Introducing Scikit-Learn

Numerous Python libraries offer reliable executions of various machine learning algorithms. Among the most well-known is Scikit-Learn, a package that offers effective renditions of numerous widely used algorithms. Scikit-Learn is distinguished by an organized, consistent, and streamlined API in addition to extensive and helpful online documentation. One advantage of this consistency is that it makes transitioning to a new model or algorithm quite simple once you grasp the fundamental usage and syntax of Scikit-Learn for a particular type of model.

- ## Data Representation in Scikit-Learn

Since the goal of machine learning is to build models from data, we'll begin by talking about how data can be encoded so that a computer can understand it. Tables of data are the most natural way to conceptualize data in Scikit-Learn.

1- Data as table:

A basic table is a two-dimensional data grid where the rows correspond to the different components of the dataset and the columns show the quantities associated with each component. Take the Iris dataset, which Ronald Fisher renowned examined in 1936. With the seaborn package, we can get this dataset as a Pandas DataFrame:

Date	Open	High	Low	Close	Adj Close	Volume
2000-09-25	0.941964	0.991071	0.929688	0.955357	0.809837	435551200
2000-09-26	0.952009	0.977679	0.917411	0.918527	0.778617	290936800
2000-09-27	0.924107	0.941964	0.861607	0.873884	0.740774	402259200
2000-09-28	0.880580	0.960938	0.859375	0.955357	0.809837	979585600
2000-09-29	0.503348	0.517857	0.453125	0.459821	0.389781	7421640800
...	...	...	...	...	...	...
2024-01-17	181.270004	182.929993	180.300003	182.679993	182.679993	47317400
2024-01-18	186.089996	189.139999	185.830002	188.630005	188.630005	78005800
2024-01-19	189.330002	191.949997	188.820007	191.559998	191.559998	68741000
2024-01-22	192.300003	195.330002	192.259995	193.889999	193.889999	60133900
2024-01-23	195.020004	195.750000	193.829895	195.179993	195.179993	42056581

Figure 1

Figure 1 row of the data refers to a single observed stock prices in the dataset. In general, we will refer to the rows of the matrix as samples, and the number of rows as n_samples.

Likewise, each column of the data refers to a particular quantitative piece of information that describes each sample. In general, we will refer to the columns of the matrix as features, and the number of columns as n_features.

2- Features:

It is evident from this table layout that the data may be conceptualized as a two-dimensional numerical array or matrix, which we shall refer to as the features matrix. This characteristics matrix is customarily kept in a variable called X. A NumPy array or a Pandas DataFrame typically houses the features matrix, which is supposed to be two-dimensional and has the shape [n_samples, n_features] such as Figure 2. However, certain Scikit-Learn models also support SciPy sparse matrices.

Date	Open	High	Low	Close	Adj Close	Volume
2000-09-25	0.941964	0.991071	0.929688	0.955357	0.809837	435551200
2000-09-26	0.952009	0.977679	0.917411	0.918527	0.778617	290936800
2000-09-27	0.924107	0.941964	0.861607	0.873884	0.740774	402259200
2000-09-28	0.880580	0.960938	0.859375	0.955357	0.809837	979585600
2000-09-29	0.503348	0.517857	0.453125	0.459821	0.389781	7421640800
...	...	...	...	...	...	...
2024-01-17	181.270004	182.929993	180.300003	182.679993	182.679993	47317400
2024-01-18	186.089996	189.139999	185.830002	188.630005	188.630005	78005800
2024-01-19	189.330002	191.949997	188.820007	191.559998	191.559998	68741000
2024-01-22	192.300003	195.330002	192.259995	193.889999	193.889999	60133900
2024-01-23	195.020004	195.750000	193.829895	195.179993	195.179993	42056581

5868 rows × 6 columns

Figure 2

The samples (i.e., rows) always refer to the individual objects described by the dataset. For example, the sample might be a stock price, ID, document, image, sound file, video, astronomical object, or anything else you can describe with a set of quantitative measurements.

The features (i.e., columns) always refer to the distinct observations that quantitatively describe each sample. Features are generally real-valued but may be Boolean or discrete-valued in some cases (Ture or False).

