Importing and plotting data, and an example machine learning classifier

Introduction

In this practical we will cover what is always an important first step in any machine learning project, that of importing and viewing your data.

To demonstrate this we will use the 'Iris' dataset introduced in the videos and articles. We'll use some of the Python libraries we've introduced, including Scikit-learn and Pandas to import and manipulate the data, and Matplotlib for plotting and visualising the data. Finally, we will demonstrate how we might train a simple machine learning classifier using Scikit-Learn (the K-nearest neighbour classifier).

You can do this practical using a Python shell, but you might find it easier to save the code as a separate script in a text editor, or an IDE such as Spyder, then run from the command line (or within Spyder). That way you should find it easier to correct any errors, re-run the code, and experiment with changing parts of the code.

Step 1: Importing and viewing the data.

The first thing we need to do is load the Iris dataset. Scikit learn provides a helper function called **load_iris** we can access as follows:

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

The function **load_iris** returns a scikit learn **bunch** object. You can think of this as a Python dictionary where you can access the contents as attributes as well as keys.

You can view the names of these attributes/keys in same way you would a regular Python dictionary:

```
print(iris.keys())
```

As you might expect **DESCR** and **filename** contain a description of the dataset and the full path the the data file itself.

The data itself is conveniently already split into the format we want, i.e. a features matrix and target vector, contained in **data** and **target** respectively. Similarly, the feature names and target names are contained in the attributes **feature_names** and **target_names**. We can look at these as follows:

```
print('Feature names:',iris.feature_names)
print('Data shape:',iris.data.shape)
print('Target names:',iris.target_names)
print('Target shape:',iris.target.shape)
```

As you will see when you run the above code, the dataset contains 150 observations with 4 features (width and length in cm for both petals and sepals of the flowers).

The target vector has length 150, and contains the species of each plant, which consists of a list of integers (0,1 and 2 in this case) that refer to the list of target names ['setosa', 'versicolor','virginica'].

For example, if target = 0 it is the species 'setosa', if target = 1 then the species is 'versicolor', and so on.

We can now proceed by assigning the names **X** and **y** to the **data** and **target** attributes of **iris** as follows:

```
X = iris.data
y = iris.target
```

Then we can use these more convenient names to visualise the data and train a machine learning model, as we'll see in the following sections.

Aside: Pandas DataFrames.

If when we call the 'load_iris' function, we add the argument 'as_frame=True' we also get the data in the Pandas DataFrame format, contained in the attribute 'frame':

```
iris = load_iris(as_frame=True)
df = iris.frame
print(type(df))
```

Since we have a DataFrame, we can use the function **head** to look at the first few rows of our data:

```
print(df.head())
```

By using **head**, you should see that the DataFrame has both the four fields of numerical data and the categorical **target** data in the same table. To use it we will want to split it into feature data and target, as before.

Though we've already seen we can get this data in the desired features/target format directly using the **load_iris** function, it's worth looking at how we can manipulate our data if we just have a DataFrame as is likely to be the case with new data imported from an external file such as a csv file.

To get only the features we can use the handy Pandas function 'drop()' as follows:

```
X = df.drop('target', axis=1)
print(X.head())
```

The axis=1 argument is essential to drop columns rather than rows.

To extract only the target column, it is simpler still:

```
y = df['target']
```

However we choose to import the data, the important thing is we now have our features matrix called X, and target vector y ready for further use in the correct format.

Step 2: visualising the data.

