index

April 20, 2022

1 PCA for Facial Image Recognition

1.1 Introduction

In this lesson, you'll get to explore an exciting application of PCA: PCA can be used for preprocessing facial image recognition data!

1.2 Objectives

You will be able to:

- Use PCA to discover the principal components of image data
- Use the principal components of a dataset as features in a machine learning model

1.3 Load the data

First, let's load the dataset.

[6]:
```
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_olivetti_faces
```

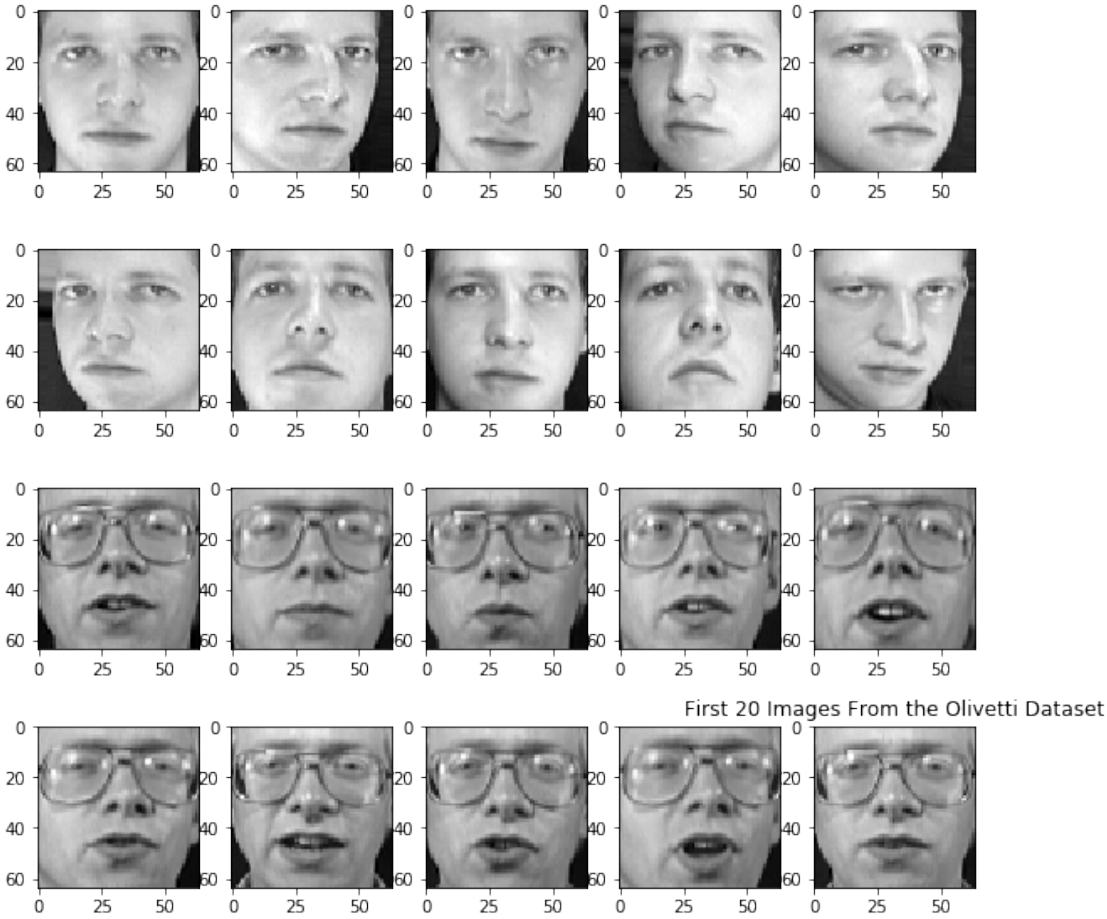
[42]:
```
data = fetch_olivetti_faces()
type(data)
```

[42]:
```
sklearn.utils.Bunch
```

1.4 Preview the images in the dataset

Next, we'll take a quick preview of the images within the dataset.

[40]:
```
fig, axes = plt.subplots(nrows=4, ncols=5, figsize=(10,10))
for n in range(20):
    i = n //5
    j = n%5
    ax = axes[i][j]
    ax.imshow(data.images[n], cmap=plt.cm.gray)
plt.title('First 20 Images From the Olivetti Dataset');
```



1.5 Train a baseline classifier

You'll soon take a look at the performance gains by using PCA as a preprocessing technique. To compare the performance, here's an out-of-the-box classifier's performance.

[15]:
```
from sklearn import svm
from sklearn.model_selection import train_test_split
```

[16]:
```
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=22)
```

[17]:
```
clf = svm.SVC(C=5, gamma=0.05)
%timeit clf.fit(X_train, y_train)
```

```
263 ms ± 3.94 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[18]:
```
train_acc = clf.score(X_train, y_train)
test_acc = clf.score(X_test, y_test)
print('Training Accuracy: {}\tTesting Accuracy: {}'.format(train_acc, test_acc))
```

```
Training Accuracy: 1.0  Testing Accuracy: 0.74
```

1.6 Grid search on the baseline classifier

To produce a more robust baseline to compare against, let's see how much performance you can squeeze by conducting a grid search to find optimal hyperparameters for the model. It's also worth timing the duration of training such a model, as PCA will drastically decrease training time and it's interesting to observe this performance gain.

> **Warning**: It's not recommended to run the cell below. (Doing so is apt to take well over an hour, depending on the particular specs of your machine.)

[8]:
```
#  This cell may take over an hour to run!
import numpy as np
from sklearn.model_selection import GridSearchCV

clf = svm.SVC()
param_grid = {'C' : np.linspace(0.1, 10, num=11),
              'gamma' : np.linspace(10**-3, 5, num=11)}

grid_search = GridSearchCV(clf, param_grid, cv=5)

%timeit grid_search.fit(X_train, y_train)
```

```
10min 17s ± 6.79 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[9]:
```
grid_search.best_estimator_.score(X_test, y_test)
```

[9]:
```
0.91
```

1.7 Preprocessing with PCA

Now it's time to perform some dimensionality reduction with PCA! To start, you can simply pick an arbitrary number of components. Later, you can compare the performance of a varying number of components.

Note that to avoid information leakage from the test set, PCA should only be fit on the training data.

[19]:
```
from sklearn.decomposition import PCA
```

[20]:
```
X[0].shape
```

[20]:
```
(4096,)
```

[21]:
```
pca = PCA(n_components=100, whiten=True)
X_pca_train = pca.fit_transform(X_train)
X_pca_train.shape
```

[21]: (300, 100)

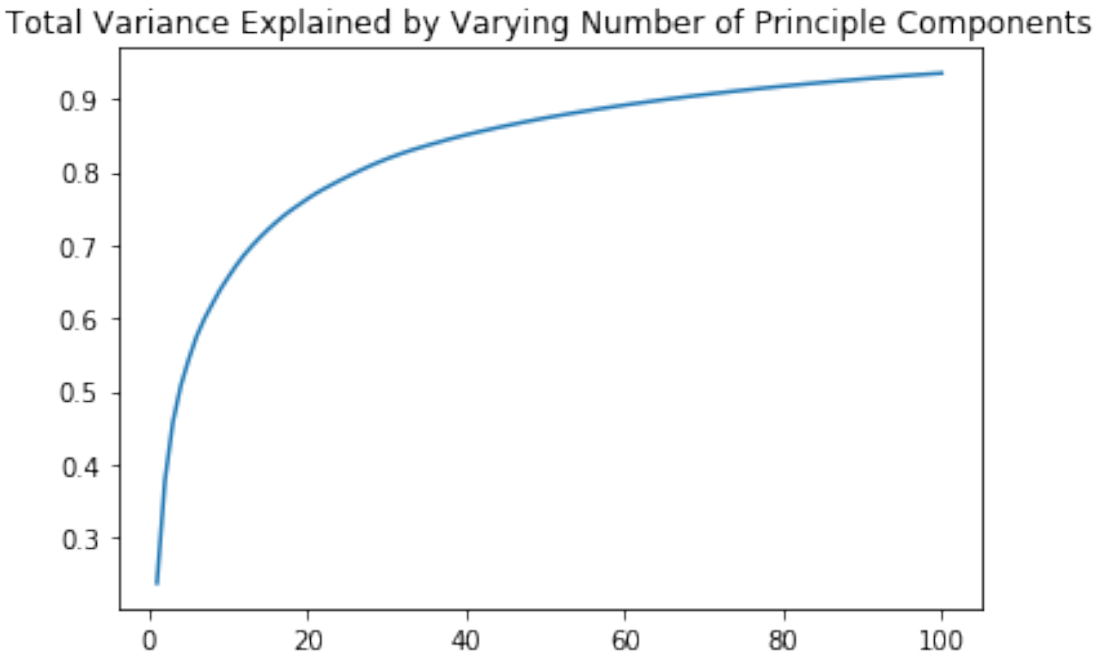
[22]:
```
X_pca_train[0].shape
```

[22]: (100,)

1.8 Explore the explained variance captured by principal components

How much of the total data was captured in these compressed representations? Take a quick look at a plot of the explained variance to explore this.

[23]:
```
plt.plot(range(1,101), pca.explained_variance_ratio_.cumsum())
plt.title('Total Variance Explained by Varying Number of Principle Components');
```



1.9 Train a classifier on the compressed dataset

Now it's time to compare the performance of a classifier trained on the compressed dataset.

[24]:
```
X_pca_test = pca.transform(X_test)
clf = svm.SVC()
%timeit clf.fit(X_pca_train, y_train)
```

```
25.9 ms ± 616 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[25]:
```
train_pca_acc = clf.score(X_pca_train, y_train)
test_pca_acc = clf.score(X_pca_test, y_test)
print('Training Accuracy: {}\tTesting Accuracy: {}'.format(train_pca_acc,␣
 ↪test_pca_acc))
```

```
Training Accuracy: 1.0  Testing Accuracy: 0.93
```

1.10 Grid search for appropriate parameters

Going further, you can also refine the model using grid search.

[27]:
```
#  This cell may take several minutes to run
import numpy as np
from sklearn.model_selection import GridSearchCV

clf = svm.SVC()

param_grid = {'C' : np.linspace(0.1, 10, num=11),
              'gamma' : np.linspace(10**-3, 5, num=11)}

grid_search = GridSearchCV(clf, param_grid, cv=5)

%timeit grid_search.fit(X_pca_train, y_train)
```

```
15.6 s ± 358 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[28]:
```
grid_search.best_params_
```

[28]:
```
{'C': 5.05, 'gamma': 0.001}
```

[29]:
```
grid_search.best_estimator_.score(X_pca_test, y_test)
```

[29]:
```
0.94
```

[30]:
```
grid_search.score(X_pca_test, y_test)
```

[30]:
```
0.94
```

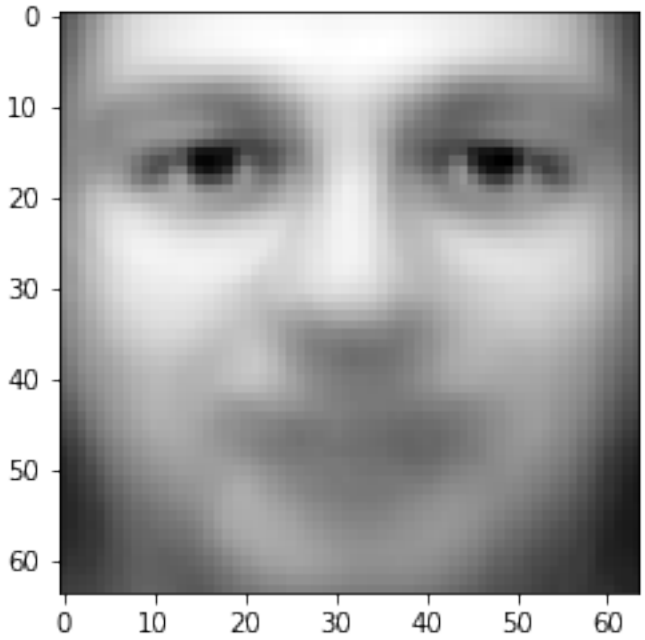
1.11 Visualize some of the features captured by PCA

