Machine Learning

|Transformation Pipelines

Prof. Dr. Mostafa Elhosseini

https://youtube.com/drmelhosseini

Agenda

- Transformation Pipelines
- Pipeline class
- make_pipeline()
- ColumnTransformer
- make_column_selector()

Transformation Pipelines

- **Systematic way to assemble** several data *preprocessing and transformation steps* into a single, unified process.
- These pipelines streamline the workflow, **ensuring** that data transformations are **applied** consistently **during both the model training and the prediction** phases.
- They also help in **maintaining clean code and enhance reproducibility**.
- There are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the **Pipeline class** to help with such sequences of transformations

Transformation Pipelines

Usage:

- Once defined, you can treat the pipeline as a single entity:
- **Fitting**: When you call pipeline.fit(X_train, y_train), each transformer's fit and transform methods are invoked sequentially, with the output of each step passed as input to the next. The final estimator's fit method is called with the output of the last transformation step.

Transformation Pipelines

Usage:

- **Predicting**: Calling **pipeline.predict(X_test)** automatically applies all the transformers' transform methods **to the test data** (in the same sequence as defined) before invoking the predict method of the final estimator.
- **Pipeline as an Estimator**: In Scikit-Learn, a pipeline itself is treated as an estimator. This means you can use it anywhere you would use a regular estimator, including in cross-validation or as part of a larger pipeline.

Transformation Pipelines

- Here is a small pipeline for numerical attributes, which will first impute then scale the input features:

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
```

- The **Pipeline constructor** takes a **list of name/estimator pairs** (2-tuples) defining a sequence of steps. The **names can be anything** you like, as long as they are unique and **don't contain double underscores** (__). They will be useful later, when we discuss hyperparameter tuning.

Transformation Pipelines

- The **estimators must all be transformers** (i.e., they must have a **fit_transform() method**), **except for the last one**, which can be anything: a transformer, a predictor, or any other type of estimator.
- **If you don’t want to name the transformers**, you can use the **make_pipeline()** function instead; it takes transformers as positional arguments and creates a Pipeline using the **names of the transformers’ classes**, in lowercase and without underscores (e.g., "simpleimputer")

Transformation Pipelines

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

- If multiple transformers have the same name, an index is appended to their names (e.g., "foo-1", "foo-2", etc.).

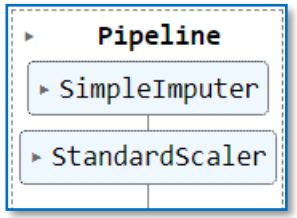
Tip

- In a Jupyter notebook, if you import sklearn and run sklearn.set_config(display="diagram"), all Scikit-Learn estimators will be **rendered as interactive diagrams**.
- This is particularly useful for visualizing pipelines.
- To visualize num_pipeline, run a cell with num_pipeline as the last line.
- Clicking an estimator will show more details.

```
from sklearn import set_config

set_config(display='diagram')

num_pipeline
```



Transformation Pipelines

- **When you call the pipeline's fit() method**, it calls fit_transform() sequentially on all the transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator, for which it just calls the fit() method.

Transformation Pipelines

- The pipeline exposes the same methods as the final estimator.
- In this example the last estimator is a StandardScaler, which is a transformer, so the pipeline also acts like a transformer. If you call the pipeline's **transform() method**, it will sequentially **apply all the transformations** to the data.
- If the last estimator were a predictor instead of a transformer, then the pipeline would have a **predict() method** rather than a transform() method. Calling it would sequentially apply all the transformations to the data and pass the result to the predictor's predict() method.

Transformation Pipelines

- Let's call the pipeline's fit_transform() method and look at the output's first two rows, rounded to two decimal places

```
housing_num_prepared = num_pipeline.fit_transform(housing_num)
housing_num_prepared[:2].round(2)
```

```
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.6 , -0.7 ,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

- As you saw earlier, if you want to recover a nice DataFrame, you can use the pipeline's **get_feature_names_out()** method

