Dimensionality Reduction

Dr Rob Collins

Version 3, 21st July 2021

Introduction

This tutorial is focussed on 'Dimensionality Reduction'. It is quite common in data-sets that although the data set may have many features ('dimensions'), some of these features may be highly correlated. Another way of thinking about this is that using both features ('dimensions') provides little more information than using one of them alone.

Data may often appear to occupy many dimensions becuase of the way it is represented and stored. However, the data is in-fact of much lower dimensionality.

In this case it is useful to reduce the dimensionality of the data-set. That is to 'squeeze' the information from the data-set and represent it as data that has far fewer dimensions. The lower dimensional representation will be smaller to store and will often require significantly lower resources to process - since there is less data.

Sometimes, some of the dimensions in data really only represent random noise in the original data. Thus, dimensionality reduction can also achieve noise reduced. This will be illustrated in one of the examples below.

Instructions for Students

This workbook includes some empty code-blocks. In those cases you are required to create the code for the block based on your previous learning on this course. This activity is indicated in the workbook below with a '**Student Task**' indicator.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

1 Demonstration of low-dimensional data being embedded in a higher-dimensional representation

This part of the workshop provides a visualisation that helps show how low-dimensional data might be 'embedded' in a higher dimensional representation.

First, as always, import the libraries needed for this part of the workshop:

In [1]:
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import random as rand
import numpy as np
%matplotlib
```

```
Using matplotlib backend: Qt5Agg
```

This code sets up a 3D data visualisation. On running this code you should only see a window with axis - there is no data yet.

In [2]:
```
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
```

Here we create a two-dimensional data set. Then add data in 3rd (z) dimension which is a simple linear function of the original 2-D data-set with a small amount of Gaussian noise. Obviously,

In [3]:
```
xGrad = 5
yGrad = 7
noiseSD = .05

x = []
y = []
z=[]

for i in range(100):
    xItem = rand.randint(1,200)
    yItem = rand.randint(1,200)
    x.append(xItem)
    y.append(yItem)
    noise = np.random.normal(0, scale=50.0)
    z.append((xItem * xGrad) + (yItem * yGrad) + noise)
```

Finally, we plot the data as a 3D chart. Note that you can **rotate the view of the chart by clicking and dragging** on it. As you do so, notice that whilst the data is represented in a 3D space - the points generally lie in a flat plane.

Any variation above and below the plane is just random 'noise' (error). This three dimensional representation provides no useful information that could not be represented in two dimensions.

In [4]:
```
ax.scatter(x, y, z, c='r', marker='o')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

fig.show()
```

2 Principle Component Analysis (PCA) for MNIST

In this section we will look at an algorythm that is used to reduce the dimensions of data sets.

In this case we will be using a series of images of handwritten digits. This data set is called MNIST and is frequently used in Machine Learning tutorials for such tasks as building tools for handwritting analysis.

An important thing to remember when working through the following section is that we are using an **unsupervised learning** algorythm. When we process the images of the digits we are not giving the algorythm examples of 'the correct answer'. Rather, we are purely squeezing information from the raw data.

As always, we start by loading the required libraries:

In [5]:
```
%matplotlib
import numpy as np                          # A powerful library for fast numerical calculations
import matplotlib.pyplot as plt             # Used to plot data
import seaborn as sns; sns.set()            # Statistical data visualisation
from sklearn.decomposition import PCA       # sklearn is a commonly-used library of Machine Learning a
```

```
Using matplotlib backend: Qt5Agg
```

In this case, the data set we want to use is built into sklearn. That means that we can access it easily and directly.

In [6]:
```
from sklearn.datasets import load_digits
digits = load_digits()
```

We review the shape of the data ..

In [7]:
```
digits.data.shape
```

Out[7]:
```
(1797, 64)
```

That is, there are 1797 images, each with **64 dimensions**. The dimensions represent an image which is 8x8 pixels in size. Each pixel represents a grey-scale.

The following command provides a view of the raw data for the first of the images. You can see that it consists of 64 numbers. Each number is in the range 0..255 and represents a 'grey-scale'.

Student Task : Experiment by changing the index in the code below to look at other digits in their raw form.

In [8]:
```
digits.data[0]
```

Out[8]:
```
array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
       15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
       12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
        0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
       10., 12.,  0.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

Sklean provides a function that enables you to look at an image of the image represented by the data. The code to do this is included in the code-block below.

