Adapded for class presentation by Claudio Sartori - University of Bologna

Introducing Scikit-Learn

Scikit-Learn

- package that provides efficient versions of a large number of common algorithms
- clean, uniform, and streamlined API
- very useful and complete online documentation.
  - 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

Contents

Data Representation in Scikit-Learn

Data as table

- a two-dimensional grid of data
  - rows represent individual elements of the data set
  - columns represent quantities related to each of these elements
- Example: Iris dataset
  - analyzed by Ronald Fisher in 1936
  - download this dataset in the form of a Pandas `DataFrame`

In [ ]:
```
import seaborn as sns
import pandas as pd
```

Download the **Iris** dataset at the url `https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data` or from your local file, if you already have it. The file does not have header, use as column names the list below, inspect the text file to see which character is used as separator.

`'sepal length', 'sepal width', 'petal length', 'petal width', 'species'`

Use the dataframe name `iris`. Show the head of `iris`

As an alternative way of loading the data, you can use this utility included in scikit-learn

--> Insert your code in new cell below

In [ ]:
```
url = './iris.csv'
iris = pd.read_csv(url, names=['sepal length', 'sepal width', 'petal length', 'peta
iris.head()
```

Out[ ]:

	sepal length	sepal width	petal length	petal width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

- each row refers to a single observed flower
  - the number of rows is the total number of flowers in the dataset.
  - *sample*: a single row
  - `n_samples` : number of rows
- each column refers to a piece of information that describes each sample
  - *feature*: a single column `n_features` : the number of columns
    - each column has a data type: number (continuous), boolean, discrete (nominal or ordinal, represented with integers or strings)

Features matrix

The part of the data matrix containing the `unsupervised attributes`

Usually in *scikit-learn* documentation referred as `X`

Can be a:

- two-dimensional numpy array with shape `[n_samples, n_features]`
- SciPy `sparse matrix`
- Pandas `DataFrame`

The matrix cases require uniform data types in columns

Target array

label or *target* array, by convention usually called `y`

- usually one dimensional, with length `n_samples`,
- generally contained in a NumPy array or Pandas `Series`.
- may have continuous numerical values, or discrete classes/labels
- usually it the quantity we want to *predict from the data*

- in statistical terms, it is the dependent variable

In the example we may wish to construct a model that can predict the species of flower based on the other measurements

