sebastianraschka

ABOUT BOOKS DEEP LEARNING ELSEWHERE PROJECTS RESOURCES BLOG

MACHINE LEARNING FAQ

[RSS]

Implementing a Principal Component Analysis (PCA)

– in Python, step by step

Apr 13, 2014
by Sebastian Raschka

Sections

- Differences between the step by step approach and matplotlib.mlab.PCA()
- Using the PCA() class from the sklearn.decomposition library to confirm our results

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 nn dd-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 dd-dimensional dataset by projecting it onto a $(k)(k)$-dimensional subspace (where $k < d k < d$). So, how do we know what size we should choose for kk, 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 of the eigenvalues are much much higher than others, we might be interested in keeping only those eigenvectors with the much larger eigenvalues, since they contain more information about our data distribution. Vice versa, eigenvalues that are close to 0 are less informative and we might consider in dropping those when we construct the new feature subspace.

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$$d$-dimensional samples ignoring the class labels
2. Compute the $d$$d$-dimensional mean vector (i.e., the means for every dimension of the whole dataset)
3. Compute the scatter matrix (alternatively, the covariance matrix) of the whole data set
4. Compute eigenvectors ($\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_d$$\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_d$) and corresponding eigenvalues ($\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2, ..., \boldsymbol{\lambda}_d$$\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2, ..., \boldsymbol{\lambda}_d$)
5. Sort the eigenvectors by decreasing eigenvalues and choose $k$$k$ eigenvectors with the largest eigenvalues to form a $d \times k$$d \times k$ dimensional matrix $\mathbf{W}$ $\mathbf{W}$ (where every column represents an eigenvector)
6. Use this $d \times k$$d \times k$ eigenvector matrix to transform the samples onto the new subspace. This can be summarized by the mathematical equation: $\mathbf{y} = \mathbf{W}^T \times \mathbf{x}$$\mathbf{y} = \mathbf{W}^T \times \mathbf{x}$ (where $\mathbf{x}$$\mathbf{x}$ is a $d \times 1$$d \times 1$-dimensional vector representing one sample, and $\mathbf{y}$$\mathbf{y}$ is the transformed $k \times 1$$k \times 1$-dimensional sample in the new subspace.)

Generating some 3-dimensional sample data

For the following example, we will generate 40 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., 20) samples of our data set are labeled $\omega_1$$\omega_1$ (class 1) and the other half $\omega_2$$\omega_2$ (class 2).

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

$\mathbf{\Sigma_1} = \Sigma_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{\Sigma_2} = \Sigma_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(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.

```
import numpy as np

np.random.seed(234234782384239784) # random seed for consistency

# A reader pointed out that Python 2.7 would raise a
# "ValueError: object of too small depth for desired array".
# This can be avoided by choosing a smaller random seed, e.g. 1
# or by completely omitting this line, since I just used the random seed for
# consistency.

mu_vec1 = np.array([0,0,0])
cov_mat1 = np.array([[1,0,0],[0,1,0],[0,0,1]])
class1_sample = np.random.multivariate_normal(mu_vec1, cov_mat1, 20).T
assert class1_sample.shape == (3,20), "The matrix has not the dimensions 3x20"

mu_vec2 = np.array([1,1,1])
cov_mat2 = np.array([[1,0,0],[0,1,0],[0,0,1]])
class2_sample = np.random.multivariate_normal(mu_vec2, cov_mat2, 20).T
assert class2_sample.shape == (3,20), "The matrix has not the dimensions 3x20"
```

Using the code above, we created two $3 \times 20$$3 \times 20$ datasets - one dataset for each class $\omega_1$$\omega_1$ and $\omega_2$$\omega_2$ -

where each column can be pictured as a 3-dimensional vector $\pmb x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$\pmb x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ so that

