2024/09/11

Introducing Scikit-Learn

Several Python libraries provide solid implementations of a range of machine-learning algorithms. One of the best-known is `Scikit-Learn`, a package that provides efficient versions of many common algorithms. `Scikit-Learn` is characterized by a clean, uniform, and streamlined API and by very useful and complete online documentation. **A benefit of this uniformity is that once you understand the basic use and syntax of `Scikit-Learn` for one type of model, switching to a new model or algorithm is very straightforward.**

This section provides an overview of the `Scikit-Learn` API; a solid understanding of these API elements will form the foundation for understanding the deeper practical discussion of machine learning algorithms and approaches in the following chapters.

We will start by covering data representation in `Scikit-Learn`, and followed by covering the Estimator (algorithm) API.

1. Data Representation in Scikit-Learn

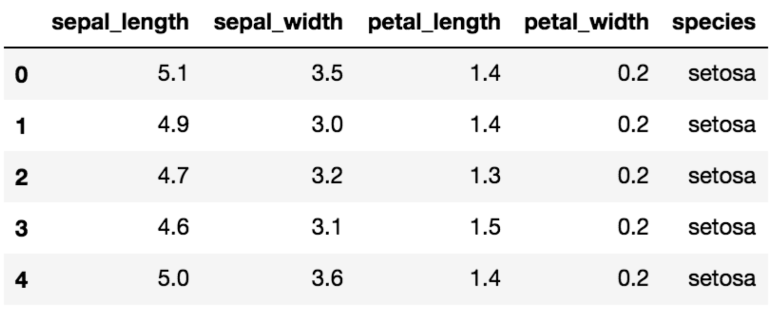
Machine learning is about creating models from data: for that reason, we'll start by discussing how data can be represented to be understood by the computer. The best way to think about data within `Scikit-Learn` is in terms of **tables of data**.

Data as table

A basic table is a two-dimensional grid of data in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements. For example, consider the Iris dataset as we mentioned before. We can download this dataset in the form of a Pandas `DataFrame` using the Seaborn library[1]:

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

[1] Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. See https://seaborn.pydata.org



	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Here each row of the data refers to a single observed flower, and the number of rows is the total number of flowers in the dataset. In general, we will refer to the rows of the matrix as *samples* and the number of rows as `n_samples`. In the Iris set, we have `n_samples=150`.

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`.

Features matrix

This table layout makes clear that the information can be thought of as a two-dimensional numerical array or matrix, which we will call the features matrix. By convention, this features matrix is often stored in a variable named `X`. The features matrix is assumed to be two-dimensional, with shape `[n_samples, n_features]`, and is most often contained in a NumPy array or a Pandas `DataFrame`.

