Pipeline trong Sklearn

Pipeline trong Sklearn

Như đã phân tích ở các bài trước, để xây dựng một mô hình học máy có tính hiệu quả trong thực tế chúng ta cần có một luồng xử lý rõ ràng và thống nhất. Thông thường, một luồng xử lý tổng quát sẽ gồm các bước sau: tiền xử lý dữ liệu thô thành các đặc trưng hữu ích, chuyển hóa dữ liệu để phù hợp với từng thuật toán học máy cụ thể, huấn luyện dữ liệu và đánh giá mô hình. Trong bài này, chúng ta sẽ cùng nhau tìm hiểu về đối tượng giúp ta xây dựng và quản lý một luồng các công việc từ tiền xử lý tới huấn luyện mô hình. Đó là đối tượng *pipeline* của thư viện *sklearn* trong ngôn ngữ lập trình Python.

Scikit-learn, một công cụ xử lý thông minh

Chúng ta đều biết rằng có rất nhiều gói thư viện học máy trong Python. Một trong số chúng là thư viện *Scikit-learn*, hay còn biết đến là *sklearn* trong *pip*. Thư viện này chứa rất nhiều thuật toán học máy và các công cụ tiền xử lý dữ liệu hiệu quả. Về mặt tổng quát, *sklearn* chứa các công cụ hữu ích sau:

- Bộ công cụ trích xuất đặc trưng để chuyển dữ liệu thô sang các đặc trưng mang lại tri thức cho mô hình.
- Bộ công cụ tiền xử lý để làm sạch dữ liệu và tăng cường tính hiệu quả của dữ liệu với việc thêm vào dữ liệu các thông tin bổ sung.
- Các thuật toán học máy giám sát để dự đoán các giá trị hoặc phân lớp dữ liệu.
- Các thuật toán học máy bán giám sát để cấu trúc dữ liệu và nhận dạng các mẫu.
- Pipelines để tổng hợp các công cụ khác nhau vào một khối code duy nhất.

Tất cả các bộ công cụ đó rất cần thiết để xây dựng các ứng dụng học máy. Tuy nhiên, trong bài viết này, chúng ta chỉ tập trung vào một đối tượng đặc biệt là pipeline.

Pipeline là gì?

Một *pipeline* trong *sklearn* là một tập các chuỗi thuật toán để trích xuất đặc trưng, tiền xử lý, chuyển hóa và huấn luyện dữ liệu sử dụng các thuật toán học máy cụ thể.

Dưới đây là cách bạn khởi tạo một pipeline trong *sklearn*

```
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction.text import CountVectorizer

pipeline = Pipeline(steps=[
  ('vectorize', CountVectorizer()),
  ('classify', DecisionTreeClassifier())
])
```

Mỗi *pipeline* bao gồm một vài bước nhất định, mỗi bước bao gồm một tham số là tên của bước, tham số còn lại là bộ chuyển đổi dữ liệu tương ứng (thường gọi là transformer). Bước cuối cùng trong một *pipeline* được gọi là *estimator*. Một *estimator* có thể là một thuật toán phân lớp, một thuật toán hồi quy, một mạng nơ-ron hay có thể là một thuật toán học máy không giám sát.

Để huấn luyện *estimator* tại bước cuối cùng của *pipeline*, bạn phải gọi phương thức *fit* của *pipeline* và cung cấp dữ liệu để huấn luyện.

```
# train model

raw_features = ['Hello friend', 'Happy', 'Sad']

raw_labels = ['Greetings', 'Emotions', 'Emotions']

pipeline.fit(raw_features, raw_labels)
```

Bởi vì bạn đang sử dụng một *pipeline* mà bạn có thể truyền vào dữ liệu thô nên các bước tiền xử lý dữ liệu sẽ được thực hiện tự động trước khi được truyền đến *estimator* để huấn luyện. Trong trường hợp tổng quát, bạn có thể phải tự định nghĩa các bước tiền xử lý dữ liệu như là xử lý dữ liệu bị khuyết, chuẩn hóa hay chuẩn tắc dữ liệu.