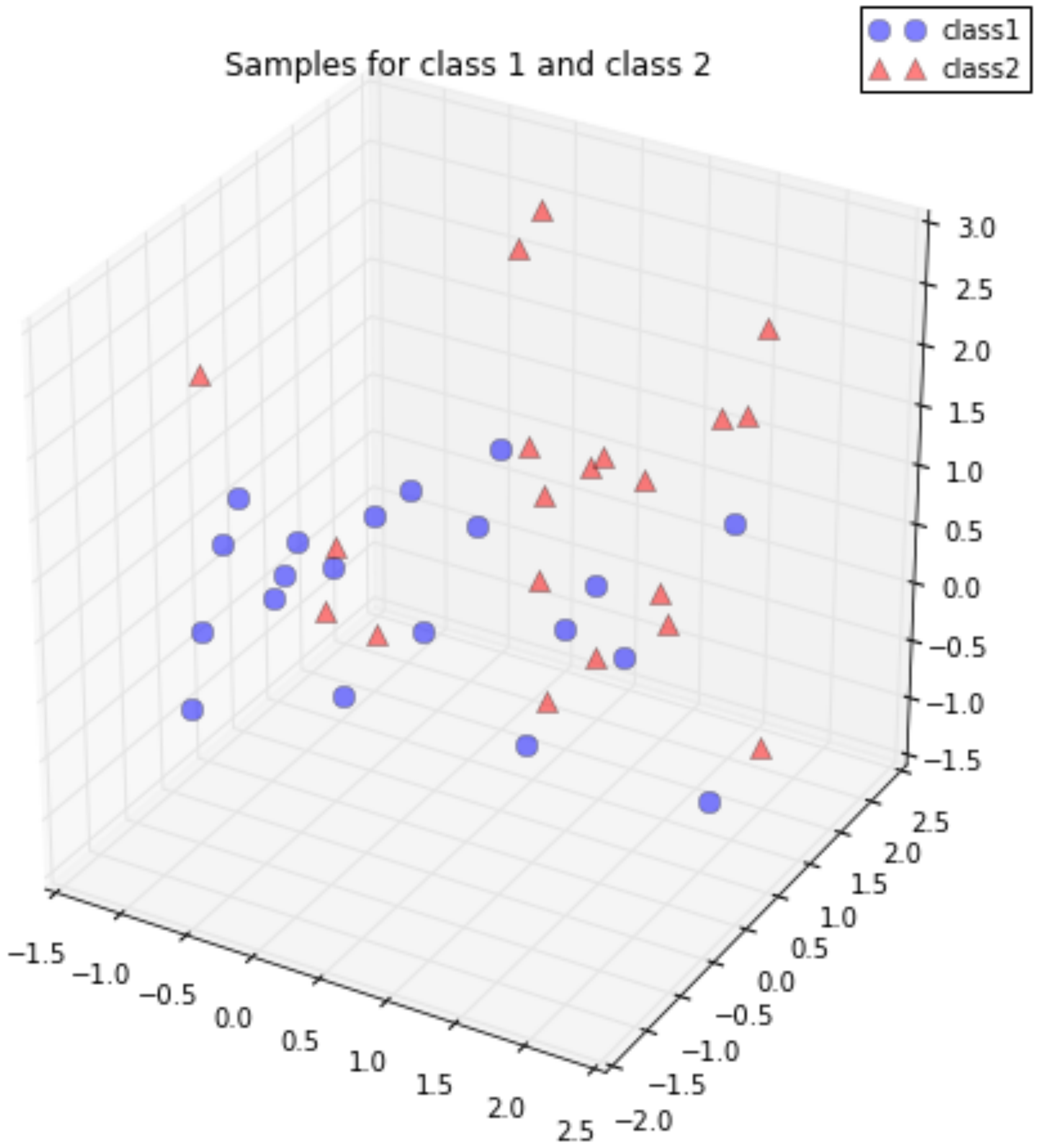
our dataset will have the form

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

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

Just to get a rough idea how the samples of our two classes $\omega_1 \omega_1$ and $\omega_2 \omega_2$ are distributed, let us plot them in a 3D scatter plot.

```
%pylab 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=
ax.plot(class2_sample[0,:], class2_sample[1,:], class2_sample[2,:], '^', markersize=

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 $3 \times 403 \times 40$-dimensional array.

