Building an image classifier in Scikit-Learn

If you completed the practical in week two you will have seen how to import and pre-process an image dataset, specifically a set of images containing several flower species. You may find it useful in this practical to refer to the code you should have saved from the previous week. In addition, you should also already have the flower image data saved in a specific folder you can refer to in the code.

As before there will be snippets of code that you should piece together in a new script, but this time we will encourage you to try and write some new code yourself. If you get stuck there is a model solution on the GitHub repository:

https://github.com/LAR/PhenoDataCampp/tree/main/MachineLearning/Code

Step 1: Importing and formatting the data.

Features matrix from image data

Complete the short script below so that it performs the steps indicated by the comments. All the individual steps are in the practical last week. The places where you'll need to add code are highlighted in green as follows: …..

The comments to help you are highlighted in red.

A quick note on the **enumerate** function if you haven't seen it before. This is a quick way in python to keep a count of what position in the list you are currently at. In the example below the count is stored in the variable **i** at each step of the loop.

```
# first import the required packages
import glob
import cv2
import numpy as np

# get a list of files with the .jpg extension from your flower data folder
…..
```

```
# set the number of rows and columns to 128
nrow = .....
ncol = .....

# calculate the number of features (rows x columns x 3 colour channels)
n_features = .....
# find the number of samples (how many images?)
n_samples = .....

# Make a matrix full of zeros with n_samples number of rows
# and n_features numbers of columns
X = .....

# loop through the list of files
for i,im_name in enumerate(img_list):
    # open the image using cv2 (OpenCV)
    img = .....

    # resize the image to be nrow by ncol pixels
    img_resized = .....

    # flatten the image data into a 1D array
    img_flat = .....

    # add the flattened data to the ith row of matrix X
    X[i,:] = .....
```

Target data

When you have completed the code above you should have a features matrix called **X** which contains pixel data from the images in the folder. We now need to add the label data. You may have noticed the images are grouped and ordered by species, with 80 instances of each species. So e.g. the first 80 images are of Daffodils, the second 80 images are of Tigerlilies, and so on.

To construct a target vector with all the species names we can just use a list of the flower species names as follows:

```
species_list = ['Daffodil','Tigerlily','Tulip','Daisy']
y=[]
```

```
for species in species_list:
    y=y+[species]*80
```

All this does is add 80 duplicates of each item in the species list to the end of the target vector **y**

Step 2: Naïve Bayes Classifier – pixel data

Now we have our features matrix **X** and target vector **y** we are ready to train a machine learning model.

Test and Training sets

The first step is to split the data into training and test subsets (as in the practical in Week1):

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=7)
```

Initialising and training a model

Next, we need to initialise a model. We will use a Gaussian Naïve Bayes classifier. One advantage of Naïve Bayes is that we don't need to provide any parameters when we set it up (refer back to the video for a reminder).

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
```

Now, we need to train the model. We can do this with the following line:

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

Model Evaluation

Finally, we can make predictions using the test data, and check their accuracy:

```
ymodel = model.predict(Xtest)

from sklearn.metrics import accuracy_score
score = accuracy_score(ytest, ymodel)
print(score)
```

This gives us a crude measure of how well the classifier is doing – around 60 to 70% accurate depending on the random selection of test and training sets (try this out for yourself by changing the **random_state** variable). Better than what you would expect just by chance, which for four species is 1 in 4, or 25%, but still room for improvement.

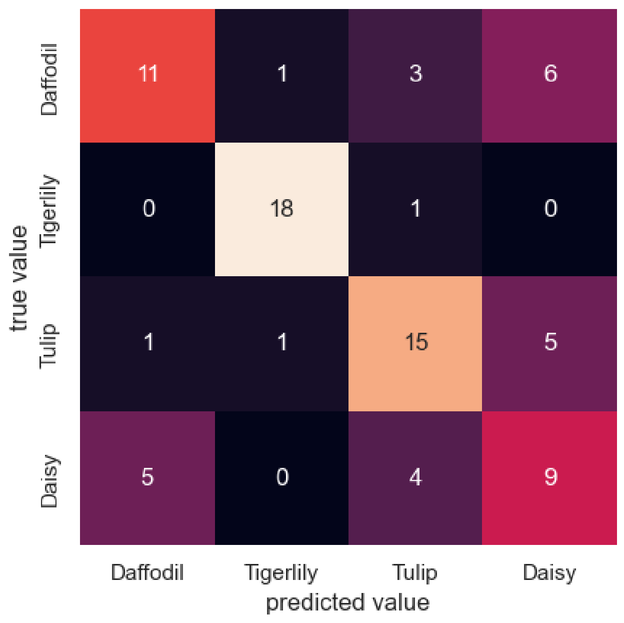
As we saw in the videos and articles, to see more clearly where the model is failing, we can look at the **confusion matrix**. The following code will make and display a confusion matrix:

```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(ytest, y_model)


from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
        display_labels=species_list)
disp.plot()

plt.show()
```



