

Scikit-learn

Scikit-learn is an open-source Python library that implements a range of machine learning, pre-processing, cross-validation, and visualization algorithms using a unified interface.

Features of scikit-learn:

- ✓ Simple and efficient tools for data mining and data analysis. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means, etc.
- ✓ Accessible to everybody and reusable in various contexts.
- ✓ Built on the top of NumPy, SciPy, and matplotlib.
- ✓ Open source, commercially usable – BSD license.

Installation:

- ✓ Before installing scikit-learn, ensure that you have NumPy and SciPy installed. Once you have a working installation of NumPy and SciPy, the easiest way to install scikit-learn is using pip:

✓ `pip install scikit-learn`

Step-1: Data Preprocessing in Scikit-learn:

Data preprocessing is the process of transforming data into a useful, manageable and understandable format. Scikit-learn has package named `preprocessing` for this purpose.

Preprocessing steps:

1. Load data
2. Data analysis
3. Handle missing values
4. Encode categorical features
6. Scale numeric features

1. Load data:

A dataset is nothing but a collection of data. A dataset generally has two main components:

Features: The variables of data are called its features. They are also known as predictors, inputs or attributes. They can be more than one and hence represented by a **feature matrix** ('X' is a common notation to represent feature matrix). A list of all the feature names is termed **feature names**.

Response: It is the output variable that basically depends upon the feature variables. They are also known as target, label or output. We generally have a single response column and it is represented by a **response vector** ('y' is a common notation to represent response vector). All the possible values taken by a response vector are termed **target names**.

Scikit-learn have few example datasets like iris and digits for classification and the Boston house prices for regression.

Example:

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

```

from sklearn import preprocessing
students = [[85,      'M',      'verygood' ],
            [95,      'F',      'excellent'],
            [75,      None,     'good'      ],
            [np.NaN,   'M',      'average'   ],
            [70,      'M',      'good'      ],
            [np.NaN,   None,     'verygood' ],
            [92,      'F',      'verygood' ],
            [98,      'M',      'excellent']]

df = pd.DataFrame(students)
df.columns = ['marks', 'gender', 'result']
print (df)

```

Output:

	marks	gender	result
0	85.0	M	verygood
1	95.0	F	excellent
2	75.0	None	good
3	NaN	M	average
4	70.0	M	good
5	NaN	None	verygood
6	92.0	F	verygood
7	98.0	M	excellent

2. Data analysis:

Let's start by understanding our dataset.

2.1. Show null columns

```

Print(df.info( ))

```

```

Data columns (total 3 columns):
#   Column  Non-Null Count  Dtype
---  ------  -
0   marks    6 non-null    float64
1   gender    6 non-null    object
2   result    8 non-null    object
dtypes: float64(1), object(2)

```

2.2. Show missing values

It is noted that the dataset contains missing values, we can use the function `isna()`, which returns if an cell of the dataset is NaN or not. Then we can count how many missing values there are for each column.

```

Print( df.isnull( ).sum( ))

```

```

marks      2
gender      2
result      0
dtype: int64

```

3. Handle Missing Data

We can handle missing data in two ways:

- ✓ Remove it (dropping)
- ✓ Imputing it from the whole dataset (using the mean, median, mode...)

3.1. Dropping missing data

We could drop rows and columns with missing values, at risk of losing too much information.

```
print(df.dropna())
```

	marks	gender	result
0	85.0	M	verygood
1	95.0	F	excellent
4	70.0	M	good
6	92.0	F	verygood
7	98.0	M	excellent

However, it is preferable to impute data from the entire dataset.

3.2. Imputing missing data

`SimpleImputer` class is used to impute / replace the numerical or categorical missing data related to one or more features with appropriate values such as following:

Mean: When `SimpleImputer()` is invoked without any arguments, it defaults to using the mean strategy. Missing values get replaced with the mean along each column. This strategy can only be used with numeric data.

Median: Missing values get replaced with the median along each column. This strategy can only be used with numeric data.

Most frequent (mode): Missing values get replaced with the most frequent value along each column. This strategy can be used with strings or numeric data.

Constant: Missing values get replaced with the `fill_value`. This strategy can be used with strings or numeric data.