3- Target Array:

In expansion to the highlight lattice X, we moreover by and large work with a name or target cluster, which by tradition we are going ordinarily call y. The target cluster is ordinarily one dimensional, with length n_samples, and is by and large contained in a NumPy cluster or Pandas Arrangement. The target cluster may have ceaseless numerical values, or discrete classes/labels. Whereas a few Scikit-Learn estimators do handle different target values within the form of a two-dimensional, [n_samples, n_targets] target cluster, we are going basically be working with the common case of a one-dimensional target cluster.

Frequently one point of disarray is how the target cluster contrasts from the other highlights columns. The recognizing highlight of the target cluster is that it is more often than not the amount we need to foresee from the information:

in measurable terms, it is the subordinate variable. For this case, within the going before information we may wish to build a demonstration that can anticipate the species of blossom based on the other estimations; in this case, the species column would be considered the target cluster.

With this target array in intellect, we can utilize Seaborn (to helpfully visualize the information as Figure 3:

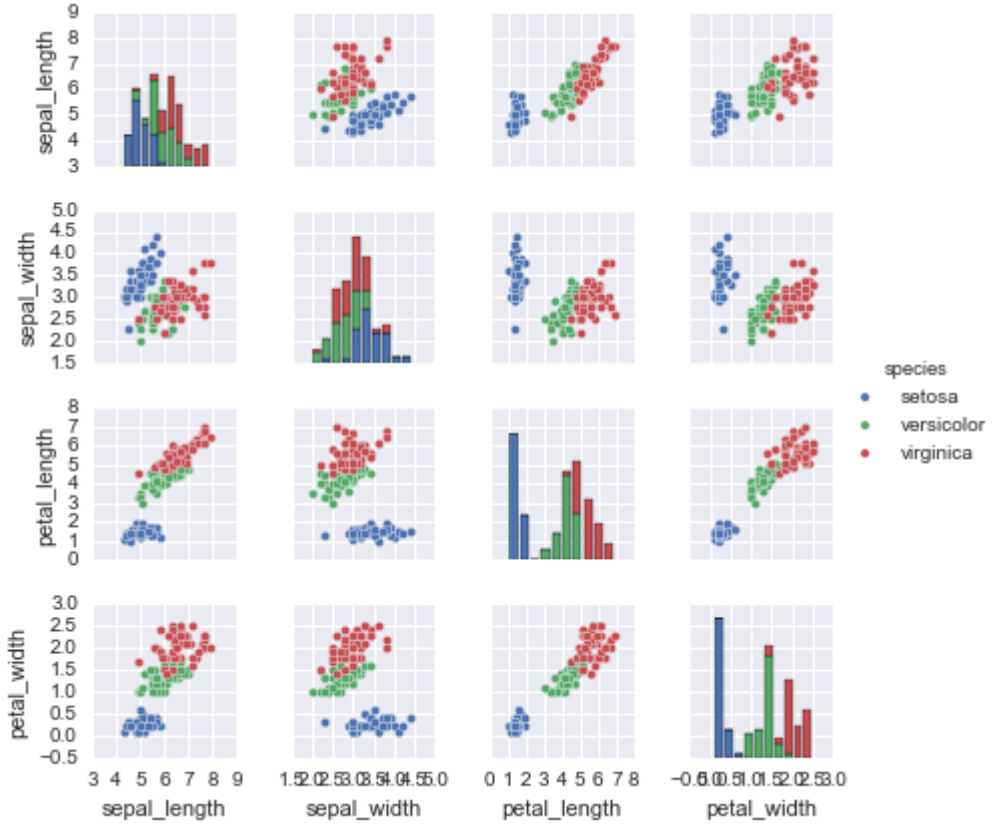

Figure 3

- Scikit-Learn's Estimator API

The Scikit-Learn API is designed with the following guiding principles in mind, as outlined in the Scikit-Learn API paper:

- Consistency: All objects share a common interface drawn from a limited set of methods, with consistent documentation.

- Inspection: All specified parameter values are exposed as public attributes.