```
all_samples = np.concatenate((class1_sample, class2_sample), axis=1)
assert all_samples.shape == (3,40), "The matrix has not the dimensions 3x40"
```

2. Computing the d-dimensional mean vector

```
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:
 [[ 0.50576644]
 [ 0.30186591]
 [ 0.76459177]]
```

3. a) Computing the Scatter Matrix

The scatter matrix is computed by the following equation:

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

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

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

```
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
print('Scatter Matrix:\n', scatter_matrix)
```

```
Scatter Matrix:
 [[ 48.91593255   7.11744916   7.20810281]
 [  7.11744916  37.92902984   2.7370493 ]
 [  7.20810281   2.7370493   35.6363759 ]]
```

3. b) Computing the Covariance Matrix (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 $\frac{1}{N-1} \frac{1}{N-1}$ (here: $\frac{1}{40-1} = \frac{1}{39} \frac{1}{40-1} = \frac{1}{39}$) for the covariance matrix. Thus, their ***eigenspaces*** will be identical (identical eigenvectors, only the eigenvalues are scaled differently by a constant factor).

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

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

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

```
    Covariance Matrix:
     [[ 1.25425468  0.1824987   0.18482315]
     [ 0.1824987   0.97253923  0.07018075]
     [ 0.18482315  0.07018075  0.91375323]]
```

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.

```
# 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 identical'

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

```
    Eigenvector 1: 
    [[-0.84190486]
     [-0.39978877]
     [-0.36244329]]
    Eigenvalue 1 from scatter matrix: 55.398855957302445
    Eigenvalue 1 from covariance matrix: 1.4204834860846791
    Scaling factor:  39.0
    ----------------------------------------
    Eigenvector 2: 
    [[-0.44565232]
     [ 0.13637858]
     [ 0.88475697]]
    Eigenvalue 2 from scatter matrix: 32.42754801292286
    Eigenvalue 2 from covariance matrix: 0.8314755900749456
    Scaling factor:  39.0
    ----------------------------------------
    Eigenvector 3: 
    [[ 0.30428639]
     [-0.90640489]
     [ 0.29298458]]
    Eigenvalue 3 from scatter matrix: 34.65493432806495
    Eigenvalue 3 from covariance matrix: 0.8885880596939733
    Scaling factor:  39.0
    ----------------------------------------
```

Checking the eigenvector-eigenvalue calculation

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

$$\pmb\Sigma\pmb{v} = \lambda\pmb{v}$$

$$\pmb\Sigma\pmb{v} = \lambda\pmb{v}$$

where

$$\begin{aligned} \pmb\Sigma &= \mathit{Covariance\ matrix} \\ \pmb{v} &= \mathit{Eigenvector} \\ \lambda &= \mathit{Eigenvalue} \end{aligned}$$

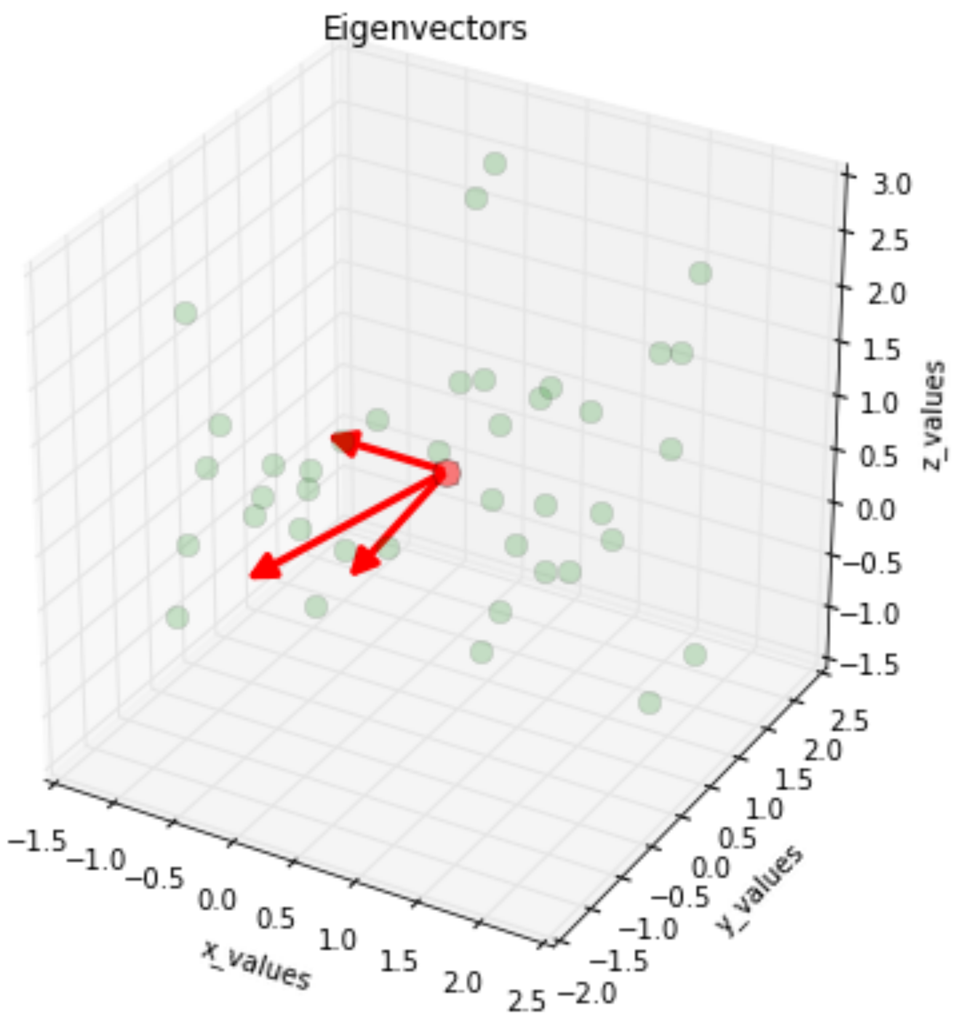
$$\pmb\Sigma = \mathit{Covariance\ matrix} \pmb{v} = \mathit{Eigenvector} \lambda = \mathit{Eigenvalue}$$

```
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_val_sc[i] * e
                                         decimal=6, err_msg='', verbose=True)
```