The confusion matrix should look something like the above, if not exactly the same.

By looking at the confusion matrix we can see that the Naïve Bayes model using the pixel data is quite good at identifying images of Tulips and Tigerlilies but does less well with the images of Daffodils and Daisies.

Step 3: Naïve Bayes Classifier – HOG features

Now let's try using the HOG features we talked about last week in our classifier rather than the raw pixel data.

We can reuse much of the code from before, but this time within the loop rather than adding the raw pixel data to our features array, we will add the HOG extracted features. We'll also need to recalculate the number of features in our reduced HOG representation.

If we keep the same image dimensions as before this depends on the settings we intend to use in the HOG extraction.

If we first resize our image to 128x128 pixels, then choose 16x16 pixels per cell, we will end up with 128/16 x 128/16 = 8 x 8 = 64 cells.

If we choose 8 orientations per cell we will end up with 64 x 8 = 512 features. In the code below we can work this out directly using some new variables, **pix_per_cell** and **orients**:

```
import glob
import cv2
import numpy as np
img_list = glob.glob('FlowerData/*.jpg')
nrow = 128
ncol = 128

pix_per_cell = 16
orients = 8
# calculate the number of features, making sure it returns an integer
n_features = int((nrow/pix_per_cell) * (ncol/pix_per_cell) * orients)
print('number of HOG features should be',n_features)
n_samples = len(img_list)
```

Now we can make a new empty features matrix as before, and loop through all the images as before. This time though, rather than just flattening the raw pixel data into the array, we perform the HOG feature extraction and add that instead. Read through the code below and make sure you understand where and how the HOG extraction is done.

```
from skimage.feature import hog
# Make the empty features matrix
X = np.zeros((n_samples,n_features))

# loop through the list of files
for i,im_name in enumerate(img_list):
    # open the image
    img = cv2.imread(im_name)
    # resize the image
    img_resized = cv2.resize(img,(nrow,ncol))
    # get the HOG features
    img_hog = hog(img_resized, orientations=orients,
                  pixels_per_cell=(pix_per_cell, pix_per_cell),
                  cells_per_block=(1, 1),visualize=False, channel_axis=-1)
    # add the data to the ith row of matrix X
    X[i,:] = img_hog
```

Once you have the HOG features extracted to your matrix **X**, you can set up and train and evaluate the model in exactly the same way as before. The target vector **y** will remain the same. Try doing this for yourself. Remember to split your data in to training and test sets!

If you get stuck there is an example solution in the Github repository.

How does the model trained on the HOG features compare with the raw pixel data? Why do you think it might perform less well?

It's possible that by reducing our features from several thousand raw pixels to just 512 Hog features we have lost too much detail in the detail to build a good classifier. We're also throwing away all the colour data, when classifying flowers that information is likely to be useful. Also, the Naïve Bayes algorithm is probably not the best algorithm for this particular task.

The aim of this example is just to demonstrate some of the basic processes in building a classifier, and some of the decisions you need to make along the way.

Step 4 (challenge): Other Classifier Methods

Instead of using a Naïve Bayes Classifier try using the Stochastic Gradient Descent classifier in SciKit-Learn. To import the function you will need:

```
from sklearn.linear_model import SGDClassifier
```

Then to try it out on the same data you will just need to change one line of code (HINT: your aim is to replace the function **GaussianNB** with the function **SGDClasifier**). For more information on the stochastic gradient descent classifier, and the different setting or hyperparameters you can use see https://scikit-learn.org/stable/modules/sgd.html