Sebastian Raschka
last updated: 02/07/2017

- Link to the containing GitHub Repository: https://github.com/rasbt/pattern_classification (https://github.com/rasbt/pattern_classification)

Stepping through a Principal Component Analysis

(using Python's `numpy` and `matplotlib`)

Sections

Introduction

The main purposes of a principal component analysis are the analysis of data to identify patterns and finding patterns to reduce the dimensions of the dataset with minimal loss of information.

Here, our desired outcome of the principal component analysis is to project a feature space (our dataset consisting of n d-dimensional samples) onto a smaller subspace that represents our data "well". A possible application would be a pattern classification task, where we want to reduce the computational costs and the error of parameter estimation by reducing the number of dimensions of our feature space by extracting a subspace that describes our data "best".

Principal Component Analysis (PCA) Vs. Multiple Discriminant Analysis (MDA)

Both Multiple Discriminant Analysis (MDA) and Principal Component Analysis (PCA) are linear transformation methods and closely related to each other. In PCA, we are interested to find the directions (components) that maximize the variance in our dataset, where in MDA, we are additionally interested to find the directions that maximize the separation (or discrimination) between different classes (for example, in pattern classification problems where our dataset consists of multiple classes. In contrast two PCA, which ignores the class labels).
***In other words, via PCA, we are projecting the entire set of data (without class labels) onto a different subspace, and in MDA, we are trying to determine a suitable subspace to distinguish between patterns that belong to different classes. Or, roughly speaking in PCA we are trying to find the axes with maximum variances where the data is most spread (within a class, since PCA treats the whole data set as one class), and in MDA we are additionally maximizing the spread between classes. ***
In typical pattern recognition problems, a PCA is often followed by an MDA.

What is a "good" subspace?

Let's assume that our goal is to reduce the dimensions of a d-dimensional dataset by projecting it onto a (k)-dimensional subspace (where $k \;<\; d$). So, how do we know what size we should choose for k, and how do we know if we have a feature space that represents our data "well"?
Later, we will compute eigenvectors (the components) from our data set and collect them in a so-called scatter-matrix (or alternatively calculate them from the covariance matrix). Each of those eigenvectors is associated with an eigenvalue, which tell us about the "length" or "magnitude" of the eigenvectors. If we observe that all the eigenvalues are of very similar magnitude, this is a good indicator that our data is already in a "good" subspace. Or if some

Summarizing the PCA approach

Listed below are the 6 general steps for performing a principal component analysis, which we will investigate in the following sections.

1. Take the whole dataset consisting of d-dimensional samples ignoring the class labels
2. Compute the d-dimensional mean vector (i.e., the means for every dimension of the whole dataset)
3. Compute the scatter matrix (alternatively, the covariance matrix) of the whole data set
4. Compute eigenvectors ($\pmb e_1, \; \pmb e_2, \; ..., \; \pmb e_d$) and corresponding eigenvalues ($\pmb\lambda_1, \; \pmb\lambda_2, \; ..., \; \pmb\lambda_d$)
5. Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a $d \times k$ dimensional matrix $\pmb W\;$ (where every column represents an eigenvector)
6. Use this $d \times k$ eigenvector matrix to transform the samples onto the new subspace. This can be summarized by the mathematical equation: $\pmb y = \pmb W^T \times \pmb x$ (where $\pmb x$ is a $d \times 1$-dimensional vector representing one sample, and $\pmb y$ is the transformed $k \times 1$-dimensional sample in the new subspace.)

Generating some 3-dimensional sample data

For the following example, we will generate $2N = 80$ 3-dimensional samples randomly drawn from a multivariate Gaussian distribution.
Here, we will assume that the samples stem from two different classes, where one half (i.e., N=40) samples of our data set are labeled ω_1 (class 1) and the other half ω_2 (class 2).

$$\boldsymbol{\mu_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{\mu_2} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \text{(sample means)}$$

$$\boldsymbol{\Sigma_1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \boldsymbol{\Sigma_2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{(covariance matrices)}$$

Why are we chosing a 3-dimensional sample?

The problem of multi-dimensional data is its visualization, which would make it quite tough to follow our example principal component analysis (at least visually). We could also choose a 2-dimensional sample data set for the following examples, but since the goal of the PCA in an "Diminsionality Reduction" application is to drop at least one of the dimensions, I find it more intuitive and visually appealing to start with a 3-dimensional dataset that we reduce to an 2-dimensional dataset by dropping 1 dimension.

In [1]:

```python
import numpy as np

np.random.seed(0)
N=40

mu_vec1 = np.array([0, 0, 0])
cov_mat1 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 0.1]])
class1_sample = np.random.multivariate_normal(mu_vec1, cov_mat1, N).T
assert class1_sample.shape == (3, N), "The matrix has not the dimension

mu_vec2 = np.array([1, 1, 1])
cov_mat2 = np.array([[1, 0, 0],[0, 1, 0], [0, 0, 0.1]])
class2_sample = np.random.multivariate_normal(mu_vec2, cov_mat2, N).T
assert class2_sample.shape == (3, N), "The matrix has not the dimension
```

