```
In [1]: import numpy,sklearn,sklearn.datasets,utils
        %matplotlib inline
```

Principal Component Analysis

In this exercise, we will experiment with two different techniques to compute the PCA components of a dataset:

- **Standard PCA**: The standard technique based on eigenvalue decomposition.
- **Iterative PCA**: A technique that iteratively optimizes the PCA objective.

We consider a random subset of the Labeled Faces in the Wild (LFW) dataset, readily accessible from sklearn, and we apply some basic preprocessing to discount strong variations of luminosity and contrast.

```
In [2]: X = sklearn.datasets.fetch_lfw_people(resize=0.5)['images']
        X = X[numpy.random.mtrand.RandomState(1).permutation(len(X))[:150]]*1.0
        X = X - X.mean(axis=(1,2),keepdims=True)
        X = X / X.std(axis=(1,2),keepdims=True)
        print(X.shape)
```

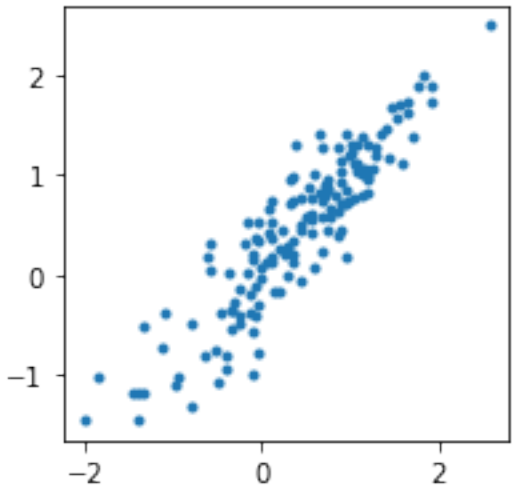
```
(150, 62, 47)
```

Two functions are provided for your convenience and are available in `utils.py` that is included in the zip archive. The functions are the following:

- `utils.scatterplot` produces a scatter plot from a two-dimensional data set.
- `utils.render` takes an array of data points or objects of similar shape, and renders them in the IPython notebook.

Some demo code that makes use of these functions is given below.

```
In [3]: utils.scatterplot(X[:,32,20],X[:,32,21]) # Plot relation between adjacent pixels
        utils.render(X[:30],15,2,vmax=5)         # Display first 10 examples in the data
```



PCA with Eigenvalue Decomposition (15 P)

Principal components can be found by solving the eigenvalue problem

$$S\boldsymbol{w} = \lambda \boldsymbol{w}.$$

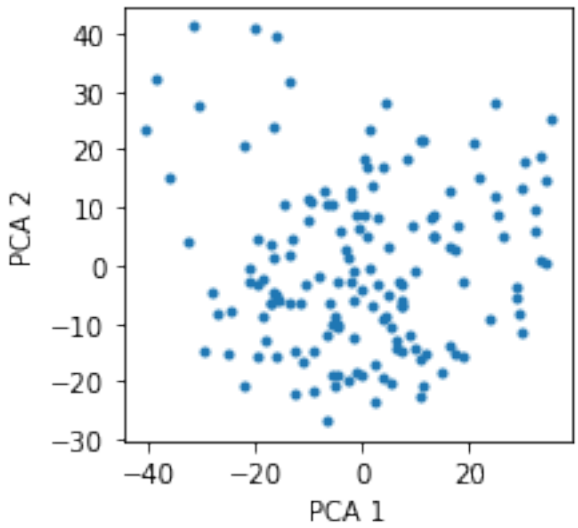
where $S = \sum_{k=1}^{N} (\boldsymbol{x}_k - \boldsymbol{m})(\boldsymbol{x}_k - \boldsymbol{m})^\top$ is the scatter matrix, and where $\boldsymbol{m} = \frac{1}{N} \sum_{k=1}^{N} \boldsymbol{x}_k$ is the mean vector.
Tasks:

- **Compute the principal components of the data using the function** `numpy.linalg.eigh`**.**
- **Measure the computational time required to find the principal components. Use the function** `time.time()` **for that purpose. Do not include in your estimate the computation overhead caused by loading the data, plotting and rendering.**
- **Plot the projection of the dataset on the first two principal components using the function** `utils.scatterplot`**.**
- **Visualize the 60 leading principal components using the function** `utils.render`**.**

Note that if the algorithm runs for more than 1 minute, there may be some error in your implementation.

```
In [4]: ### REPLACE BY YOUR CODE
        import solutions; solutions.basic(X)
        ###
```

```
Time: 4.555 seconds
```



When looking at the scatter plot, we observe that much more variance is expressed in the first two principal components than in individual dimensions as it was plotted before. When looking at the principal components themselves which we render as images, we can see that the first principal components correspond to low-frequency filters that select for coarse features, and the following principal components capture progressively higher-frequency information and are also becoming more noisy.

Iterative PCA (15 P)

The standard PCA method based on eigenvalues is quite expensive to compute. Instead, the power iteration algorithm looks only for the first component and finds it using an iterative procedure. It starts with an initial weight vector $\boldsymbol{w}$, and repeatedly applies the update rule

$$\boldsymbol{w} \leftarrow S\boldsymbol{w} \,/\, \|S\boldsymbol{w}\|.$$

Like for standard PCA, the objective that iterative PCA optimizes is $J(\boldsymbol{w}) = \boldsymbol{w}^\top S\boldsymbol{w}$ subject to the unit norm constraint for $\boldsymbol{w}$. We can therefore keep track of the progress of the algorithm after each iteration.

Tasks:

- **Implement the iterative PCA algorithm. Use as a stopping criterion the value of $J(w)$ between two iterations increasing by less than one.**
- **Print the value of the objective function $J(w)$ at each iteration.**
- **Measure the time taken to find the principal component.**
- **Visualize the the eigenvector w obtained after convergence using the function `utils.render`.**

Note that if the algorithm runs for more than 1 minute, there may be some error in your implementation.

```
In [5]: ### REPLACE BY YOUR CODE
        import solutions; solutions.iterative(X)
        ###
```

```
iteration  0   J(w) =      0.691
iteration  1   J(w) =    131.834
iteration  2   J(w) =    222.571
iteration  3   J(w) =    246.011
iteration  4   J(w) =    259.190
```

```
iteration  5   J(w) =    269.695
iteration  6   J(w) =    277.481
iteration  7   J(w) =    282.657
iteration  8   J(w) =    285.833
iteration  9   J(w) =    287.683
iteration 10   J(w) =    288.730
iteration 11   J(w) =    289.311
stopping criterion satisfied
Time: 0.826 seconds
```

We observe that the computation time has decreased significantly. The difference of performance becomes larger as the number of dimensions increases. We can observe that the principal component is the same (sometimes up to a sign flip) as the one obtained by standard PCA.