Student Task : Experiment by changing the index in the code below to look at other digits as images.

In [9]:
```
plt.gray()
plt.matshow(digits.images[0])
plt.show()
```

Now, whilst the digits are represented in a 64 dimensional data structure .. there are not really 64 dimensions of real 'information' here. Hand-written digits are very constrained - they fall into a rather simple graphical pattern.

A 64 dimensional array of number in the range 0-255 can represent $1x10^{154}$ different data values! This means that the image representation of a digit is highly redundant.

We can reduce the number of dimensions for the data-set whilst still retaining much of the useful information.

The sklean 'pca' function can be used to perform a 'Principle Component Analysis'. This transforms the data into a lower dimensional space. Dimensions are ordered so that the first dimension contains the more information than any other dimension. The second dimension contains more information than the third .. and so on. Dimensions are thus ordered by the amount of information they contain.

We can reduce the number of dimensions in the MNIST data set from 64 to just 3 dimensions.

In [10]:
```
pca = PCA(3)  #
projected = pca.fit_transform(digits.data)
```

The resulting data is not longer 1797 rows with 64 dimensions .. but 1797 rows each with 3 dimensions...

In [11]:
```
print(digits.data.shape)
print(projected.shape)
```

```
(1797, 64)
(1797, 3)
```

The data actually looks like this ..

In [12]:
```
projected
```

Out[12]:
```
array([[ -1.25946492,  21.27488028,  -9.46304245],
       [  7.95761288, -20.76870367,   4.43951839],
       [  6.99192094,  -9.95598135,   2.95854171],
       ...,
       [ 10.80128356,  -6.96025196,   5.59955362],
       [ -4.87209815,  12.42395005, -10.17085099],
       [ -0.34439154,   6.36555425,  10.77369358]])
```

Even in just 3 dimenions there is still enough information to do quite a good job of separating the image data into clusters for each digit. We can see this by plotting a 3D chart. In this chart, each point has a colour depending on the value of the original digit 0, 1 .. 9.

In [13]:
```
x = projected[:,0]
y = projected[:,1]
z = projected[:,2]
```

In [14]:
```
fig2 = plt.figure(figsize=(6,6))
ax2 = fig2.add_subplot(111, projection='3d')
```

In [15]:
```
ax2.scatter(x, y,z,
            c=digits.target,
            cmap=plt.cm.get_cmap('Spectral', 10))

ax2.set_xlabel('X Label')
ax2.set_ylabel('Y Label')
ax2.set_zlabel('Z Label')

fig2.show()
```

We can even do the same thing with just 2 dimensions:

In [16]:
```
pca = PCA(2)  #
projected = pca.fit_transform(digits.data)
```

And again, plot the results:

