Scikit learn

Wednesday, 13 November 2024 6:26 PM

- **fit**: Trains the model or transformer on your data.
- **predict**: Makes predictions on new data, used with estimators like classifiers or regressors.
- **transform**: Applies a transformation.
- **Pipeline**: Combines multiple steps, making code cleaner and ensuring consistent data processing across all stages.

Using fit and predict with a Classifier

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

# Load data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Fit the model
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Predict
predictions = clf.predict(X_test)
print(predictions)
```

Explanation

- train_test_split: Used to split the dataset into training and testing sets.
- LogisticRegression: The classifier used for our model.
- load_iris: Loads the Iris dataset, a commonly used dataset for classification.

    - X, y = load_iris(return_X_y=True)
- Here, X is the feature data (input), and y is the target data (output).
- return_X_y=True gives us X and y directly.

    - X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
- The data is split into a training set (70%) and a test set (30%).
- random_state=42 ensures reproducibility.

    - clf = LogisticRegression()

  We instantiate a LogisticRegression model and store it in clf.

    - clf.fit(X_train, y_train)
- fit trains the model using X_train (features) and y_train (target).
- During training, the model learns to classify based on patterns in the data.

    - predictions = clf.predict(X_test)

- predictions = clf.predict(X_test)

- We use predict on the X_test data to generate predictions.
- predictions is an array with predicted class labels for each sample in X_test.

  - print(predictions)

- Displays the predictions, which are the model's classification of the X_test samples.

Using fit and transform with a Transformer

```
from sklearn.preprocessing import StandardScaler
import numpy as np

# Data
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Fit and transform
scaler = StandardScaler()
scaler.fit(data)  # Learns the mean and std
transformed_data = scaler.transform(data)  # Transforms the data
print(transformed_data)
```

Import Necessary Libraries:

- StandardScaler is used for standardizing features by removing the mean and scaling to unit variance.

data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

- We create a 2D array, data, to demonstrate scaling.

scaler = StandardScaler()
StandardScaler is initialized, which will later calculate the mean and standard deviation.

scaler.fit(data)
fit calculates the mean and standard deviation for each feature column.

transformed_data = scaler.transform(data)

- transform scales the data using the previously calculated mean and standard deviation.
- transformed_data now contains the standardized version of data.

print(transformed_data)
Displays the transformed data.

Using a Pipeline to Chain Steps Together

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load data
X, y = load_iris(return_X_y=True)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create a pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),          # Step 1: Scale features
    ('classifier', LogisticRegression())    # Step 2: Fit logistic regression
])

# Fit pipeline on training data
pipeline.fit(X_train, y_train)

# Predict with pipeline
predictions = pipeline.predict(X_test)
print(predictions)
```

- Pipeline from sklearn.pipeline lets us chain together transformations and a model.
- We also import StandardScaler, LogisticRegression, and the dataset utilities as in previous examples.

We load the Iris dataset and split it into training and testing sets as before.

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),          # Step 1: Scale features
    ('classifier', LogisticRegression())    # Step 2: Fit logistic regression
])
```

- We create a Pipeline object that contains two steps:
- scaler: Uses StandardScaler to standardize the data.
- classifier: Uses LogisticRegression as the classifier.
- Each step has a name ('scaler', 'classifier') and an instance of a transformer or model.

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

- fit applies each step in sequence:
- First, it scales X_train using StandardScaler.
- Then, it trains the LogisticRegression model on the scaled data.
- The pipeline manages the transformations and model fitting automatically, ensuring consistent processing.

```
predictions = pipeline.predict(X_test)
```

When we call predict, the pipeline applies the same scaling transformation to X_test and then makes predictions with the LogisticRegression model.

```
print(predictions)
```

Displays the predictions, just as before, but this time done through the pipeline.

Key Benefits of Using Pipelines

- Consistency: Ensures the same preprocessing steps are applied to both training and test data.

- Simplified Code: Reduces code complexity by chaining multiple steps.
- Hyperparameter Tuning: Pipelines allow easy integration with GridSearchCV and RandomizedSearchCV for hyperparameter tuning across multiple steps.