- Limited object hierarchy: Only algorithms are represented by Python classes; datasets are represented in standard formats (NumPy arrays, Pandas DataFrames, SciPy sparse matrices) and parameter names use standard Python strings.

- Composition: Many machine learning tasks can be expressed as sequences of more fundamental algorithms, and Scikit-Learn makes use of this wherever possible.

-
- Sensible defaults: When models require user-specified parameters, the library defines an appropriate default value.
-

In practice, these principles make Scikit-Learn very easy to use,once the basic principles are understood. Every machine-learning algorithm in Scikit-Learn is implemented via the Estimator API, which provides a consistent interface for a wide range of machine-learning applications.

1- Basics of the API:

Most commonly, the steps in using the Scikit-Learn estimator API are as follows

1. Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
2. Choose model hyperparameters by instantiating this class with desired values.
3. Arrange data into a features matrix and target vector following the discussion above.
4. Fit the model to your data by calling the fit() method of the model instance.
5. Apply the Model to new data:
    - For supervised learning, often we predict labels for unknown data using the predict() method.
    - For unsupervised learning, we often transform or infer properties of the data using the transform() or predict() method.

We will now step through several simple examples of applying supervised and unsupervised learning methods.

2- Supervised learning example: Simple linear regression

As an example of this process, let's consider a simple linear regression—that is, the common case of fitting a line to (x,y) data. We will use the following simple data for our regression example as Figure 4:

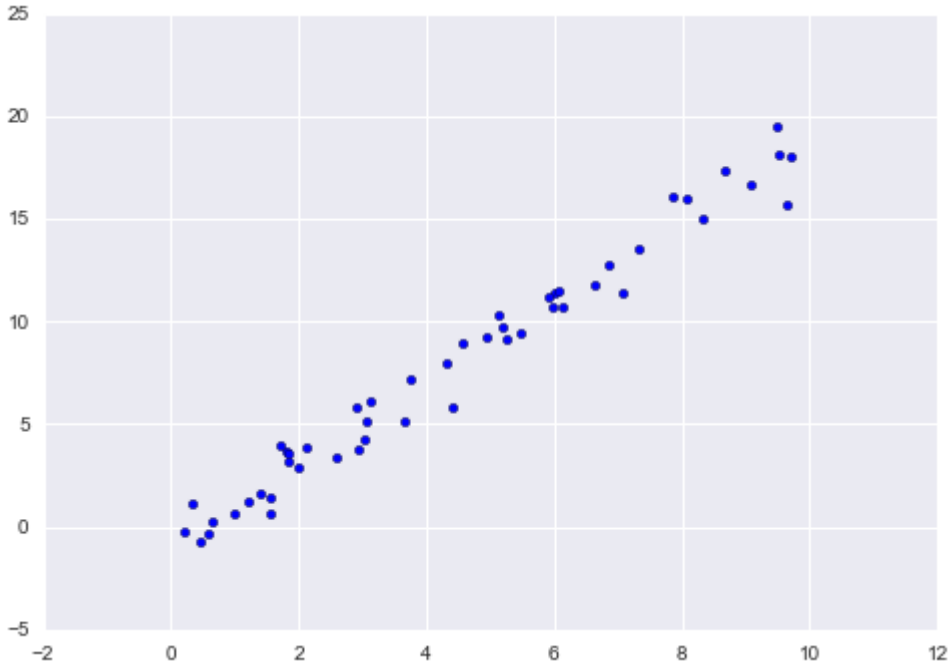

Figure 4

Choose a class of model

In Scikit-Learn, every class of model is represented by a Python class. So, for example, if we would like to compute a simple linear regression model, we can import the linear regression class:

```
from sklearn.linear_model import LinearRegression
```

Figure 5

Choose model hyperparameters

An important point is that a class of model is not the same as an instance of a model.

Once we have decided on our model class, there are still some options open to us. Depending on the model class we are working with, we might need to answer one or more questions like the following:

- Would we like to fit for the offset (i.e., y-intercept)?
- Would we like the model to be normalized?
- Would we like to preprocess our features to add model flexibility?
- What degree of regularization would we like to use in our model?
- How many model components would we like to use?