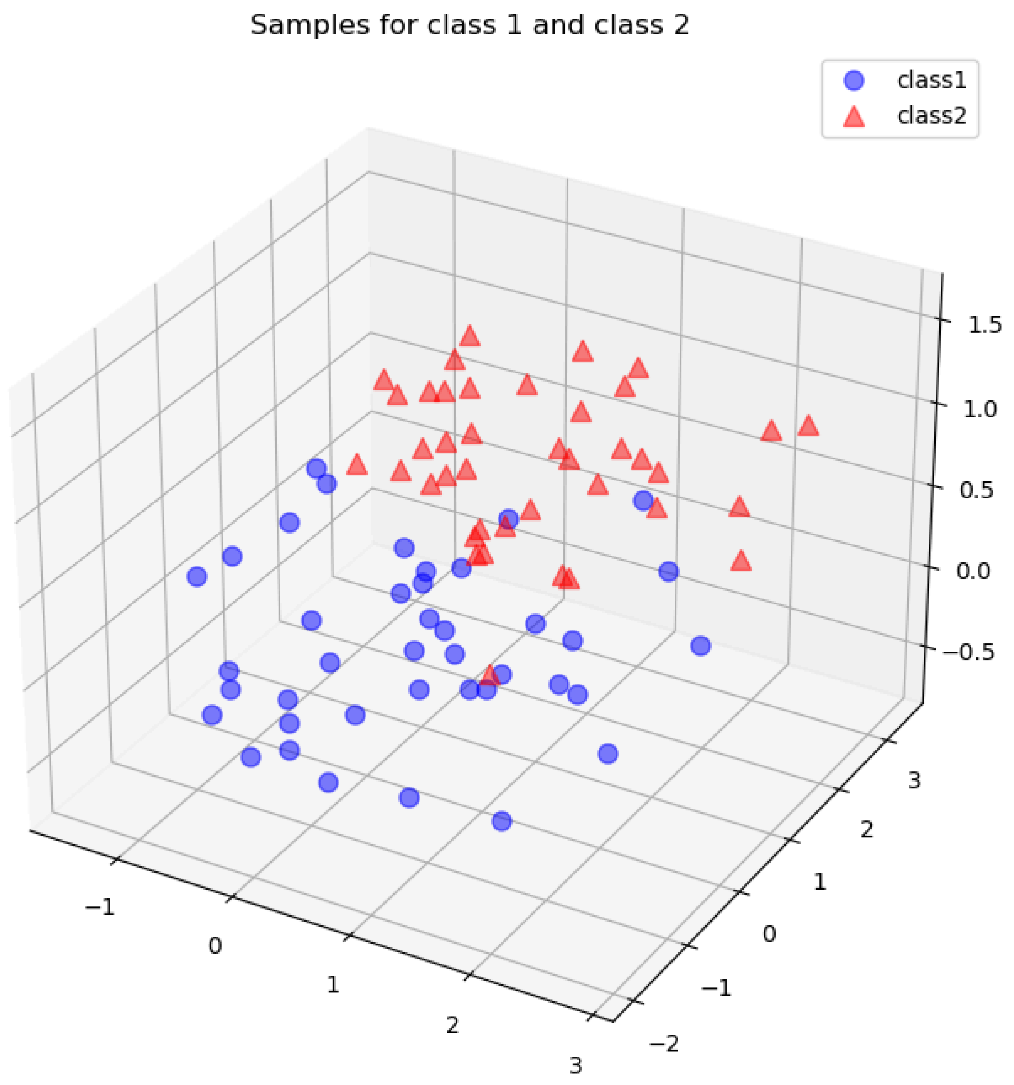
Using the code above, we created two 3×20 datasets - one dataset for each class ω_1 and ω_2 -

where each column can be pictured as a 3-dimensional vector $\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ so that our dataset will have the form

$$\boldsymbol{X} = \begin{pmatrix} x_{1_1} & x_{1_2} & \ldots & x_{1_{20}} \\ x_{2_1} & x_{2_2} & \ldots & x_{2_{20}} \\ x_{3_1} & x_{3_2} & \ldots & x_{3_{20}} \end{pmatrix}$$

Just to get a rough idea how the samples of our two classes ω_1 and ω_2 are distributed, let us plot them in a 3D scatter plot.

In [2]:
```
%matplotlib inline

from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d import proj3d

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
plt.rcParams['legend.fontsize'] = 10
ax.plot(class1_sample[0, :], class1_sample[1, :],
        class1_sample[2, :], 'o', markersize=8,
        color='blue', alpha=0.5, label='class1')
ax.plot(class2_sample[0, :], class2_sample[1, :],
        class2_sample[2, :], '^', markersize=8,
        alpha=0.5, color='red', label='class2')

plt.title('Samples for class 1 and class 2')
ax.legend(loc='upper right')

plt.show()
```



1. Taking the whole dataset ignoring the class labels

Because we don't need class labels for the PCA analysis, let us merge the samples for our 2 classes into one 3×40-dimensional array.

In [3]:

```
all_samples = np.concatenate((class1_sample, class2_sample), axis=1)
assert all_samples.shape == (3, 2*N), "The matrix has not the dimension
```

2. Computing the d-dimensional mean vector

In [4]:

```
mean_x = np.mean(all_samples[0, :])
mean_y = np.mean(all_samples[1, :])
mean_z = np.mean(all_samples[2, :])

mean_vector = np.array([[mean_x],[mean_y],[mean_z]])

print('Mean Vector:\n', mean_vector)
```

```
Mean Vector:
 [[0.62126397]
 [0.63102898]
 [0.46545201]]
```

3. a) Computing the Scatter Matrix S

The scatter matrix is computed by the following equation:

$$S = \sum_{k=1}^{n} (\boldsymbol{x}_k - \boldsymbol{\mu})\ (\boldsymbol{x}_k - \boldsymbol{\mu})^T$$

where $\boldsymbol{\mu}$ is the mean vector

$$\boldsymbol{\mu} = \frac{1}{n} \sum_{k=1}^{n}\ \boldsymbol{x}_k$$

In [5]:
```
scatter_matrix = np.zeros((3, 3))
for i in range(all_samples.shape[1]):
    scatter_matrix += (all_samples[:, i].reshape(3, 1)\
                       - mean_vector).dot((all_samples[:, i].reshape(3,
                                           - mean_vector).T)
print('Scatter Matrix:\n', scatter_matrix)
```

```
Scatter Matrix:
 [[ 78.84955958   7.16472646  15.00557512]
 [  7.16472646 101.75582353  19.70553213]
 [ 15.00557512  19.70553213  27.3546555 ]]
```

3. b) Computing the Covariance Matrix $\Sigma = C = \frac{1}{N-1} S$

(alternatively to the scatter matrix)

Alternatively, instead of calculating the scatter matrix, we could also calculate the covariance matrix using the in-built `numpy.cov()` function. The equations for the covariance matrix and scatter matrix are very similar, the only difference is, that we use the scaling factor $\frac{1}{N-1}$ (here: $\frac{1}{40-1} = \frac{1}{39}$) for the covariance matrix. Thus, their ***eigenspaces*** will be identical (identical eigenvectors, only the eigenvalues are scaled differently by a constant factor).

$$\Sigma_i = \begin{bmatrix} \sigma_{11}^2 & \sigma_{12}^2 & \sigma_{13}^2 \\ \sigma_{21}^2 & \sigma_{22}^2 & \sigma_{23}^2 \\ \sigma_{31}^2 & \sigma_{32}^2 & \sigma_{33}^2 \end{bmatrix}$$

In [6]:
```
cov_mat = np.cov([all_samples[0, :],
                  all_samples[1, :],
                  all_samples[2, :]])
print('Covariance Matrix:\n', cov_mat)
```

```
Covariance Matrix:
 [[0.99809569 0.09069274 0.18994399]
 [0.09069274 1.2880484  0.24943712]
 [0.18994399 0.24943712 0.34626146]]
```

Scaled Scatter Matrix S

In [7]:
```
scatter_matrix/(N-1)
```

Out[7]:
```
array([[2.02178358, 0.18371093, 0.38475834],
       [0.18371093, 2.60912368, 0.50527005],
       [0.38475834, 0.50527005, 0.70140142]])
```

4. Computing eigenvectors and corresponding eigenvalues

To show that the eigenvectors are indeed identical whether we derived them from the scatter or the covariance matrix, let us put an `assert` statement into the code. Also, we will see that the eigenvalues were indeed scaled by the factor 39 when we derived it from the scatter matrix.

In [8]:
```
# eigenvectors and eigenvalues for the from the scatter matrix
eig_val_sc, eig_vec_sc = np.linalg.eig(scatter_matrix)

# eigenvectors and eigenvalues for the from the covariance matrix
eig_val_cov, eig_vec_cov = np.linalg.eig(cov_mat)

for i in range(len(eig_val_sc)):
    eigvec_sc = eig_vec_sc[:, i].reshape(1, 3).T
    eigvec_cov = eig_vec_cov[:,i].reshape(1, 3).T
    assert eigvec_sc.all() == eigvec_cov.all(), 'Eigenvectors are not i

    print('Eigenvector {}: \n{}'.format(i+1, eigvec_sc))
    print('Eigenvalue {} from scatter matrix: {}'.format(i+1, eig_val_s
    print('Eigenvalue {} from covariance matrix: {}'.format(i+1, eig_va
    print('Scaling factor: ', eig_val_sc[i]/eig_val_cov[i])
    print(40 * '-')
```

```
Eigenvector 1:
[[ 0.21616489]
 [ 0.20989637]
 [-0.95352832]]
Eigenvalue 1 from scatter matrix: 19.615191401686616
Eigenvalue 1 from covariance matrix: 0.24829356204666597
Scaling factor:  79.00000000000001
----------------------------------------
Eigenvector 2:
[[-0.91708422]
 [ 0.37874599]
 [-0.12453118]]
Eigenvalue 2 from scatter matrix: 77.92821618940803
Eigenvalue 2 from covariance matrix: 0.9864331163216218
Scaling factor:  78.99999999999991
----------------------------------------
Eigenvector 3:
[[-0.33500638]
 [-0.90138504]
 [-0.27436423]]
Eigenvalue 3 from scatter matrix: 110.41663103073263
Eigenvalue 3 from covariance matrix: 1.3976788738067416
Scaling factor:  79.00000000000003
----------------------------------------
```

Checking the eigenvector-eigenvalue calculation

Let us quickly check that the eigenvector-eigenvalue calculation is correct and satisfy the equation

$$\mathbf{\Sigma}\boldsymbol{v} = \lambda\boldsymbol{v}$$

where
$\mathbf{\Sigma} = Covariance\ matrix$
$\boldsymbol{v} =\ \ Eigenvector$
$\lambda =\ \ Eigenvalue$

In [9]:

