HANDS-ON TUTORIALS

Step by Step Tutorial of Sci-kit Learn Pipeline

Use Pipelines to streamline your data science project right now!

James Ho Jan 9 · 5 min read★

Photo by Myriam Jessier on Unsplash

Most of the data science projects (as keen as I am to say all of them) require a certain level of data cleaning and preprocessing to make the most of the machine learning models. Some common preprocessing or transformations are:

a. Imputing missing values

b. Removing outliers

c. Normalising or standardising numerical features

d. Encoding categorical features

Sci-kit learn has a bunch of functions that support this kind of transformation, such as StandardScaler, SimpleImputer…etc, under the preprocessing package.

A typical and simplified data science workflow would like

1. Get the training data
2. Clean/preprocess/transform the data
3. Train a machine learning model
4. Evaluate and optimise the model
5. Clean/preprocess/transform new data
6. Fit the model on new data to make predictions.

You may notice that data preprocessing has to be done at least twice in the workflow. As tedious and time-consuming as this step is, how nice it would be if only we could automate this process and apply it to all of the future new datasets.

The good news is: YES, WE ABSOLUTELY CAN! With the scikit learn pipeline, we can easily systemise the process and therefore make it extremely reproducible. Following I'll walk you through the process of using scikit learn pipeline to make your life easier.

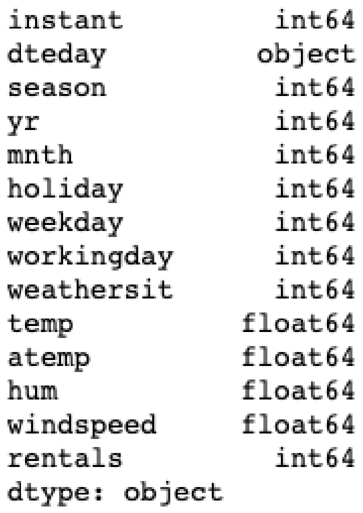
***Disclaimer: the purpose of this piece is to understand the usage of scikit learn pipeline, not to train a perfect machine learning model.*

Read in data

We'll be using the 'daily-bike-share' data from Microsoft's fantastic machine learning studying material.

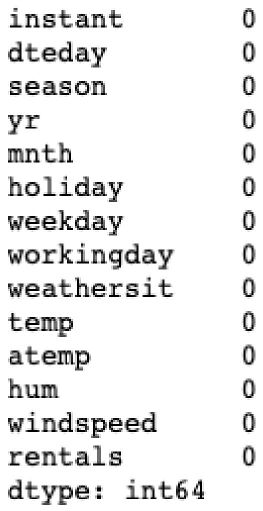
```
import pandas as pd
import numpy as np
```

```
data =
pd.read_csv('https://raw.githubusercontent.com/MicrosoftDocs/ml-
basics/master/data/daily-bike-share.csv')
data.dtypes
```



```
instant         int64
dteday         object
season          int64
yr              int64
mnth            int64
holiday         int64
weekday         int64
workingday      int64
weathersit      int64
temp          float64
atemp         float64
hum           float64
windspeed     float64
rentals         int64
dtype: object
```

```
data.isnull().sum()
```



```
instant       0
dteday        0
season        0
yr            0
mnth          0
holiday       0
weekday       0
workingday    0
weathersit    0
temp          0
atemp         0
hum           0
windspeed     0
rentals       0
dtype: int64
```

Luckily this dataset doesn't have missing values. Although it seems all features are numeric, there are actually some categorical features we need to identify. Those are *['season', 'mnth', 'holiday', 'weekday', 'workingday', 'weathersit']*. For the sake of illustration, I'll still treat it as having missing values. Let's filter out some obviously useless features first.

```
data = data[['season'
             , 'mnth'
             , 'holiday'
             , 'weekday'
             , 'workingday'
             , 'weathersit'
             , 'temp'
             , 'atemp'
             , 'hum'
             , 'windspeed'
             , 'rentals']]
```