Here is the Python code sample representing the usage of `SimpleImputer` for replacing numerical missing values.

```
from sklearn.impute import SimpleImputer

#imputing with mean value
imputer = SimpleImputer(missing_values=np.NaN, strategy='mean')
df.marks = imputer.fit_transform(df['marks'].values.reshape(-1,1))[:,0]
print(df.marks)
```

Output:

```
0    85.000000
1    95.000000
2    75.000000
3    85.833333
4    70.000000
5    85.833333
6    92.000000
7    98.000000
Name: marks, dtype: float64
```

Imputing with median value

```
imputer = SimpleImputer(missing_values=np.NaN, strategy='median')
df.marks = imputer.fit_transform(df['marks'].values.reshape(-1,1))[:,0]
print(df.marks)
```

Output:

```
0    85.0
1    95.0
2    75.0
3    88.5
4    70.0
5    88.5
6    92.0
7    98.0
Name: marks, dtype: float64
```

Imputing with most frequent / mode value

```
imputer = SimpleImputer(missing_values=np.NaN, strategy='most_frequent')
df.marks = imputer.fit_transform(df['marks'].values.reshape(-1,1))[:,0]
print(df.marks)
```

Output:

```
0    85.0
1    95.0
2    75.0
3    70.0
4    70.0
5    70.0
6    92.0
7    98.0
Name: marks, dtype: float64
```

Imputing with constant value;

```
imputer = SimpleImputer(missing_values=np.NaN, strategy='constant', fill_value=80)
df.marks = imputer.fit_transform(df['marks'].values.reshape(-1,1))[:,0]
print(df.marks)
```

Output:

```
0    85.0
1    95.0
2    75.0
3    80.0
4    70.0
5    80.0
6    92.0
7    98.0
Name: marks, dtype: float64
```

#SimpleImputer for imputing Categorical Missing Data

```
imputer = SimpleImputer(missing_values=None, strategy='most_frequent')
df.gender = imputer.fit_transform(df['gender'].values.reshape(-1,1))[:,0]
print(df.gender)
```

Output:

```
0    M
1    F
2    M
3    M
4    M
5    M
6    F
7    M
Name: gender, dtype: object
```

```
imputer = SimpleImputer(missing_values=None, strategy='constant', fill_value='F')
df.gender = imputer.fit_transform(df['gender'].values.reshape(-1,1))[:,0]
print(df.gender)
```

Output:

```
0    M
1    F
2    F
3    M
4    M
5    F
6    F
7    M
Name: gender, dtype: object
```

4. Encode categorical features

When dealing with classification problems, we need to encode categorical features numerically on a continuous scale. It requires categorical features to be converted into binary arrays (0s, 1s).

To convert categorical features with Scikit-learn, use `OneHotEncoder()` with the `fit_transform()` method.

```

from sklearn.preprocessing import OneHotEncoder
df['marks'] = OneHotEncoder().fit_transform(df[['marks']]).toarray()[:,1]
df['gender'] = OneHotEncoder().fit_transform(df[['gender']]).toarray()[:,1]
df['result'] = OneHotEncoder().fit_transform(df[['result']]).toarray()[:,1]
print(df)

```

Output:

	marks	gender	result
0	0.0	1.0	0.0
1	0.0	0.0	1.0
2	1.0	0.0	0.0
3	0.0	1.0	0.0
4	0.0	1.0	0.0
5	0.0	0.0	0.0
6	0.0	0.0	0.0
7	0.0	1.0	1.0

5. Scale numeric features/Normalization

In different datasets, the ranges within features are not necessarily the same. To interpret properly, these features need to be on the same scale. So, we need to **normalize** the data.

To normalize the data, we can use

- **MinMaxScaler** transforms each feature to a given range (e.g. from 0 to 1)
- **StandardScaler** standardizes the features by removing the mean and scaling to unit variance so that each feature has $\mu = 0$ and $s = 1$.

MinMaxScaler: MinMaxScaler() put all numeric values on a scale from 0 to 1.

```

from sklearn.preprocessing import MinMaxScaler
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.NaN, strategy='mean')
df.marks = imputer.fit_transform(df['marks'].values.reshape(-1,1))[:,0]
df['marks'] = MinMaxScaler().fit_transform(df['marks'].values.reshape(-1,1))[:,0]
print(df['marks'])

```

Output:

0	0.535714
1	0.892857
2	0.178571
3	0.565476
4	0.000000
5	0.565476
6	0.785714
7	1.000000

Name: marks, dtype: float64

StandardScaler: StandardScaler() put all numeric values on a scale where the mean equals 0 and the standard deviation equals 1.

```

from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.NaN, strategy='mean')
df.marks = imputer.fit_transform(df['marks'].values.reshape(-1,1))[:,0]
df['marks'] = StandardScaler().fit_transform(df['marks'].values.reshape(-1,1))[:,0]
print(df['marks'])

```

Output:

```
0    -0.093255
1     1.025800
2    -1.212309
3     0.000000
4    -1.771836
5     0.000000
6     0.690084
7     1.361516
Name: marks, dtype: float64
```

Step 2: Splitting the dataset

To check the accuracy of our model, we can split the dataset into two pieces-**a training set** and a **testing set**. Use the training set to train the model and testing set to test the model. After that, we can evaluate how well our model did.

Advantages of train/test split:

- The model can be trained and tested on different data than the one used for training.
- Response values are known for the test dataset, hence predictions can be evaluated
- Testing accuracy is a better estimate than training accuracy of out-of-sample performance.

The **train_test_split** function takes several arguments, those are:

- **X, y**: These are the feature matrix and response vector which need to be split.
- **test_size**: It is the ratio of test data to the given data..
- **random_state**: If you use random_state = some_number, then you can guarantee that your split will be always the same. This is useful if you want reproducible results,

```
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, random_state=2)
print(X,y,X_train, X_test, y_train, y_test)
```

X		y	
0	85.0	0	0.0
1	95.0	1	1.0
2	75.0	2	0.0
3	70.0	3	0.0
4	70.0	4	0.0
5	70.0	5	0.0
6	92.0	6	0.0
7	98.0	7	1.0
Name: marks, dtype: float64		Name: result, dtype: float64	

X_train	y_train
<pre>marks 1 95.0 6 92.0 0 85.0 4 70.0 3 70.0 5 70.0 Name: result, dtype: float64</pre>	<pre>1 1.0 6 0.0 0 0.0 4 0.0 3 0.0 5 0.0 Name: result, dtype: float64</pre>
X_test	y_test
<pre>marks 7 98.0 2 75.0 Name: result, dtype: float64</pre>	<pre>7 1.0 2 0.0 Name: result, dtype: float64</pre>

Step 3: Training the model

Create KNN (K nearest neighbors) classifier:

Now, it's time to train some prediction models using our dataset. Scikit-learn provide a wide range of machine learning algorithms that have a unified/consistent interface for fitting, predicting accuracy, etc. The example given below uses KNN (K nearest neighbors) classifier.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train).values.reshape(-1,1)[: ,0]

# making predictions on the testing set
y_pred = knn.predict(X_test)

# comparing actual response values with predicted response values
from sklearn import metrics
print("kNN model accuracy:", metrics.accuracy_score(y_test, y_pred))
```

```
-----
kNN model accuracy: 0.5
-----
```


Create a logistic regression:

Now, let's create a [logistic regression](#) from the dataset. For the purpose of simplicity, I will not add training and testing datasets. I just want to show how to run a basic logistic regression machine learning model with Scikit-learn.

```
from sklearn.linear_model import LogisticRegression
lr=LogisticRegression()
lr.fit((X_train), (y_train))
y_pred = lr.predict(X_test)
from sklearn import metrics
print("LR model accuracy:", metrics.accuracy_score(y_test, y_pred))
```

```
-----
LR model accuracy: 0.5
-----
```

load the iris dataset as an example

```
from sklearn.datasets import load_iris
iris = load_iris()

# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target

# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
random_state=1)

# training the model on training set
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# making predictions on the testing set
y_pred = knn.predict(X_test)

# comparing actual response values (y_test) with predicted response values
(y_pred)
from sklearn import metrics
print("kNN model accuracy:", metrics.accuracy_score(y_test, y_pred))
```

```
kNN model accuracy: 0.9833333333333333
```