While this model may have lost some accuracy, it is clearly much faster to train. Let's take a moment to visualize some of the information captured by PCA. Specifically, you'll take a look at two perspectives. First, you'll take a look at visualizing the feature means. Second, you'll get to visualize the compressed encodings of the dataset.

1.11.1 Visualize feature means

While it is a very simple mathematical model, just observing the mean values of the features produces quite an informative picture:

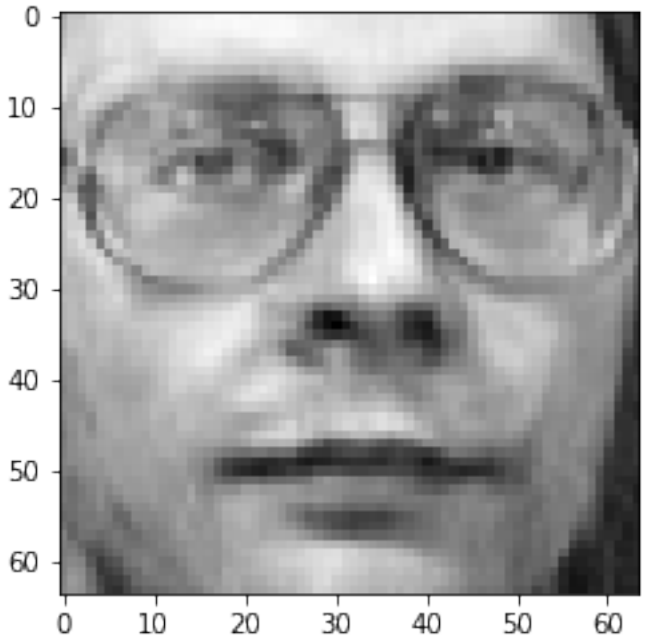
[31]:
```
plt.imshow(X.mean(axis=0).reshape(data.images[0].shape), cmap=plt.cm.gray);
```



1.11.2 Visualize compressed representations

Visualizing the components from PCA is slightly tricky, as they have new dimensions which may not correspond accurately to the 64x64 size of the original images. Fortunately, scikit-learn provides a useful `.inverse_transformation()` method to PCA allowing you to reproject the compressed dataset back to the original size. This allows you to observe what features are retrieved and encapsulated within the principal components.

[43]:
```
fig, axes
plt.imshow(pca.inverse_transform(X_pca_train[0]).reshape(64,64), cmap=plt.cm.
↪gray);
```



To make this even more interesting, take a look at some of the varied levels of detail based on varying number of principle components:

[53]:
```
fig, axes = plt.subplots(ncols=4, nrows=3, figsize=(10,10))
ax = axes[0][0]
ax.set_title('Original Image')
ax.imshow(X_train[0].reshape(64,64), cmap=plt.cm.gray)
for n in range(1,12):
    i = n //4
    j = n%4
    ax = axes[i][j]
    ax.set_title('Re')
    n_feats = n*10
    pca = PCA(n_components=n_feats)
    pca.fit(X_train)
    compressed = pca.transform(X_train)
    ax.set_title('Recovered Image from\n{} principle components'.
 ↪format(n_feats))
    ax.imshow(pca.inverse_transform(compressed[0]).reshape(64,64), cmap=plt.cm.
 ↪gray)
#     print(compressed.shape)
plt.tight_layout()
```

1.12 Summary

Awesome! In this lesson, you saw how you can use PCA to reduce the dimensionality of a complex dataset. In the next lab, you'll get a chance to put these same procedures to the test in working with the MNIST dataset.