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.

```
%pylab inline

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

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
        FancyArrowPatch.draw(self, renderer)

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, col
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, l
    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:

```
for ev in eig_vec_sc:
    numpy.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 kk eigenvectors.

```
# 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 eigenvalues
for i in eig_pairs:
    print(i[0])
```

```
    55.3988559573
    34.6549343281
    32.4275480129
```

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 $d \times kd \times k$-dimensional eigenvector matrix $\boldsymbol{W}\boldsymbol{W}$.

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

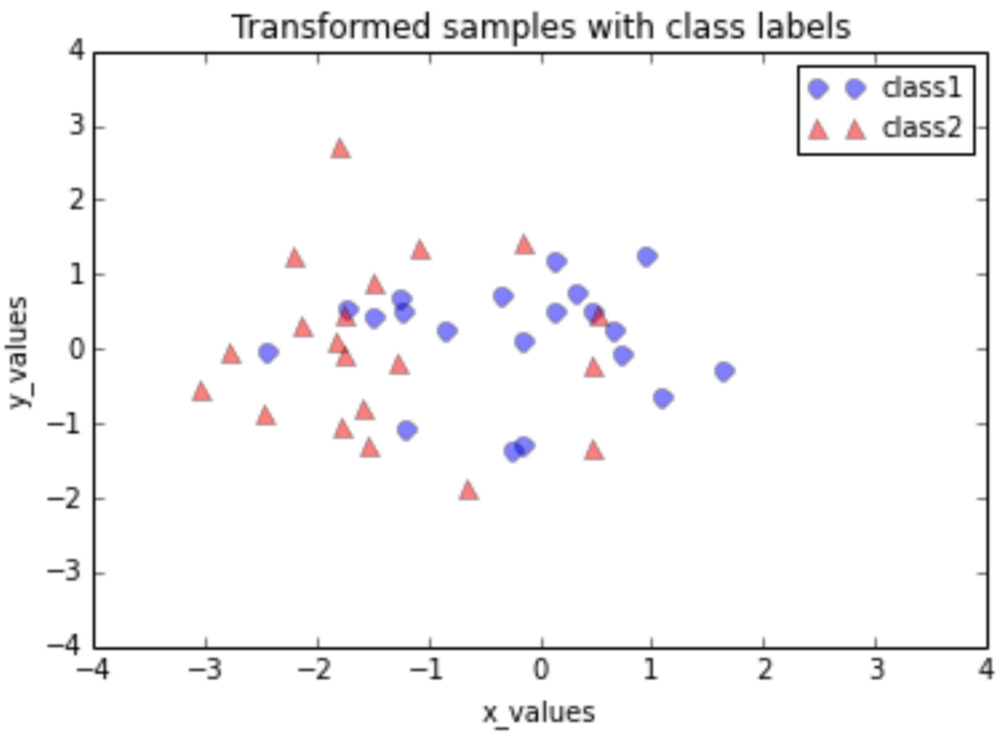
```
    Matrix W:
     [[-0.84190486  0.30428639]
     [-0.39978877 -0.90640489]
     [-0.36244329  0.29298458]]
```

