Using ColumnTransformer to combine data processing steps

Create cohesive pipelines for processing data where different columns require different techniques

Allison Honold Feb 22, 2020 · 6 min read ★

This scikit-learn tool comes in extremely handy, but also has some quirks of its own. Today we'll be using it to transform data on ferry wait time for the Edmonds-Kingston route of the Washington State Ferries. (Thank you WSF for the data!). Full disclosure: we're just going to use a small portion of the data set today.

More full disclosure — a warning from scikit-learn: "**Warning:** The compose.ColumnTransformer class is experimental and the API is subject to change."

Washington State Ferry. Photo by Bryan Hanson on Unsplash

General Idea and Use

ColumnTransformers come in handy when you are creating a data pipeline where different columns need different transformations. Perhaps you have a combination of categorical and numeric features. Perhaps you want to use different imputation strategies to fill NaNs in different numeric columns. You could transform each column separately and then stitch them together, or you can use ColumnTransformer to do that work for you.

Here's a basic example. In this case, our input features are weekday (0–6 Monday-Sunday), hour (0–23), and maximum, average, and minimum daily temperature. I want to standard scale the temperature features and one hot encode the date features.

Assuming I have my input and target DataFrames (X_train, y_train) already loaded:

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline

# define column transformer and set n_jobs to use all cores
col_transformer = ColumnTransformer(
                    transformers=[
                        ('ss', StandardScaler(), ['max_temp',
                                                  'avg_temp',
                                                  'min_temp']),
                        ('ohe', OneHotEncoder(), ['weekday',
                                                  'hour'])
                    ],
                    remainder='drop',
                    n_jobs=-1
                    )
```

We are then ready to transform!

we get:

```
<465x30 sparse matrix of type '<class 'numpy.float64'>'
        with 2325 stored elements in Compressed Sparse Row format>
```

More likely, you'll add the ColumnTransformer as a step in your Pipeline:

```
lr = LinearRegression()

pipe = Pipeline([
            ("preprocessing", col_transformer),
            ("lr", lr)
       ])

pipe.fit(X_train, y_train)
```

And now your pipe is ready to make predictions! Or to be used in cross validation without leaking information across slices.

Note that we need to indicate the column in the format expected by the transformer. If the transformer expects a 2D array, pass a list of string columns (even if it is only one column — eg. `['col1']`). If the transformer expects a 1D array, pass just the string column name — eg. `'col1'`.

But things aren't always this easy — maybe your data set has null values and needs multiple transformations on the same column, you want a custom transformer, or you want to dig deeper into feature importances, maybe not all OneHotEncoder categories are practically guaranteed to be present in all data slices.

Tip 1: Use pipelines for any columns that need multiple transformations

The first time I used ColumnTransformer, I thought that it would perform the transformations in order, and that I could start with SimpleImputing my NaNs on whatever columns, then StandardScale(r) an overlapping subset of columns, then OneHotEncode another overlapping subset of columns, etc, etc. **I was wrong.** If you

```
# define transformers
si_0 = SimpleImputer(strategy='constant', fill_value=0)
ss = StandardScaler()
ohe = OneHotEncoder()

# define column groups with same processing
cat_vars = ['weekday', 'hour']
num_vars = ['max_temp', 'avg_temp', 'min_temp']

# set up pipelines for each column group
categorical_pipe = Pipeline([('si_0', si_0), ('ohe', ohe)])
numeric_pipe = Pipeline([('si_0', si_0), ('ss', ss)])

# set up columnTransformer
col_transformer = ColumnTransformer(
                    transformers=[
                        ('nums', numeric_pipe, num_vars),
                        ('cats', categorical_pipe, cat_vars)
                    ],
                    remainder='drop',
                    n_jobs=-1
                    )
```

Tip 2: Keep track of your column names

From the scikit-learn docs: “The order of the columns in the transformed feature matrix follows the order of how the columns are specified in the `transformers` list. Columns of the original feature matrix that are not specified are dropped from the resulting transformed feature matrix, unless specified in the `passthrough` keyword. Those columns specified with `passthrough` are added at the right to the output of the transformers.”

So for the examples above, the preprocessed array columns are:

```
['max_temp', 'avg_temp', 'min_temp, 'weekday_0', 'weekday_1',
'weekday_2', 'weekday_3', 'weekday_4', 'weekday_5', 'weekday_6',
'hour_0', 'hour_1', 'hour_2', 'hour_3', 'hour_4', 'hour_5',
'hour_6', 'hour_7', 'hour_8', 'hour_9', 'hour_10', 'hour_11',
'hour_12', 'hour_13', 'hour_14', 'hour_15', 'hour_16', 'hour_17',
'hour_18', 'hour_19', 'hour_20', 'hour_21', 'hour_22', 'hour_23']
```

