Note: Go to the end to download the full example code or to run this example in your browser via JupyterLite or Binder

Column Transformer with Mixed Types

This example illustrates how to apply different preprocessing and feature extraction pipelines to different subsets of features, using `ColumnTransformer`. This is particularly handy for the case of datasets that contain heterogeneous data types, since we may want to scale the numeric features and one-hot encode the categorical ones.

In this example, the numeric data is standard-scaled after mean-imputation. The categorical data is one-hot encoded via `OneHotEncoder`, which creates a new category for missing values. We further reduce the dimensionality by selecting categories using a chi-squared test.

In addition, we show two different ways to dispatch the columns to the particular pre-processor: by column names and by column data types.

Finally, the preprocessing pipeline is integrated in a full prediction pipeline using `Pipeline`, together with a simple classification model.

```
# Author: Pedro Morales <part.morales@gmail.com>
#
# License: BSD 3 clause
```

```
import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.datasets import fetch_openml
from sklearn.feature_selection import SelectPercentile, chi2
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler

np.random.seed(0)
```

Load data from https://www.openml.org/d/40945

```
X, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)

# Alternatively X and y can be obtained directly from the frame attribute:
# X = titanic.frame.drop('survived', axis=1)
# y = titanic.frame['survived']
```

Use `ColumnTransformer` by selecting column by names

We will train our classifier with the following features:

Numeric Features:

- `age`: float;
- `fare`: float.

Categorical Features:

- `embarked`: categories encoded as strings `{'C', 'S', 'Q'}`;
- `sex`: categories encoded as strings `{'female', 'male'}`;
- `pclass`: ordinal integers `{1, 2, 3}`.

We create the preprocessing pipelines for both numeric and categorical data. Note that `pclass` could either be treated as a categorical or numeric feature.

```
numeric_features = ["age", "fare"]
numeric_transformer = Pipeline(
    steps=[("imputer", SimpleImputer(strategy="median")), ("scaler", StandardScaler())]
)

categorical_features = ["embarked", "sex", "pclass"]
categorical_transformer = Pipeline(
    steps=[
        ("encoder", OneHotEncoder(handle_unknown="ignore")),
        ("selector", SelectPercentile(chi2, percentile=50)),
    ]
)
preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, numeric_features),
        ("cat", categorical_transformer, categorical_features),
    ]
)
```

Append classifier to preprocessing pipeline. Now we have a full prediction pipeline.

Toggle Menu

```
clf = Pipeline(
    steps=[("preprocessor", preprocessor), ("classifier", LogisticRegression())]
)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

Out:
```
model score: 0.798
```

HTML representation of `Pipeline` (display diagram)

When the `Pipeline` is printed out in a jupyter notebook an HTML representation of the estimator is displayed:

```
clf
```

```
▾ Pipeline
                                    ['age', 'fare']),
                                   ('cat',
                                    Pipeline(steps=[('encoder',
                                                     OneHotEncoder(handle_unknown='ignore')),
                                                    ('selector',
                                                     SelectPercentile(percentile=50,
                                                                      score_func=<function chi2 at
0x7fb8b6ad4b80>))]),
                                    ['embarked', 'sex',
                                     'pclass'])])),

  ▾ preprocessor: ColumnTransformer
  ColumnTransformer(transformers=[('num',
                                   Pipeline(steps=[('imputer',
                                                    SimpleImputer(strategy='median')),
                                                   ('scaler', StandardScaler())]),
                                   ['age', 'fare']),
                                  ('cat',
                                   Pipeline(steps=[('encoder',
                                                    OneHotEncoder(handle_unknown='ignore')),
                                                   ('selector',
                                                    SelectPercentile(percentile=50,

  ▾ num                                  ▾ cat
  ['age', 'fare']                        ['embarked', 'sex', 'pclass']
    ▾ SimpleImputer                        ▾ OneHotEncoder
    SimpleImputer(strategy='median')       OneHotEncoder(handle_unknown='ignore')
      ▾ StandardScaler                     ▾ SelectPercentile
      StandardScaler()                     SelectPercentile(percentile=50, score_func=<function chi2 at
                                           0x7fb8b6ad4b80>)

                        ▾ LogisticRegression
                        LogisticRegression()
```

Use `ColumnTransformer` by selecting column by data types

When dealing with a cleaned dataset, the preprocessing can be automatic by using the data types of the column to decide whether to treat a column as a numerical or categorical feature. **`sklearn.compose.make_column_selector`** gives this possibility. First, let's only select a subset of columns to simplify our example.