6. Transforming the samples onto the new subspace

In the last step, we use the $2 \times 32 \times 3$-dimensional matrix $\boldsymbol{W}\boldsymbol{W}$ that we just computed to transform our samples onto the new subspace via the equation $\boldsymbol{y} = \boldsymbol{W}^T \times \boldsymbol{x}\boldsymbol{y} = \boldsymbol{W}^T \times \boldsymbol{x}$.

```
transformed = matrix_w.T.dot(all_samples)
assert transformed.shape == (2,40), "The matrix is not 2x40 dimensional."
```

```
plt.plot(transformed[0,0:20], transformed[1,0:20], 'o', markersize=7, color='blue',
plt.plot(transformed[0,20:40], transformed[1,20:40], '^', markersize=7, color='red',
plt.xlim([-4,4])
plt.ylim([-4,4])
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 matplotlib.mlab library

Now, that we have seen how a principal component analysis works, we can use the in-built `PCA()` class from the `matplotlib` library for our convenience in future applications. Unfortunately, the original documentation (http://matplotlib.sourceforge.net/api/mlab_api.html#matplotlib.mlab.PCA) is very sparse; a better documentation can be found here: https://www.clear.rice.edu/comp130/12spring/pca/pca_docs.shtml.

And the original code implementation of the `PCA()` class can be viewed at: https://sourcegraph.com/github.com/matplotlib/matplotlib/symbols/python/lib/matplotlib/mlab/PCA

Class attributes of `PCA()`

```
    Attrs:
```

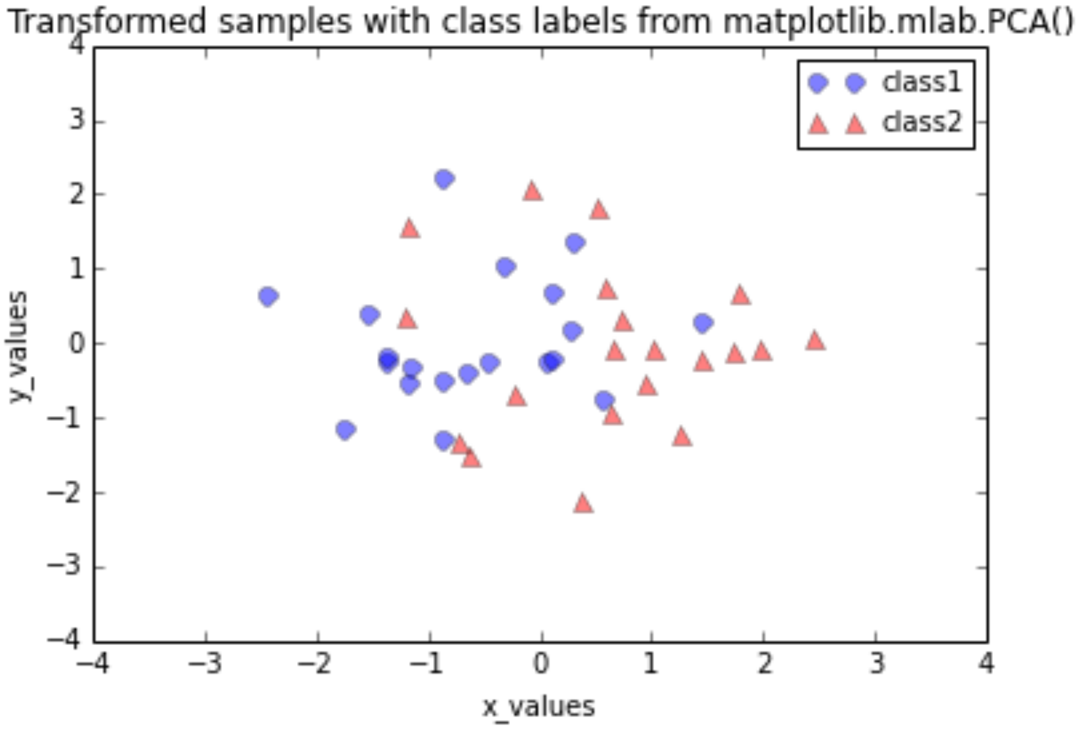
```
    a : a centered unit sigma version of input a