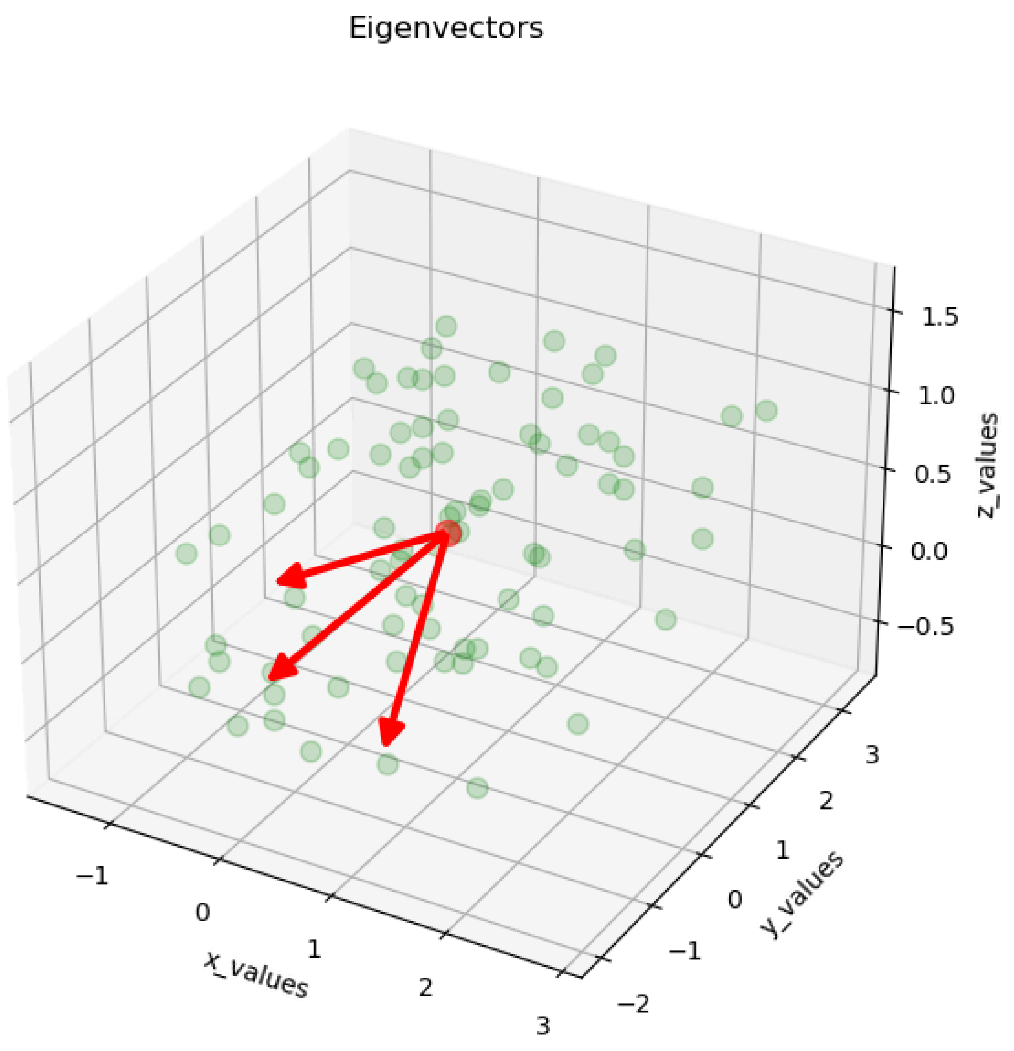
```
for i in range(len(eig_val_sc)):
    eigv = eig_vec_sc[:, i].reshape(1, 3).T
    np.testing.assert_array_almost_equal(scatter_matrix.dot(eigv), eig_
                                         decimal=6, err_msg='', verbose
```

Visualizing the eigenvectors

And before we move on to the next step, just to satisfy our own curiosity, we plot the eigenvectors centered at the sample mean.

In [11]:

```
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d import proj3d
from matplotlib.patches import FancyArrowPatch


class Arrow3D(FancyArrowPatch):
    def __init__(self, xs, ys, zs, *args, **kwargs):
        super().__init__((0,0), (0,0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def do_3d_projection(self, renderer=None):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, self.axes.
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))

        return np.min(zs)

fig = plt.figure(figsize=(7, 7))
ax = fig.add_subplot(111, projection='3d')

ax.plot(all_samples[0, :], all_samples[1, :], all_samples[2, :],
        'o', markersize=8, color='green', alpha=0.2)
ax.plot([mean_x], [mean_y], [mean_z],
        'o', markersize=10, color='red', alpha=0.5)

for v in eig_vec_sc.T:
    a = Arrow3D([mean_x, v[0]], [mean_y, v[1]],
                [mean_z, v[2]], mutation_scale=20,
                lw=3, arrowstyle="-|>", color="r")
    ax.add_artist(a)

ax.set_xlabel('x_values')
ax.set_ylabel('y_values')
ax.set_zlabel('z_values')

plt.title('Eigenvectors')

plt.show()
```



5.1. Sorting the eigenvectors by decreasing eigenvalues

We started with the goal to reduce the dimensionality of our feature space, i.e., projecting the feature space via PCA onto a smaller subspace, where the eigenvectors will form the axes of this new feature subspace. However, the eigenvectors only define the directions of the new axis, since they have all the same unit length 1, which we can confirm by the following code:

In [12]:
```
for ev in eig_vec_sc:
    np.testing.assert_array_almost_equal(1.0, np.linalg.norm(ev))
    # instead of 'assert' because of rounding errors
```

So, in order to decide which eigenvector(s) we want to drop for our lower-dimensional subspace, we have to take a look at the corresponding eigenvalues of the eigenvectors. Roughly speaking, the eigenvectors with the lowest eigenvalues bear the least information

about the distribution of the data, and those are the ones we want to drop.
The common approach is to rank the eigenvectors from highest to lowest corresponding eigenvalue and choose the top k eigenvectors.

In [13]:
```
# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_val_sc[i]),
              eig_vec_sc[:, i]) for i in range(len(eig_val_sc))]

# Sort the (eigenvalue, eigenvector) tuples from high to low
eig_pairs.sort(key=lambda x: x[0], reverse=True)

# Visually confirm that the list is correctly sorted by decreasing eiger
for i in eig_pairs:
    print(i[0])
```

```
110.41663103073263
77.92821618940803
19.615191401686616
```

5.2. Choosing *k* eigenvectors with the largest eigenvalues

For our simple example, where we are reducing a 3-dimensional feature space to a 2-dimensional feature subspace, we are combining the two eigenvectors with the highest eigenvalues to construct our $d \times k$-dimensional eigenvector matrix $\boldsymbol{W}$.

In [14]:
```
matrix_w = np.hstack((eig_pairs[0][1].reshape(3, 1),
                      eig_pairs[1][1].reshape(3, 1)))
print('Matrix W:\n', matrix_w)
```

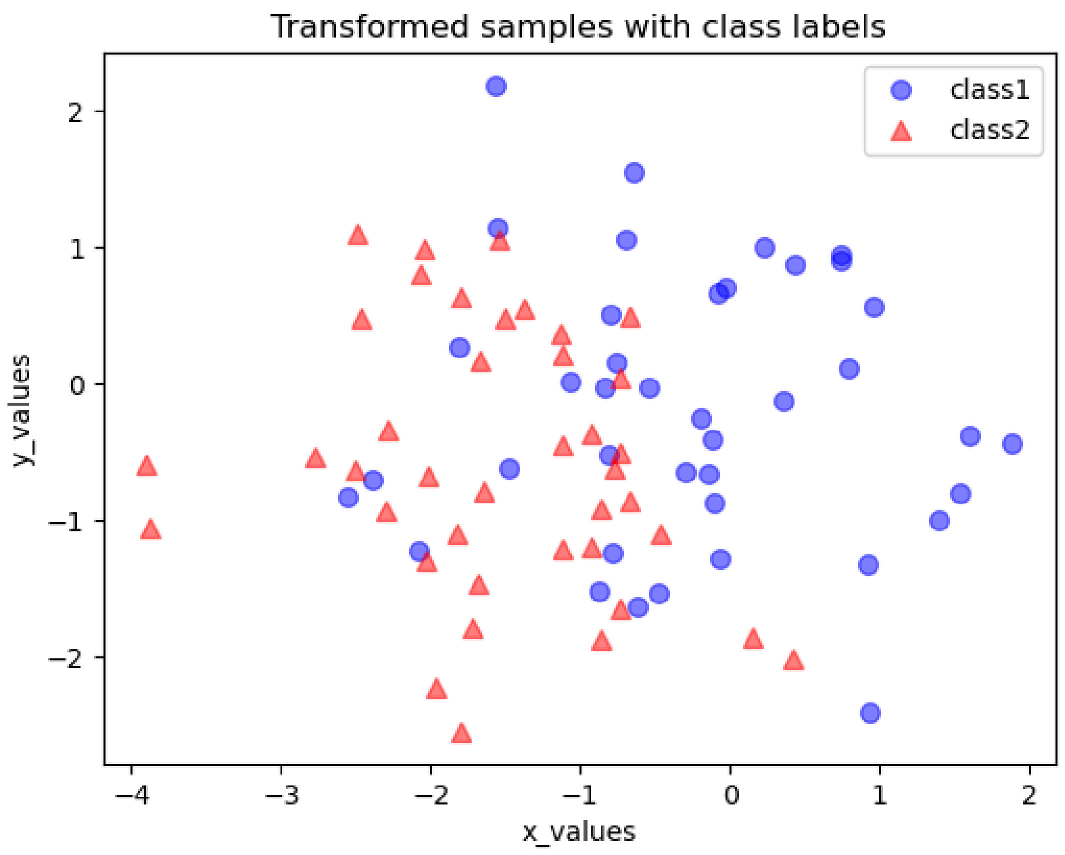
```
Matrix W:
 [[-0.33500638 -0.91708422]
 [-0.90138504  0.37874599]
 [-0.27436423 -0.12453118]]
```

6. Transforming the samples onto the new subspace

In the last step, we use the 2×3-dimensional matrix $\boldsymbol{W}$ that we just computed to transform our samples onto the new subspace via the equation $\boldsymbol{y} = \boldsymbol{W}^T \times \boldsymbol{x}$.

In [15]:
```
transformed = matrix_w.T.dot(all_samples)
assert transformed.shape == (2, 2*N), "The matrix is not 2x40 dimensiona
```

In [16]:
```
plt.plot(transformed[0, 0:N], transformed[1, 0:N],
         'o', markersize=7, color='blue',
         alpha=0.5, label='class1')
plt.plot(transformed[0, N:2*N], transformed[1, N:2*N], '^',
         markersize=7, color='red', alpha=0.5, label='class2')

plt.xlabel('x_values')
plt.ylabel('y_values')
plt.legend()
plt.title('Transformed samples with class labels')

plt.show()
```



Using the PCA() class from the sklearn.decomposition library to confirm our results

In order to make sure that we have not made a mistake in our step by step approach, we will use another library that doesn't rescale the input data by default.
Here, we will use the PCA class from the `scikit-learn` machine-learning library. The

documentation can be found here: [http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html (http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html)](http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html).

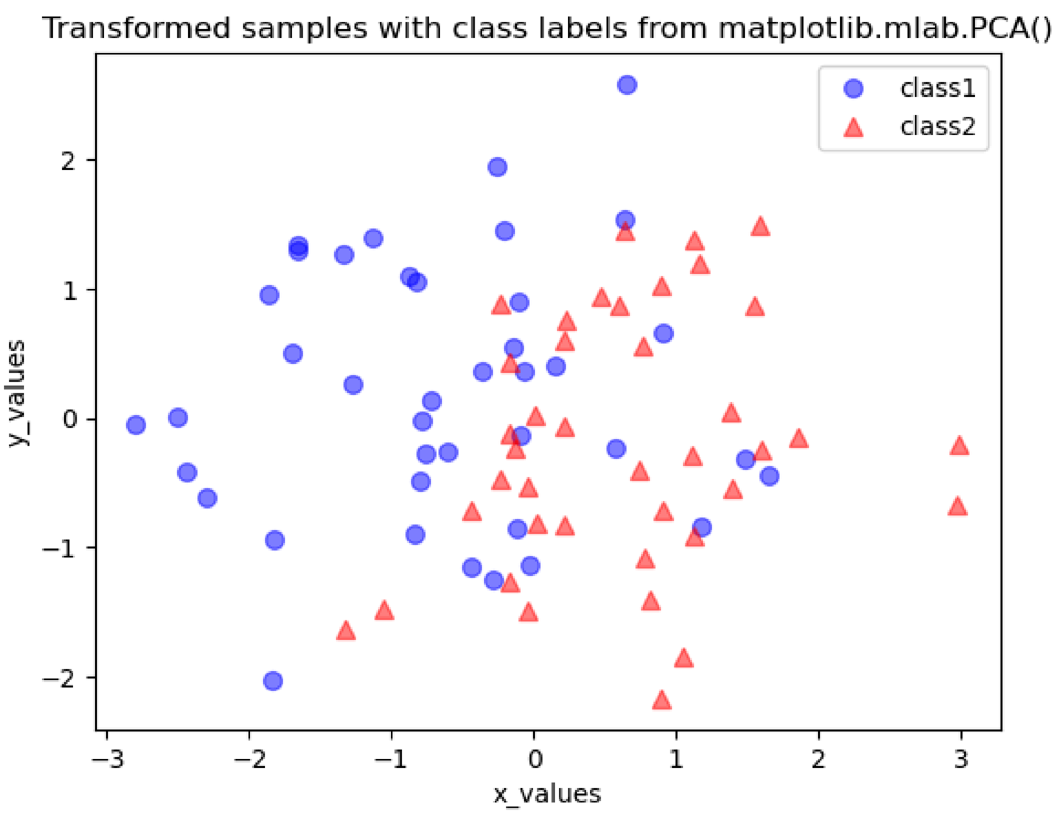
For our convenience, we can directly specify to how many components we want to reduce

Next, we just need to use the `.fit_transform()` in order to perform the dimensionality reduction.

In [17]:
```
np.shape(all_samples)
```

Out[17]:
```
(3, 80)
```

In [18]:
```
from sklearn.decomposition import PCA as sklearnPCA

sklearn_pca = sklearnPCA(n_components=2, svd_solver='full')
sklearn_transf = sklearn_pca.fit_transform(all_samples.T)

plt.plot(sklearn_transf[0:N, 0],sklearn_transf[0:N, 1],
         'o', markersize=7, color='blue', alpha=0.5, label='class1')
plt.plot(sklearn_transf[N:2*N, 0], sklearn_transf[N:2*N, 1],
         '^', markersize=7, color='red', alpha=0.5, label='class2')

plt.xlabel('x_values')
plt.ylabel('y_values')

plt.legend()
plt.title('Transformed samples with class labels from matplotlib.mlab.P(

plt.show()
```



Depending on your computing environmnent, you may find that the plot above is the exact mirror image of the plot from out step by step approach. This is due to the fact that the signs of the eigenvectors can be either positive or negative, since the eigenvectors are scaled to the unit length 1, both we can simply multiply the transformed data by $\times(-1)$ to revert the mirror image.

Please note that this is not an issue: If v is an eigenvector of a matrix Σ, we have,

$$\Sigma v = \lambda v,$$

where λ is our eigenvalue. Then $-v$ is also an eigenvector that has the same eigenvalue, since

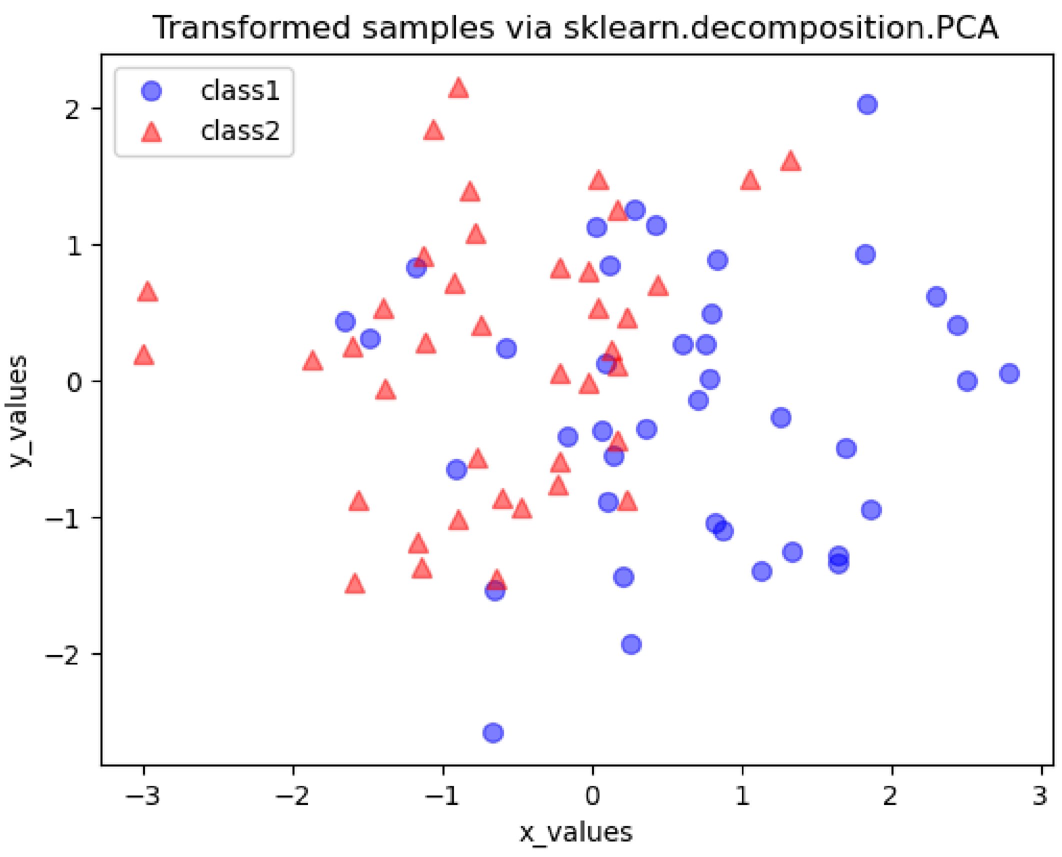
$$\Sigma(-v) = -\Sigma v = -\lambda v = \lambda(-v).$$