The measurements of the flower components are the `features array`

The `species` column can be considered the target array

Visualization

Use Seaborn (see Visualization With Seaborn) to visualize the data

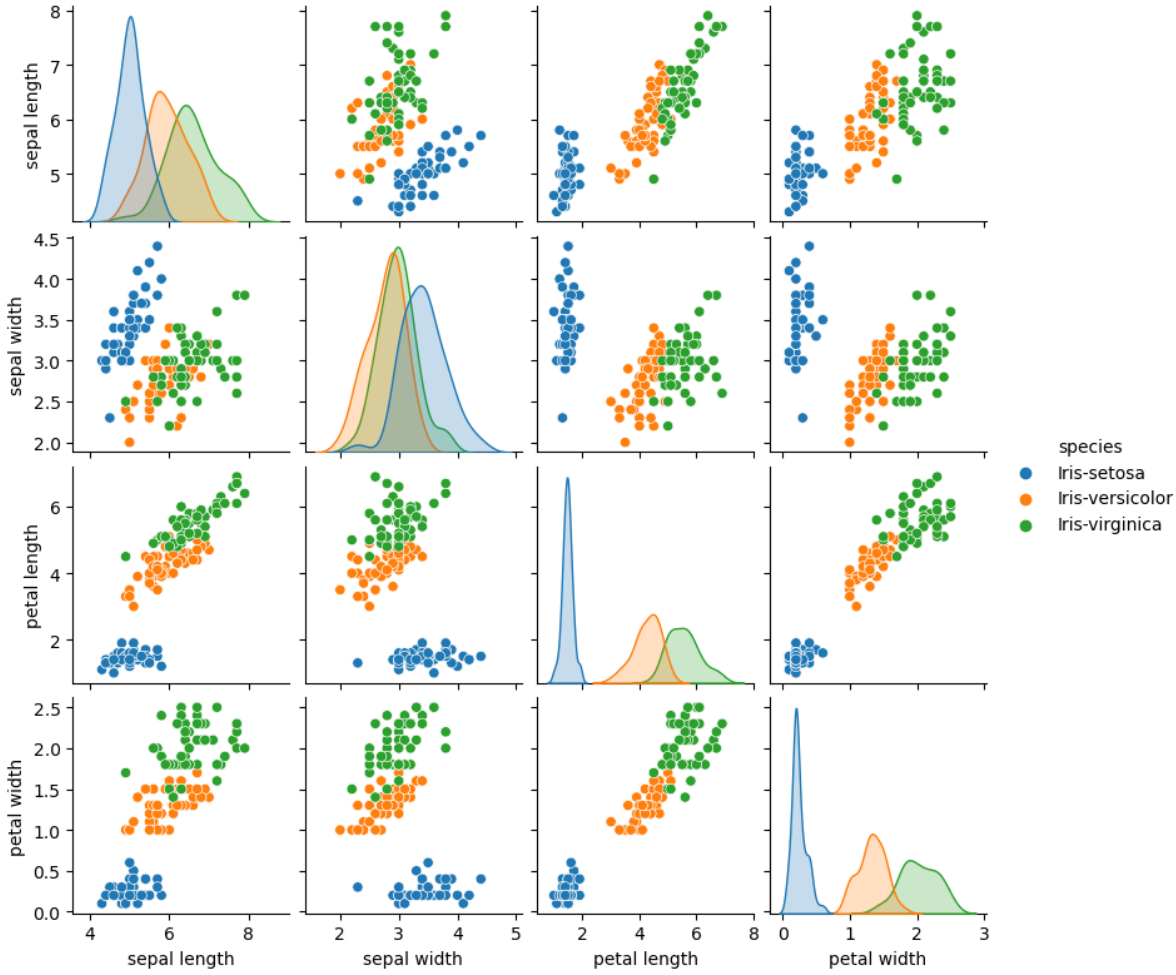
Below we need to prepare the environment for plotting information on the dataset.

1. issue the command `%matplotlib inline` In this way, the output of plotting commands is displayed inline within frontends like the Jupyter notebook, directly below the code cell that produced it. The resulting plots will then also be stored in the notebook document.
2. import `seaborn` giving it the 'nickname' `sns`
3. call the `pairplot` function of `seaborn` on the `iris` dataset, with parameters
   - `hue = 'species'`, this sets the meaning of the color in the plot of the points of the dataset
   - `height = 2`, this sets the size of the plots

--> insert your code in a new cell below this one

In [ ]:
```
%matplotlib inline
sns.pairplot(iris, hue='species',
             height=2)
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x20ba097dac0>
```



For use in Scikit-Learn, we will extract the features matrix and target array from the `DataFrame`. We can do this using some of the Pandas `DataFrame` operations discussed in the Chapter 3 of the above mentioned book.

For example, the `.drop` method allows to drop a column or row by name; remember to specify the axis to use, which is 1 for columns.

Preparing features and target

Store in X the content of `iris` excluding the column `species`. Verify the shape

--> insert your code in a new cell below this one

In [ ]:
```
X = iris.drop('species', axis=1)
print(X.shape)
```

```
(150, 4)
(150,)
```

Store in `y` the column `species` of `iris`. Verify the shape

--> insert your code in a new cell below this one

In [ ]:
```
Y = iris['species']
print(Y.shape)
```

```
(150,)
```

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

following diagram:  Figure With this data properly formatted, we can move on to consider the estimator API of Scikit-Learn:

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 `DataFrame`s, 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.