    numrows, numcols: the dimensions of a

    mu : a numdims array of means of a

    sigma : a numdims array of atandard deviation of a

    fracs : the proportion of variance of each of the principal components

    Wt : the weight vector for projecting a numdims point or array into PCA space

    Y : a projected into PCA space
```

Also, it has to be mentioned that the `PCA()` class expects a `np.array()` as input where: `'we assume data in a is organized with numrows>numcols')`, so that we have to transpose our dataset.

`matplotlib.mlab.PCA()` keeps all $d d$-dimensions of the input dataset after the transformation (stored in the class attribute `PCA.Y`), and assuming that they are already ordered ("Since the PCA analysis orders the PC axes by descending importance in terms of describing the clustering, we see that fracs is a list of monotonically decreasing values.", https://www.clear.rice.edu/comp130/12spring/pca/pca_docs.shtml) we just need to plot the first 2 columns if we are interested in projecting our 3-dimensional input dataset onto a 2-dimensional subspace.

```
from matplotlib.mlab import PCA as mlabPCA

mlab_pca = mlabPCA(all_samples.T)

print('PC axes in terms of the measurement axes scaled by the standard deviations:\n

plt.plot(mlab_pca.Y[0:20,0],mlab_pca.Y[0:20,1], 'o', markersize=7, color='blue', alp
plt.plot(mlab_pca.Y[20:40,0], mlab_pca.Y[20:40,1], '^', markersize=7, color='red', a

plt.xlabel('x_values')
plt.ylabel('y_values')
plt.xlim([-4,4])
plt.ylim([-4,4])
plt.legend()
plt.title('Transformed samples with class labels from matplotlib.mlab.PCA()')

plt.show()
```

```
PC axes in terms of the measurement axes scaled by the standard deviations:
 [[ 0.65043619  0.53023618  0.54385876]
 [-0.01692055  0.72595458 -0.68753447]
 [ 0.75937241 -0.43799491 -0.48115902]]
```



Differences between the step by step approach and matplotlib.mlab.PCA()

When we plot the transformed dataset onto the new 2-dimensional subspace, we observe that the scatter plots from our step by step approach and the `matplotlib.mlab.PCA()` class do not look identical. This is due to the fact that `matplotlib.mlab.PCA()` class ***scales the variables to unit variance*** prior to calculating the covariance matrices. This will/could eventually lead to different variances along the axes and affect the contribution of the variable to principal components.

One example where a scaling would make sense would be if one variable was measured in the unit **inches** where the other variable was measured in **cm**.
However, for our hypothetical example, we assume that both variables have the same (arbitrary) unit, so that we skipped the step of scaling the input data.

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

confirm our results

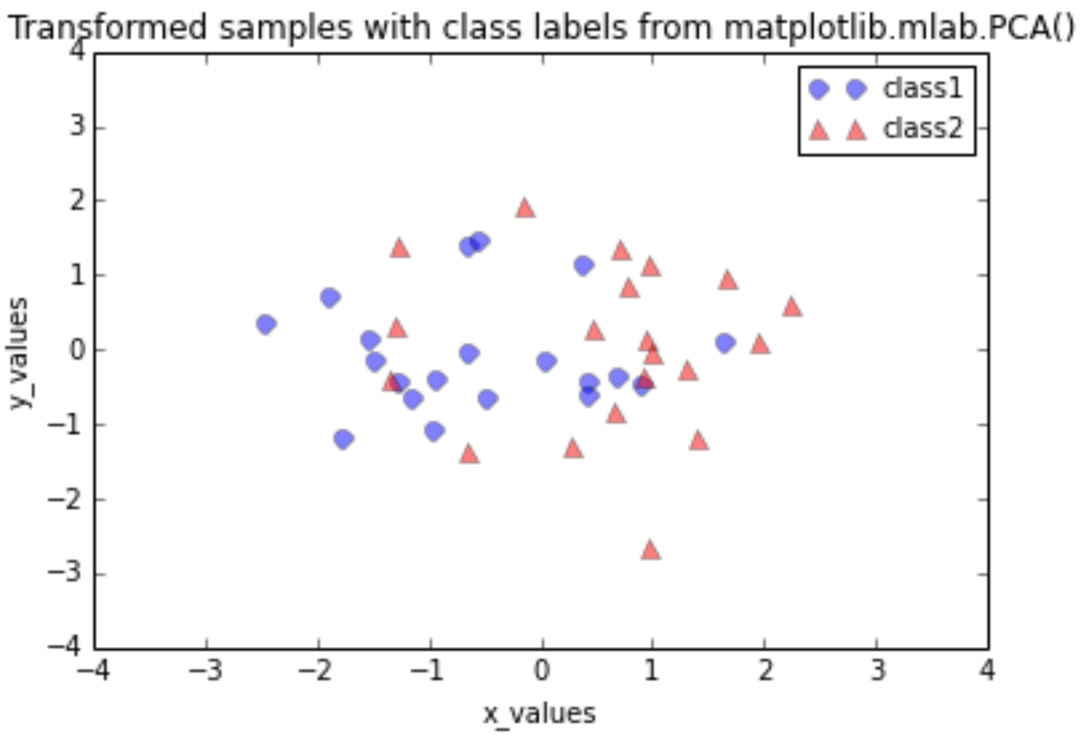
In order to make sure that we have not made a mistake in our step by step approach, we will use another library that doesn't rescale the input data by default.
Here, we will use the PCA class from the `scikit-learn` machine-learning library. The documentation can be found here:
http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

For our convenience, we can directly specify to how many components we want to reduce our input dataset via the `n_components` parameter.