Target array

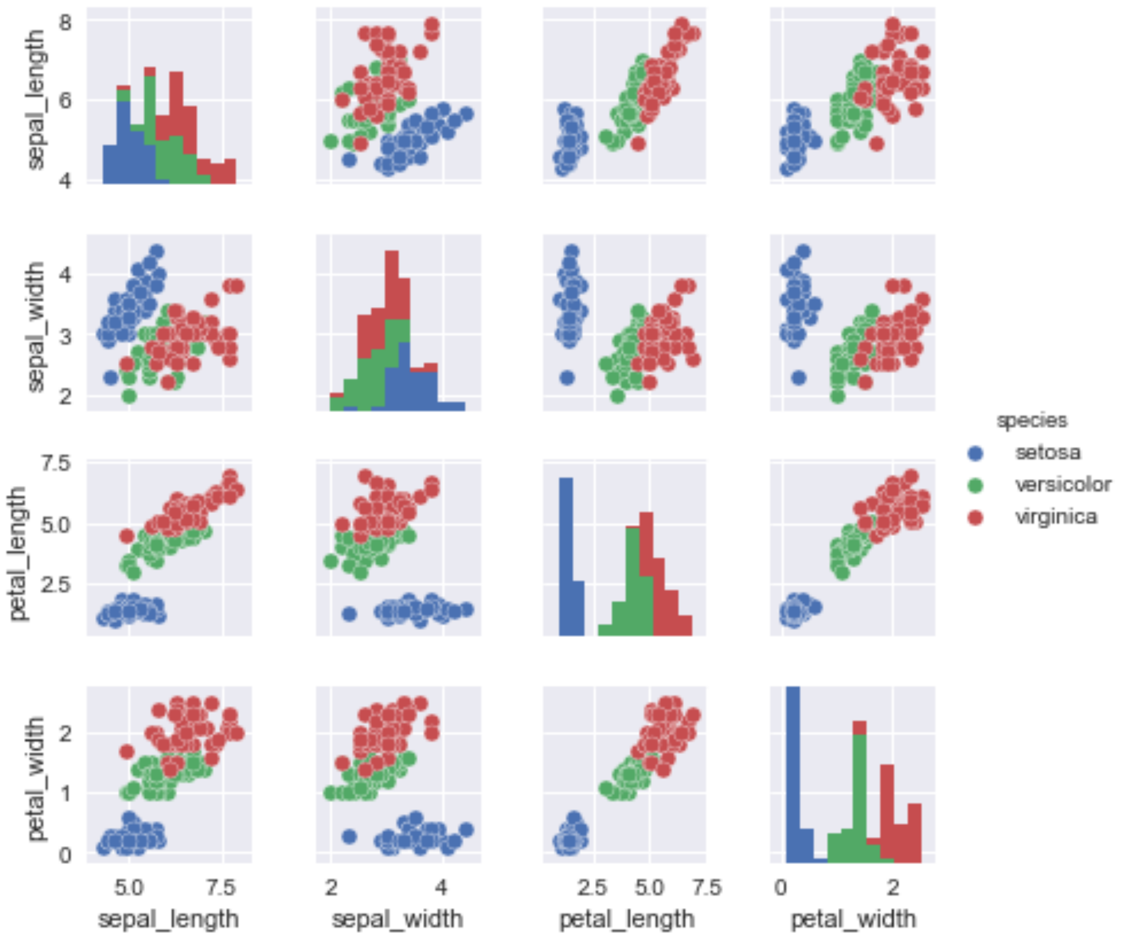
In addition to the feature matrix `X`, we also generally work with a label or target array, which by convention we will usually call `y`.

The target array is usually one dimensional, with length `n_samples`, and is generally contained in a NumPy array or Pandas `Series`. The target array may have continuous numerical values, or discrete classes/labels. While some `Scikit-Learn` estimators do handle multiple target values in the form of a two-dimensional, `[n_samples, n_targets]` target array, we will primarily be working with the common case of a one-dimensional target array.

The target array is the quantity we want to predict from the data. For example, in the preceding data we may wish to construct a model that can predict the species of flower based on the other measurements; in this case, the species column is the target array.

With this target array in mind, we can use Seaborn to conveniently visualize the data. Visualizing the multidimensional relationships among the samples is as easy as calling `sns.pairplot`:

```
%matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', size=1.5);
```



For use in `Scikit-Learn`, we will extract the features matrix and target array from the `DataFrame`, which we can do using some of the Pandas `DataFrame` operations:

```
X_iris = iris.drop('species', axis=1)
X_iris.shape
```

The parameter `axis=1` indicates to drop columns of the labels.

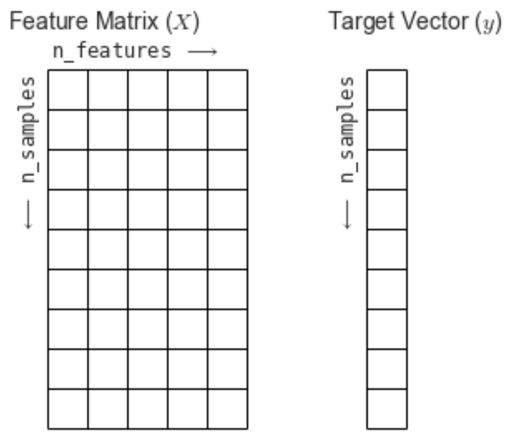
Q: what is the output `X_iris.shape`?