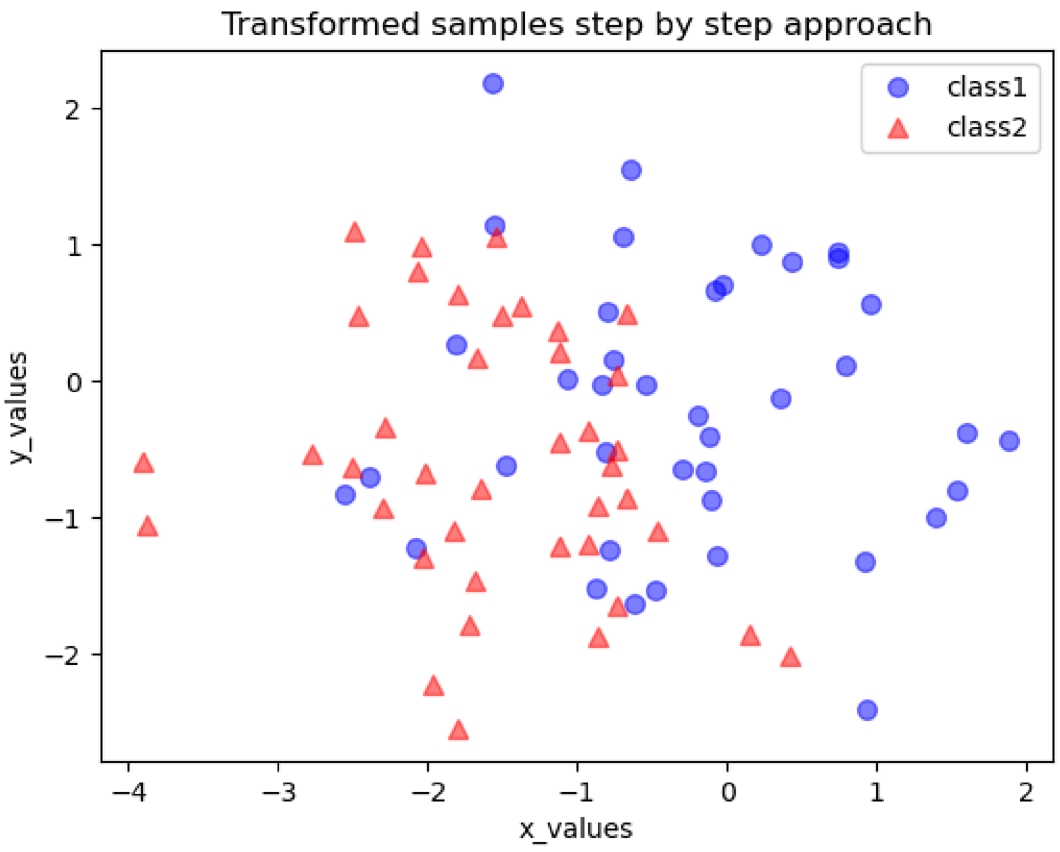
Also, see the note in the scikit-learn documentation:

> **Due to implementation subtleties of the Singular Value Decomposition (SVD), which is used in this implementation, running fit twice on the same matrix can lead to principal components with signs flipped (change in direction). For this reason, it is important to always use the same estimator object to transform data in a consistent fashion.**

(http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html (http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html))

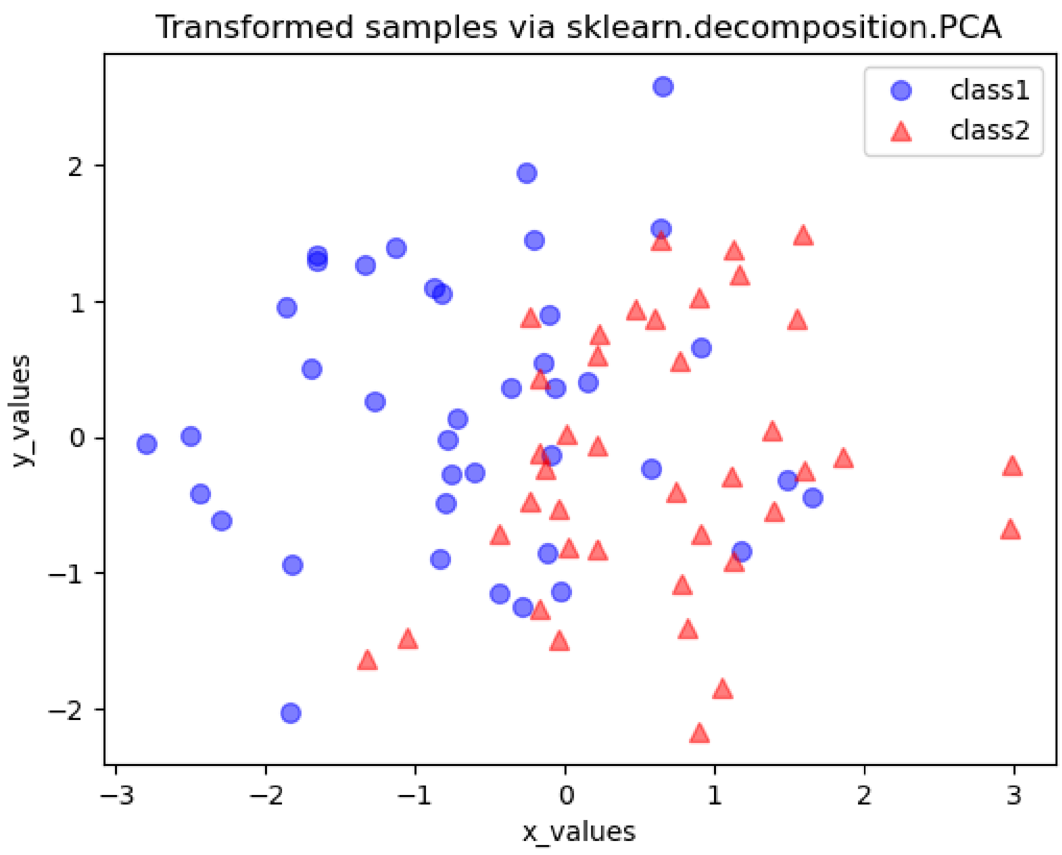
In [19]:
```
# sklearn.decomposition.PCA

sklearn_transf *= (-1)

plt.plot(sklearn_transf[0:N, 0], sklearn_transf[0:N, 1] , 'o',
         markersize=7, color='blue', alpha=0.5, label='class1')
plt.plot(sklearn_transf[N:2*N, 0], sklearn_transf[N:2*N, 1] , '^',
         markersize=7, color='red', alpha=0.5, label='class2')
plt.xlabel('x_values')
plt.ylabel('y_values')
plt.legend()
plt.title('Transformed samples via sklearn.decomposition.PCA')
plt.show()

# step by step PCA
plt.plot(transformed[0, 0:N], transformed[1, 0:N],
         'o', markersize=7, color='blue', alpha=0.5, label='class1')
plt.plot(transformed[0, N:2*N], transformed[1, N:2*N],
         '^', markersize=7, color='red', alpha=0.5, label='class2')

plt.xlabel('x_values')
plt.ylabel('y_values')
plt.legend()
plt.title('Transformed samples step by step approach')
plt.show()
```