```
    n_components : int, None or string

    Number of components to keep. if n_components is not set all components are kept
        n_components == min(n_samples, n_features)
        if n_components == ‘mle’, Minka’s MLE is used to guess the dimension if 0 <
        select the number of components such that the amount of variance that needs
        is greater than the percentage specified by n_components
```

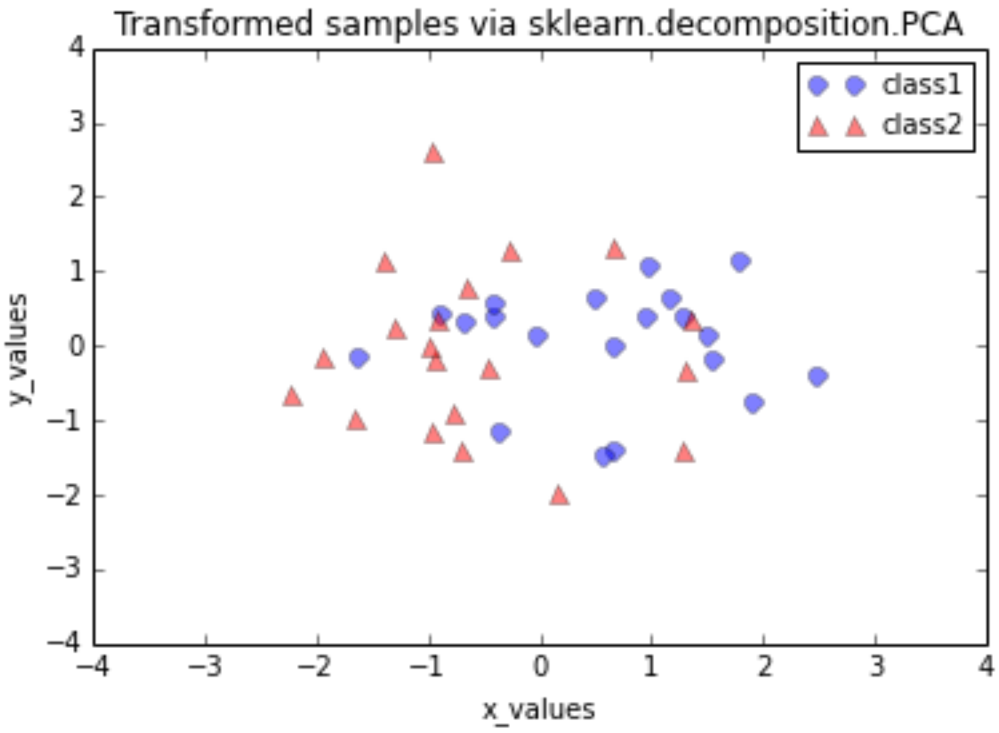
Next, we just need to use the `.fit_transform()` in order to perform the dimensionality reduction.

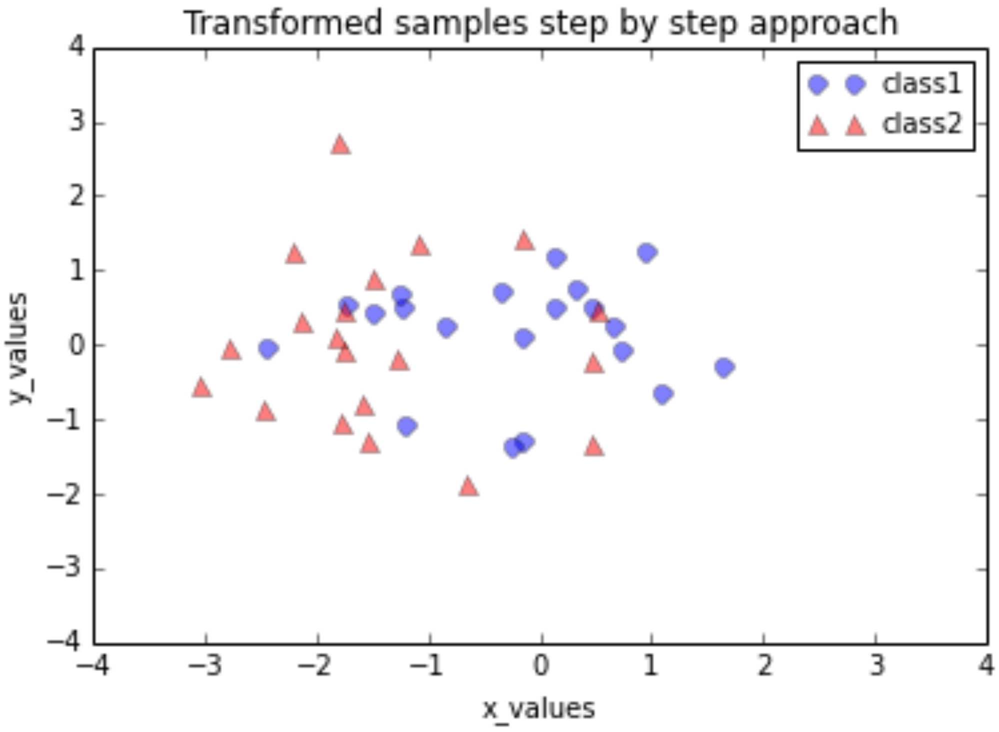
```
from sklearn.decomposition import PCA as sklearnPCA