Split the data

Before creating the pipline, we need to split the data into training set and testing set first.

```
from sklearn.model_selection import train_test_split
X = data.drop('rentals',axis=1)
y = data['rentals']

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=123)
```

Create the pipeline

The main parameter of a pipeline we'll be working on is '**steps**'. From the documentation, it is a '*list of (name, transform) tuples (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator.*'

It's easier to just have a glance at what the pipeline should look like:

```
Pipeline(steps=[('name_of_preprocessor', preprocessor),
                ('name_of_ml_model', ml_model())])
```

The 'preprocessor' is the complex bit, we have to create that ourselves. Let's crack on!

Preprocessor

The packages we need are as follow:

```
from sklearn.preprocessing import StandardScaler, OrdinalEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
```

Firstly, we need to define the transformers for both numeric and categorical features. A transforming step is represented by a tuple. In that tuple, you first define the name of the transformer, and then the function you want to apply. The order of the tuple will be the order that the pipeline applies the transforms. Here, we first deal with missing values, then standardise numeric features and encode categorical features.

```
numeric_transformer = Pipeline(steps=[
       ('imputer', SimpleImputer(strategy='mean'))
      ,('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
       ('imputer', SimpleImputer(strategy='constant'))
      ,('encoder', OrdinalEncoder())
])
```

The next thing we need to do is to specify which columns are numeric and which are categorical, so we can apply the transformers accordingly. We apply the transformers to features by using ColumnTransformer. Applying the transformers to features is our preprocessor. Similar to pipeline, we pass a list of tuples, which is composed of *(‘name’, ‘transformer’, ‘features’)*, to the parameter ‘*transformers*’.

```
numeric_features = ['temp', 'atemp', 'hum', 'windspeed']

categorical_features = ['season', 'mnth', 'holiday', 'weekday',
'workingday', 'weathersit']

preprocessor = ColumnTransformer(
   transformers=[
    ('numeric', numeric_transformer, numeric_features)
   ,('categorical', categorical_transformer, categorical_features)
])
```

Some people would create the list of numeric/categorical features based on the data type, like the following:

```
numeric_features = data.select_dtypes(include=['int64',
'float64']).columns

categorical_features = data.select_dtypes(include=
['object']).drop(['Loan_Status'], axis=1).columns
```

I personally don't recommend this, because if you have categorical features disguised as numeric data type, e.g. this dataset, you won't be able to identify them. Only use this method when you're 100% sure that only numeric features are numeric data types.

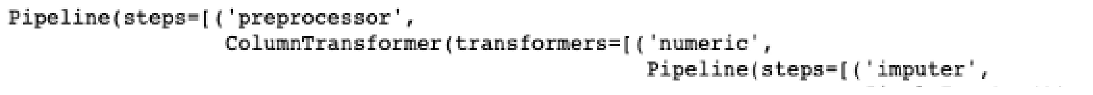
Estimator

After assembling our preprocessor, we can then add in the estimator, which is the machine learning algorithm you'd like to apply, to complete our preprocessing and training pipeline. Since in this case, the target variable is continuous, I'll apply Random Forest Regression model here.

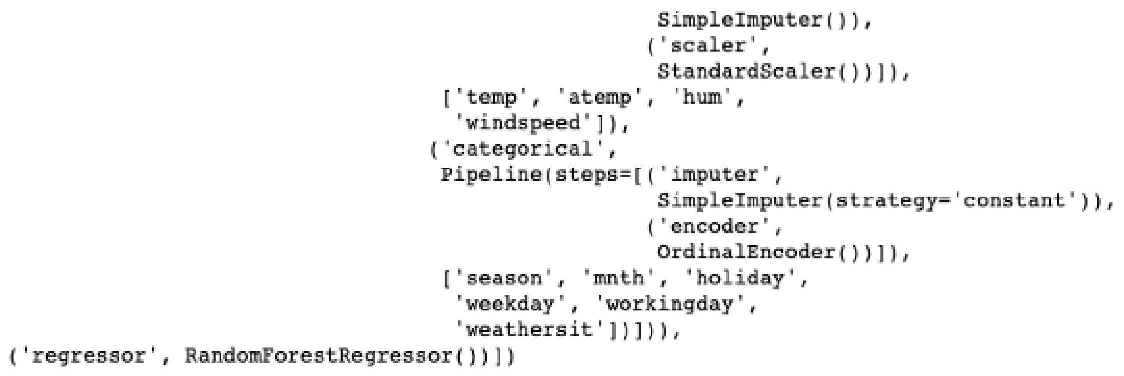
```
from sklearn.ensemble import RandomForestRegressor

pipeline = Pipeline(steps = [
               ('preprocessor', preprocessor)
              ,('regressor',RandomForestRegressor())
           ])
```