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared, columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index)
```

Transformation Pipelines

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared, columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index)
```

```
df_housing_num_prepared.head(2)
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
13096	-1.423037	1.013606	1.861119	0.311912	1.368167	0.137460	1.394812	-0.936491
14973	0.596394	-0.702103	0.907630	-0.308620	-0.435925	-0.693771	-0.373485	1.171942

Transformation Pipelines

- **Pipelines support indexing**; for example, **pipeline[1]** returns the second estimator in the pipeline, and **pipeline[:-1]** returns a Pipeline object containing all but the last estimator.
- You can also access the estimators via the **steps attribute**, which is a list of **name/estimator pairs**, or via the **named_steps** dictionary attribute, which maps the names to the estimators.
- For example, num_pipeline["simpleimputer"] returns the estimator named "simpleimputer".

Transformation Pipelines

```
num_pipeline.steps
```

```
[('simpleimputer', SimpleImputer(strategy='median')),
 ('standardscaler', StandardScaler())]
```

```
num_pipeline[1]
```

StandardScaler
StandardScaler()

```
num_pipeline[:-1]
```

Pipeline
SimpleImputer

```
num_pipeline.named_steps["simpleimputer"]
```

SimpleImputer
SimpleImputer(strategy='median')

```
num_pipeline.set_params(simpleimputer__strategy="median")
```

Pipeline
SimpleImputer
StandardScaler

ColumnTransformer

- So far, we have handled the categorical columns and the numerical columns separately.
- It would be **more convenient** to have a **single transformer** capable of handling all columns, applying the appropriate transformations to each column. For this, you can use a **ColumnTransformer**.
- For example, the following ColumnTransformer will apply **num_pipeline** (the one we just defined) to the **numerical attributes** and **cat_pipeline** to the **categorical attribute**:

ColumnTransformer

```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```

- First we import the ColumnTransformer class, then we define the list of numerical and categorical column names and construct a simple pipeline for categorical attributes. Lastly, we construct a ColumnTransformer.
- Its constructor requires a list of triplets (3-tuples), each containing a name (which must be unique and not contain double underscores), a transformer, and a list of names (or indices) of columns that the transformer should be applied to

Tip

- Instead of using a transformer, you can specify the string "**drop**" if you want the columns to be dropped, or you can specify "passthrough" if you want the columns to be left untouched.
- By default, the remaining columns (i.e., the ones that were not listed) will be dropped, but you can set the **remainder** hyperparameter to any transformer (or to "passthrough") if you want these columns to be handled differently.

ColumnTransformer

- Since listing all the column names is not very convenient, Scikit-Learn provides a **make_column_selector()** function that returns a selector function you can use to automatically select all the features of a given type, such as numerical or categorical.
- You can pass this selector function to the ColumnTransformer instead of column names or indices.

ColumnTransformer

- Moreover, if you don’t care about naming the transformers, you can use make_column_transformer(), which chooses the names for you, just like make_pipeline() does. For example, the following code creates the same ColumnTransformer as earlier, except the transformers are automatically named "pipeline-1" and "pipeline-2" instead of "num" and "cat"

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)
```

ColumnTransformer

- Now we're ready to apply this ColumnTransformer to the housing data

```
housing_prepared = preprocessing.fit_transform(housing)
```

ColumnTransformer

- we can get a DataFrame out if we want