sklearn_pca = sklearnPCA(n_components=2)
sklearn_transf = sklearn_pca.fit_transform(all_samples.T)

plt.plot(sklearn_transf[0:20,0],sklearn_transf[0:20,1], 'o', markersize=7, color='bl
plt.plot(sklearn_transf[20:40,0], sklearn_transf[20:40,1], '^', markersize=7, color=

plt.xlabel('x_values')
plt.ylabel('y_values')
plt.xlim([-4,4])
plt.ylim([-4,4])
plt.legend()
plt.title('Transformed samples with class labels from matplotlib.mlab.PCA()')

plt.show()
```



The plot above seems to be 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 $\times(-1)$ $\times (- 1)$ to revert the mirror image.

```
sklearn_transf = sklearn_transf * (-1)

# sklearn.decomposition.PCA
plt.plot(sklearn_transf[0:20,0],sklearn_transf[0:20,1], 'o', markersize=7, color='bl
plt.plot(sklearn_transf[20:40,0], sklearn_transf[20:40,1], '^', markersize=7, color=
plt.xlabel('x_values')
plt.ylabel('y_values')
plt.xlim([-4,4])
plt.ylim([-4,4])
plt.legend()
plt.title('Transformed samples via sklearn.decomposition.PCA')
plt.show()

# step by step PCA
plt.plot(transformed[0,0:20], transformed[1,0:20], 'o', markersize=7, color='blue',
plt.plot(transformed[0,20:40], transformed[1,20:40], '^', markersize=7, color='red',
plt.xlim([-4,4])
plt.ylim([-4,4])
plt.xlabel('x_values')
plt.ylabel('y_values')
plt.legend()
plt.title('Transformed samples step by step approach')
plt.show()
```





Looking at the 2 plots above, the distributions along the component axes look identical, only the center of the data is slightly different. If we want to mimic the results produced by scikit-learn's `PCA` class, we can subtract the mean vectors from the samples `X` to center the data at the coordinate system's origin (thanks to a suggestion by Alexander Guth) – that is, replacing the transformation `transformed = matrix_w.T.dot(all_samples)` by `transformed = matrix_w.T.dot(all_samples - mean_vector)` .

Have feedback on this post? I would love to hear it. Let me know and send me a tweet or email.

Processing math: 100%