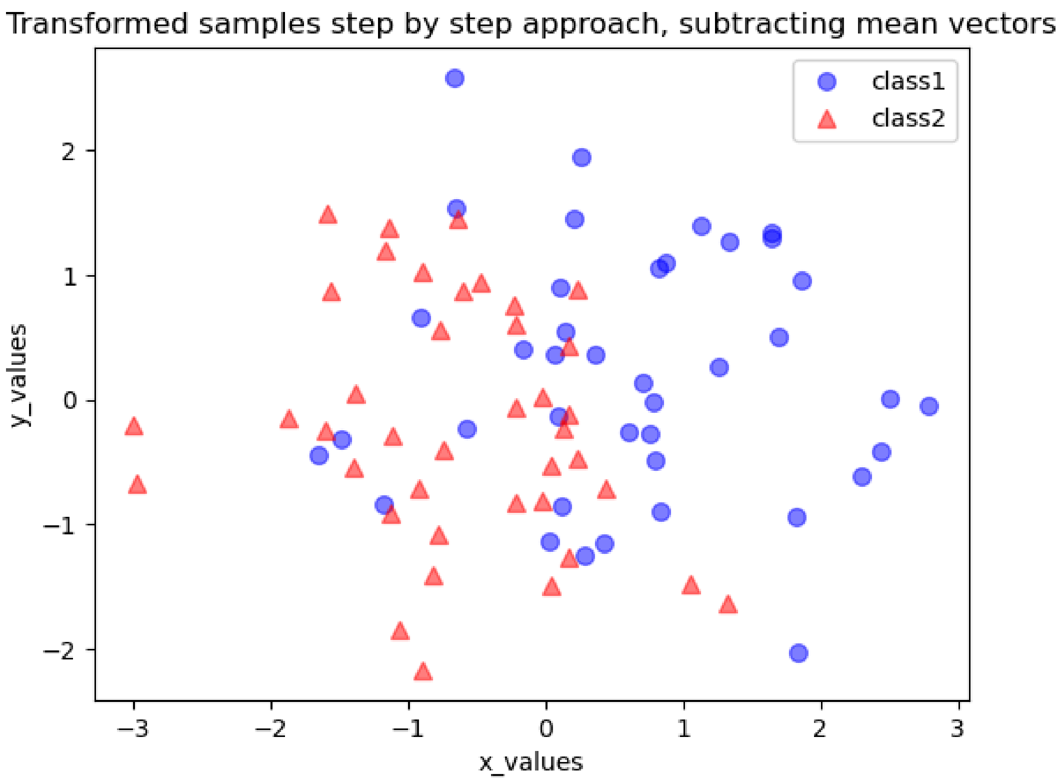



Transformed samples step by step approach
class1
class2
y_values
x_values
2
1
0
−1
−2
−4
−3
−2
−1
0
1
2

In [20]:

```
# sklearn.decomposition.PCA

sklearn_transf *= (-1)

plt.plot(sklearn_transf[0:N, 0], sklearn_transf[0:N, 1] , 'o',
         markersize=7, color='blue', alpha=0.5, label='class1')
plt.plot(sklearn_transf[N:2*N, 0], sklearn_transf[N:2*N, 1] , '^',
         markersize=7, color='red', alpha=0.5, label='class2')
plt.xlabel('x_values')
plt.ylabel('y_values')
plt.legend()
plt.title('Transformed samples via sklearn.decomposition.PCA')
plt.show()

# step by step PCA

transformed = matrix_w.T.dot(all_samples - mean_vector)

plt.plot(transformed[0, 0:N], transformed[1, 0:N],
         'o', markersize=7, color='blue', alpha=0.5, label='class1')
plt.plot(transformed[0, N:2*N], transformed[1, N:2*N],
         '^', markersize=7, color='red', alpha=0.5, label='class2')

plt.xlabel('x_values')
plt.ylabel('y_values')
plt.legend()
plt.title('Transformed samples step by step approach, subtracting mean 
plt.show()
```




Transformed samples step by step approach, subtracting mean vectors
class1
class2
x_values
y_values
−3
−2
−1
0
1
2
3
2
1
0
−1
−2

```
In [ ]:
```

```
In [ ]:
```