```
col_transformer.named_transformers_['ohe'].get_feature_names()
```

Here, 'ohe' is the name of my transformer in the first example. Unfortunately, transformers that don't create more features/columns don't typically have this method, and ColumnTransformer relies on this attribute of its interior transformers. If you are using only transformers that have this method, then you can call `col_transformer.get_feature_names()` to easily get them all. I haven't had this opportunity yet, but we might at some point. Or maybe this column tracking functionality will be added to a future ColumnTransformer release.

Note: If you are using pipelines (like in tip #1), you'll need to dig a little deeper, and use the Pipeline attribute `named_steps`. In this case:

```
col_transformer.named_transformers_['cats'].named_steps['ohe']\
     .get_feature_names()
```

Tip 3: Feel free to create your own transformers

ColumnTransformer works with any transformer, so feel free to create your own. We're not going to go too deep into custom transformers today, but there is a caveat when using custom transformers with ColumnTransformer that I wanted to point out.

For our ferry project, we can extract the date features with a custom transformer:

```
from sklearn.base import TransformerMixin, BaseEstimator

class DateTransformer(TransformerMixin, BaseEstimator):
    """Extracts features from datetime column

    Returns:
      hour: hour
      day: Between 1 and the number of days in the month
      month: Between 1 and 12 inclusive.
      year: four-digit year
      weekday: day of the week as an integer. Mon=0 and Sun=6
   """
```

```
def transform(self, x, y=None):
        result = pd.DataFrame(x, columns=['date_hour'])
        result['hour'] = [dt.hour for dt in result['date_hour']]
        result['day'] = [dt.day for dt in result['date_hour']]
        result['month'] = [dt.month for dt in result['date_hour']]
        result['year'] = [dt.year for dt in result['date_hour']]
        result['weekday'] = [dt.weekday() for dt in
                             result['date_hour']]
        return result[['hour', 'day', 'month', 'year', 'weekday']]

def get_feature_names(self):
        return ['hour','day', 'month', 'year', 'weekday']
```

Note that ColumnTransformer “sends” the columns as a numpy array. To convert these timestamps from strings, I cast them as a pandas DataFrame (maybe not the most elegant solution).

Note that ColumnTransformer “sends” all of the specified columns to our transformer together. This means that you need to design your transformer to take and transform multiple columns at the same time, or make sure to send each column in a separate line of the ColumnTransformer. Since our custom transformer is only designed for process a single column, we would need to tailor our ColumnTransformer like this (assuming we want to re-use it in a situation with two datetime columns that we want to expand):

```
transformers=[('dates1', DateTransformer, ['start_date'])

ct = ColumnTransformer(
          transformers=[
              ('dates1', DateTransformer, ['start_date']),
              ('dates2', DateTransformer, ['end_date'])
          ])
```

Tip 4: Be proactive with “rare” categorical features or flags

The key here is that your model is expecting the same number of features in the training set, the testing set, and your production inputs.

If we have any rare categorical features that end up not present in each of these groups, the default OneHotEncoding settings are going to produce different numbers of columns for the different input sets.

columns will again be different after the preprocessing stage.

This can raise a couple of different errors including:

```
ValueError: Found unknown categories [0] in column 0 during
transform
```

and

```
ValueError: The features [0] have missing values in transform but
have no missing values in fit.
```

For the OneHotEncoding issue, you can list the categories when you initialize the ohe. If you had two categorical features with the first having categories 'one' and 'two', and the second having 'March', 'April', you could indicate this way:

```
OneHotEncoder(categories=[['one', 'two'], ['March', 'April']]).
```

For the SimpleImputer, you can not use a flag, drop the columns with NaN (if it is so few), adjust your train-test split (and ensure that your production inputs account for this difference), or create your own transformer that builds on SimpleImputer by adding a flag column regardless of the presence of NaNs.

This data prep step feels a little unsatisfying today, as we don't have any conclusions or fun facts derived from our data set (yet). But as you all know, this is an essential step on our way to predicting the ferry wait time (or whatever else you want to predict/classify/etc).

As always, check out the GitHub repo for the full code. Happy coding!

Photo by Patrick Robinson on Unsplash

Columntransformer Python Data Preparation Pipeline One Hot Encoder