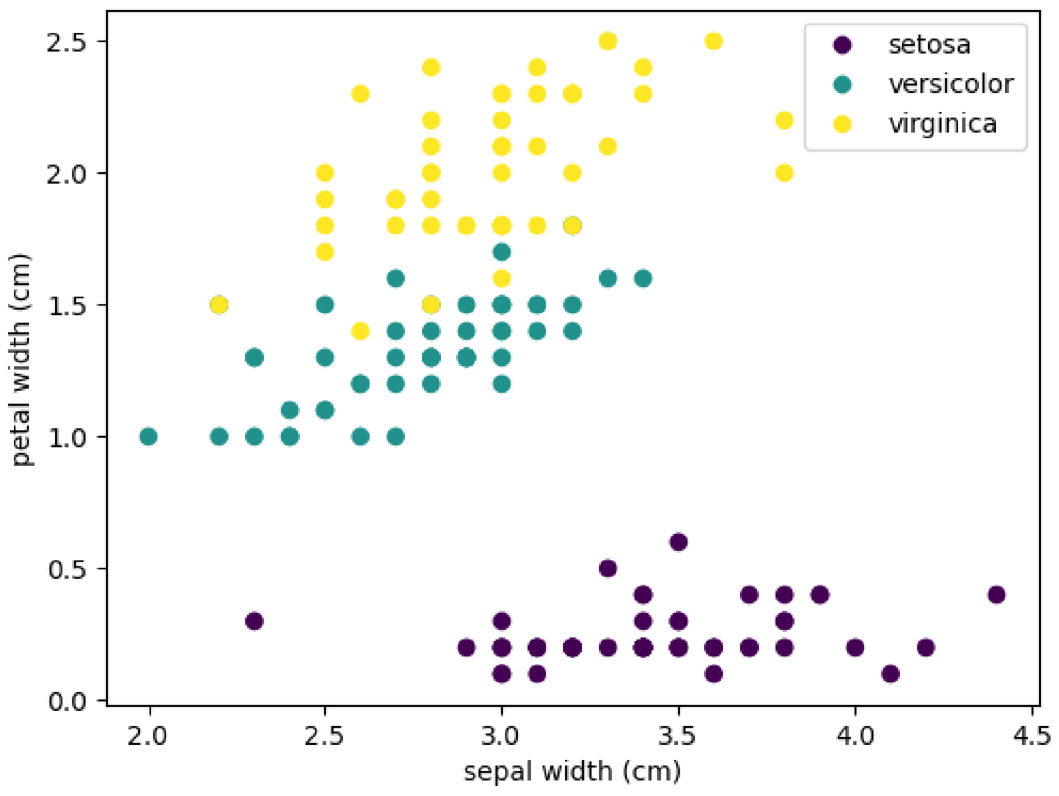
We've already seen how we can look at or display the raw data and the feature names in our terminal window. More useful however is to visualise our data using some plotting tool, which we will do here using Matplotlib.

For any dataset there are numerous ways one can visualise the data contained in it. We'll just look at one simple one here, a scatter plot of petal width versus sepal width, which we'll colour using the target data.

```
import matplotlib.pyplot as plt
xy_plot = plt.scatter(X['sepal width (cm)'],X['petal width (cm)'],c=y)
plt.xlabel('sepal width (cm)')
plt.ylabel('petal width (cm)')
plt.legend(handles=xy_plot.legend_elements()[0],
           labels=list(iris.target_names))
plt.show()
```

The first line imports the pyplot module, the second makes the plot, the third and fourth add axis labels, the fifth line constructs the legend, and the last line displays the plot in a pop-up window.

When you display the plot, it should look something like this:



Already we can see that the three species appear to form distinct clusters based on just two features. Of course, there are more features in the dataset than those visualised here, and often there will many more than can be shown on a simple scatter plot like this. But taking some time to explore and visualise your data can help inform the choices you will need to make when selecting a machine learning model.

Step 3: Training a simple machine learning model.

The aim of machine learning in this case is to take the iris data and use it to make a model that will predict the species of new data-points not previously seen by the model.

A simple model we can try is the K-nearest neighbour (KNN) method (see the article/video for details of the method). To import and set up the model it's as simple as the following code:

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=3)
```

The important parameter we need to supply when we initialise the model is **n_neighbours**, our value for K or the number of nearest neighbours considered by the algorithm. In this case we have chosen K=3.

Then we need to split our data into **test** and **training** subsets to avoid overfitting (more on this later in the course):

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, y_train, y_test = train_test_split(X, y , random_state=13)
```

The random_state parameter allows us to pick the same 'random' subset of data each to time and reproduce our results for the purposes of this practical. You should probably avoid doing this ordinarily.

Once we have our training and test subsets, to train our model on the training data we just need the following:

```
model.fit(Xtrain,y_train)
```

And that's it – we have trained a machine learning model! But how well does it perform?

Step 4: Evaluating the results.

We can use the model to make predictions on the test data, and check how accurate it is using the data in **y_test** (scikit-learn has a handy function to calculate this)

```
from sklearn.metrics import accuracy_score
y_predicted = model.predict(Xtest)
score = accuracy_score(y_test,y_predicted)
print(score)
```

You should find the fitted model is around 90-95% accurate (the score is the proportion of correct predictions, so is between 0 and 1).

What does this look like though? Which instances have been misclassified?