A:

```
(150, 4)
```

```
y_iris = iris['species']
```

To summarize, the expected layout of features and target values is visualized in the following diagram:



With this data properly formatted, we can move on to consider the estimator API of `Scikit-Learn`.

2. Scikit-Learn's Estimator (Algorithm) API

The `Scikit-Learn` API is designed with the following guiding principles in mind:

- **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`) 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-Lear`n is implemented via the Estimator API, which provides a consistent interface for a wide range of machine learning applications.

Basics of the API

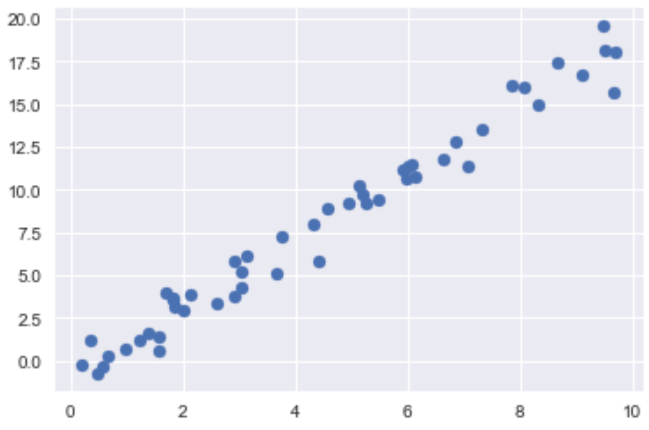
Most commonly, the steps in using the `Scikit-Learn` estimator API are as follows (we will step through a handful of detailed examples later).

1. Choose a class of model by importing the appropriate estimator class from `Scikit-Learn`.
2. Choose model **hyperparameters**[2] 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 two simple examples of applying supervised learning methods.

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:



```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50) #normal distribution
plt.scatter(x, y);
```

With this data in place, we can use the recipe outlined earlier. Let's walk through the process:

[2] In machine learning, **hyperparameters** refer to parameters that must be set before the model is fit to data.

1. 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
```

2. 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., 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** (the 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 the sequel.

   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
```

   Keep in mind that when the model is instantiated, the only action is the storing of these hyperparameter values. In particular, we have not yet applied the model to any data: the `Scikit-Learn` API makes very clear the distinction between the **choice of model and application of the model to data**.

3. Arrange data into a feature 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 #output (50,1)
```

The `newaxis` object is used in all slicing operations to create an axis of length one.

4. Fit the model to your data

Now it is time to apply our model to data. This can be done with the `fit()` method of the model:

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

This `fit()` command causes a number of model-dependent internal computations to take place. 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; for example in this linear model, we have the following:

```
model.coef_
model.intercept_
```

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

5. 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:

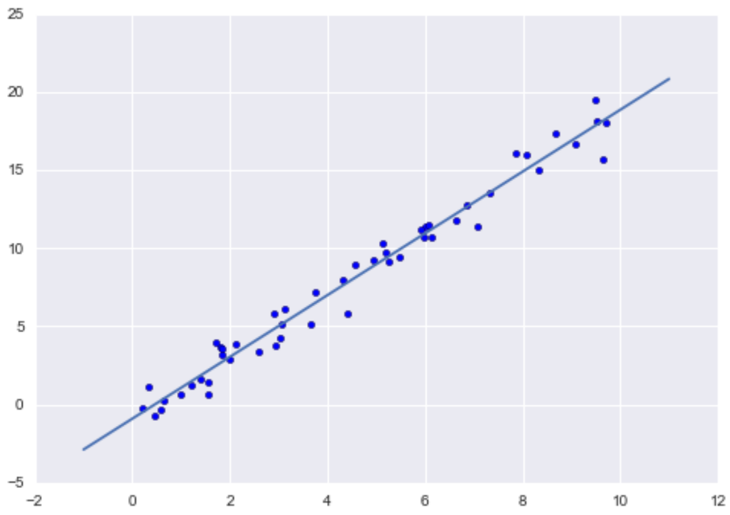
```
xfit = np.linspace(-1, 11)
```

As before, we need to reshape 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)
```