To create the model, similar to what we used to do with a machine learning algorithm, we use the 'fit' function of pipeline.

```
rf_model = pipeline.fit(X_train, y_train)
print (rf_model)
```



```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('numeric',
                                                  Pipeline(steps=[('imputer',
```



```
                                                 SimpleImputer()),
                                                ('scaler',
                                                 StandardScaler())]),
                                 ['temp', 'atemp', 'hum',
                                  'windspeed']),
                                ('categorical',
                                 Pipeline(steps=[('imputer',
                                                 SimpleImputer(strategy='constant')),
                                                ('encoder',
                                                 OrdinalEncoder())]),
                                 ['season', 'mnth', 'holiday',
                                  'weekday', 'workingday',
                                  'weathersit'])])),
                ('regressor', RandomForestRegressor())])
```

Use the normal methods to evaluate the model.

```
from sklearn.metrics import r2_score

predictions = rf_model.predict(X_test)

print (r2_score(y_test, predictions))
>> 0.7355156699663605
```

Use the model

To maximise reproducibility, we'd like to use this model repeatedly for our new incoming data. Let's save the model by using 'joblib' package to save it as a pickle file.

```
import joblib

joblib.dump(rf_model, './rf_model.pkl')
```

Now we can call this pipeline, which includes all sorts of data preprocessing we need and the training model, whenever we need it.

```
# In other notebooks
rf_model = joblib.load('PATH/TO/rf_model.pkl')

new_prediction = rf_model.predict(new_data)
```

Conclusion

Before knowing scikit learn pipeline, I always had to redo the whole data preprocessing and transformation stuff whenever I wanted to apply the same model to different datasets. It was a really tedious process. I tried to write a function to do all of them, but the result wasn't really satisfactory and didn't save me a lot of workloads.

Scikit learn pipeline really makes my workflows smoother and more flexible. For example, you can easily compare the performance of a number of algorithms like:

```
regressors = [
    regressor_1()
   ,regressor_2()
   ,regressor_3()
   ....]

for regressor in regressors:
    pipeline = Pipeline(steps = [
               ('preprocessor', preprocessor)
              ,('regressor',regressor)
           ])
    model = pipeline.fit(X_train, y_train)
    predictions = model.predict(X_test)
    print (regressor)
    print (f('Model r2 score:{r2_score(predictions, y_test)}')
```

, or adjust the preprocessing/transforming methods. For instance, use 'median' value to fill missing values, use a different scaler for numeric features, change to one-hot encoding instead of ordinal encoding to handle categorical features, hyperparameter tuning, etc.

```
numeric_transformer = Pipeline(steps=[
       ('imputer', SimpleImputer(strategy='median'))
      ,('scaler', MinMaxScaler())
])

categorical_transformer = Pipeline(steps=[
       ('imputer', SimpleImputer(strategy='constant'))
      ,('encoder', OneHotEncoder())
])

pipeline = Pipeline(steps = [
               ('preprocessor', preprocessor)
              ,('regressor',RandomForestRegressor(n_estimators=300
                                                                ,max_depth=10))
              ])
```

All of the above adjustments can now be done as simply as changing a parameter in the functions.

I hope you find this helpful and any comments or advice are welcome!