Hyperparameters

The machine learning algorithms are designed to learn from the data the *parameters* that will be used at run time by the algorithms implementing the tasks to perform at the best on data similar to those used in learning.

For example, a *decision tree* (and in particular all the tests placed in the nodes) are the parameters of a *decision tree classifier*

The learning process is also controlled by other parameters (e.g. to control the *overfitting*) which cannot be direcly learned from the data, but are chosen *before* the learning process. Those are called **hyperparameters**

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 in the sections that follow).

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.
    - or in the first attempt use the default 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.

Supervised learning example: Iris classification

Let's take a look at another example of this process, 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 the *Decision Tree* algorithm, with the standard parameter values. We would like to evaluate the model on data it has not seen before, and so 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

1. Import the method `train_test_split` from `sklearn.model_selection`
2. Generate the variables `Xtrain, Xtest, ytrain, ytest` by calling the function `train_test_split` with parameters `X` and `y`, and the additional parameter `random_state = 1`
3. Show the shape of the resulting variables

--> insert your code in a new cell below this one

In [ ]:
```
from sklearn.model_selection import  train_test_split
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, random_state = 1)
print(Xtrain.shape, Xtest.shape, Ytrain.shape, Ytest.shape)
```

```
(112, 4) (38, 4) (112,) (38,)
```

With the data arranged, we can follow our recipe to predict the labels:

1. choose the model class, it will be `DecisionTreeClassifier`, imported from `sklearn.tree`
2. instantiate the `model` as a `DecisionTreeClassifier` whithout any hyperparameter, we will use the defaults
3. fit the `model` to data, calling its method `fit` with parameters `Xtrain, ytrain`
4. predict the target `ytrain_model` using the `predict` method of `model` on the `Xtrain` data

--> insert your code in a new cell below this one

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

model = DecisionTreeClassifier()
```

```
np.random.seed(15)
model.fit(Xtrain, Ytrain)
Ytrain_model = model.predict(Xtrain)
```

We can use the `accuracy_score` utility to see the fraction of predicted training set labels that match their true value.

Import the `accuracy_score` from `sklearn.metrics` and call it on `ytrain, ytrain_model`

--> insert your code in a new cell below this one

In [ ]:
```
from sklearn.metrics import accuracy_score
print("The accuracy on training set is " + str(round(accuracy_score(Ytrain, Ytrain_
```

```
The accuracy on training set is 100.0%
```

Finally, predict the new target `ytest_model` using the `predict` method of `model` on the `Xtest` data, then compute the accuracy on the test set

--> insert your code in a new cell below this one

In [ ]:
```
Ytest_model = model.predict(Xtest)
print("The accuracy on testing set is " + str(round(accuracy_score(Ytest, Ytest_mo
```

```
The accuracy on testing set is 97.37%
```

Show the Decision Tree

To show the Decision Tree we will need a few imports

```
from matplotlib import pyplot
from sklearn.tree import plot_tree
from matplotlib.pyplot import figure
```

We will start setting the *figure size* with the `figure` function, taking as argument `figsize` and a list of two values in inches, try and error for the measures you like.

We will then use the `plot_tree` function of `sklearn.tree`. It takes as argument the *fitted model*\, in our case `model` and several arguments to control how the tree is displayed.

I suggest the arguments below, you can try freely configurations and omissions of the parameters, to use the defaults. The parameters must follow the model variable and be separated by commas, the order is not relevant, since the parameters are named.

`filled=True` \ `feature_names = ['sepal length', 'sepal width', 'petal length', 'petal width']` \ `class_names = ['setosa', 'versicolor', 'virginica']` \ `rounded = True` \ `proportion = True` \ `rotate = False`

--> insert your code in a new cell below this one

In [ ]:
```
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
```

```
plt.figure(figsize=(10, 10))
plot_tree(model, filled=True, feature_names=['sepal length', 'sepal width', 'petal
                  class_names = ['setosa', 'versicolor', 'virginica'], rounded = Tru
print()
```