pipe['reduce_dim']
PCA()
```

Pipeline's `named_steps` attribute allows accessing steps by name with tab completion in interactive environments:

```
>>> pipe.named_steps.reduce_dim is pipe['reduce_dim']
True
```

A sub-pipeline can also be extracted using the slicing notation commonly used for Python Sequences such as lists or strings (although only a step of 1 is permitted). This is convenient for performing only some of the transformations (or their inverse):

Toggle Menu

```
>>> pipe[:1]
Pipeline(steps=[('reduce_dim', PCA())])
>>> pipe[-1:]
Pipeline(steps=[('clf', SVC())])
```

Nested parameters

Parameters of the estimators in the pipeline can be accessed using the `<estimator>__<parameter>` syntax:

```
>>> pipe.set_params(clf__C=10)
Pipeline(steps=[('reduce_dim', PCA()), ('clf', SVC(C=10))])
```

This is particularly important for doing grid searches:

```
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = dict(reduce_dim__n_components=[2, 5, 10],
...                   clf__C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

Individual steps may also be replaced as parameters, and non-final steps may be ignored by setting them to `'passthrough'`:

```
>>> from sklearn.linear_model import LogisticRegression
>>> param_grid = dict(reduce_dim=['passthrough', PCA(5), PCA(10)],
...                   clf=[SVC(), LogisticRegression()],
...                   clf__C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

The estimators of the pipeline can be retrieved by index:

```
>>> pipe[0]
PCA()
```

or by name:

```
>>> pipe['reduce_dim']
PCA()
```

To enable model inspection, `Pipeline` has a `get_feature_names_out()` method, just like all transformers. You can use pipeline slicing to get the feature names going into each step:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectKBest
>>> iris = load_iris()
>>> pipe = Pipeline(steps=[
...    ('select', SelectKBest(k=2)),
...    ('clf', LogisticRegression())])
>>> pipe.fit(iris.data, iris.target)
Pipeline(steps=[('select', SelectKBest(...)), ('clf', LogisticRegression(...))])
>>> pipe[:-1].get_feature_names_out()
array(['x2', 'x3'], ...)
```

You can also provide custom feature names for the input data using `get_feature_names_out`:

```
>>> pipe[:-1].get_feature_names_out(iris.feature_names)
array(['petal length (cm)', 'petal width (cm)'], ...)
```

Examples:

- Pipeline ANOVA SVM
- Sample pipeline for text feature extraction and evaluation
- Pipelining: chaining a PCA and a logistic regression
- Explicit feature map approximation for RBF kernels
- SVM-Anova: SVM with univariate feature selection
- Selecting dimensionality reduction with Pipeline and GridSearchCV
- Displaying Pipelines

See Also:

- Composite estimators and parameter spaces

Toggle Menu

6.1.1.2. Notes

Calling `fit` on the pipeline is the same as calling `fit` on each estimator in turn, `transform` the input and pass it on to the next step. The pipeline has all the methods that the last estimator in the pipeline has, i.e. if the last estimator is a classifier, the `Pipeline` can be used as a classifier. If the last estimator is a transformer, again, so is the pipeline.

6.1.1.3. Caching transformers: avoid repeated computation

Fitting transformers may be computationally expensive. With its `memory` parameter set, `Pipeline` will cache each transformer after calling `fit`. This feature is used to avoid computing the fit transformers within a pipeline if the parameters and input data are identical. A typical example is the case of a grid search in which the transformers can be fitted only once and reused for each configuration.

The parameter `memory` is needed in order to cache the transformers. `memory` can be either a string containing the directory where to cache the transformers or a joblib.Memory object:

```
>>> from tempfile import mkdtemp
>>> from shutil import rmtree
>>> from sklearn.decomposition import PCA
>>> from sklearn.svm import SVC
>>> from sklearn.pipeline import Pipeline
>>> estimators = [('reduce_dim', PCA()), ('clf', SVC())]
>>> cachedir = mkdtemp()
>>> pipe = Pipeline(estimators, memory=cachedir)
>>> pipe
Pipeline(memory=...,
         steps=[('reduce_dim', PCA()), ('clf', SVC())])
>>> # Clear the cache directory when you don't need it anymore
>>> rmtree(cachedir)
```

Warning: Side effect of caching transformers
Using a `Pipeline` without cache enabled, it is possible to inspect the original instance such as:

```
>>> from sklearn.datasets import load_digits
>>> X_digits, y_digits = load_digits(return_X_y=True)
>>> pca1 = PCA()
>>> svm1 = SVC()
>>> pipe = Pipeline([('reduce_dim', pca1), ('clf', svm1)])
>>> pipe.fit(X_digits, y_digits)
Pipeline(steps=[('reduce_dim', PCA()), ('clf', SVC())])
>>> # The pca instance can be inspected directly
>>> print(pca1.components_)
    [[-1.77484909e-19  ... 4.07058917e-18]]
```

Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. In following example, accessing the `PCA` instance `pca2` will raise an `AttributeError` since `pca2` will be an unfitted transformer. Instead, use the attribute `named_steps` to inspect estimators within the pipeline:

```
>>> cachedir = mkdtemp()
>>> pca2 = PCA()
>>> svm2 = SVC()
>>> cached_pipe = Pipeline([('reduce_dim', pca2), ('clf', svm2)],
...                        memory=cachedir)
>>> cached_pipe.fit(X_digits, y_digits)
Pipeline(memory=...,
         steps=[('reduce_dim', PCA()), ('clf', SVC())])
>>> print(cached_pipe.named_steps['reduce_dim'].components_)
    [[-1.77484909e-19  ... 4.07058917e-18]]
>>> # Remove the cache directory
>>> rmtree(cachedir)
```

Examples:

- Selecting dimensionality reduction with Pipeline and GridSearchCV

6.1.2. Transforming target in regression

`TransformedTargetRegressor` transforms the targets `y` before fitting a regression model. The predictions are mapped back to the original space via an inverse transform. It takes as an argument the regressor that will be used for prediction, and the transformer that will be applied to the target variable:

Toggle Menu

```
>>> import numpy as np
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.compose import TransformedTargetRegressor
>>> from sklearn.preprocessing import QuantileTransformer
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.model_selection import train_test_split
>>> X, y = fetch_california_housing(return_X_y=True)
>>> X, y = X[:2000, :], y[:2000]  # select a subset of data
>>> transformer = QuantileTransformer(output_distribution='normal')
>>> regressor = LinearRegression()
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                   transformer=transformer)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: 0.61
>>> raw_target_regr = LinearRegression().fit(X_train, y_train)
>>> print('R2 score: {0:.2f}'.format(raw_target_regr.score(X_test, y_test)))
R2 score: 0.59
```

For simple transformations, instead of a Transformer object, a pair of functions can be passed, defining the transformation and its inverse mapping:

```
>>> def func(x):
...     return np.log(x)
>>> def inverse_func(x):
...     return np.exp(x)
```

Subsequently, the object is created as:

```
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                   func=func,
...                                   inverse_func=inverse_func)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: 0.51
```

By default, the provided functions are checked at each fit to be the inverse of each other. However, it is possible to bypass this checking by setting `check_inverse` to `False`:

```
>>> def inverse_func(x):
...     return x
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                   func=func,
...                                   inverse_func=inverse_func,
...                                   check_inverse=False)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {0:.2f}'.format(regr.score(X_test, y_test)))
R2 score: -1.57
```

Note: The transformation can be triggered by setting either `transformer` or the pair of functions `func` and `inverse_func`. However, setting both options will raise an error.

Examples:

- Effect of transforming the targets in regression model

6.1.3. FeatureUnion: composite feature spaces

`FeatureUnion` combines several transformer objects into a new transformer that combines their output. A `FeatureUnion` takes a list of transformer objects. During fitting, each of these is fit to the data independently. The transformers are applied in parallel, and the feature matrices they output are concatenated side-by-side into a larger matrix.

When you want to apply different transformations to each field of the data, see the related class `ColumnTransformer` (see user guide).

`FeatureUnion` serves the same purposes as `Pipeline` - convenience and joint parameter estimation and validation.

`FeatureUnion` and `Pipeline` can be combined to create complex models.

(A `FeatureUnion` has no way of checking whether two transformers might produce identical features. It only produces a union when the feature sets are disjoint, and making sure they are is the caller's responsibility.)

Toggle Menu

6.1.3.1. Usage

A `FeatureUnion` is built using a list of `(key, value)` pairs, where the `key` is the name you want to give to a given transformation (an arbitrary string; it only serves as an identifier) and `value` is an estimator object:

```
>>> from sklearn.pipeline import FeatureUnion
>>> from sklearn.decomposition import PCA
>>> from sklearn.decomposition import KernelPCA
>>> estimators = [('linear_pca', PCA()), ('kernel_pca', KernelPCA())]
>>> combined = FeatureUnion(estimators)
>>> combined
FeatureUnion(transformer_list=[('linear_pca', PCA()),
                               ('kernel_pca', KernelPCA())])
```

Like pipelines, feature unions have a shorthand constructor called `make_union` that does not require explicit naming of the components.

Like `Pipeline`, individual steps may be replaced using `set_params`, and ignored by setting to `'drop'`:

```
>>> combined.set_params(kernel_pca='drop')
FeatureUnion(transformer_list=[('linear_pca', PCA()),
                               ('kernel_pca', 'drop')])
```

Examples:

- Concatenating multiple feature extraction methods

6.1.4. ColumnTransformer for heterogeneous data

Many datasets contain features of different types, say text, floats, and dates, where each type of feature requires separate preprocessing or feature extraction steps. Often it is easiest to preprocess data before applying scikit-learn methods, for example using pandas. Processing your data before passing it to scikit-learn might be problematic for one of the following reasons:

1. Incorporating statistics from test data into the preprocessors makes cross-validation scores unreliable (known as *data leakage*), for example in the case of scalers or imputing missing values.
2. You may want to include the parameters of the preprocessors in a parameter search.

The `ColumnTransformer` helps performing different transformations for different columns of the data, within a `Pipeline` that is safe from data leakage and that can be parametrized. `ColumnTransformer` works on arrays, sparse matrices, and pandas DataFrames.

To each column, a different transformation can be applied, such as preprocessing or a specific feature extraction method:

```
>>> import pandas as pd
>>> X = pd.DataFrame(
...     {'city': ['London', 'London', 'Paris', 'Sallisaw'],
...      'title': ["His Last Bow", "How Watson Learned the Trick",
...                "A Moveable Feast", "The Grapes of Wrath"],
...      'expert_rating': [5, 3, 4, 5],
...      'user_rating': [4, 5, 4, 3]})
```

For this data, we might want to encode the `'city'` column as a categorical variable using `OneHotEncoder` but apply a `CountVectorizer` to the `'title'` column. As we might use multiple feature extraction methods on the same column, we give each transformer a unique name, say `'city_category'` and `'title_bow'`. By default, the remaining rating columns are ignored (`remainder='drop'`):