```
subset_feature = ["embarked", "sex", "pclass", "age", "fare"]
X_train, X_test = X_train[subset_feature], X_test[subset_feature]
```

Then, we introspect the information regarding each column data type.

```
X_train.info()
```

Toggle Menu

Out:
```
<class 'pandas.core.frame.DataFrame'>
Index: 1047 entries, 1118 to 684
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   embarked  1045 non-null   category
 1   sex       1047 non-null   category
 2   pclass    1047 non-null   int64
 3   age       841 non-null    float64
 4   fare      1046 non-null   float64
dtypes: category(2), float64(2), int64(1)
memory usage: 35.0 KB
```

We can observe that the `embarked` and `sex` columns were tagged as `category` columns when loading the data with `fetch_openml`. Therefore, we can use this information to dispatch the categorical columns to the `categorical_transformer` and the remaining columns to the `numerical_transformer`.

Note: In practice, you will have to handle yourself the column data type. If you want some columns to be considered as `category`, you will have to convert them into categorical columns. If you are using pandas, you can refer to their documentation regarding Categorical data.

```
from sklearn.compose import make_column_selector as selector

preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, selector(dtype_exclude="category")),
        ("cat", categorical_transformer, selector(dtype_include="category")),
    ]
)
clf = Pipeline(
    steps=[("preprocessor", preprocessor), ("classifier", LogisticRegression())]
)


clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
clf
```

Out:
```
model score: 0.798
```

```
Pipeline

Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num',
                                                  Pipeline(steps=[('imputer',
                                                                   SimpleImputer(strategy='median')),
                                                                  ('scaler',
                                                                   StandardScaler())]),
                                                  <sklearn.compose._column_transformer.make_column_selector object at
0x7fb881e94130>),
                                                 ('cat',
                                                  Pipeline(steps=[('encoder',

preprocessor: ColumnTransformer

ColumnTransformer(transformers=[('num',
                                 Pipeline(steps=[('imputer',
                                                  SimpleImputer(strategy='median')),
                                                 ('scaler', StandardScaler())]),
                                 <sklearn.compose._column_transformer.make_column_selector object at
0x7fb881e94130>),
                                ('cat',
                                 Pipeline(steps=[('encoder',
                                                  OneHotEncoder(handle_unknown='ignore')),
                                                 ('selector',

num
SimpleImputer
SimpleImputer(strategy='median')
StandardScaler
StandardScaler()

cat
<sklearn.compose._column_transformer.make_column_selector object at
0x7fb881e940d0>
OneHotEncoder
OneHotEncoder(handle_unknown='ignore')
SelectPercentile
SelectPercentile(percentile=50, score_func=<function chi2 at
0x7fb8b6ad4b80>)

LogisticRegression
LogisticRegression()
```

The resulting score is not exactly the same as the one from the previous pipeline because the dtype-based selector treats the `pclass` column as a numeric feature

Toggle Menu tegorical feature as previously:

```
selector(dtype_exclude="category")(X_train)
```

Out:
```
['pclass', 'age', 'fare']
```

```
selector(dtype_include="category")(X_train)
```

Out:
```
['embarked', 'sex']
```

Using the prediction pipeline in a grid search

Grid search can also be performed on the different preprocessing steps defined in the `ColumnTransformer` object, together with the classifier's hyperparameters as part of the `Pipeline`. We will search for both the imputer strategy of the numeric preprocessing and the regularization parameter of the logistic regression using **`RandomizedSearchCV`**. This hyperparameter search randomly selects a fixed number of parameter settings configured by `n_iter`. Alternatively, one can use **`GridSearchCV`** but the cartesian product of the parameter space will be evaluated.

```
param_grid = {
    "preprocessor__num__imputer__strategy": ["mean", "median"],
    "preprocessor__cat__selector__percentile": [10, 30, 50, 70],
    "classifier__C": [0.1, 1.0, 10, 100],
}

search_cv = RandomizedSearchCV(clf, param_grid, n_iter=10, random_state=0)
search_cv
```

```
▾ RandomizedSearchCV
                                                    ('s...
score_func=<function chi2 at 0x7fb8b6ad4b80>))]),
<sklearn.compose._column_transformer.make_column_selector object at 0x7fb881e940d0>)])),
                                                ('classifier',
                                                 LogisticRegression())]),
                   param_distributions={'classifier__C': [0.1, 1.0, 10, 100],
                                        'preprocessor__cat__selector__percentile': [10,
                                                                                    30,
▾ estimator: Pipeline
                                                  StandardScaler())]),
                                 <sklearn.compose._column_transformer.make_column_selector object at
0x7fb881e94130>),
                                ('cat',
                                 Pipeline(steps=[('encoder',
                                                  OneHotEncoder(handle_unknown='ignore')),
                                                 ('selector',
                                                  SelectPercentile(percentile=50,
                                                                   score_func=<function chi2 at
0x7fb8b6ad4b80>))]),
▾ preprocessor: ColumnTransformer
ColumnTransformer(transformers=[('num',
                                 Pipeline(steps=[('imputer',
                                                  SimpleImputer(strategy='median')),
                                                 ('scaler', StandardScaler())]),
                                 <sklearn.compose._column_transformer.make_column_selector object at
0x7fb881e94130>),
                                ('cat',
                                 Pipeline(steps=[('encoder',
                                                  OneHotEncoder(handle_unknown='ignore')),
                                                 ('selector',
▾ num
<sklearn.compose._column_transformer.make_column_selector object at 0x7fb881e94130>
▾ SimpleImputer
SimpleImputer(strategy='median')
▾ StandardScaler
StandardScaler()
▾ cat
<sklearn.compose._column_transformer.make_column_selector object at 0x7fb881e940d0>
▾ OneHotEncoder
OneHotEncoder(handle_unknown='ignore')
▾ SelectPercentile
SelectPercentile(percentile=50, score_func=
<function chi2 at 0x7fb8b6ad4b80>)
▾ LogisticRegression
LogisticRegression()
```

Calling 'fit' triggers the cross-validated search for the best hyper-parameters combination:

Toggle Menu

```
search_cv.fit(X_train, y_train)

print("Best params:")
print(search_cv.best_params_)
```

Out:
```
Best params:
{'preprocessor__num__imputer__strategy': 'mean', 'preprocessor__cat__selector__percentile': 30, 'classifier__C': 100}
```

The internal cross-validation scores obtained by those parameters is:

```
print(f"Internal CV score: {search_cv.best_score_:.3f}")
```

Out:
```
Internal CV score: 0.786
```

We can also introspect the top grid search results as a pandas dataframe:

```
import pandas as pd

cv_results = pd.DataFrame(search_cv.cv_results_)
cv_results = cv_results.sort_values("mean_test_score", ascending=False)
cv_results[
    [
        "mean_test_score",
        "std_test_score",
        "param_preprocessor__num__imputer__strategy",
        "param_preprocessor__cat__selector__percentile",
        "param_classifier__C",
    ]
].head(5)
```

	mean_test_score	std_test_score	param_preprocessor__num__imputer__strategy	param_preprocessor__cat__selector__percentile	param_classifier__C
7	0.786015	0.031020	mean	30	100
0	0.785063	0.030498	median	30	1.0
4	0.785063	0.030498	mean	10	10
2	0.785063	0.030498	mean	30	1.0
3	0.783149	0.030462	mean	30	0.1

The best hyper-parameters have be used to re-fit a final model on the full training set. We can evaluate that final model on held out test data that was not used for hyperparameter tuning.

```
print(
    "accuracy of the best model from randomized search: "
    f"{search_cv.score(X_test, y_test):.3f}"
)
```

Out:
```
accuracy of the best model from randomized search: 0.798
```

Total running time of the script: (0 minutes 1.207 seconds)

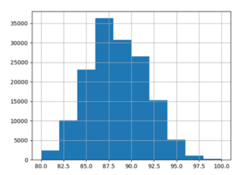
launch binder

launch lite

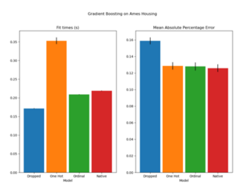
Download Jupyter notebook: plot_column_transformer_mixed_types.ipynb

Download Python source code: plot_column_transformer_mixed_types.py

Related examples

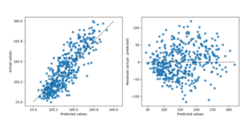


Comparing Target Encoder with Other Encoders

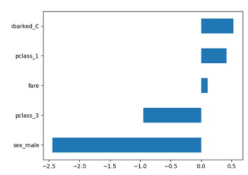


Categorical Feature Support in Gradient Boosting

Displaying Pipelines



Release Highlights for scikit-learn 1.2



Introducing the set_output API

Toggle Menu

Gallery generated by Sphinx-Gallery

Toggle Menu