In [17]:
```
plt.figure(figsize=(8,8))
plt.scatter(projected[:, 0], projected[:, 1],
            c=digits.target, edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('Spectral', 10))
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```

So even with just 2 dimensions, there seems to be enough information to separate the digits into clusters.

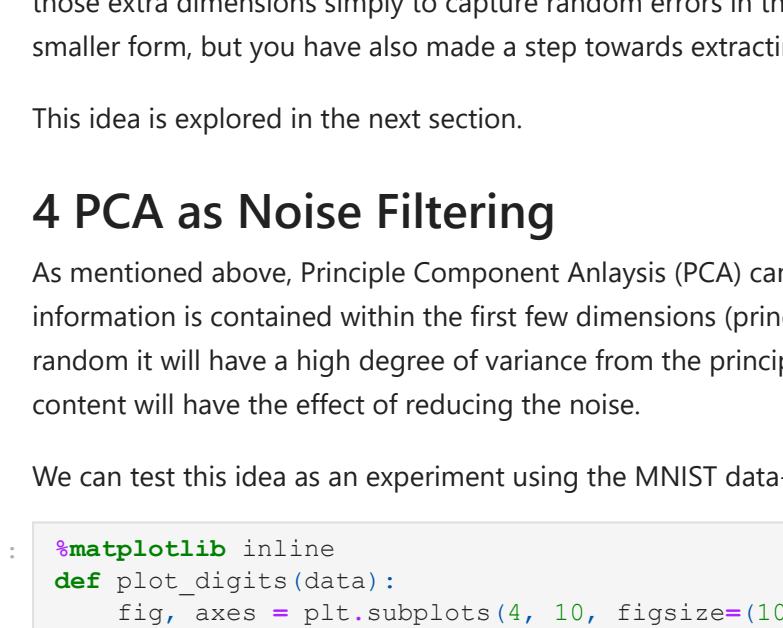
3 Choosing the number of components

A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data. This can be determined by looking at the cumulative explained variance ratio as a function of the number of components:

It is interesting to see how much information is contained in each of the dimensions generated through PCA. The following chart shows the cumulative total of information provided by each dimension ('component'). From the graph you will see that over 70% of the variation is explained by just 10 components ('dimensions').

In [18]:
```
%matplotlib inline
pca = PCA().fit(digits.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
np.cumsum(pca.explained_variance_ratio_)
```

Out[18]:
```
array([0.14890594, 0.28509365, 0.40303959, 0.48713938, 0.54496353,
       0.59413263, 0.6372925 , 0.67390623, 0.70743871, 0.73822677,
       0.76195018, 0.78467714, 0.80289578, 0.82063433, 0.83530534,
       0.84940249, 0.86258838, 0.87506976, 0.88524694, 0.89430312,
       0.9031985 , 0.91116973, 0.91884467, 0.9260737 , 0.93303259,
       0.9389934 , 0.94474955, 0.94990113, 0.95479652, 0.9590854 ,
       0.96282146, 0.96635421, 0.96972105, 0.97300135, 0.97608455,
       0.97902234, 0.98158823, 0.98386565, 0.98608843, 0.98820273,
       0.99010182, 0.99168835, 0.99319995, 0.99460574, 0.99577196,
       0.99684689, 0.99781094, 0.99858557, 0.99914278, 0.99954711,
       0.99975703, 0.99983951, 0.99989203, 0.99994255, 0.99997555,
       0.99998798, 0.99999503, 0.99999804, 0.99999911, 0.99999966,
       1.        , 1.        , 1.        , 1.        ])
```



This plot can help you understand the level of redundancy present in higher dimensional data-sets.

With just 30 dimensions (less than half the total) you still obtain over 95% of the 'real' information (or 'variance' as it is properly called in the context of PCA).

It is also worth considering that the remaining 5% of 'information' may also actually be 'noise' in the data. That is .. you may need all of those extra dimensions simply to capture random errors in the data! It is not just that you have squeezed the amount of information into a smaller form, but you have also made a step towards extracting **useful** information.

This idea is explored in the next section.

4 PCA as Noise Filtering

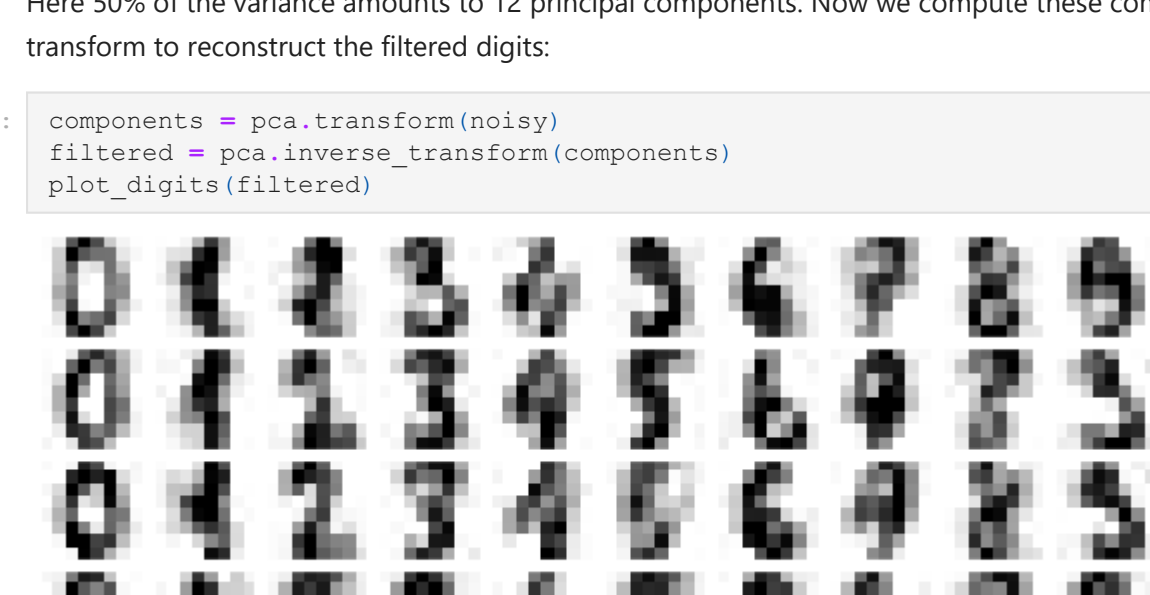
As mentioned above, Principle Component Anlaysis (PCA) can also be a way to reduce 'noise' in data. The idea is that most of the information is contained within the first few dimensions (principle components).Noise is technically speaking 'information' but since it is random it will have a high degree of variance from the principle components. Thus, removing some of the dimensions with low information content will have the effect of reducing the noise.

We can test this idea as an experiment using the MNIST data-set again:

In [19]:
```
%matplotlib inline
def plot_digits(data):
    fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                             subplot_kw={'xticks':[], 'yticks':[]},
                             gridspec_kw=dict(hspace=0.1, wspace=0.1))
    for i, ax in enumerate(axes.flat):
        ax.imshow(data[i].reshape(8, 8),
                  cmap='binary', interpolation='nearest',
                  clim=(0, 16))
plot_digits(digits.data)
```

Now lets add some random noise to create a noisy dataset, and re-plot it:

In [20]:
```
np.random.seed(42)
noisy = np.random.normal(digits.data, 4)
plot_digits(noisy)
```



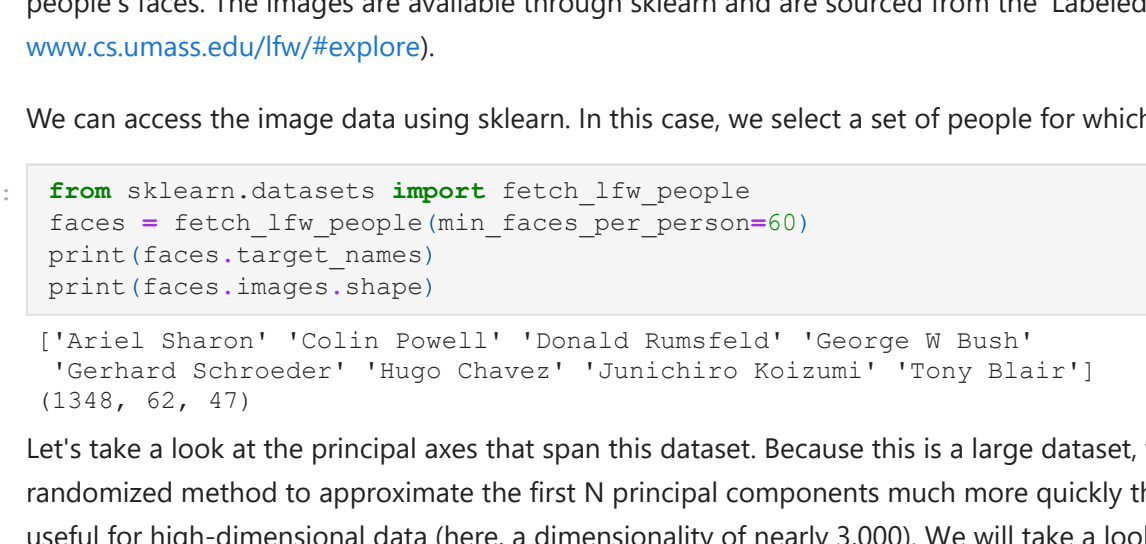
It's clear by eye that the images are noisy, and contain spurious pixels. Let's train a PCA on the noisy data, requesting that the projection preserve 50% of the variance:

In [21]:
```
pca = PCA(0.50).fit(noisy)
pca.n_components_
```

Out[21]:
```
12
```

Here 50% of the variance amounts to 12 principal components. Now we compute these components, and then use the inverse of the transform to reconstruct the filtered digits:

In [22]:
```
components = pca.transform(noisy)
filtered = pca.inverse_transform(components)
plot_digits(filtered)
```



This signal preserving/noise filtering property makes PCA a very useful feature selection routine—for example, rather than training a classifier on very high-dimensional data, you might instead train the classifier on the lower-dimensional representation, which will automatically serve to filter out random noise in the inputs.

5 PCA on Facial Image Data

In this part of the workshop we will apply the same un-supervised learning algorythm (Principle Component Analysis) to images of famous people's faces. The images are available through sklearn and are sourced from the 'Labeled Faces in the Wild' database (http://vis-www.cs.umass.edu/lfw/#explore).

We can access the image data using sklearn. In this case, we select a set of people for which there are more than 60 images per person:

In [23]:
```
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)
```

Let's take a look at the principal axes that span this dataset. Because this is a large dataset, we will use RandomizedPCA—it contains a randomized method to approximate the first N principal components much more quickly than the standard PCA estimator, and thus is very useful for high-dimensional data (here, a dimensionality of nearly 3,000). We will take a look at the first 150 components:

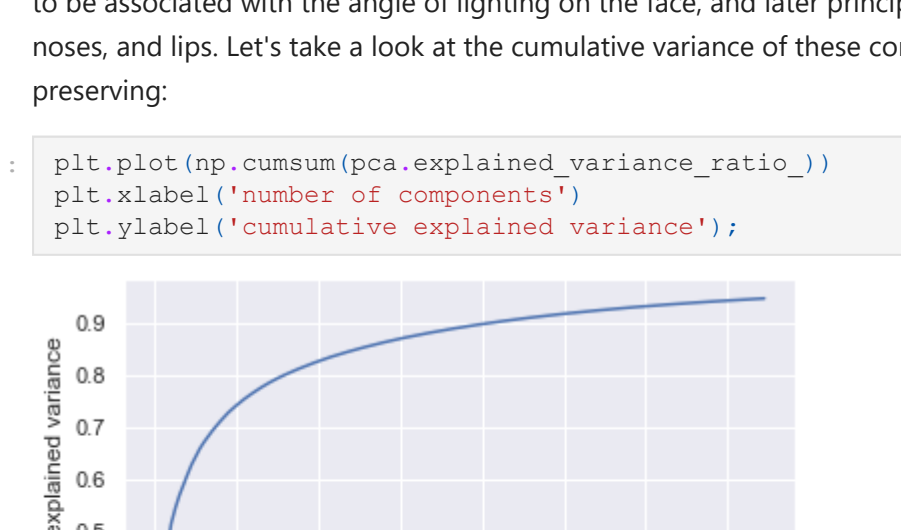
In [24]:
```
from sklearn.decomposition import PCA as RandomizedPCA