To find out we can look at those values in Xtest for which the predicted answer does not equal the true value:

```
X_misclass = Xtest[y_test != y_predicted]
```

We can then plot our results to visualise where the misclassified items are as follows:

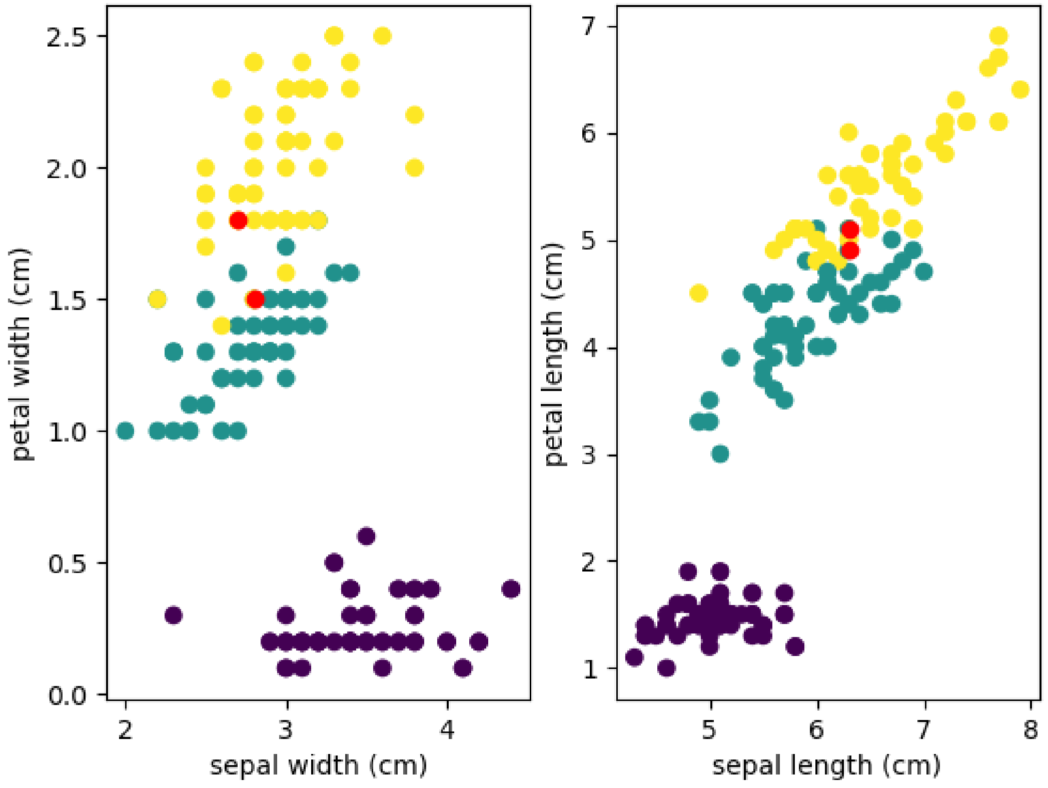
```
# set up and select first of two subplots
plt.subplot(1,2,1)

# plot test data (sepal width vs petal width)
plt.scatter(Xtest['sepal width (cm)'],
            Xtest['petal width (cm)'],c=y_test)
# plot misclassified items
plt.scatter(X_misclass['sepal width (cm)'],
            X_misclass['petal width (cm)'],c='r')
#label axes
plt.xlabel('sepal width (cm)')
plt.ylabel('petal width (cm)')

# move to second subplot
plt.subplot(1,2,2)
# plot test data (sepal length vs petal length)
plt.scatter(Xtest['sepal length (cm)'],
            Xtest['petal length (cm)'],c=y_predicted)
# plot misclassified items
plt.scatter(X_misclass['sepal length (cm)'],
            X_misclass['petal length (cm)'],c='r')
# label axes
plt.xlabel('sepal length (cm)')
plt.ylabel('petal length (cm)')

# show plot
plt.show()
```

Your plot should look something like this:



Perhaps unsurprisingly, the misclassified points are in the areas where the yellow and green (virginica and versicolor) clusters appear to overlap. Nonetheless, the model has done a good job of predicting the species based on just the size of the sepals and petals in the dataset.

We'll look at more machine learning algorithms, discuss their strengths and weaknesses, and how we might evaluate and improve our model results later in the course.

Optional:

Try changing the random state used to split the data into training and test subsets (use a different integer value). Do you always get the same number of misclassified points? What does this suggest about the importance of validating your models on as much data as possible?