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

```
plt.scatter(x, y)
plt.plot(xfit, yfit);
```



<u>Recipe Recap and Summary</u>: **<u>import/instantiate/fit/predict</u>**

The process of doing machine learning using `Scikit-Learn` consists of the following **four** steps:

1. Choose a `class` of model by `import`ing the appropriate estimator class from `Scikit-Learn`.

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

2. Choose model hyperparameters by instantiating this `class` with desired values. For example

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

3. Fit the model to your data by calling the `fit()` method of the model instance.

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

4. Apply the Model to new data:

- For supervised learning, we often <u>predict labels for unknown data</u> using the `predict()` method.

```
yfit = model.predict(Xfit)
```

Remark: Hyperparameters vs. Parameters

1. A hyperparameter is a property of a learning algorithm, usually (but not always) having a numerical value. That value influences the way the algorithm works. Hyperparameters aren't learned by the algorithm itself from data. They have to be set by you before running the algorithm.
2. Parameters are variables that define the model learned by the learning algorithm. Parameters are directly modified by the learning algorithm based on the training data. The goal of learning is to find such values of parameters that make the model optimal in a certain sense.

Supervised learning example: Simple classification

Let's take a look at another example of this import/instantiate/fit/predict **recipe**, using the Iris dataset we discussed earlier. Our question will be this: **given a model trained on a portion of the Iris data, how well can we predict the remaining labels**?

For this task, we will use a support vector machine classifier (SVC). We would like to evaluate the model on data it has not seen before.

(Why evaluate the model on data it has not seen before? Conceptual Analogy) Before the final exam, I may hand out some practice problems and solutions to the class. Although these problems are not the exact ones that will appear on the exam, studying them will help you do better. They are the 'training set' in your learning.

If my goal is to help you do better in the exam, why not give out the exam problems themselves? Well, nice try ☺.

Q: Why not?
A: Doing well in the exam is not the goal. **The goal is for you to learn the course material.** The exam is merely a way to gauge how well you have learned the material. If the exam problems are known ahead of time, your performance on them will no longer accurately gauge how well you have learned. The same distinction between training and testing happens in learning from data.

We will **split the data into a training set and a testing set**. This could be done by hand, but it is more convenient to use the `train_test_split` utility function from `Scikit-Learn`:

```
from sklearn.datasets import load_iris
iris = load_iris()
X_iris, y_iris = iris.data, iris.target
```

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
random_state=1)
```

With the data arranged, we can follow our import/instantiate/fit/predict **recipe** to predict the labels:

```
from sklearn.svm import SVC # 1. choose "Support vector classifier"
model = SVC(kernel='rbf', gamma=0.01, C=10) # 2. instantiate model
model.fit(Xtrain, ytrain)                   # 3. fit model to data
y_model = model.predict(Xtest)              # 4. predict on new data
```

Finally, we can use the `accuracy_score` utility to see the fraction of predicted labels that match their true value:

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
```

```
0.9736842105263158
```

The `accuracy_score` function computes the accuracy. If $\hat{y}_i$ is the predicted value of the i^{th} sample and y_i is the corresponding true value, then the fraction of correct predictions over N samples is defined as:

$$\text{accuracy}(y_i, \hat{y}_i) = \frac{1}{N} \sum_{n=1}^{N} 1(\hat{y}_i = y_i)$$

The $1(\hat{y}_i = y_i)$ is an indicator function; it is 1.0 if $\hat{y}_i = y_i$ and 0.0 if $\hat{y}_i \neq y_i$.

With an accuracy topping 97%, we see that this classification algorithm is effective for this dataset.

3. Summary

In this section we have covered the essential features of the `Scikit-Learn` data representation, and the estimator API. Regardless of the type of estimator, the same **import/instantiate/fit/predict** process holds. Armed with this information about the estimator API, you can explore the `Scikit-Learn` documentation and begin trying out various models on your data. See http://scikit-learn.org/stable.