```
housing_prepared_fr = pd.DataFrame(
    housing_prepared,
    columns=preprocessing.get_feature_names_out(),
    index=housing.index)
housing_prepared_fr.head(2)
```

	pipeline-1__longitude	pipeline-1__latitude	pipeline-1__housing_median_age	pipeline-1__total_rooms	pipeline-1__total_bedrooms	pipeline-1__population	pipeline-1__households	pipeline-1__median_income	pipeline-2__ocean_proximity_<1H OCEAN	pipeline-2__ocean_proximity_INLAND	pipeline-2__ocean_proximity_ISLAND	pipeline-2__ocean_proximity_NEAR BAY	pipeline-2__ocean_proximity_NEAR OCEAN
13096	-1.423037	1.013606	1.861119	0.311912	1.368167	0.137460	1.394812	-0.936491	0.0	0.0	0.0	1.0	0.0
14973	0.596394	-0.702103	0.907630	-0.308620	-0.435925	-0.693771	-0.373485	1.171942	1.0	0.0	0.0	0.0	0.0

ColumnTransformer

- Great! We have a preprocessing pipeline that takes the **entire training dataset** and **applies each transformer** to the appropriate columns, then *concatenates the transformed columns* **horizontally** (transformers must never change the number of rows).
- Once again this returns a NumPy array, but you can get the column names using preprocessing.get_feature_names_out() and wrap the data in a nice DataFrame as we did before.

Note

- The **OneHotEncoder** returns a **sparse** matrix and the num_pipeline returns a dense matrix.
- When there is such a **mix of sparse and dense matrices**, the ColumnTransformer **estimates the density of the final matrix** (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, sparse_threshold=0.3). In this example, it returns a dense matrix.

Conclusion

- Your project is going really well and you're almost ready to train some models! You now want to create a **single pipeline that will perform all the transformations** you've experimented with up to now.
- Let's recap what the pipeline will do and why:

Conclusion

Let’s recap what the pipeline will do and why:

- Missing values in numerical features will be imputed by replacing them with the median, as most ML algorithms don’t expect missing values. In categorical features, missing values will be replaced by the most frequent category.
- The categorical feature will be one-hot encoded, as most ML algorithms only accept numerical inputs.

Conclusion

Let’s recap what the pipeline will do and why:

- The categorical feature will be one-hot encoded, as most ML algorithms only accept numerical inputs.
- A few ratio features will be computed and added: bedrooms_ratio, rooms_per_house, and people_per_house. Hopefully these will better correlate with the median house value, and thereby help the ML models.

Conclusion

Let’s recap what the pipeline will do and why:

- A few cluster similarity features will also be added. These will likely be more useful to the model than latitude and longitude.
- Features with a long tail will be replaced by their logarithm, as most models prefer features with roughly uniform or Gaussian distributions.
- All numerical features will be standardized, as most ML algorithms prefer when all features have roughly the same scale.

Conclusion

- The code that builds the pipeline to do all of this should look familiar to you by now:

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_name(function_transformer, feature_names_in):
    return ["ratio"]  # feature names out

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy="median"),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler())

log_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(np.log, feature_names_out="one-to-one"),
    StandardScaler())
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                     StandardScaler())
preprocessing = ColumnTransformer([
        ("bedrooms", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
        ("rooms_per_house", ratio_pipeline(), ["total_rooms", "households"]),
        ("people_per_house", ratio_pipeline(), ["population", "households"]),
        ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
                               "households", "median_income"]),
        ("geo", cluster_simil, ["latitude", "longitude"]),
        ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
    ],
    remainder=default_num_pipeline)  # one column remaining: housing_median_age
```

Conclusion

- If you run this ColumnTransformer, it performs all the transformations and outputs a NumPy array with 24 features:

```
housing_prepared = preprocessing.fit_transform(housing)
housing_prepared.shape
```

```
(16512, 24)
```

```
preprocessing.get_feature_names_out()
```

```
array(['bedrooms__ratio', 'rooms_per_house__ratio',
       'people_per_house__ratio', 'log__total_bedrooms',
       'log__total_rooms', 'log__population', 'log__households',
       'log__median_income', 'geo__Cluster 0 similarity',
       'geo__Cluster 1 similarity', 'geo__Cluster 2 similarity',
       'geo__Cluster 3 similarity', 'geo__Cluster 4 similarity',
       'geo__Cluster 5 similarity', 'geo__Cluster 6 similarity',
       'geo__Cluster 7 similarity', 'geo__Cluster 8 similarity',
       'geo__Cluster 9 similarity', 'cat__ocean_proximity_<1H OCEAN',
       'cat__ocean_proximity_INLAND', 'cat__ocean_proximity_ISLAND',
       'cat__ocean_proximity_NEAR BAY', 'cat__ocean_proximity_NEAR OCEAN',
       'remainder__housing_median_age'], dtype=object)
```