pca = RandomizedPCA(150, svd_solver='randomized')
pca.fit(faces.data)
```

Out[24]:
```
PCA(n_components=150, svd_solver='randomized')
```

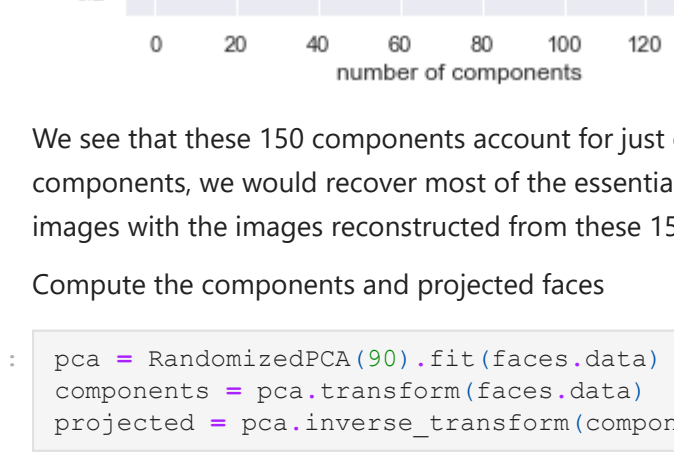
In this case, it can be interesting to visualize the images associated with the first several principal components (these components are technically known as "eigenvectors," so these types of images are often called "eigenfaces"). As you can see in this figure, they are as creepy as they sound:

In [25]:
```
fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i, ax in enumerate(axes.flat):
    ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```