These are examples of the important choices that must be made once the model class is selected. These choices are often represented as hyperparameters, or parameters that must be set before the model is fit to data. In Scikit-Learn, hyperparameters are chosen by passing values at model instantiation. We will explore how you can quantitatively motivate the choice of hyperparameters in Hyperparameters and Model Validation.

For our linear regression example, we can instantiate the LinearRegression class and specify that we would like to fit the intercept using the fit_intercept hyperparameter:

```
model = LinearRegression(fit_intercept=True)
model
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Figure 6

Be beyond any doubt that when the demonstrate is instantiated, the as it were activity is the putting away of these hyperparameter values. In specific, we have not however connected the demonstrate to any information:

the Scikit-Learn API makes exceptionally clear the qualification between choice of show and application of show to information.

Arrange data into a features matrix and target vector

Previously we detailed the Scikit-Learn data representation, which requires a two-dimensional features matrix and a one-dimensional target array. Here our target variable y is already in the correct form (a length-n_samples array), but we need to massage the data x to make it a matrix of size [n_samples, n_features]. In this case, this amounts to a simple reshaping of the one-dimensional array:

```
X = x[:, np.newaxis]
X.shape
```

```
(50, 1)
```

Figure 7

Fit the model to your data

In Figure 8, a apply our model to data. This can be done with the `fit()` method of the model:

```
model.fit(X, y)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Figure 8

This fit() command causes a number of model-dependent internal computations to take place, and the results of these computations are stored in model-specific attributes that the user can explore. In Scikit-Learn, by convention all model parameters that were learned during the fit() process have trailing underscores in Figure 9;

```
model.coef_
```

```
array([ 1.9776566])
```

```
model.intercept_
```

```
-0.90331072553111635
```

Figure 9

These two parameters represent the slope and intercept of the simple linear fit to the data. Comparing to the data definition, we see that they are very close to the input slope of 2 and intercept of -1.

One question that frequently comes up regards the uncertainty in such internal model parameters. In general, Scikit-Learn does not provide tools to draw conclusions from internal model parameters themselves: interpreting model parameters is much more a statistical modeling question than a machine learning question. Machine learning rather focuses on what the model predicts. If you would like to dive into the meaning of fit parameters within the model, other tools are available, including the Statsmodels Python package.

Predict labels for unknown data

Once the model is trained, the main task of supervised machine learning is to evaluate it based on what it says about new data that was not part of the training set. In Scikit-Learn, this can be done using the predict() method. For the sake of this example, our "new data" will be a grid of x values, and we will ask what y values the model predicts:

```
xfit = np.linspace(-1, 11)
```

Figure 10

As before, we need to coerce these x values into a [n_samples, n_features] features matrix, after which we can feed it to the model:

```
Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)
```

Figure 11

Finally, let's visualize the results by plotting first the raw data in Figure 12, and then this model fit:

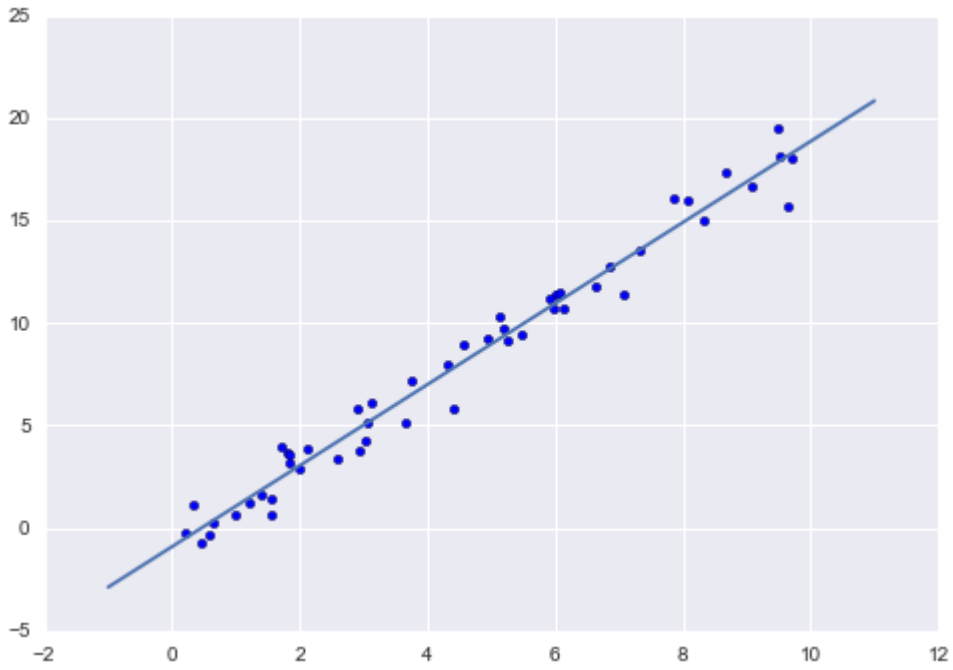

Figure 12

3- Unsupervised learning example: Iris dimensionality

As an example of an unsupervised learning problem, let's take a look at reducing the dimensionality of the Iris data so as to more easily visualize it. Recall that the Iris data is four dimensional: there are four features recorded for each sample.

The task of dimensionality reduction is to ask whether there is a suitable lower-dimensional representation that retains the essential features of the data. Often dimensionality reduction is used as an aid to visualizing data: after all, it is much easier to plot data in two dimensions than in four dimensions or higher!

Here we will use principal component analysis (PCA; see In Depth: Principal Component Analysis), which is a fast linear dimensionality reduction technique. We will ask the model to

return two components—that is, a two-dimensional representation of the data. Following the sequence of steps outlined earlier, we have:

```
from sklearn.decomposition import PCA
model = PCA(n_components=2)
model.fit(X_iris)
X_2D = model.transform(X_iris)
```

Figure 13

Now let's plot the results. A quick way to do this is to insert the results into the original Iris DataFrame, and use Seaborn's lmplot to show the results:

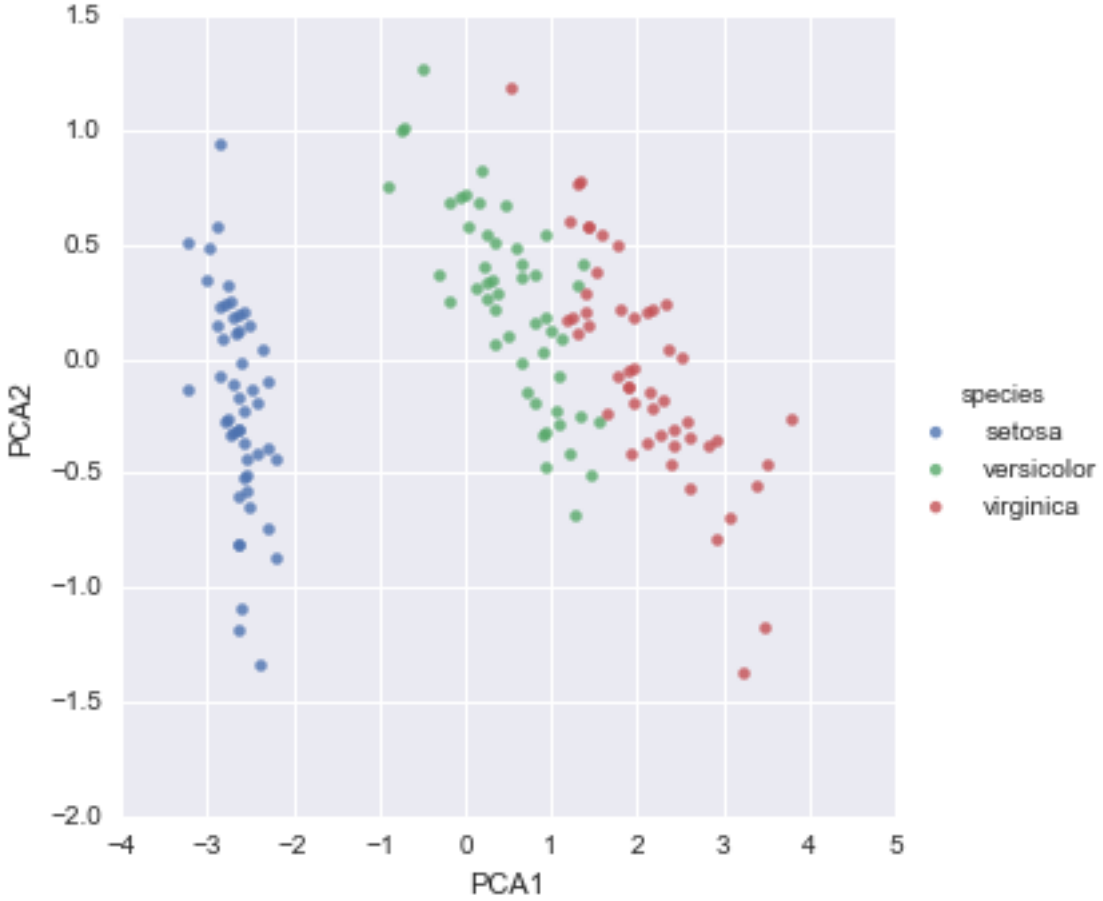

Figure 14

We see that within the two-dimensional representation, the species are decently well isolated, indeed in spite of the fact that the PCA calculation had no information of the species names! This shows to us that a generally direct classification will likely be successful on the dataset, as we saw some time recently.

4- Unsupervised learning: Iris clustering

A clustering algorithm attempts to find distinct groups of data without reference to any labels. Here we will use a powerful clustering method called a Gaussian mixture model (GMM), discussed in more detail in In Depth: Gaussian Mixture Models. A GMM attempts to model the data as a collection of Gaussian blobs.

We can fit the Gaussian mixture model as follows:

```
from sklearn.mixture import GMM
model = GMM(n_components=3,
            covariance_type='full')
model.fit(X_iris)
y_gmm = model.predict(X_iris)
```

Figure 15

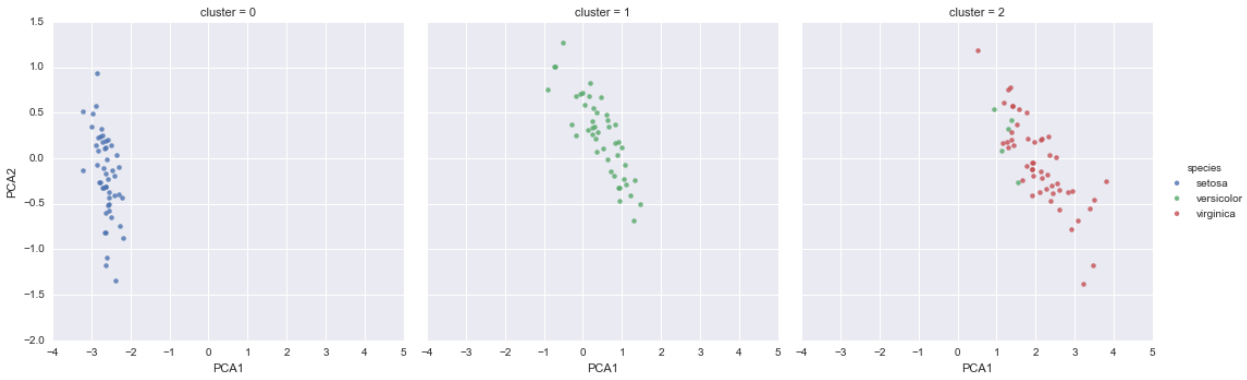

Figure 16

By splitting the data by cluster number, we see exactly how well the GMM algorithm has recovered the underlying label: the setosa species is separated perfectly within cluster 0, while there remains a small amount of mixing between versicolor and virginica. This means that even without an expert to tell us the species labels of the individual flowers, the measurements of these flowers are distinct enough that we could automatically identify the presence of these different groups of species with a simple clustering algorithm! This sort of algorithm might further give experts in the field clues as to the relationship between the samples they are observing.