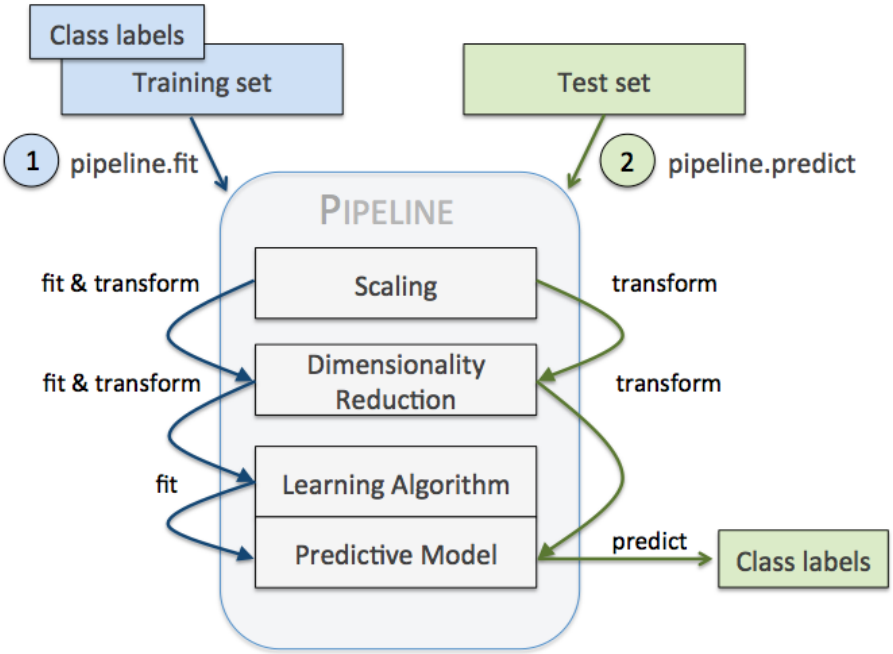
Một khi bạn đã huấn luyện dữ liệu bằng cách sử dụng *estimator* trong *pipeline*, bạn có thể sử dụng *pipeline* đó để dự đoán đầu ra cho dữ liệu mới bằng cách sử dụng phương thức *predict*.

```
# test model

sample_sentence = ['Hi friend']


outcome = pipeline.predict(sample_sentence)
```

Một điểm quan trọng mà chúng ta nên nhớ là khi dự đoán một đầu ra thì các bước xử lý trước đó cho dữ liệu huấn luyện sẽ được áp dụng giống hệt với các dữ liệu mới. Ta có thể xét ví dụ sau đây:



Hình 1. Ví dụ về một pipeline trong sklearn (link ảnh: https://codetudau.com/content/images/2017/12/pipeline-diagram.png)

Hình 1 mô tả một pipeline gồm các bước là scaling dữ liệu, giảm chiều dữ liệu và huấn luyện mô hình. Các bước này cũng được áp dụng lần lượt trên bộ dữ liệu kiểm thử để có được kết quả dự đoán khớp với mô hình. Đây là một tính năng rất hữu dụng của một *pipeline*. Nhờ đó mà bạn không phải lo lắng về sự sai khác trong các bước tiền xử lý dữ liệu giữa quá trình huấn luyện và và quá trình dự đoán, kiểm thử. Nó hoàn toàn được xử lý một cách tự động.

Xây dựng một pipeline tủy chỉnh

Tạo một lớp tùy chỉnh từ lớp TransformerMixin

Tất cả các thành phần trong *sklearn* đều có thể được sử dụng trong một *pipeline*. Tuy nhiên, có thể có xử lý nào đó trên dữ liệu mà bạn muốn không có sẵn trong *pipeline*. Lúc này chúng ta cần xây dựng mô hình của chúng ta bằng cách tự định nghĩa ra các thành phần trong nó:

```
from sklearn.base import TransformerMixin

class MyCustomStep(TransformerMixin):
    def transform(X, **kwargs):
        pass

    def fit(X, y=None, **kwargs):
        return self
```

Một thành phần pipeline được xác định như là một lớp **TransformerMixin** với các phương thức quan trọng sau:

- *fit* – sử dụng dữ liệu đầu vào để huấn luyện transformer
- *transform* – lấy các đặc trưng đầu vào và chuyển hóa chúng thành dạng khác

Phương thức *fit* được dùng để huấn luyện transformer. Như trong ví dụ ban đầu, phương thức này được sử dụng bởi các thành phần như là *CountVectorizer* để cài đặt các ánh xạ từ các từ sang các phần tử của vector. Nó thu được cả các đặc trưng và các đầu ra mong đợi.

Phương thức *transform* chỉ thu được các đặc trưng cần được chuyển hóa thành. Nó trả về một bộ các đặc trưng đã được chuyển hóa.

Ta cần lưu ý rằng, các transformer trong *pipeline* không được phép xóa bỏ hoặc thêm các điểm dữ liệu khác vào bộ dữ liệu đầu vào. Nếu bạn cần thiết phải sử dụng tính năng này, bạn nên tạo ra một transformer nằm bên ngoài *pipeline*.

Sử dụng FuntionTransformer

Một cách khác để tùy chỉnh một transformer trong *pipeline* là sử dụng lớp ***FunctionTransformer***. Lớp này có nhiệm vụ khởi tạo một transformer từ một hàm khác (hàm này có thể là một hàm do người dùng định nghĩa hoặc một hàm của một đối tượng) và

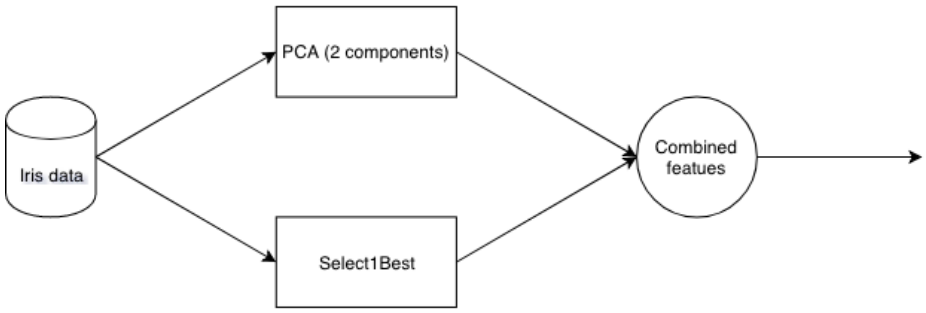
nó trả về kết quả của hàm đó. Điều này có ý nghĩa trong các transformer có tính chuyển đổi phi trạng thái như là tính *logarit* của tần số, thực hiện *scaling* dữ liệu,…

Ví dụ dưới đây minh họa cho việc sử dụng *FunctionTransformer* với hàm *log1p* của *numpy* và hàm *drop_first_component* do người dùng tự định nghĩa:

```
import numpy as np
from sklearn.preprocessing import FunctionTransformer
# drop first column
def drop_first_component(X):
    return X[:,1:]
# Create data
X = np.arange(10).reshape((5, 2))
# FunctionTransformer with log1p of numpy
print("****** Comparison of result of FunctionTransformer using object function and result of executing object function ******")
print(FunctionTransformer(np.log1p).transform(X) == np.log1p(X))
# Function Transformer with user-defined function
# Show original data
print("****** FunctionTransform using user-defined function ******")
print("Original array:")
print(X)
print("Transformed array:")
print(FunctionTransformer(drop_first_component).fit_transform(X))
```

FeatureUnion

Thông thường, chúng ta tạo một *pipeline* chứa các bước xử lý tuần tự để giải quyết bài toán học máy của mình. Tuy nhiên, đôi khi ta phải tách một bước thành một vài bước để thực hiện song song và tổng hợp kết quả của chúng lại. Điều này thường thấy ở khâu trích xuất đặc trưng khi sử dụng các transformer. Trong trường hợp này, chúng ta sử dụng lớp FeatureUnion để kết hợp các kết quả của các transformer đã thực thi song song. Ví dụ dưới đây sẽ chỉ cho bạn thấy cách sử dụng *FeatureUnion* để kết hợp các đặc trưng thu được từ hai transformer là PCA và lựa chọn đơn biến (*SelectKBest*). Kết quả thu được là ta được bộ đặc trưng mới cho dữ liệu hoa *Iris* gồm 3 đặc trưng so với số lượng đặc trưng ban đầu là 4.



Hình 2. Minh họa về FeatureUnion

```
# Author: Andreas Mueller <amueller@ais.uni-bonn.de>]
# Modifier: Phan Doan Cuong <cuongpd_58@vnu.edu.vn>
#
# License: BSD 3 clause

from __future__ import print_function
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.datasets import load_iris
```

```
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest

iris = load_iris()

X, y = iris.data, iris.target

# This dataset is way too high-dimensional. Better do PCA:
pca = PCA(n_components=2)

# Maybe some original features where good, too?
selection = SelectKBest(k=1)

# Build estimator from PCA and Univariate selection:
combined_features = FeatureUnion([("pca", pca), ("univ_select", selection)])

# Use combined features to transform dataset:
X_features = combined_features.fit(X, y).transform(X)
print("Original space has", X.shape[1], "features")
print("Combined space has", X_features.shape[1], "features")
```

Ứng dụng pipeline sử dụng grid search

Pipeline là một cách xây dựng hữu ích cho việc xây dựng các phần mềm học máy. Và với những thành phần có thể mở rộng được, bạn có thể tùy chỉnh để chúng hoạt động tốt hơn, vượt ra ngoài phạm vi của gói thư viện *sklearn*.

Có một mẹo hay mà tôi muốn giới thiệu với các bạn đó là bạn có thể sử dụng *pipeline* để thực thi một *tìm kiếm lưới* (*grid search*). Với một thuật toán *grid search*, bạn có thể để cho máy tính tự tìm ra các *siêu tham số* (*hyperparameters*) tối ưu cho thuật toán học máy mà bạn sử dụng một cách tự động.

Các *hyperparameters* là các tham số mà bạn cài đặt trước khi huấn luyện một mô hình, hay còn gọi là tham số của các thuật toán. Ví dụ như là chỉ số *learning rate* của một thuật toán *gradient descent* hay số lượng *nơ-ron* trong một tầng *mạng nơ-ron*.

Với *grid search* bạn có thể chỉ định một vài biến thể của các tham số mô hình và để cho máy tính huấn luyện các mô hình ứng với các bộ tham số đã cài đặt để tìm ra bộ tham số tối ưu cho thuật toán.

Đầu tiên, bạn phải định nghĩa một tập các tham số mà bạn muốn tối ưu với *grid search*. Để cài đặt giá trị cho các bộ tham số, ta cần tạo ra một *dictionary* chứa các thành phần là các giá trị của một tham số trong một bước nhất định trong pipeline theo mẫu sau:

stepname__model__param=[values]

Trong đó:

- *stepname* là tên của bước mà bạn đã định nghĩa trong pipeline
- *model* là tên mô hình mà bạn sử dụng để chuyển hóa dữ liệu
- *param* là tên của tham số ứng với *model* đã chọn
- *[values]* là một mảng chứa các giá trị của tham số *param*
- Ký hiệu hai dấu gạch dưới (__) dùng để phân cách các thành phần trong mẫu trên

```
# Do grid search over k, n_components, copy and C:
svm = SVC(kernel="linear")
pipeline = Pipeline([("features", combined_features), ("svm", svm)])
param_grid = dict(features__pca__n_components=[1, 2, 3],
                  features__pca__copy=[True,False],
                  features__univ_select__k=[1, 2],
                  svm__C=[0.1, 1, 10])

grid_search = GridSearchCV(pipeline, param_grid=param_grid, cv=5, verbose=10)
grid_search.fit(X, y)
```

```
print(grid_search.best_estimator_)
```

Xét trong ví dụ trên, ta đang muốn tối ưu bước *features* trong pipeline sử dụng 2 mô hình là *pca* và *univ_select*. Với mô hình pca, ta đang tối ưu hai tham số là *n_components* và *copy* với các giá trị tương ứng là *1, 2, 3* và *True, False*. Tương tự, ta cũng đang tối ưu tham số *k* trong mô hình *univ_select* và tham số *C* trong mô hình *svm*. Sau khi tiến hành chạy grid search, kết quả ta thu được là một estimator tối ưu ứng với bộ tham số *n_components* = 3, *copy* = True, *k* = 1 và *C* = 10.

Sau khi bạn đã định nghĩa được các tham số cho *grid search*, bạn cần truyền chúng vào cùng với *pipeline* để có được một đối tượng của lớp *GridSearchCV*.

Khi bạn gọi phương thức *fit* trong *grid_search*, nó sẽ bắt đầu quá trình tìm kiếm. Sau khi quá trình này kết thúc, bạn có thể nhìn thấy kết quả bằng việc gọi ra thuộc tính *cv_results_* trong đối tượng *grid_search*. Thuộc tính *cv_results_* này chứa một *dictionary* với các thành phần thú vị sau:

- *rank_test_score*: Chứa bảng xếp hạng cho mỗi mô hình theo thứ tự mà chúng được thực thi trước đó. Chỉ số của mô hình có xếp hạng là 1 là mô hình tốt nhất.
- *params*: Chứa các bộ tham số sử dụng cho mỗi mô hình theo thứ tự mà chúng được thực thi trước đó. Nếu bạn biết mô hình tốt nhất rồi thì bạn có thể tìm được bộ tham số tương ứng cho *pipeline* với chỉ số tương ứng.
- *mean_test_score*: Chứa các điểm số kiểm thử cho mỗi mô hình theo thứ tự mà chúng được thực thi. Bạn có thể nhìn vào giá trị này để biết rằng các mô hình của bạn có tốt hay không.

Bạn cũng cần chú ý rằng nếu bạn đang xây dựng một *transformer* của riêng bạn thì bạn cần triển khai một phương thức nữa để hỗ trợ cho thuật toán *grid search*:

```
from sklearn.base import TransformerMixin


class MyCustomStep(TransformerMixin):

    def transform(X, **kwargs):

        pass

    def fit(self,X, y=None, **kwargs):

        return self

    def get_params(**kwargs):

        return {}
```

Phương thức *get_params* trả về các tham số cho thành phần *pipeline* của bạn. Nếu bạn có bất kỳ tham số nào mà bạn muốn sửa đổi bởi *grid search*, đây là nơi để bạn thêm chúng vào.

Kết luận

Chúng ta có thể khẳng định tương đối chắc chắn rằng chức năng của grid search là thứ rất hữu dụng nếu bạn muốn tìm ra các cách để các mô hình học máy của bạn trở nên tốt hơn trong thực tế.

Pipeline trong *sklearn* cung cấp một cách khởi tạo quan trọng khi nói đến việc sử dụng các thuật toán học máy từ gói thư viện này.

Tham khảo:

http://scikit-learn.org/stable/modules/compose.html

http://fizzylogic.nl/2017/11/07/learn-how-to-build-flexible-machine-learning-pipelines-in-sklearn/

http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html

http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.FunctionTransformer.html

Tác giả:

Cương Phan Đoàn,

+84989 906 612

https://www.linkedin.com/in/cuong-phan-5a8462134/