

Face Recognition (from Python Data Science Handbook by Jake VanderPlas)

As an example of support vector machines in action, let's take a look at the facial recognition problem. We will use the Labeled Faces in the Wild dataset, which consists of several thousand collated photos of various public figures. A fetcher for the dataset is built into Scikit-Learn:

In [61]:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)
```

```
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

In [62]:

```
unique_elements, counts_elements = np.unique(faces.target, return_counts=True)
print(unique_elements)
print(counts_elements)
```

```
[0 1 2 3 4 5 6 7]
[ 77 236 121 530 109  71  60 144]
```

In []:

Let's plot a few of these faces to see what we're working with:

In [63]:

```
fig, ax = plt.subplots(3, 5)
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='bone')
    axi.set(xticks=[], yticks=[],
            xlabel=faces.target_names[faces.target[i]])
```



Each image contains [62×47] or nearly 3,000 pixels. We could proceed by simply using each pixel value as a feature, but often it is more effective to use some sort of preprocessor to extract more meaningful features; here we will use a principal component analysis (see [In Depth: Principal Component Analysis \(05.09-Principal-Component-Analysis.ipynb\)](#)) to extract 150 fundamental components to feed into our support vector machine classifier. We can do this most straightforwardly by packaging the preprocessor and the classifier into a single pipeline:

In [64]:

```
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline

pca = PCA(n_components=150, whiten=True, random_state=42)
svc = SVC(kernel='rbf', class_weight='balanced')
model = make_pipeline(pca, svc)
```

In []:

For the sake of testing our classifier output, we will split the data into a training and testing set:

In [65]:

```
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import scale, StandardScaler
X_train, X_test, y_train, y_test = train_test_split(faces.data, faces.target,
                                                    random_state=42)
```

Finally, we can use a grid search cross-validation to explore combinations of parameters. Here we will adjust `C` (which controls the margin hardness) and `gamma` (which controls the size of the radial basis function kernel), and determine the best model:

In [66]:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'svc__C': [1, 5, 10, 50],
              'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}
grid = GridSearchCV(model, param_grid)

grid.fit(X_train, y_train)
print(grid.best_params_)
print(grid.best_score_)
```

```
{'svc__C': 10, 'svc__gamma': 0.001}
0.8279129883431693
```

The optimal values fall toward the middle of our grid; if they fell at the edges, we would want to expand the grid to make sure we have found the true optimum.

Now with this cross-validated model, we can predict the labels for the test data, which the model has not yet seen:

In [67]:

```
model = grid.best_estimator_
yfit = model.predict(Xtest)
```

Let's take a look at a few of the test images along with their predicted values:

In [68]:

```
fig, ax = plt.subplots(4, 6)
for i, axi in enumerate(ax.flat):
    axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
    axi.set(xticks=[], yticks=[])
    axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                  color='black' if yfit[i] == ytest[i] else 'red')
fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);
```

Predicted Names; Incorrect Labels in Red



Out of this small sample, our optimal estimator mislabeled only a single face (Bush's face in the bottom row was mislabeled as Blair). We can get a better sense of our estimator's performance using the classification report, which lists recovery statistics label by label:

In [69]:

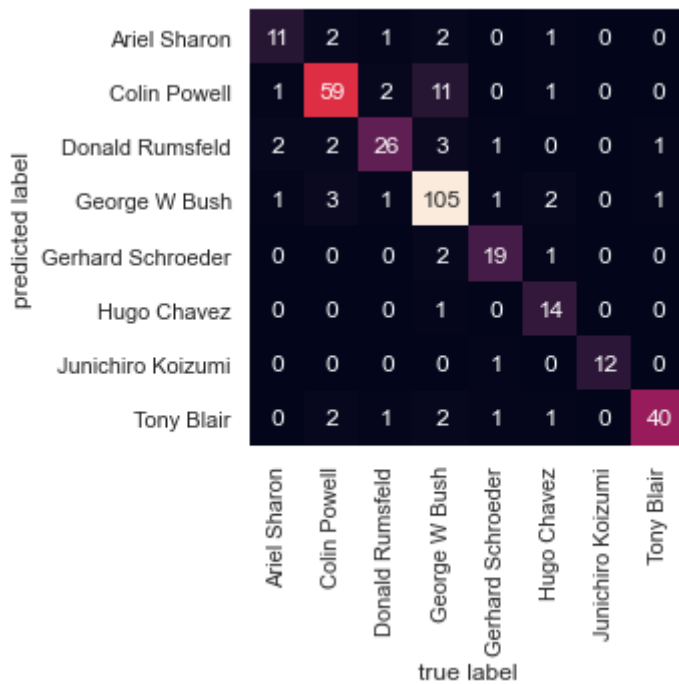
```
from sklearn.metrics import classification_report
print(classification_report(ytest, yfit,
                          target_names=faces.target_names))
```

	precision	recall	f1-score	support
Ariel Sharon	0.65	0.73	0.69	15
Colin Powell	0.80	0.87	0.83	68
Donald Rumsfeld	0.74	0.84	0.79	31
George W Bush	0.92	0.83	0.88	126
Gerhard Schroeder	0.86	0.83	0.84	23
Hugo Chavez	0.93	0.70	0.80	20
Junichiro Koizumi	0.92	1.00	0.96	12
Tony Blair	0.85	0.95	0.90	42
accuracy			0.85	337
macro avg	0.83	0.84	0.84	337
weighted avg	0.86	0.85	0.85	337

We might also display the confusion matrix between these classes:

In [70]:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns; sns.set()
mat = confusion_matrix(ytest, yfit)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=faces.target_names,
            yticklabels=faces.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```



This helps us get a sense of which labels are likely to be confused by the estimator.

For a real-world facial recognition task, in which the photos do not come pre-cropped into nice grids, the only difference in the facial classification scheme is the feature selection: you would need to use a more sophisticated algorithm to find the faces, and extract features that are independent of the pixellation. For this kind of application, one good option is to make use of [OpenCV \(http://opencv.org\)](http://opencv.org), which, among other things, includes pre-trained implementations of state-of-the-art feature extraction tools for images in general and faces in particular.

In []:

In []:

DISREGARD THE FOLLOWING

In [50]:

```
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import scale, StandardScaler
```

In [53]:

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=60, resize=0.7)
image_shape = people.images[0].shape
fix, axes = plt.subplots(2, 5, figsize=(15, 8),
subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
    ax.imshow(image, cmap='gray')
    ax.set_title(people.target_names[target])
```



In [54]:

```
people.images.shape
```

Out[54]:

```
(1348, 87, 65)
```

In [13]:

```
# have at most 50 images per person - otherwise too much bias
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1

X_people = people.data[mask]
y_people = people.target[mask]
# scale the grayscale values to be between 0 and 1
# instead of 0 and 255 for better numeric stability
X_people = X_people / 255.
```

In [16]:

```
from sklearn.neighbors import KNeighborsClassifier
# split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
print(X_train.shape)
# build a KNeighborsClassifier using one neighbor
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Test set score of 1-nn: {:.2f}".format(knn.score(X_test, y_test)))
```

(1547, 5655)

Test set score of 1-nn: 0.23

In [30]:

```
from sklearn.svm import SVC
svc = SVC(kernel='rbf', class_weight='balanced')
svc.fit(X_train, y_train)
print("Test set score of svc: {:.2f}".format(svc.score(X_test, y_test)))
```

Test set score of svc: 0.36

In []: