A Simple Guide to Scikit-learn Pipelines

Learn how to use pipelines in a scikit-learn machine learning workflow

Rebecca Vickery Follow
Feb 6, 2019 · 5 min read ★

Photo by Erlend Ekseth on Unsplash

In most machine learning projects the data that you have to work with is unlikely to be in the ideal format for producing the best performing model. There are quite often a number of transformational steps such as encoding categorical variables, feature scaling and normalisation that need to be performed. Scikit-learn has built in functions for most of these commonly used transformations in it's preprocessing package.

However, in a typical machine learning workflow you will need to apply all these transformations at least twice. Once when training the model and again on any new data you want to predict on. Of course you could write a function to apply them and reuse that but you would still need to run this

first and then call the model separately. Scikit-learn pipelines are a tool to simplify this process. They have several key benefits:

- They make your workflow much easier to read and understand.
- They enforce the implementation and order of steps in your project.
- These in turn make your work much more reproducible.

In the following post I am going to use a data set, taken from Analytics Vidhya's loan prediction practice problem, to describe how the pipelines work and how to implement them.

Transformers

First I have imported the train and test files into a jupyter notebook. I have dropped the 'Loan_ID' column as this will not be needed in training or prediction. I have used the pandas dtypes function to get a little information about the dataset.

```
import pandas as pd
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
train = train.drop('Loan_ID', axis=1)
train.dtypes
```

```
Gender              object
```

```
Married              object
Dependents           object
Education            object
Self_Employed        object
ApplicantIncome       int64
CoapplicantIncome   float64
LoanAmount          float64
Loan_Amount_Term    float64
Credit_History      float64
Property_Area        object
Loan_Status          object
dtype: object
```

I can see that I have both categorical and numeric variables so as a minimum I am going to have to apply a one hot encoding transformation and some sort of scaler. I am going to use a scikit-learn pipeline to perform those transformations, and at the same time apply the fit method.

Before building the pipeline I am splitting the training data into a train and test set so that I can validate the performance of the model.

```
X = train.drop('Loan_Status', axis=1)
y = train['Loan_Status']


from sklearn.model_selection import train_test_split


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

The first step in building the pipeline is to define each transformer type. The convention here is generally to create transformers for the different

variable types. In the code below I have created a numeric transformer which applies a StandardScaler, and includes a SimpleImputer to fill in any missing values. This is a really nice function in scikit-learn and has a number of options for filling missing values. I have chosen to use median but another method may result in better performance. The categorical transformer also has a SimpleImputer with a different fill method, and leverages OneHotEncoder to transform the categorical values into integers.

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder


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


categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant',
fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

Next we use the ColumnTransformer to apply the transformations to the correct columns in the dataframe. Before building this I have stored lists of the numeric and categorical columns using the pandas dtype method.

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

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

from sklearn.compose import ColumnTransformer

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])
```

Fitting the classifier

The next step is to create a pipeline that combines the preprocessor created above with a classifier. In this case I have used a simple RandomForestClassifier to start with.

```
from sklearn.ensemble import RandomForestClassifier

rf = Pipeline(steps=[('preprocessor', preprocessor),
                     ('classifier', RandomForestClassifier())])
```

You can then simply call the fit method on the raw data and the preprocessing steps will be applied followed by training the classifier.

```
rf.fit(X_train, y_train)
```

To predict on new data it is as simple as calling the predict method and the preprocessing steps will be applied followed by the prediction.

```
y_pred = rf.predict(X_test)
```

Model selection

A pipeline can also be used during the model selection process. The following example code loops through a number of scikit-learn classifiers applying the transformations and training the model.

```
from sklearn.metrics import accuracy_score, log_loss
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC, LinearSVC, NuSVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier,
AdaBoostClassifier, GradientBoostingClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import
QuadraticDiscriminantAnalysis


classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="rbf", C=0.025, probability=True),
    NuSVC(probability=True),
    DecisionTreeClassifier(),
    RandomForestClassifier(),
    AdaBoostClassifier(),
    GradientBoostingClassifier()
    ]
```

```
for classifier in classifiers:
    pipe = Pipeline(steps=[('preprocessor', preprocessor),
                           ('classifier', classifier)])
    pipe.fit(X_train, y_train)
    print(classifier)
    print("model score: %.3f" % pipe.score(X_test, y_test))
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=3, p=2,
           weights='uniform')
model score: 0.772
SVC(C=0.025, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=True, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
model score: 0.675
NuSVC(cache_size=200, class_weight=None, coef0=0.0,
   decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
   kernel='rbf', max_iter=-1, nu=0.5, probability=True, random_state=None,
   shrinking=True, tol=0.001, verbose=False)
model score: 0.797
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
model score: 0.699
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
model score: 0.748
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
          learning_rate=1.0, n_estimators=50, random_state=None)
model score: 0.748
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=0.1, loss='deviance', max_depth=3,
```

```
                max_features=None, max_leaf_nodes=None,
                min_impurity_decrease=0.0, min_impurity_split=None,
                min_samples_leaf=1, min_samples_split=2,
                min_weight_fraction_leaf=0.0, n_estimators=100,
                n_iter_no_change=None, presort='auto', random_state=None,
                subsample=1.0, tol=0.0001, validation_fraction=0.1,
                verbose=0, warm_start=False)
model score: 0.780
```

The pipeline can also be used in grid search to find the best performing parameters. To do this you first need to create a parameter grid for your chosen model. One important thing to note is that you need to append the name that you have given the classifier part of your pipeline to each parameter name. In my code above I have called this 'classifier' so I have added classifier__ to each parameter. Next I created a grid search object which includes the original pipeline. When I then call fit, the transformations are applied to the data, before a cross-validated grid-search is performed over the parameter grid.

```
param_grid = {
    'classifier__n_estimators': [200, 500],
    'classifier__max_features': ['auto', 'sqrt', 'log2'],
    'classifier__max_depth' : [4,5,6,7,8],
    'classifier__criterion' :['gini', 'entropy']}


from sklearn.model_selection import GridSearchCV


CV = GridSearchCV(rf, param_grid, n_jobs= 1)

CV.fit(X_train, y_train)
print(CV.best_params_)
print(CV.best_score_)
```

I am working quite a lot with scikit-learn machine learning projects at the moment. Before I started to use pipelines I would find that when I went back to a project to work on it again even after only a short time I would have trouble following the workflow again. Pipelines have really helped me to put together projects that are both easily repeatable and extensible. I hope that this guide helps others who are interested in learning how to use them.

Machine Learning Artificial Intelligence Data Science Python Scikit Learn