Toggle Menu

```
>>> from sklearn.compose import ColumnTransformer
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from sklearn.preprocessing import OneHotEncoder
>>> column_trans = ColumnTransformer(
...     [('categories', OneHotEncoder(dtype='int'), ['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder='drop', verbose_feature_names_out=False)

>>> column_trans.fit(X)
ColumnTransformer(transformers=[('categories', OneHotEncoder(dtype='int'),
                                 ['city']),
                                ('title_bow', CountVectorizer(), 'title')],
                  verbose_feature_names_out=False)

>>> column_trans.get_feature_names_out()
array(['city_London', 'city_Paris', 'city_Sallisaw', 'bow', 'feast',
'grapes', 'his', 'how', 'last', 'learned', 'moveable', 'of', 'the',
 'trick', 'watson', 'wrath'], ...)

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

In the above example, the `CountVectorizer` expects a 1D array as input and therefore the columns were specified as a string (`'title'`). However, `OneHotEncoder` as most of other transformers expects 2D data, therefore in that case you need to specify the column as a list of strings (`['city']`).

Apart from a scalar or a single item list, the column selection can be specified as a list of multiple items, an integer array, a slice, a boolean mask, or with a `make_column_selector`. The `make_column_selector` is used to select columns based on data type or column name:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.compose import make_column_selector
>>> ct = ColumnTransformer([
...       ('scale', StandardScaler(),
...       make_column_selector(dtype_include=np.number)),
...       ('onehot',
...       OneHotEncoder(),
...       make_column_selector(pattern='city', dtype_include=object))])
>>> ct.fit_transform(X)
array([[ 0.904...,  0.      ,  1. ,  0. ,  0. ],
       [-1.507...,  1.414...,  1. ,  0. ,  0. ],
       [-0.301...,  0.      ,  0. ,  1. ,  0. ],
       [ 0.904..., -1.414...,  0. ,  0. ,  1. ]])
```

Strings can reference columns if the input is a DataFrame, integers are always interpreted as the positional columns.

We can keep the remaining rating columns by setting `remainder='passthrough'`. The values are appended to the end of the transformation:

```
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(dtype='int'),['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder='passthrough')

>>> column_trans.fit_transform(X)
array([[1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 5, 4],
       [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 3, 5],
       [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 4, 4],
       [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 5, 3]]...)
```

The `remainder` parameter can be set to an estimator to transform the remaining rating columns. The transformed values are appended to the end of the transformation:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(), ['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder=MinMaxScaler())

>>> column_trans.fit_transform(X)[:, -2:]
array([[1. , 0.5],
       [0. , 1. ],
       [0.5, 0.5],
       [1. , 0. ]])
```

Toggle Menu

The `make_column_transformer` function is available to more easily create a `ColumnTransformer` object. Specifically, the names will be given automatically. The equivalent for the above example would be:

```
>>> from sklearn.compose import make_column_transformer
>>> column_trans = make_column_transformer(
...     (OneHotEncoder(), ['city']),
...     (CountVectorizer(), 'title'),
...     remainder=MinMaxScaler())
>>> column_trans
ColumnTransformer(remainder=MinMaxScaler(),
                  transformers=[('onehotencoder', OneHotEncoder(), ['city']),
                                ('countvectorizer', CountVectorizer(),
                                 'title')])
```

If `ColumnTransformer` is fitted with a dataframe and the dataframe only has string column names, then transforming a dataframe will use the column names to select the columns:

```
>>> ct = ColumnTransformer(
...          [("scale", StandardScaler(), ["expert_rating"])]).fit(X)
>>> X_new = pd.DataFrame({"expert_rating": [5, 6, 1],
...                       "ignored_new_col": [1.2, 0.3, -0.1]})
>>> ct.transform(X_new)
array([[ 0.9...],
       [ 2.1...],
       [-3.9...]])
```

6.1.5. Visualizing Composite Estimators

Estimators are displayed with an HTML representation when shown in a jupyter notebook. This is useful to diagnose or visualize a Pipeline with many estimators. This visualization is activated by default:

```
>>> column_trans
```

It can be deactivated by setting the `display` option in `set_config` to ‘text’:

```
>>> from sklearn import set_config
>>> set_config(display='text')
>>> # displays text representation in a jupyter context
>>> column_trans
```

An example of the HTML output can be seen in the **HTML representation of Pipeline** section of Column Transformer with Mixed Types. As an alternative, the HTML can be written to a file using `estimator_html_repr`:

```
>>> from sklearn.utils import estimator_html_repr
>>> with open('my_estimator.html', 'w') as f:
...     f.write(estimator_html_repr(clf))
```

Examples:

- Column Transformer with Heterogeneous Data Sources
- Column Transformer with Mixed Types

Toggle Menu