The results are very interesting, and give us insight into how the images vary: for example, the first few eigenfaces (from the top left) seem to be associated with the angle of lighting on the face, and later principal vectors seem to be picking out certain features, such as eyes, noses, and lips. Let's take a look at the cumulative variance of these components to see how much of the data information the projection is preserving:

In [26]:
```
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



We see that these 150 components account for just over 90% of the variance. That would lead us to believe that using these 150 components, we would recover most of the essential characteristics of the data. To make this more concrete, we can compare the input images with the images reconstructed from these 150 components:

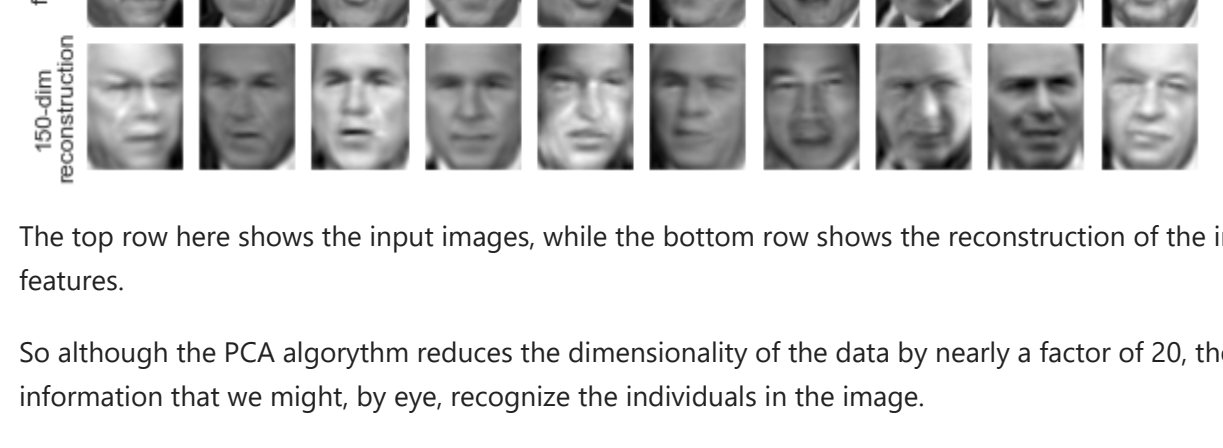
Compute the components and projected faces

In [27]:
```
pca = RandomizedPCA(90).fit(faces.data)
components = pca.transform(faces.data)
projected = pca.inverse_transform(components)
```

Plot the results

In [28]:
```
fig, ax = plt.subplots(2, 10, figsize=(10, 2.5),
                       subplot_kw={'xticks':[], 'yticks':[]},
                       gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i in range(10):
    ax[0, i].imshow(faces.data[i].reshape(62, 47), cmap='binary_r')
    ax[1, i].imshow(projected[i].reshape(62, 47), cmap='binary_r')

ax[0, 0].set_ylabel('full-dim\ninput')
ax[1, 0].set_ylabel('150-dim\nreconstruction');
```



The top row here shows the input images, while the bottom row shows the reconstruction of the images from just 150 of the ~3,000 initial features.

So although the PCA algorythm reduces the dimensionality of the data by nearly a factor of 20, the projected images contain enough information that we might, by eye, recognize the individuals in the image.

In practice, PCA may often be used as a 'pre-processor' for other algorythms. It significantly reduces the amount of data in the data-set - retaining only the most important information and reducing noise. This means that any subsequent algorythm run on the data is likely to be much more efficient.

References:

Sections of code in this tutorial has been drawn from:

- https://scikit-learn.org/stable/auto_examples/applications/plot_face_recognition.html
- https://jakevdp.github.io/PythonDataScienceHandbook/05.09-principal-component-analysis.html

In [ ]: