Abdul Qayyum

Lecturer at University of Burgundy, France

- Postdoc in Electrical and Informatics Engineering
- PhD in Electrical & Electronics Engineering
- Masters in Electronics Engineering
- Bachelor in Computer Engineering

Collaborations & Expertise:

Topic: Data Dimension Reduction Models

Instructor: Abdul Qayyum, PhD

Class: MSCV

University of Burgundy, France

Dimension Reduction Models

- In this chapter, we will cover the following topics:
- • Principal component analysis (PCA) for unsupervised data compression
- • Linear discriminant analysis (LDA) as a supervised dimensionality reduction technique for maximizing class separability
- • Nonlinear dimensionality reduction via kernel principal component analysis (KPCA)

Principal component analysis (PCA)

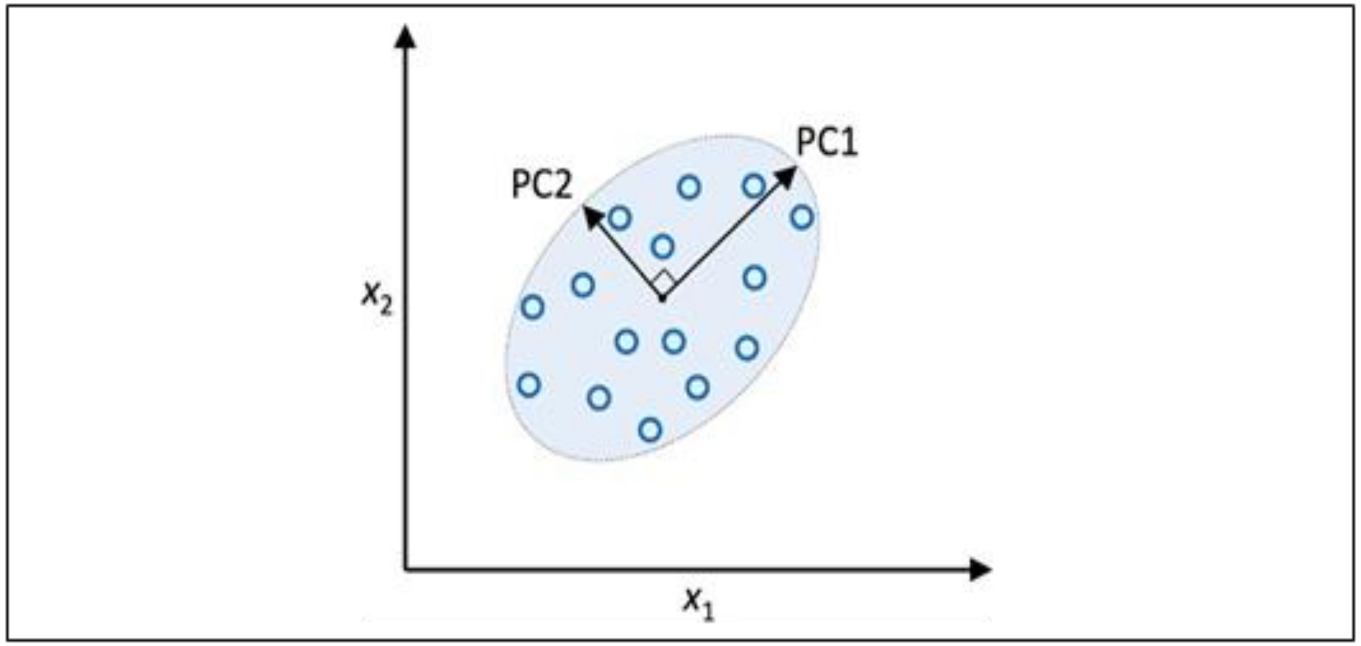
- **Unsupervised dimensionality reduction via principal component analysis**
- Similar to feature selection, we can use different feature extraction techniques to reduce the number of features in a dataset. The difference between feature selection and feature extraction is that while we maintain the original features when we use feature selection algorithms, such as sequential backward selection, we use feature extraction to transform or project the data onto a new feature space.
- In the context of dimensionality reduction, feature extraction can be understood as an approach to data compression with the goal of maintaining most of the relevant information. In practice, feature extraction is not only used to improve storage space or the computational efficiency of the learning algorithm, but can also improve the predictive performance by reducing the curse of dimensionality—especially if we are working with non-regularized models.

Principal component analysis (PCA)

- **The main steps behind principal component analysis**
- We will discuss PCA, an unsupervised linear transformation technique that is widely used across different fields, most prominently for **feature extraction and dimensionality reduction.**
- Other popular applications of PCA include exploratory data analyses and the denoising of signals in stock market trading, and the analysis of genome data and gene expression levels in the field of bioinformatics.
- PCA helps us to identify patterns in data based on the correlation between features.
- PCA aims to find the directions of maximum variance in high- dimensional data and projects the data onto a new subspace with equal or fewer dimensions than the original one.

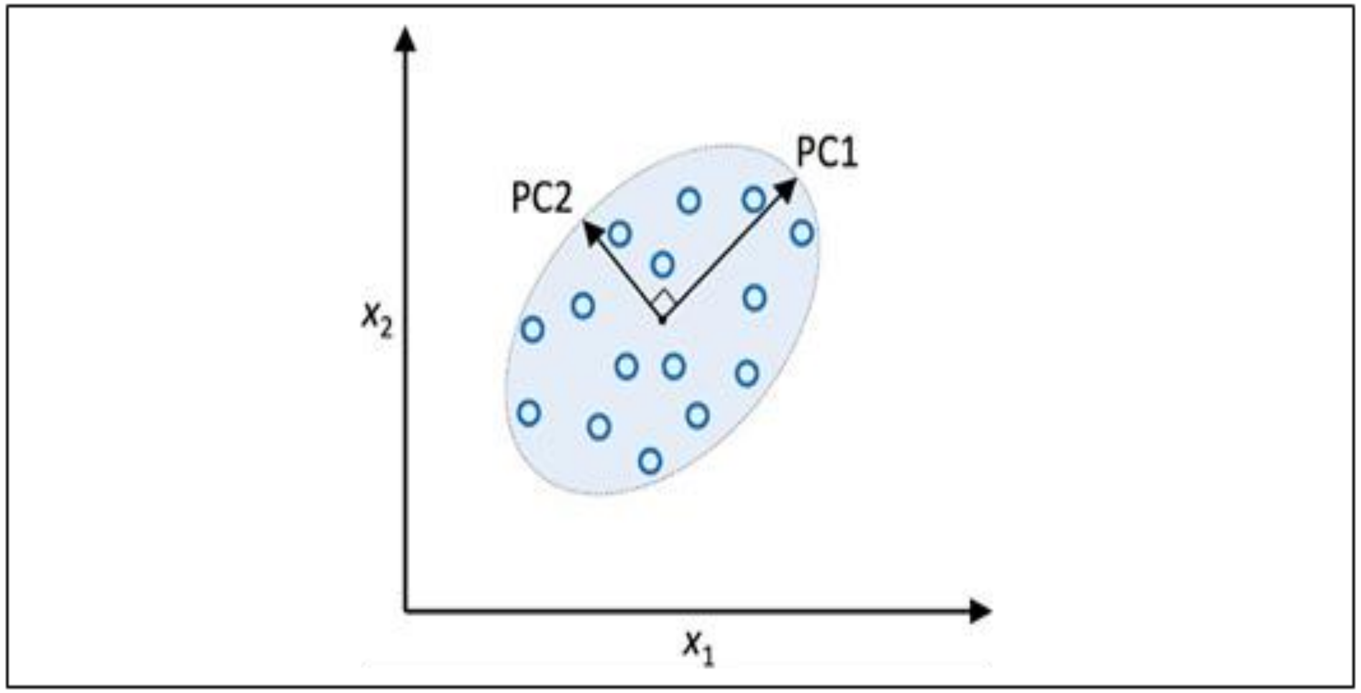
Principal component analysis (PCA)

- **The main steps behind principal component analysis**
- The orthogonal axes (principal components) of the new subspace can be interpreted as the directions of maximum variance given the constraint that the new feature axes are orthogonal to each other, as illustrated in the following figure:



Principal component analysis (PCA)

- **The main steps behind principal component analysis**
- In the preceding figure, $x1$ and $x2$ are the original feature axes, and PC1 and PC2 are the principal components.



Principal component analysis (PCA)

- **The main steps behind principal component analysis**
- If we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix, W, that allows us to map a vector, x, the features of a training example, onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space. For instance, the process is as follows.
- Suppose we have a feature vector, x:
- $\boldsymbol{x} = [x_1, x_2, x_3, \dots, x_d]$ $\quad x \in R^d$
- which is then transformed by a transformation matrix, $\boldsymbol{W} \in R^{d \times k}$
- $\boldsymbol{xW} = \boldsymbol{z}$
- resulting in the output vector:
- $\boldsymbol{z} = [z_1, z_2, z_3, \dots, z_d]$ $\quad z \in R^k$
- As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically k << d), the first principal component will have the largest possible variance.

Principal component analysis (PCA)

- **The main steps behind principal component analysis**
- Suppose we have a feature vector, x:
- $\boldsymbol{x} = [x_1, x_2, x_3, \dots, x_d]$ $\quad x \in R^d$
- which is then transformed by a transformation matrix, $\boldsymbol{W} \in R^{d \times k}$
- $\boldsymbol{xW} = \boldsymbol{z}$
- resulting in the output vector:
- $\boldsymbol{z} = [z_1, z_2, z_3, \dots, z_d]$ $\quad z \in R^k$
- As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically k << d), the first principal component will have the largest possible variance.
- All consequent principal components will have the largest variance given the constraint that these components are uncorrelated (orthogonal) to the other principal components—even if the input features are correlated, the resulting principal components will be mutually orthogonal (uncorrelated). Note that the PCA directions are highly sensitive to data scaling, and we need to standardize the features prior to PCA if the features were measured on different scales and we want to assign equal importance to all features.

Principal component analysis (PCA)

- **The main steps behind principal component analysis**
- Before looking at the PCA algorithm for dimensionality reduction in more detail, let's summarize the approach in a few simple steps:
- 1. Standardize the d-dimensional dataset.
- 2. Construct the covariance matrix.
- 3. Decompose the covariance matrix into its eigenvectors and eigenvalues.
- 4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.
- 5. Select k eigenvectors, which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
- 6. Construct a projection matrix, W, from the "top" k eigenvectors.
- 7. Transform the d-dimensional input dataset, X, using the projection matrix, W, to obtain the new k-dimensional feature subspace.
- In the following sections, we will perform a PCA step by step, using Python as a learning exercise. Then, we will see how to perform a PCA more conveniently using scikit-learn.

Principal component analysis (PCA)

- **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 ω1 (class 1) and the other half ω2 (class 2).

$$\mu_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mu_2 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\Sigma_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \Sigma_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Principal component analysis (PCA)

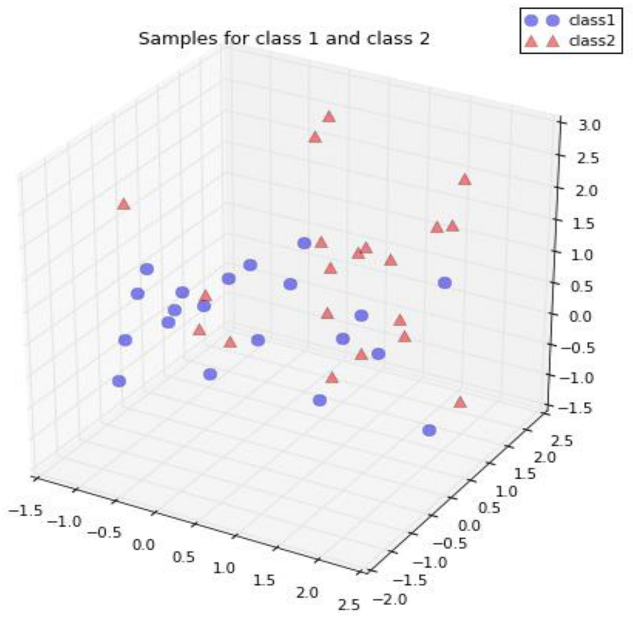
- **Generating some 3-dimensional sample data**
- For the following example, we will generate 40 3-dimensional samples randomly drawn from a multivariate Gaussian distribution.
- Here, we will assume that the samples stem from two different classes, where one half (i.e., 20) samples of our data set are labeled ω1 (class 1) and the other half ω2 (class 2).

We created two 3×20 datasets - one dataset for each class ω1 and ω2 -where each column can be pictured as a 3-dimensional vector

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

so that our dataset will have the form

$$\boldsymbol{X} = \begin{pmatrix} x_1 & x_{1,2} & x_{1,20} \\ x_2 & x_{2,1} & x_{2,20} \\ x_3 & x_{3,1} & x_{3,20} \end{pmatrix}$$



Principal component analysis (PCA)

- **Generating some 3-dimensional sample data**
- **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.**
- **2. Computing the d-dimensional mean vector**

$$\boldsymbol{X} = \begin{pmatrix} x_1 & x_{1,2} & x_{1,40} \\ x_2 & x_{2,1} & x_{2,40} \\ x_3 & x_{3,1} & x_{3,40} \end{pmatrix}$$

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

```
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]]
```

Principal component analysis (PCA)

- **Generating some 3-dimensional sample data**
- **3. a) Computing the Scatter Matrix**
- **The scatter matrix is computed by the following equation:**

$$S = \sum_{k=1}^{n} (x_k - m)(x_k - m)^T$$

m= $\frac{1}{n}\sum_{k=1}^{n} x_k$

- **Where m is the mean vector**

```
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(3,1) -
mean_vector).T)
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 ]]
```

Principal component analysis (PCA)

- **Generating some 3-dimensional sample data**
- **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 1/N−1 (here: 1/40−1=1/39) for the covariance matrix. **Thus, their eigenspaces will be identical (identical eigenvectors, only the eigenvalues are scaled differently by a constant factor).**

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

```
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]]
```

Principal component analysis (PCA)

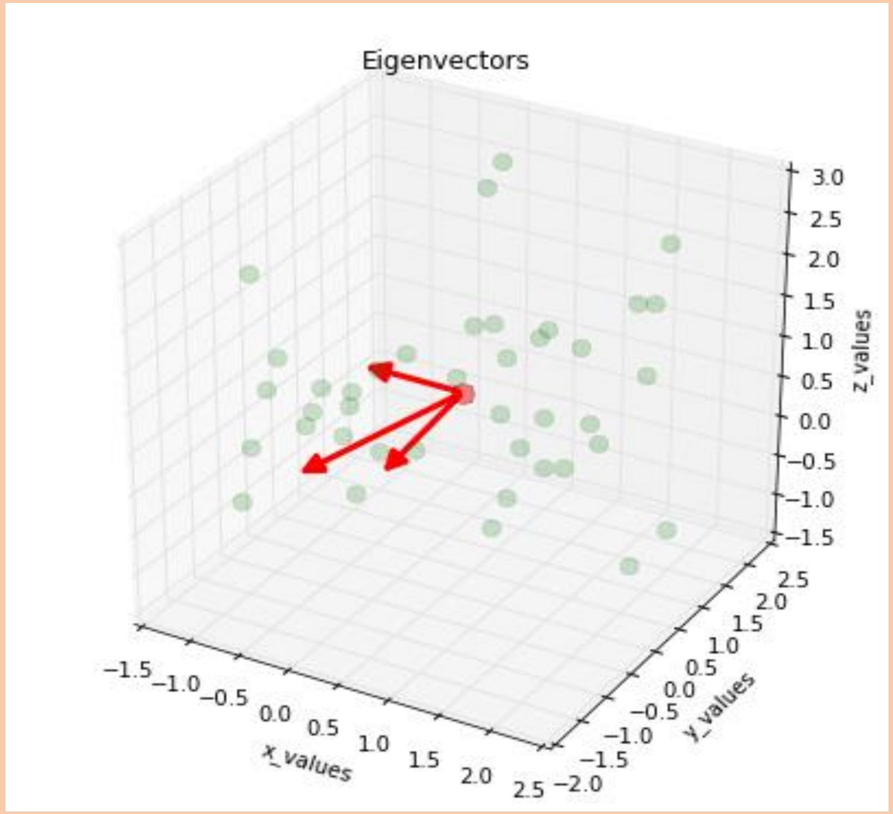
- **Generating some 3-dimensional sample data**
- **4.Computing eigenvectors and corresponding eigenvalues**
- To show that the eigenvectors are indeed identical whether we derived them from the scatter or the covariance matrix, let us put an assert statement into the code. Also, we will see that the 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 * '-')
```

Principal component analysis (PCA)

- **Generating some 3-dimensional sample data**
- **4.Computing eigenvectors and corresponding eigenvalues**

```
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
```



Principal component analysis (PCA)

- **Generating some 3-dimensional sample data**
- **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,
- 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.

```
# 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
```

Principal component analysis (PCA)

- **Generating some 3-dimensional sample data**
- **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×k-dimensional eigenvector matrix W.

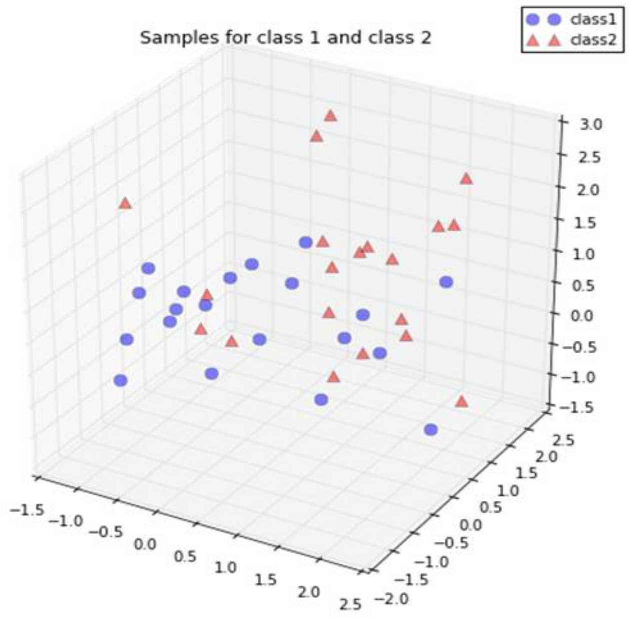
```
matrix_w = np.hstack((eig_pairs[0][1].reshape(3,1), eig_pairs[1][1].reshape(3,1)))
print('Matrix W:\n', matrix_w)
```

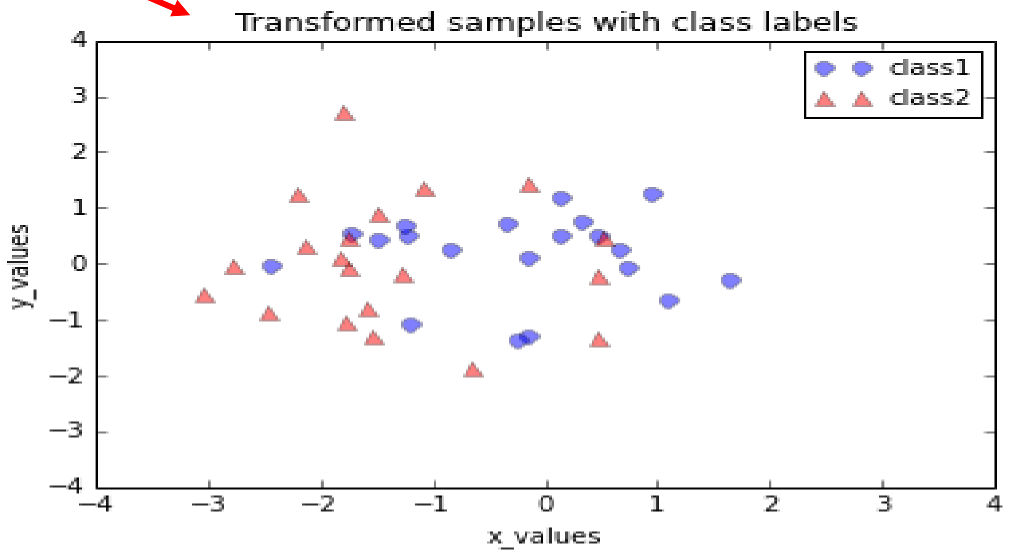
```
Matrix W:
    [[-0.84190486  0.30428639]
    [-0.39978877 -0.90640489]
    [-0.36244329  0.29298458]]
```

Principal component analysis (PCA)

- **Generating some 3-dimensional sample data**
- **6. Transforming the samples onto the new subspace**
- In the last step, we use the 2×3-dimensional matrix W that we just computed to transform our samples onto the new subspace via the equation $y = W^T x$.

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





Principal component analysis (PCA)

- **Extracting the principal components step by step**
- In this subsection, we will tackle the first four steps of a PCA:
- 1. Standardizing the data.
- 2. Constructing the covariance matrix.
- 3. Obtaining the eigenvalues and eigenvectors of the covariance matrix.
- 4. Sorting the eigenvalues by decreasing order to rank the eigenvectors.
- First, we will start by loading the Wine dataset:

```
>>> import pandas as pd
>>> df_wine =
pd.read_csv('https://archive.ics.uci.edu/ml/'
...    'machine-learning-
databases/wine/wine.data',
...    header=None)
```

Principal component analysis (PCA)

- **Extracting the principal components step by step**
- Next, we will process the Wine data into separate training and test datasets—using 70 percent and 30 percent of the data, respectively—and standardize it to unit variance:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:,
0].values
>>> X_train, X_test, y_train, y_test = \
... train_test_split(X, y, test_size=0.3,
...                  stratify=y,
...                  random_state=0)
>>> # standardize the features
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Principal component analysis (PCA)

- **Extracting the principal components step by step**
- After completing the mandatory preprocessing by executing the preceding code, let's **advance to the second step: constructing the covariance matrix.**
- The symmetric $d \times d$-dimensional covariance matrix, where d is the number of dimensions in the dataset, stores the pairwise covariances between the different features. For example, the covariance between two features, x_j and x_k , on the population level can be calculated via the following equation:
- $\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^{n} \left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$
- Here, μ_j and μ_k are the sample means of features j and k, respectively. Note that the sample means are zero if we standardized the dataset. **A positive covariance between two features indicates that the features increase or decrease together, whereas a negative covariance indicates that the features vary in opposite directions.** For example, the covariance matrix of three features can then be written as follows (note that Σ stands for the Greek uppercase letter sigma, which is not to be confused with the summation symbol):
- $\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \end{bmatrix}$

Principal component analysis (PCA)

- **Extracting the principal components step by step**
- The eigenvectors of the covariance matrix represent the principal components (the directions of maximum variance), whereas the corresponding eigenvalues will define their magnitude. In the case of the Wine dataset, we would obtain 13 eigenvectors and eigenvalues from the 13 × 13-dimensional covariance matrix.
- $\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$

Principal component analysis (PCA)

- **Extracting the principal components step by step**
- Now, for our third step, let's obtain the eigenpairs of the covariance matrix.
- As you will remember from our introductory linear algebra classes, an eigenvector,
- v, satisfies the following condition:
- $\Sigma v = \lambda\, v$
- Here, λ is a scalar: the eigenvalue. Since the manual computation of eigenvectors and eigenvalues is a somewhat tedious and elaborate task,
- we will use the linalg.eig function from NumPy to obtain the eigenpairs of the Wine covariance matrix:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n%s' % eigen_vals)
```

```
Eigenvalues
[ 4.84274532      2.41602459      1.54845825
         0.96120438      0.84166161
0.6620634       0.51828472      0.34650377
         0.3131368       0.10754642
0.21357215      0.15362835      0.1808613 ]
```

Principal component analysis (PCA)

- **Extracting the principal components step by step**
- Using the numpy.cov function, we computed the covariance matrix of the standardized training dataset. Using the linalg.eig function, we performed the eigen decomposition, which yielded a vector (eigen_vals) consisting of 13 eigenvalues and the corresponding eigenvectors stored as columns in a 13 × 13-dimensional matrix (eigen_vecs).

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n%s' % eigen_vals)
```

```
Eigenvalues
[ 4.84274532       2.41602459        1.54845825
           0.96120438        0.84166161
0.6620634          0.51828472        0.34650377
           0.3131368         0.10754642
0.21357215         0.15362835        0.1808613 ]
```

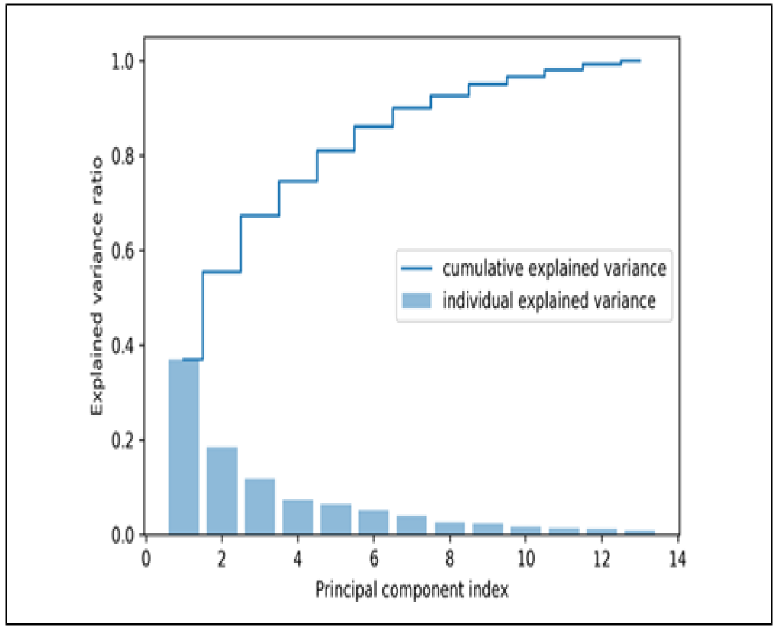
Principal component analysis (PCA)

- **Extracting the principal components step by step**
- Total and explained variance
- Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, **we only select the subset of the eigenvectors (principal components) that contains most of the information (variance).** The eigenvalues define the magnitude of the eigenvectors, so we have to sort the eigenvalues by decreasing magnitude; we are interested in the top k eigenvectors based on the values of their corresponding eigenvalues. But before we collect those k most informative eigenvectors,
- let's plot the variance explained ratios of the eigenvalues. The variance explained ratio of an eigenvalue, λj, is simply the fraction of an eigenvalue, λj, and the total sum of the eigenvalues:
- $Explained\ Variance\ ratio = \frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$

Principal component analysis (PCA)

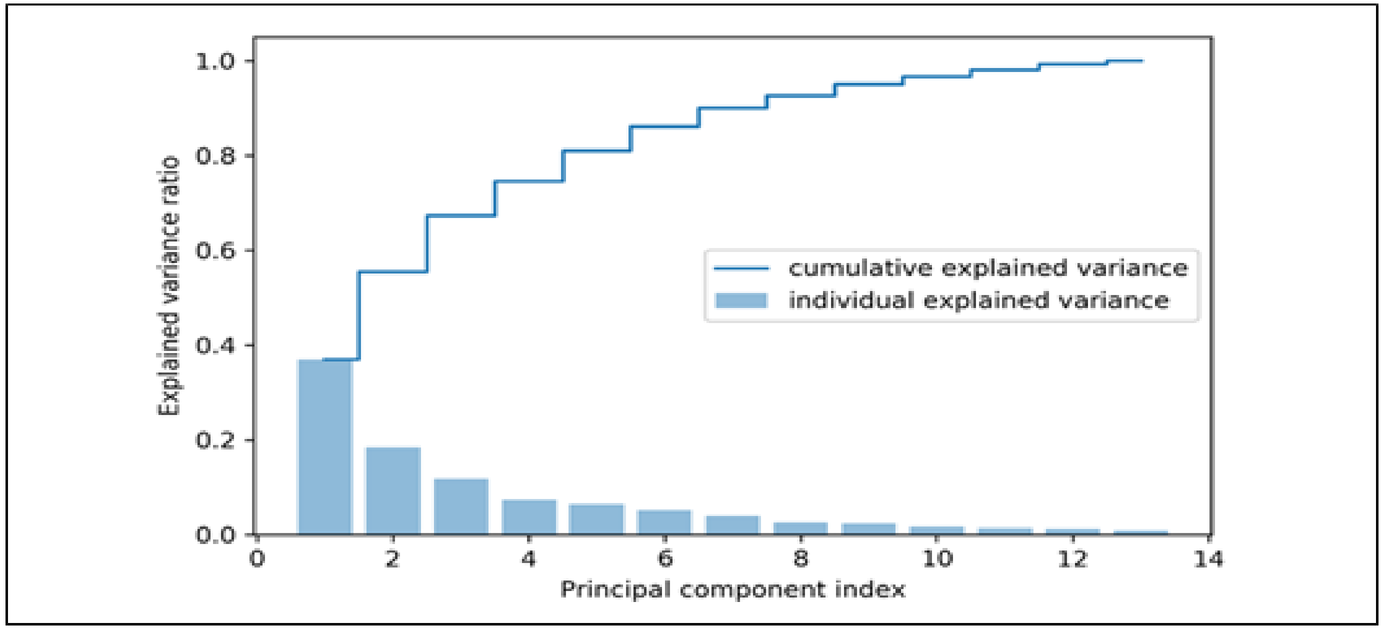
- **Extracting the principal components step by step**
- Total and explained variance
- $Explained\ Variance\ ratio = \frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$
- Using the NumPy cumsum function, we can then calculate the cumulative sum of explained variances, which we will then plot via Matplotlib's step function:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...            sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)
>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...            label='Individual explained variance')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...            label='Cumulative explained variance')
>>> plt.ylabel('Explained variance ratio')
>>> plt.xlabel('Principal component index')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```



Principal component analysis (PCA)

- **Extracting the principal components step by step**
- Total and explained variance
- $Explained\ Variance\ ratio = \frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$
- The resulting plot indicates that the first principal component alone accounts for approximately 40 percent of the variance.
- Also, we can see that the first two principal components combined explain almost 60 percent of the variance in the dataset:



Principal component analysis (PCA)

- **Feature transformation**
- Now that we have successfully decomposed the covariance matrix into eigenpairs, let's proceed with the last three steps to transform the Wine dataset onto the new principal component axes. The remaining steps we are going to tackle in this section are the following:
- 5. Select k eigenvectors, which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
- 6. Construct a projection matrix, W, from the "top" k eigenvectors.
- 7. Transform the d-dimensional input dataset, X, using the projection matrix, W, to obtain the new k-dimensional feature subspace, Or, in less technical terms, we will sort the eigenpairs by descending order of the eigenvalues, construct a projection matrix from the selected eigenvectors, and use the projection matrix to transform the data onto the lower-dimensional subspace.

Principal component analysis (PCA)

- **Feature transformation**
- Now that we have successfully decomposed the covariance matrix into eigenpairs, let's proceed with the last three steps to transform the Wine dataset onto the new principal component axes.

We start by sorting the eigenpairs by decreasing order of the eigenvalues:

```
>>> # Make a list of (eigenvalue, eigenvector) tuples
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
...             for i in range(len(eigen_vals))]
>>> # Sort the (eigenvalue, eigenvector) tuples from high to low
>>> eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

Principal component analysis (PCA)

- **Feature transformation**
- Next, we collect the two eigenvectors that correspond to the two largest eigenvalues, to capture about 60 percent of the variance in this dataset. Note that two eigenvectors have been chosen for the purpose of illustration, since we are going to plot the data via a two-dimensional scatter plot .
- **In practice, the number of principal components has to be determined by a tradeoff between computational efficiency and the performance of the classifier:**

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n', w)
Matrix W:
[[-0.13724218   0.50303478]
 [ 0.24724326   0.16487119]
 [-0.02545159   0.24456476]
 [ 0.20694508  -0.11352904]
 [-0.15436582   0.28974518]
 [-0.39376952   0.05080104]
 [-0.41735106  -0.02287338]
 [ 0.30572896   0.09048885]
 [-0.30668347   0.00835233]
 [ 0.07554066   0.54977581]
 [-0.32613263  -0.20716433]
 [-0.36861022  -0.24902536]
 [-0.29669651   0.38022942]]
```

Principal component analysis (PCA)

- **Feature transformation**
- **By executing the preceding code, we have created a 13 × 2-dimensional projection matrix, W, from the top two eigenvectors.**

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n', w)
Matrix W:
[[-0.13724218     0.50303478]
 [ 0.24724326     0.16487119]
 [-0.02545159     0.24456476]
 [ 0.20694508    -0.11352904]
 [-0.15436582     0.28974518]
 [-0.39376952     0.05080104]
 [-0.41735106    -0.02287338]
 [ 0.30572896     0.09048885]
 [-0.30668347     0.00835233]
 [ 0.07554066     0.54977581]
 [-0.32613263    -0.20716433]
 [-0.36861022    -0.24902536]
 [-0.29669651     0.38022942]]
```

Principal component analysis (PCA)

- **Feature transformation**
- Using the projection matrix, we can now transform an example, x (represented as a 13-dimensional row vector), onto the PCA subspace (the principal components one and two) obtaining $\boldsymbol{x}'$, now a two-dimensional example vector consisting of two new features:
- $\boldsymbol{x}'=\boldsymbol{x}\boldsymbol{W}$

```
>>> X_train_std[0].dot(w)
array([ 2.38299011,    0.45458499])
```

Principal component analysis (PCA)

- **Feature transformation**
- Similarly, we can transform the entire 124 × 13-dimensional training dataset onto the two principal components by calculating the matrix dot product:
- $\boldsymbol{X'} = \mathbf{X}\boldsymbol{W}$

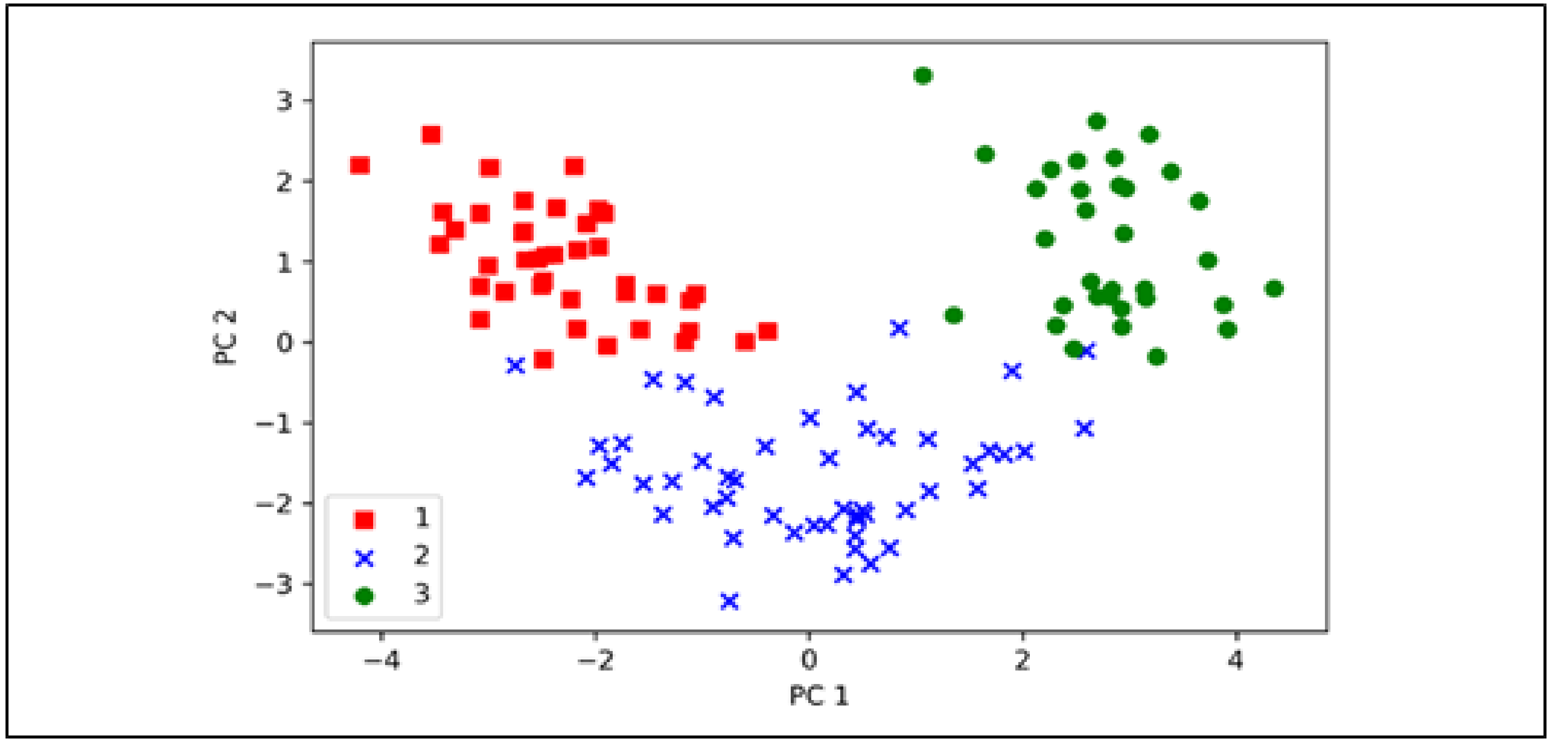
```
>>> X_train_pca =
X_train_std.dot(w)
```

Lastly, let's visualize the transformed Wine training dataset, now stored as an 124 × 2-dimensional matrix, in a two-dimensional scatterplot:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...             plt.scatter(X_train_pca[y_train==l, 0],
...             X_train_pca[y_train==l, 1],
...             c=c, label=l, marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

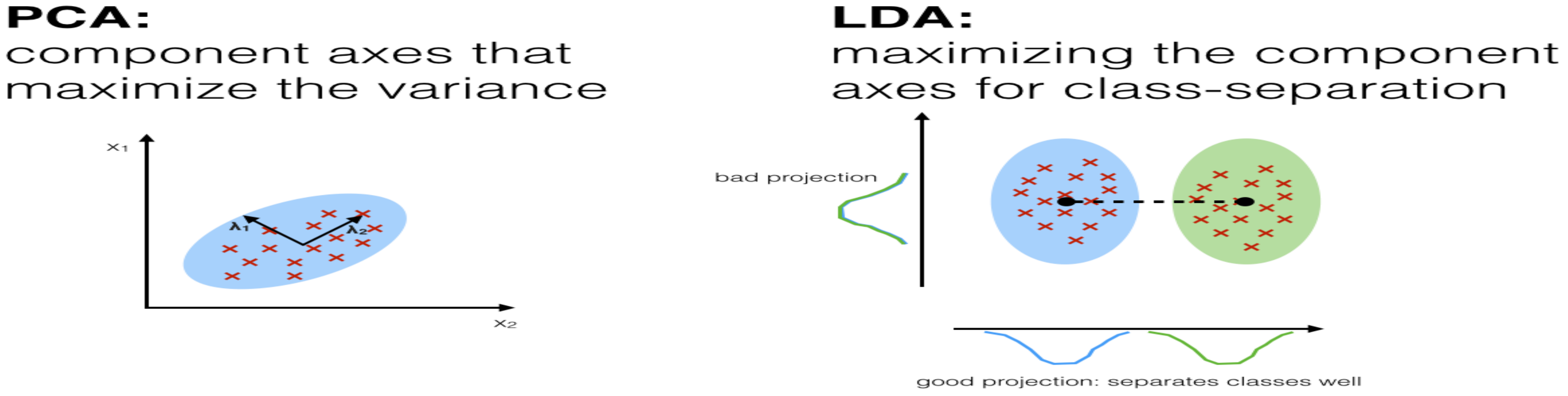
Principal component analysis (PCA)

- **Feature transformation**
- As we can see in the resulting plot, the data is more spread along the x-axis—the first principal component—than the second principal component (y-axis), which is consistent with the explained variance ratio plot. However, we can tell that a linear classifier will likely be able to separate the classes well:



Linear Discriminant Analysis(LDA)

- **Supervised data compression via linear discriminant analysis**
- LDA can be used as a technique for feature extraction to increase the computational efficiency and reduce the degree of overfitting due to the curse of dimensionality in non-regularized models.
- The general concept behind LDA is very similar to PCA, but whereas PCA attempts to find the orthogonal component axes of maximum variance in a dataset,
- the goal in LDA is to find the feature subspace that optimizes class separability.
- In the following sections, we will discuss the similarities between LDA and PCA in more detail and walk through the LDA approach step by step.

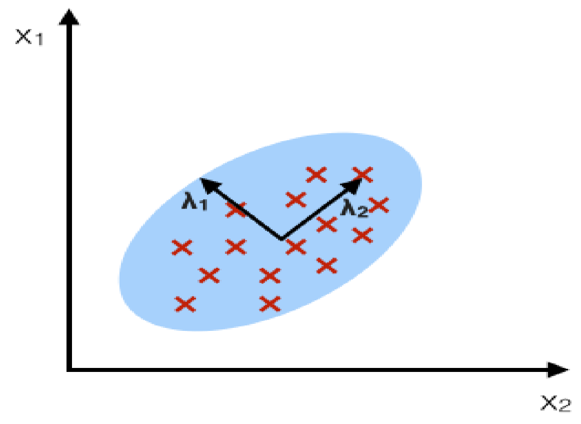

PCA:
component axes that maximize the variance
x1
x2
λ1
λ2
LDA:
maximizing the component axes for class-separation
bad projection
good projection: separates classes well

Linear Discriminant Analysis(LDA)

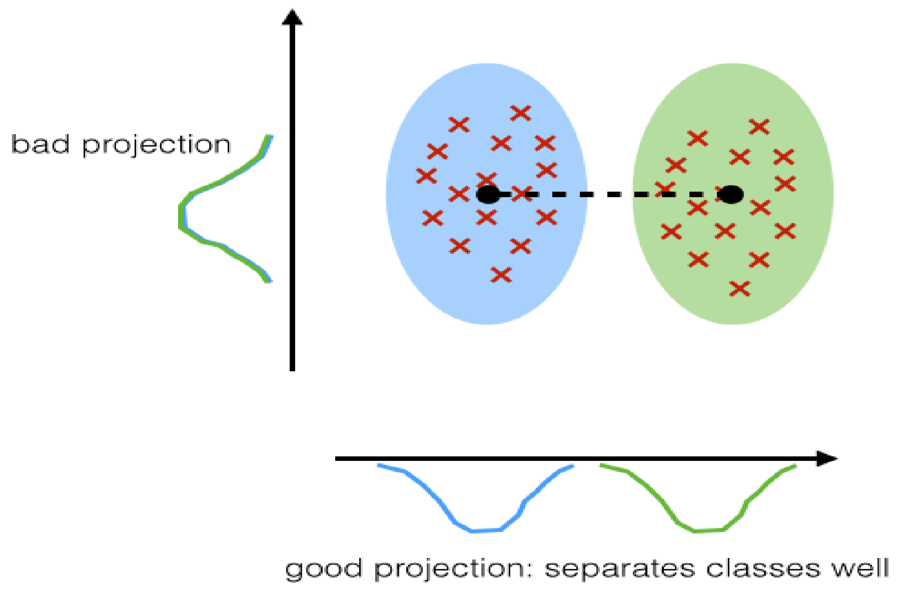
- **Principal component analysais versus Linear discriminant analysais**

- Both PCA and LDA are linear transformation techniques that can be used to reduce the number of dimensions in a dataset; the former is an unsupervised algorithm, whereas the latter is supervised. Thus, we might think that LDA is a superior feature extraction technique for classification tasks compared to PCA. However, A.M. Martinez reported that preprocessing via PCA tends to result in better classification results in an image recognition task in certain cases, for instance, if each class consists of only a small number of examples (PCA Versus LDA, A. M. Martinez and A. C. Kak, IEEE Transactions on Pattern Analysis and Machine Intelligence, 23(2): 228-233, 2001).

PCA:
component axes that maximize the variance

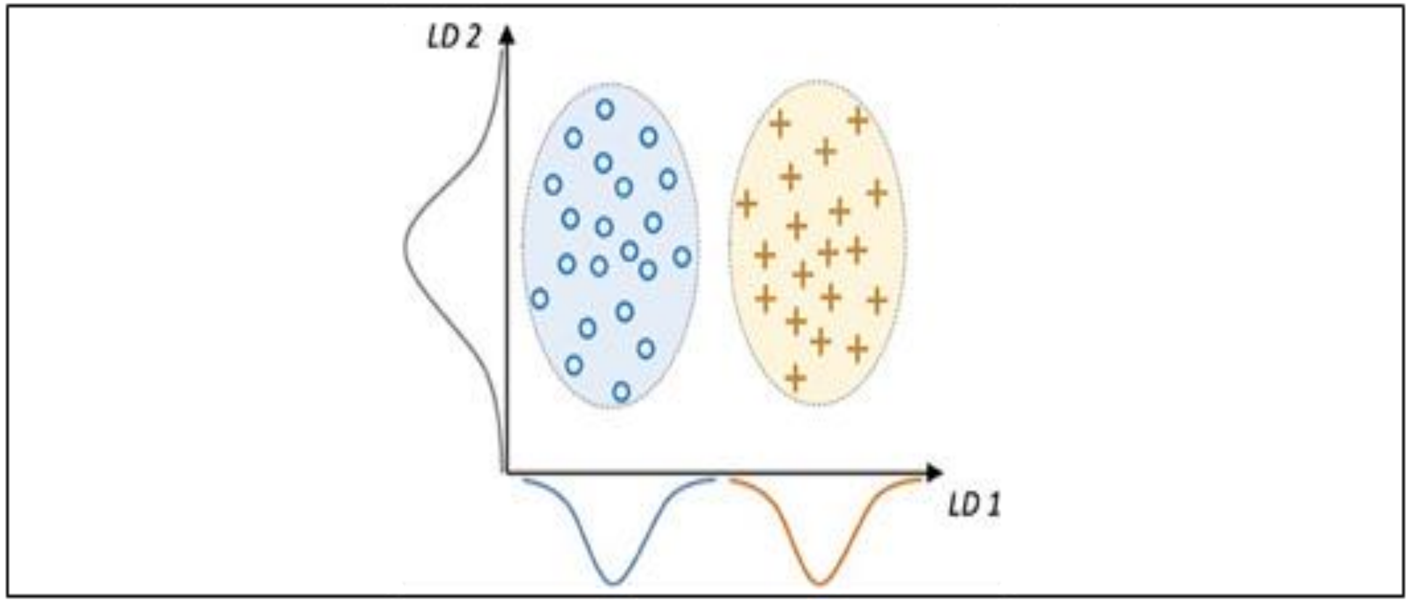


LDA:
maximizing the component axes for class-separation



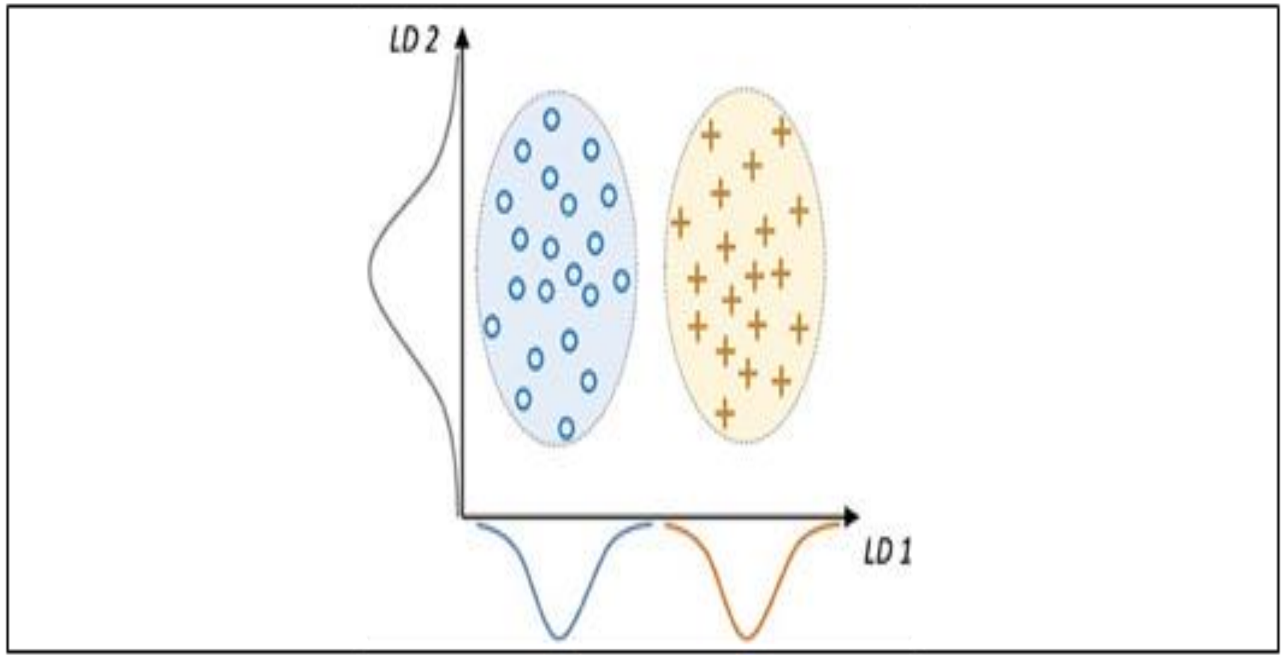
Linear Discriminant Analysis(LDA)

- **Principal component analysis versus linear discriminant analysis**
- The following figure summarizes the concept of LDA for a two-class problem.
- Examples from class 1 are shown as circles, and examples from class 2 are shown as crosses:
- A linear discriminant, as shown on the x-axis (LD 1), would separate the two normal distributed classes well. Although the exemplary linear discriminant shown on the y-axis (LD 2) captures a lot of the variance in the dataset, it would fail as a good linear discriminant since it does not capture any of the class-discriminatory information.



Linear Discriminant Analysis(LDA)

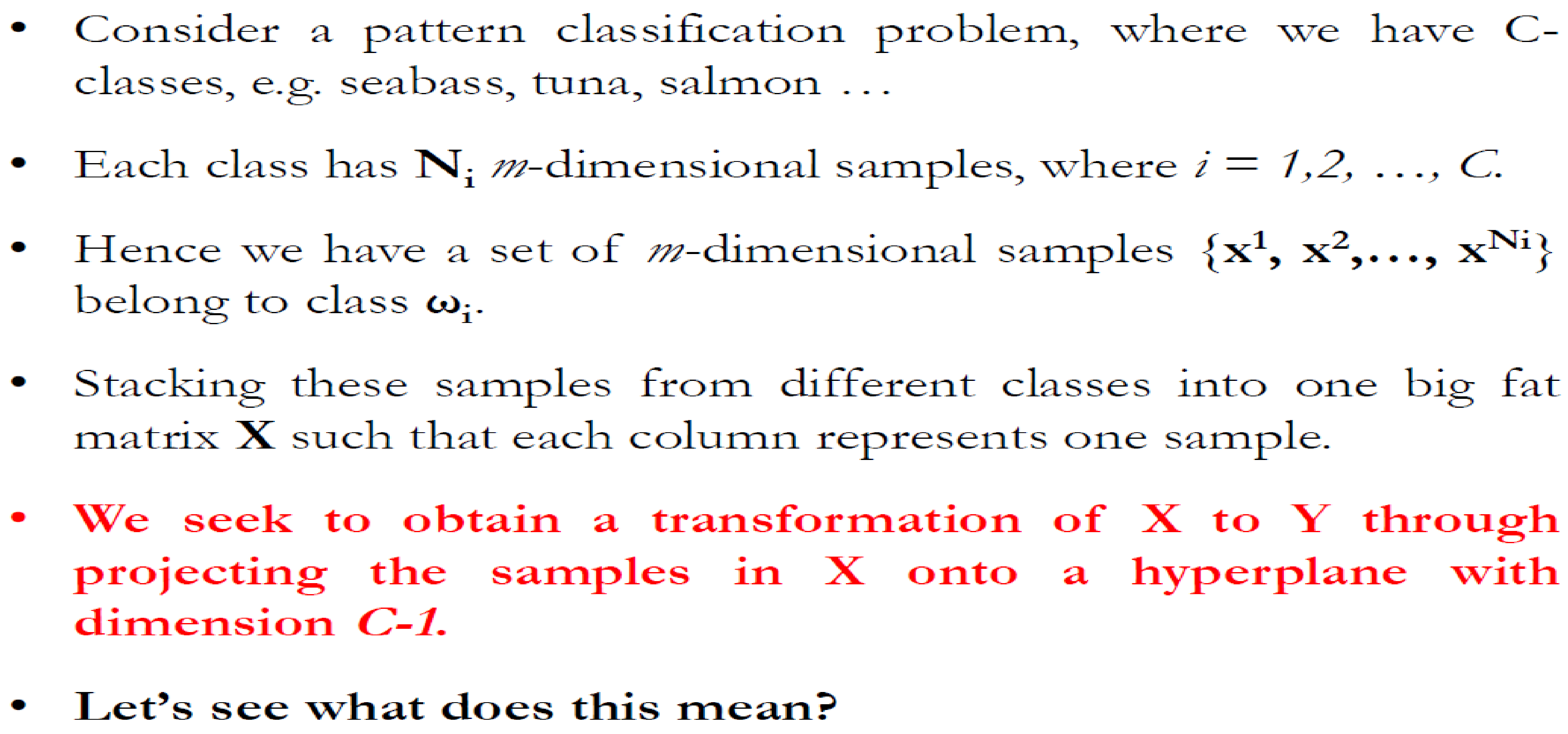
- **Principal component analysis versus linear discriminant analysis**
- One assumption in LDA is that the data is normally distributed. Also, we assume that the classes have identical covariance matrices and that the training examples are statistically independent of each other.
- However, even if one, or more, of those assumptions is (slightly) violated, LDA for dimensionality reduction can still work reasonably well (Pattern Classification 2nd Edition, R. O. Duda, P. E. Hart, and D. G. Stork, New York, 2001).



Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



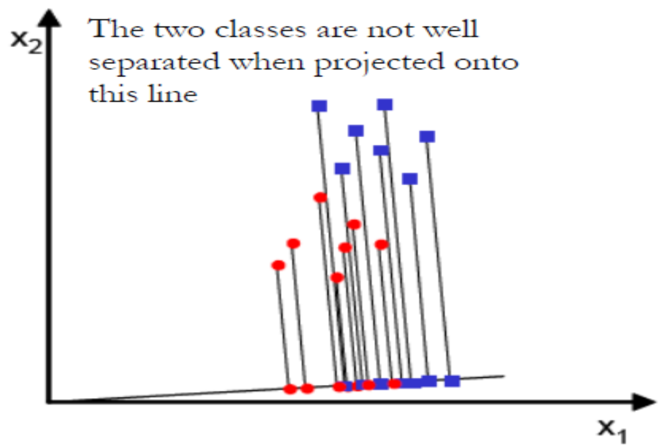
- Consider a pattern classification problem, where we have C-classes, e.g. seabass, tuna, salmon …
- Each class has $\mathbf{N_i}$ *m*-dimensional samples, where $i = 1,2, \ldots, C$.
- Hence we have a set of *m*-dimensional samples $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^{Ni}\}$ belong to class $\boldsymbol{\omega}_\mathbf{i}$.
- Stacking these samples from different classes into one big fat matrix **X** such that each column represents one sample.
- **We seek to obtain a transformation of X to Y through projecting the samples in X onto a hyperplane with dimension *C-1*.**
- **Let's see what does this mean?**

Linear Discriminant Analysis(LDA)

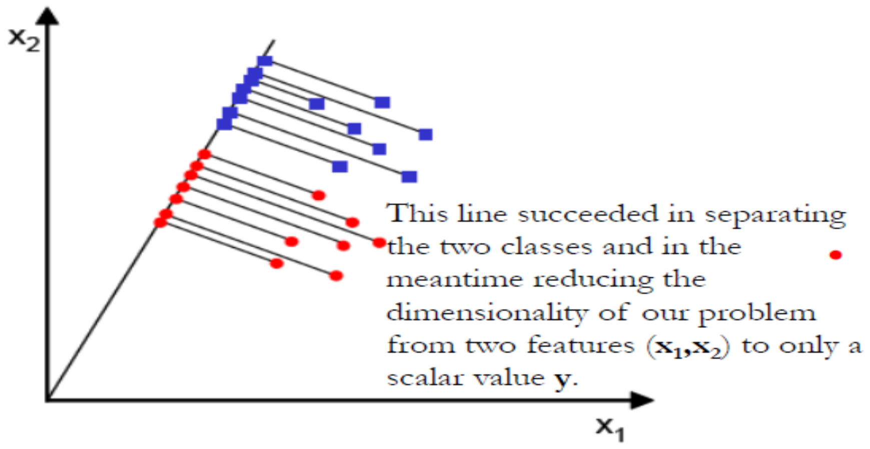
- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf

LDA … Two Classes



x_2
The two classes are not well separated when projected onto this line
x_1



x_2
This line succeeded in separating the two classes and in the meantime reducing the dimensionality of our problem from two features (x_1,x_2) to only a scalar value **y**.
x_1

- Assume we have *m*-dimensional samples $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\}$, N_1 of which belong to $\boldsymbol{\omega}_1$ and N_2 belong to $\boldsymbol{\omega}_2$.
- We seek to obtain a scalar **y** by projecting the samples **x** onto a line (C-1 space, C = 2).

$$y = w^T x \quad where \quad x = \begin{bmatrix} x_1 \\ . \\ . \\ x_m \end{bmatrix} \quad and \quad w = \begin{bmatrix} w_1 \\ . \\ . \\ w_m \end{bmatrix}$$

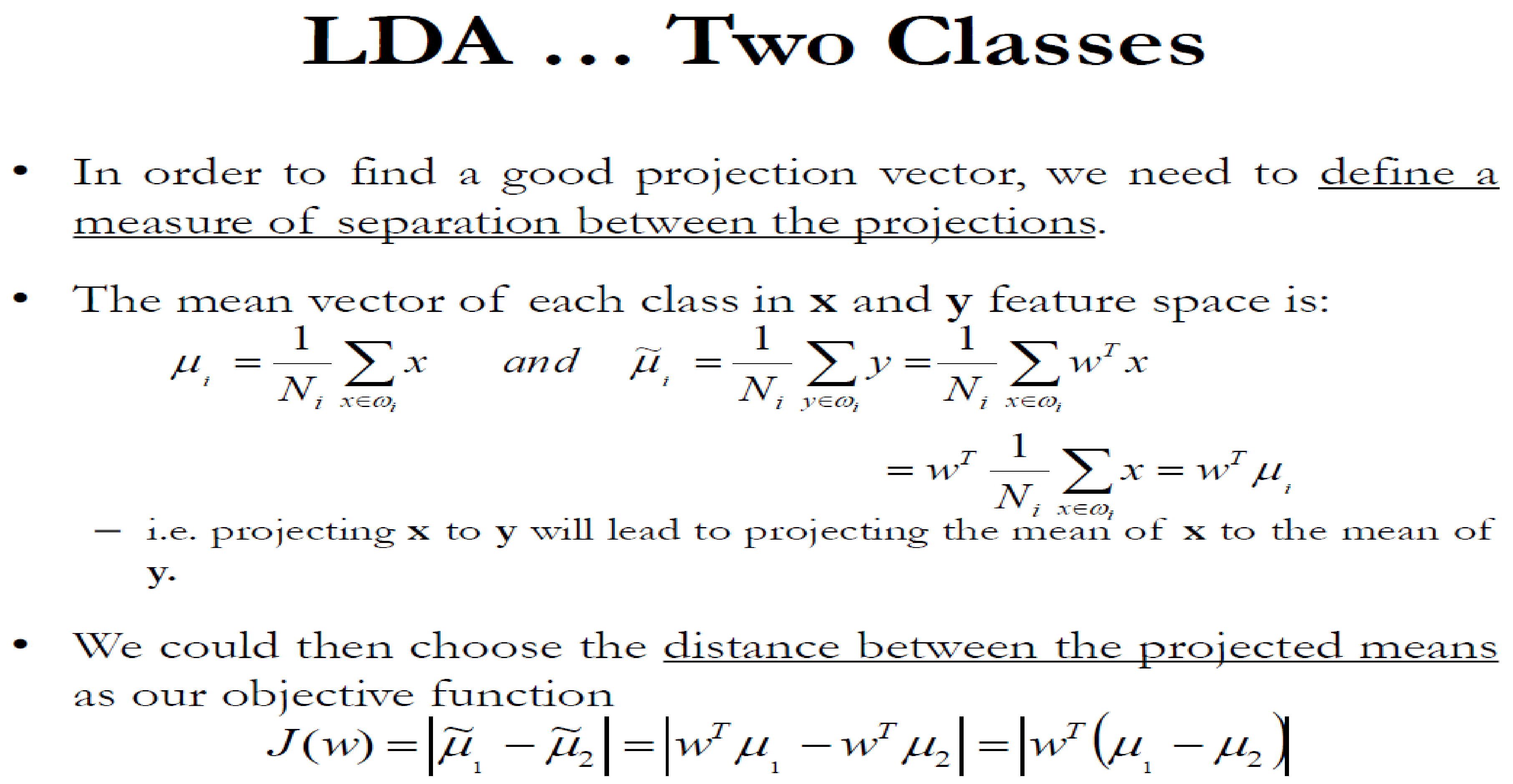
  – where **w** is the projection vectors used to project **x** to **y**.

- **Of all the possible lines we would like to select the one that maximizes the separability of the scalars.**

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes

- In order to find a good projection vector, we need to define a measure of separation between the projections.
- The mean vector of each class in **x** and **y** feature space is:

$$\mu_i = \frac{1}{N_i}\sum_{x\in\omega_i} x \qquad and \qquad \tilde{\mu}_i = \frac{1}{N_i}\sum_{y\in\omega_i} y = \frac{1}{N_i}\sum_{x\in\omega_i} w^T x$$

$$= w^T \frac{1}{N_i}\sum_{x\in\omega_i} x = w^T \mu_i$$

  – i.e. projecting **x** to **y** will lead to projecting the mean of **x** to the mean of **y**.

- We could then choose the distance between the projected means as our objective function

$$J(w) = \left|\tilde{\mu}_1 - \tilde{\mu}_2\right| = \left|w^T\mu_1 - w^T\mu_2\right| = \left|w^T(\mu_1 - \mu_2)\right|$$

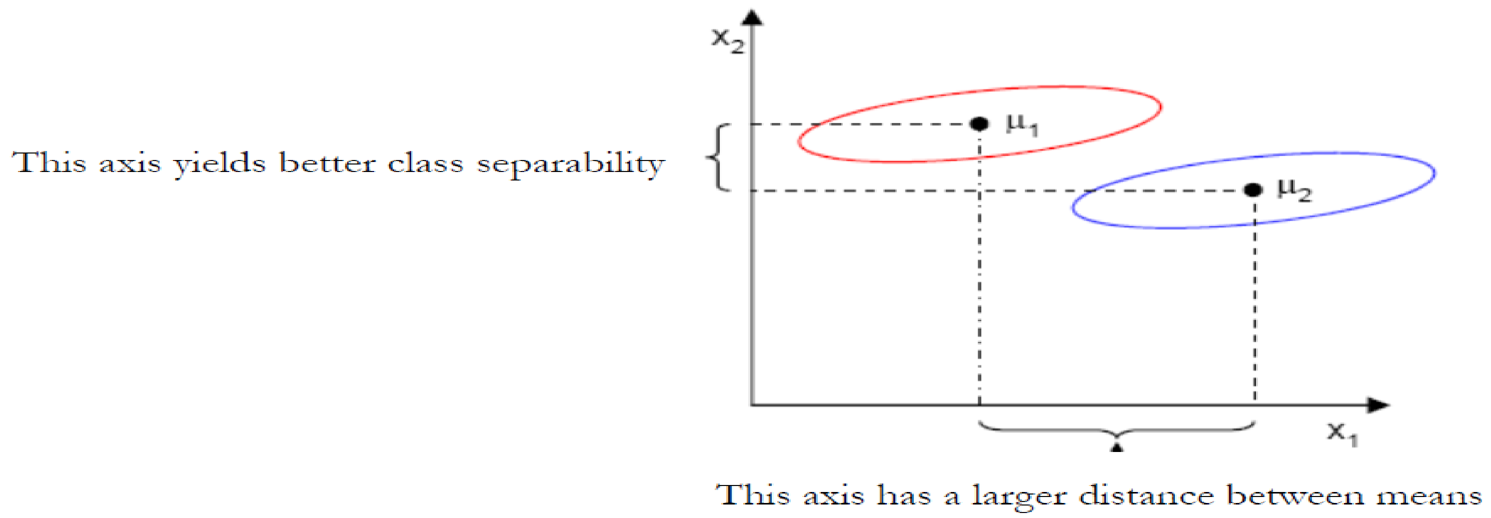
Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf

LDA … Two Classes

- However, the distance between the projected means is <u>not a very good measure</u> since it does not take into account the standard deviation within the classes.

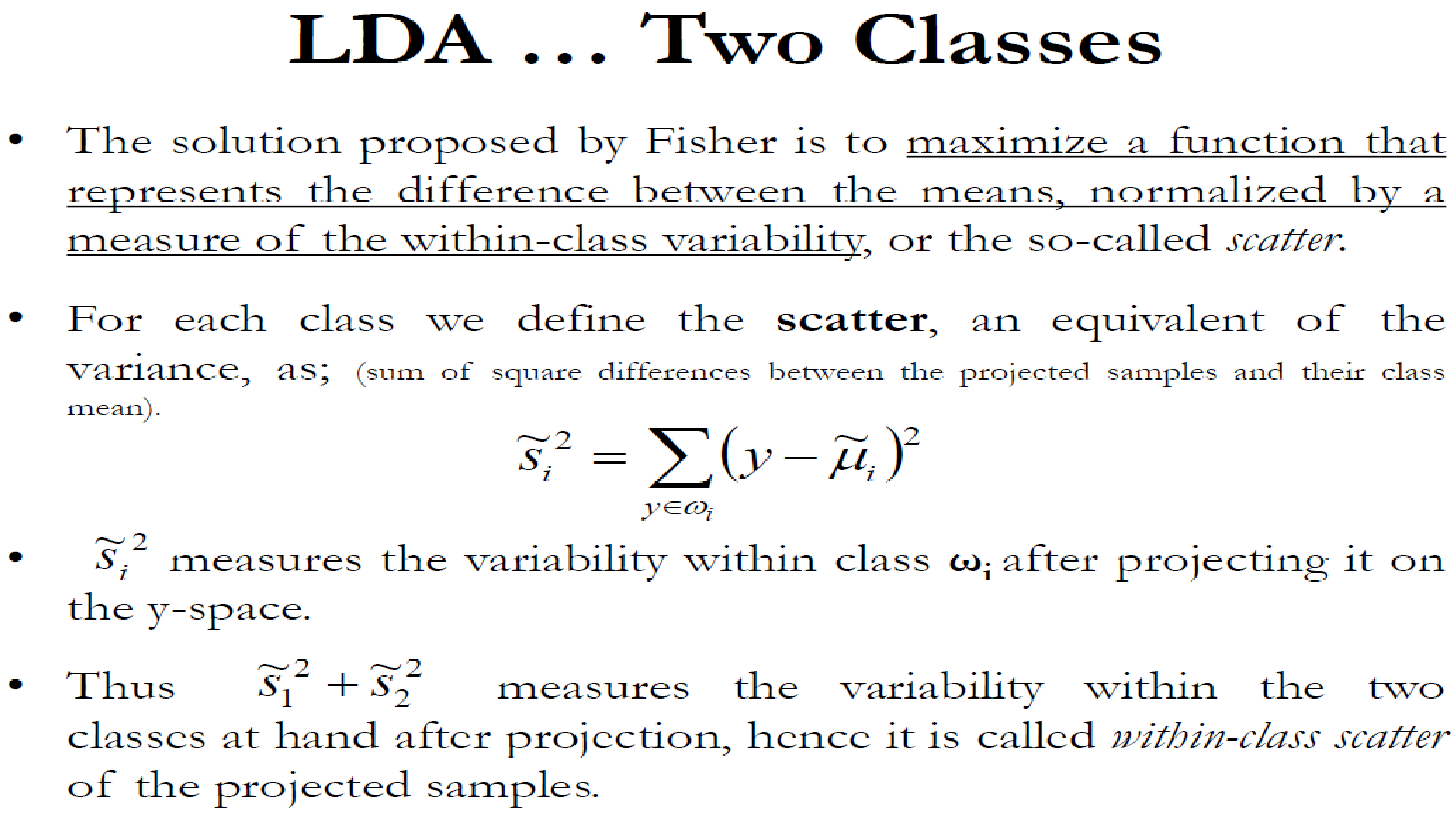


x_2
μ_1
μ_2
This axis yields better class separability
x_1
This axis has a larger distance between means

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes

- The solution proposed by Fisher is to maximize a function that represents the difference between the means, normalized by a measure of the within-class variability, or the so-called *scatter*.
- For each class we define the **scatter**, an equivalent of the variance, as; (sum of square differences between the projected samples and their class mean).

$$\tilde{s}_i^2 = \sum_{y \in \omega_i} (y - \tilde{\mu}_i)^2$$

- $\tilde{s}_i^2$ measures the variability within class $\boldsymbol{\omega}_i$ after projecting it on the y-space.
- Thus $\tilde{s}_1^2 + \tilde{s}_2^2$ measures the variability within the two classes at hand after projection, hence it is called *within-class scatter* of the projected samples.

Linear Discriminant Analysis(LDA)

- LDA

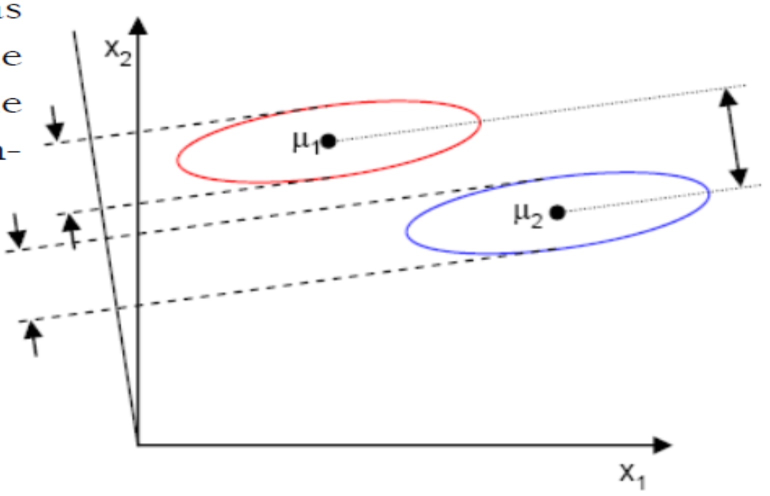
http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf

LDA … Two Classes

- The Fisher linear discriminant is defined as the linear function $\mathbf{w}^T\mathbf{x}$ that maximizes the criterion function: (the distance between the projected means normalized by the within-class scatter of the projected samples.

$$J(w) = \frac{\left|\tilde{\mu}_1 - \tilde{\mu}_2\right|^2}{\tilde{s}_1^2 + \tilde{s}_2^2}$$

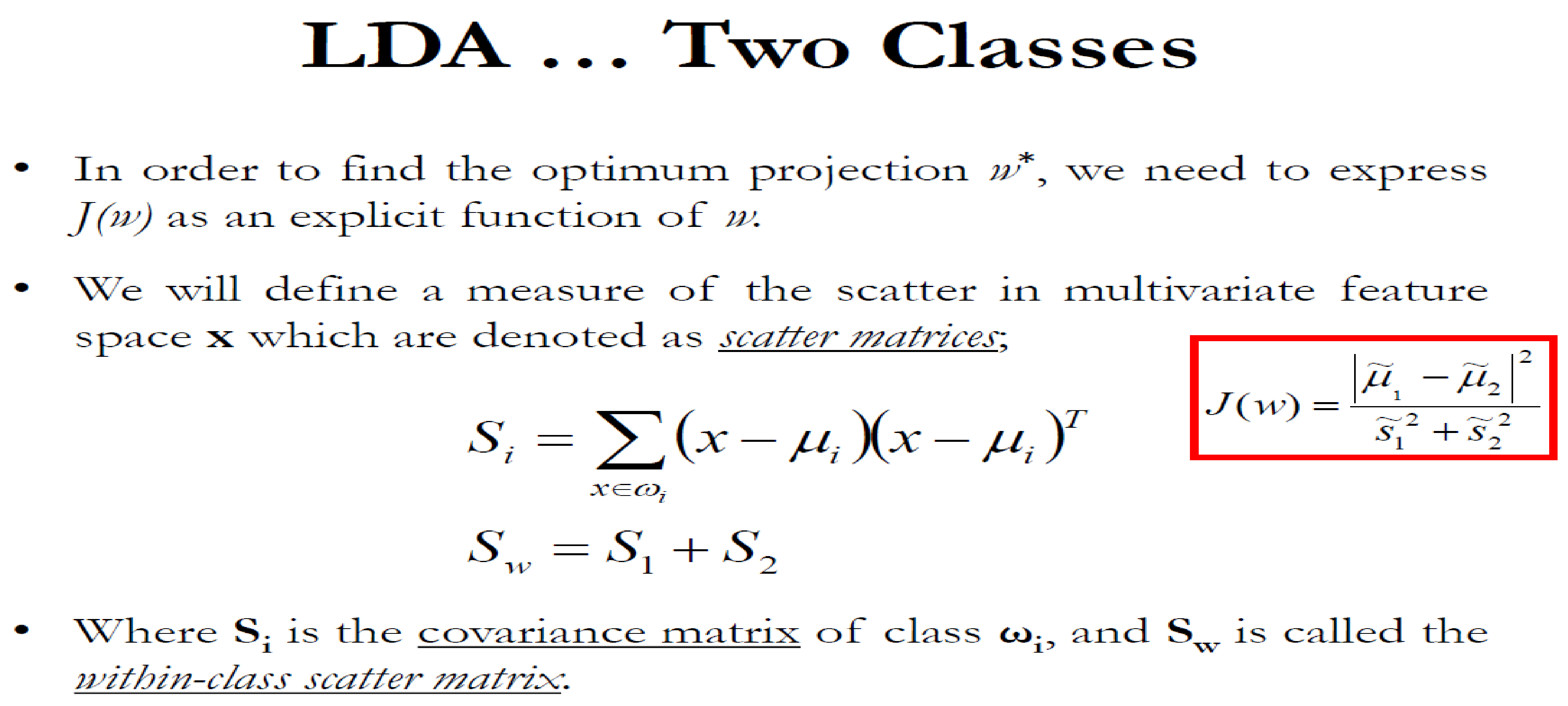
- Therefore, we will be looking for a projection where examples from the same class are projected very close to each other and, at the same time, the projected means are as farther apart as possible



Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes

- In order to find the optimum projection w^*, we need to express $J(w)$ as an explicit function of w.
- We will define a measure of the scatter in multivariate feature space **x** which are denoted as *scatter matrices*;

$$S_i = \sum_{x\in\omega_i} (x-\mu_i)(x-\mu_i)^T$$

$$S_w = S_1 + S_2$$

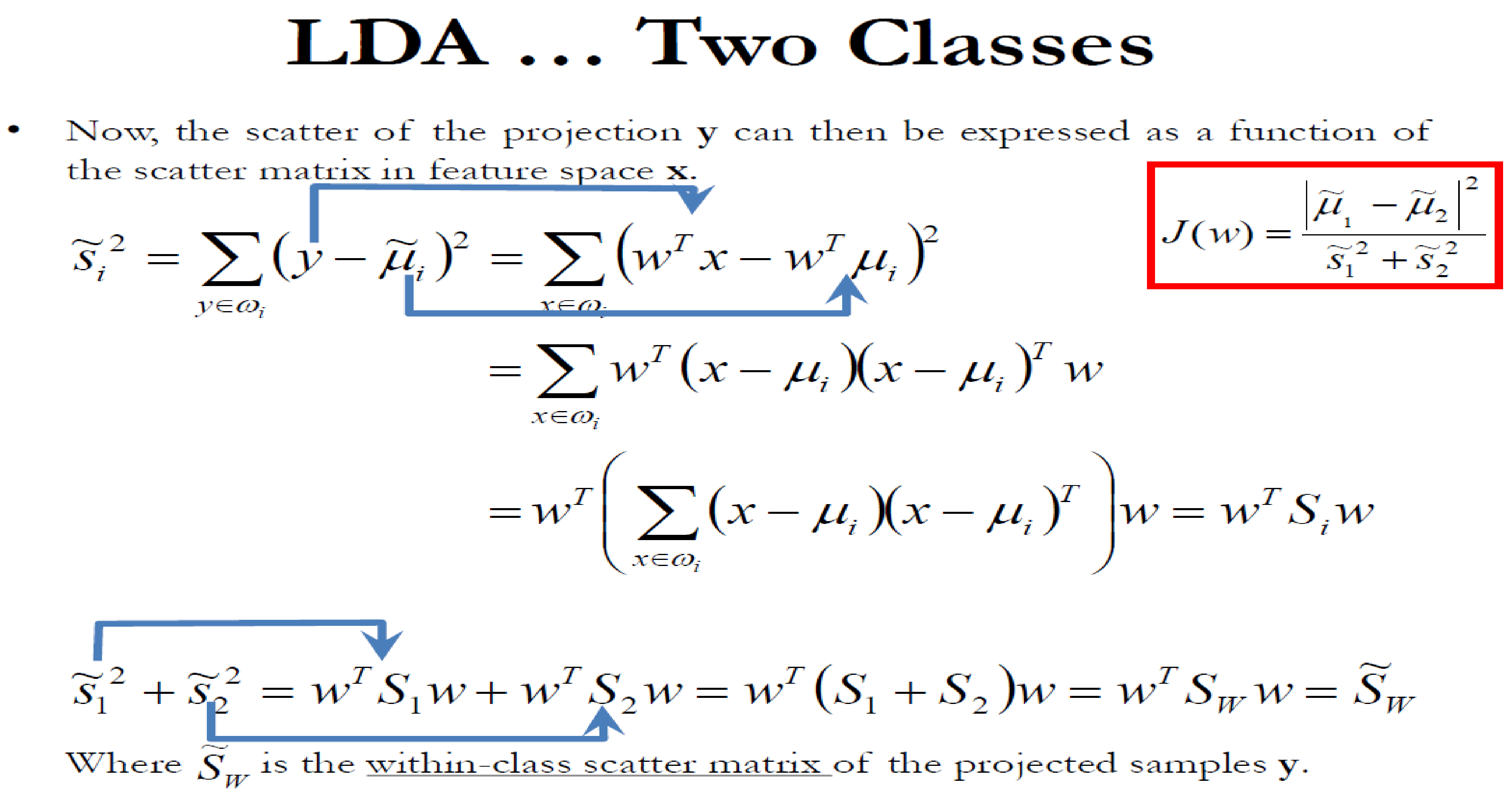
$$\boxed{J(w) = \frac{\left|\tilde{\mu}_1 - \tilde{\mu}_2\right|^2}{\tilde{s}_1^2 + \tilde{s}_2^2}}$$

- Where $\mathbf{S_i}$ is the covariance matrix of class $\boldsymbol{\omega_i}$, and $\mathbf{S_w}$ is called the *within-class scatter matrix*.

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes

- Now, the scatter of the projection **y** can then be expressed as a function of the scatter matrix in feature space **x**.

$$\tilde{s}_i^2 = \sum_{y\in\omega_i}(y-\tilde{\mu}_i)^2 = \sum_{x\in\omega_i}\left(w^T x - w^T\mu_i\right)^2$$

$$= \sum_{x\in\omega_i} w^T(x-\mu_i)(x-\mu_i)^T w$$

$$= w^T\left(\sum_{x\in\omega_i}(x-\mu_i)(x-\mu_i)^T\right)w = w^T S_i w$$

$$\boxed{J(w) = \frac{\left|\tilde{\mu}_1 - \tilde{\mu}_2\right|^2}{\tilde{s}_1^2 + \tilde{s}_2^2}}$$

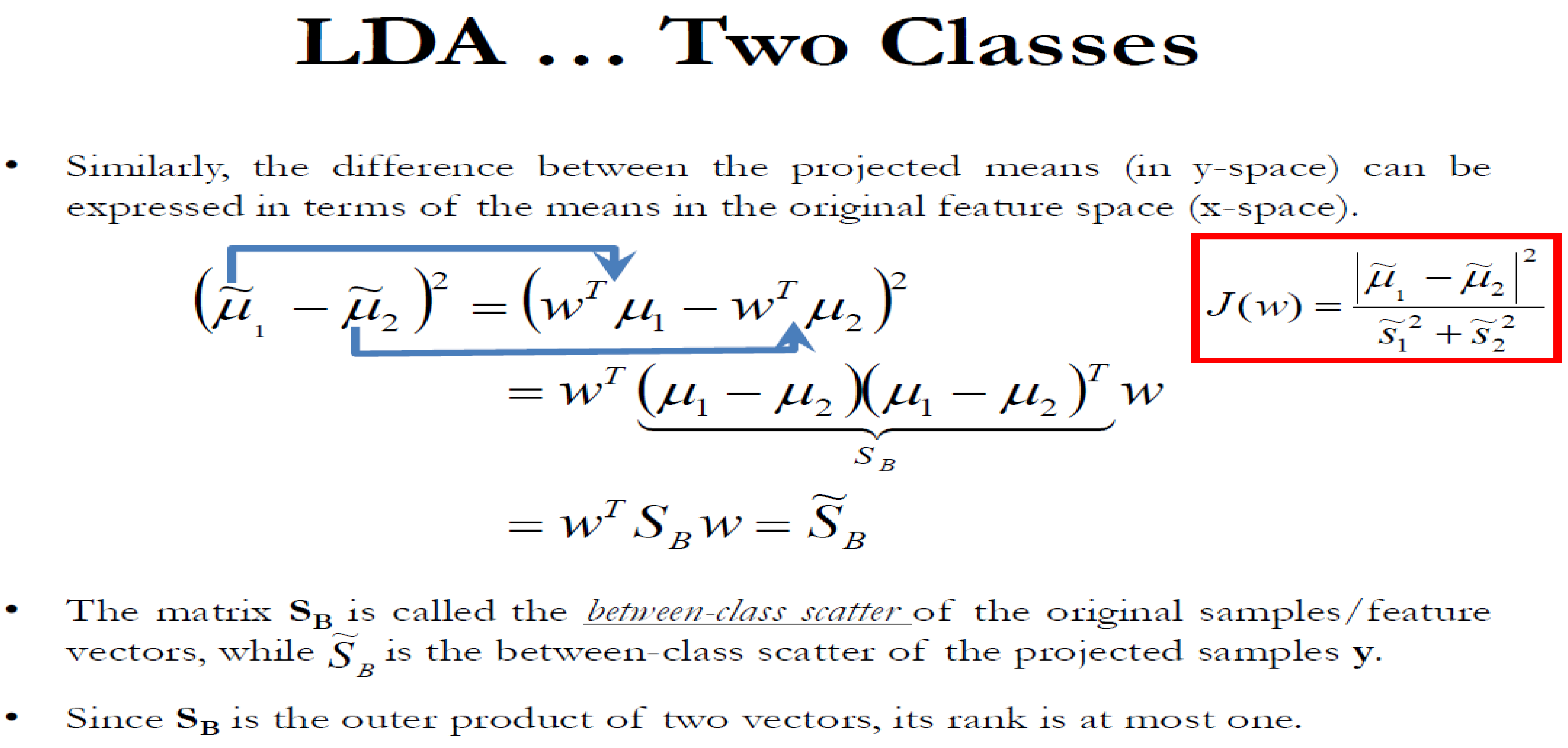
$$\tilde{s}_1^2 + \tilde{s}_2^2 = w^T S_1 w + w^T S_2 w = w^T(S_1 + S_2)w = w^T S_W w = \tilde{S}_W$$

Where $\tilde{S}_W$ is the within-class scatter matrix of the projected samples **y**.

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes

- Similarly, the difference between the projected means (in y-space) can be expressed in terms of the means in the original feature space (x-space).

$$\boxed{J(w)=\frac{\left|\tilde{\mu}_1-\tilde{\mu}_2\right|^2}{\tilde{s}_1^{\,2}+\tilde{s}_2^{\,2}}}$$

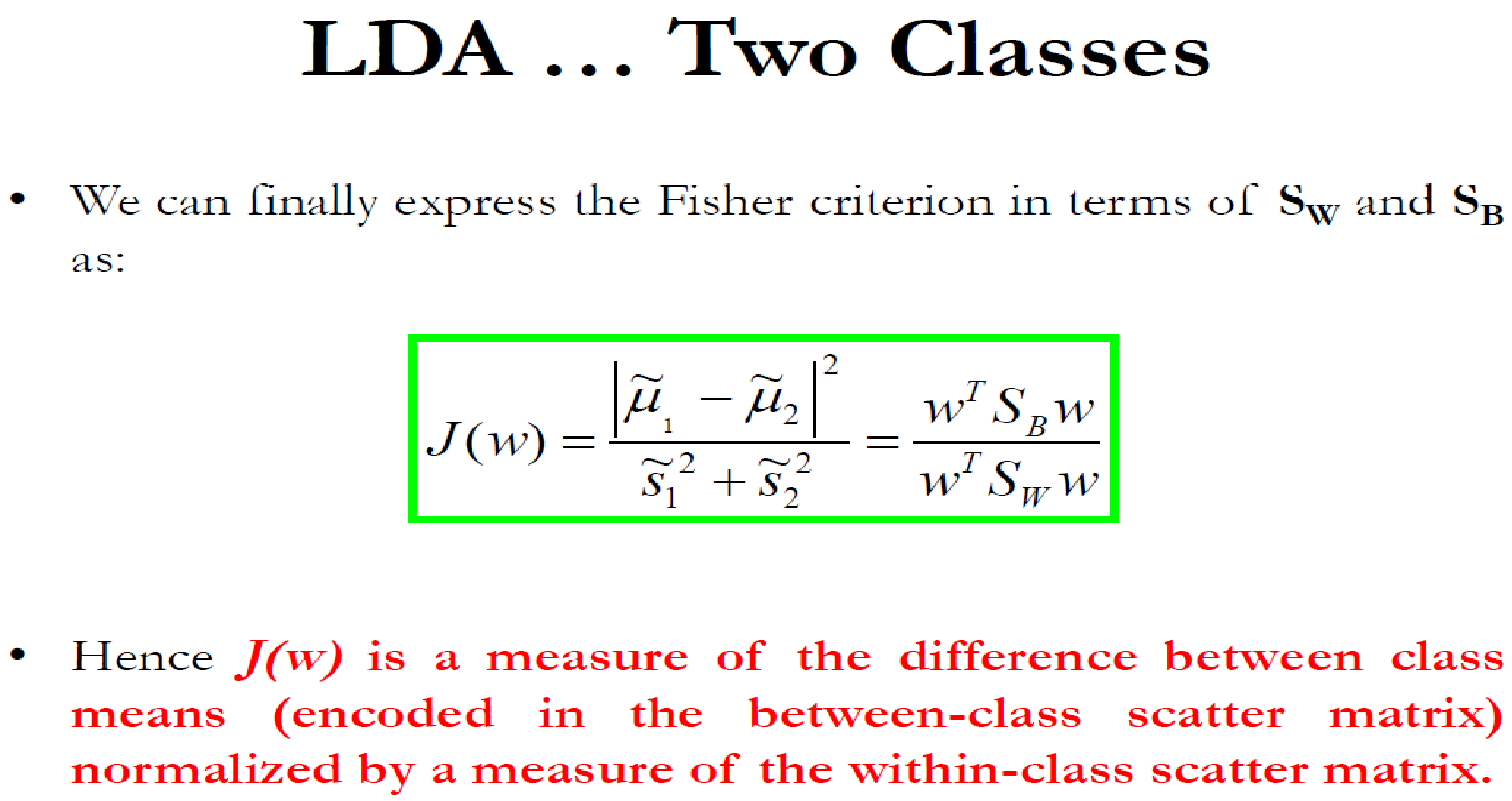
$$
\begin{aligned}
(\tilde{\mu}_1-\tilde{\mu}_2)^2 &= \left(w^T\mu_1-w^T\mu_2\right)^2\\
&= w^T\underbrace{(\mu_1-\mu_2)(\mu_1-\mu_2)^T}_{S_B}w\\
&= w^T S_B w=\tilde{S}_B
\end{aligned}
$$

- The matrix $\mathbf{S_B}$ is called the *between-class scatter* of the original samples/feature vectors, while $\tilde{S}_B$ is the between-class scatter of the projected samples **y**.
- Since $\mathbf{S_B}$ is the outer product of two vectors, its rank is at most one.

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes

- We can finally express the Fisher criterion in terms of $\mathbf{S_W}$ and $\mathbf{S_B}$ as:

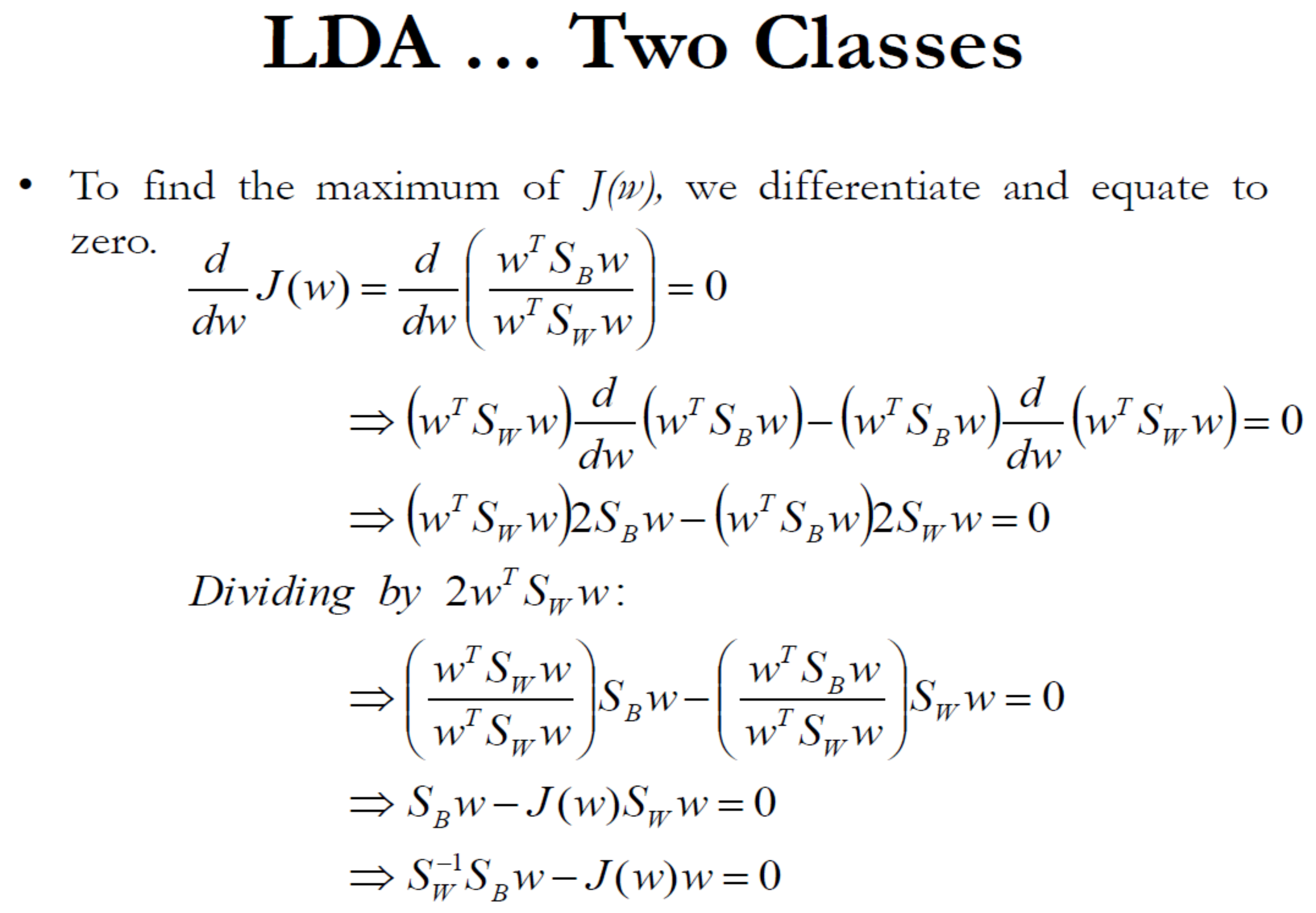
$$\boxed{J(w) = \frac{\left|\tilde{\mu}_1 - \tilde{\mu}_2\right|^2}{\tilde{s}_1^2 + \tilde{s}_2^2} = \frac{w^T S_B w}{w^T S_W w}}$$

- Hence ***J(w)*** **is a measure of the difference between class means (encoded in the between-class scatter matrix) normalized by a measure of the within-class scatter matrix.**

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes

- To find the maximum of *J(w)*, we differentiate and equate to zero.

$$\frac{d}{dw}J(w) = \frac{d}{dw}\left(\frac{w^T S_B w}{w^T S_W w}\right) = 0$$

$$\Rightarrow \left(w^T S_W w\right)\frac{d}{dw}\left(w^T S_B w\right) - \left(w^T S_B w\right)\frac{d}{dw}\left(w^T S_W w\right) = 0$$

$$\Rightarrow \left(w^T S_W w\right)2S_B w - \left(w^T S_B w\right)2S_W w = 0$$

$$Dividing\ \ by\ \ 2w^T S_W w:$$

$$\Rightarrow \left(\frac{w^T S_W w}{w^T S_W w}\right)S_B w - \left(\frac{w^T S_B w}{w^T S_W w}\right)S_W w = 0$$

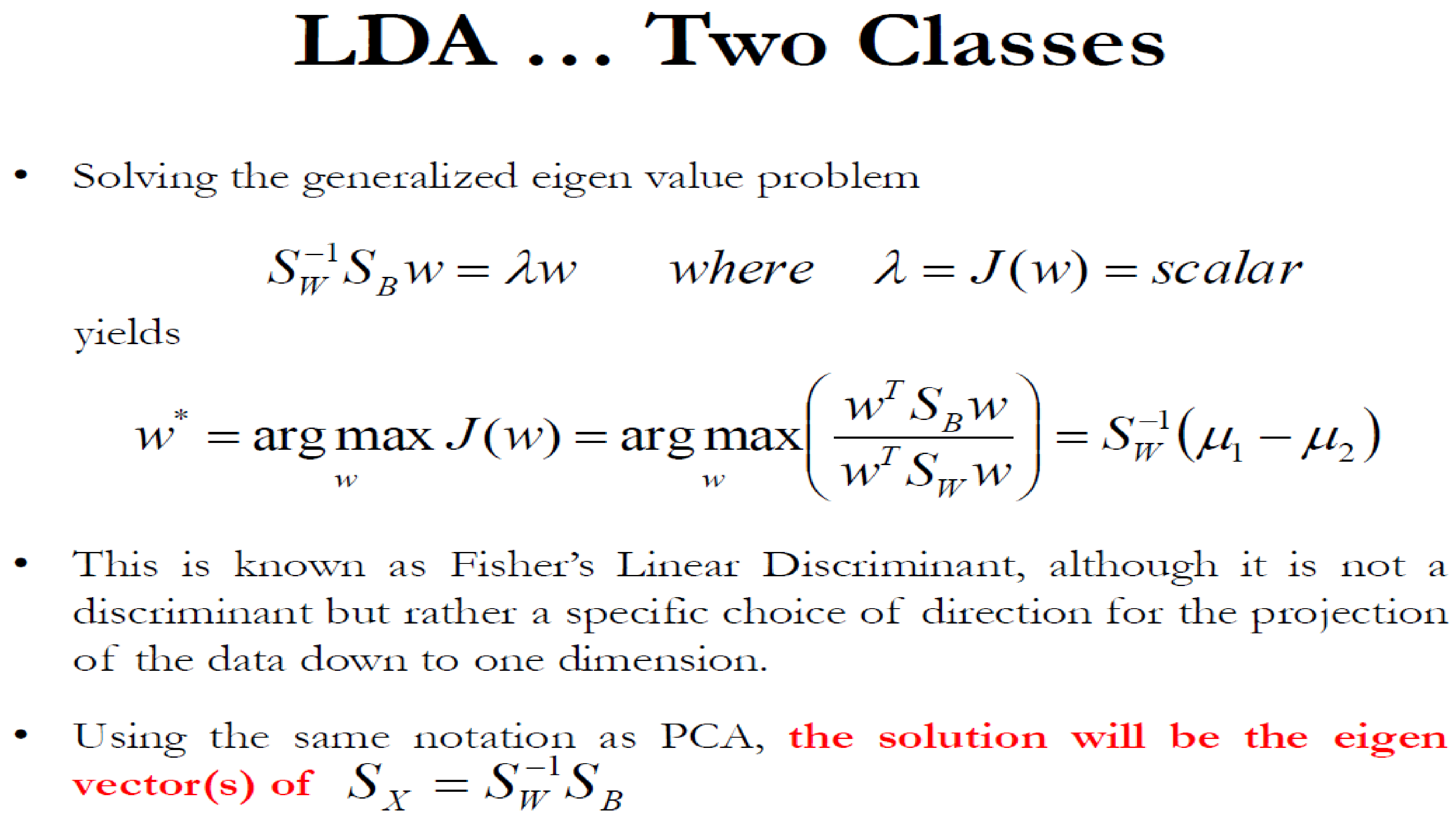
$$\Rightarrow S_B w - J(w)S_W w = 0$$

$$\Rightarrow S_W^{-1}S_B w - J(w)w = 0$$

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA ... Two Classes

- Solving the generalized eigen value problem

$$S_W^{-1} S_B w = \lambda w \qquad where \quad \lambda = J(w) = scalar$$

yields

$$w^* = \arg\max_w J(w) = \arg\max_w \left(\frac{w^T S_B w}{w^T S_W w} \right) = S_W^{-1}(\mu_1 - \mu_2)$$

- This is known as Fisher's Linear Discriminant, although it is not a discriminant but rather a specific choice of direction for the projection of the data down to one dimension.

- Using the same notation as PCA, **the solution will be the eigen vector(s) of** $S_X = S_W^{-1} S_B$

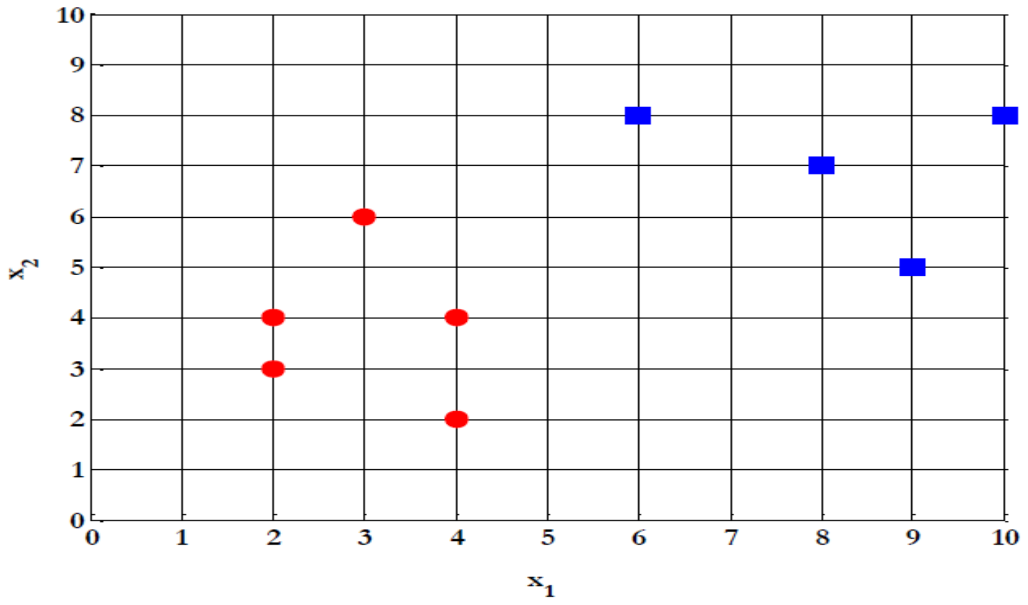
Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf

LDA ... Two Classes - Example

- Compute the Linear Discriminant projection for the following two-dimensional dataset.
  - Samples for class ω_1 : $\mathbf{X}_1=(x_1,x_2)=\{(4,2),(2,4),(2,3),(3,6),(4,4)\}$
  - Sample for class ω_2 : $\mathbf{X}_2=(x_1,x_2)=\{(9,10),(6,8),(9,5),(8,7),(10,8)\}$

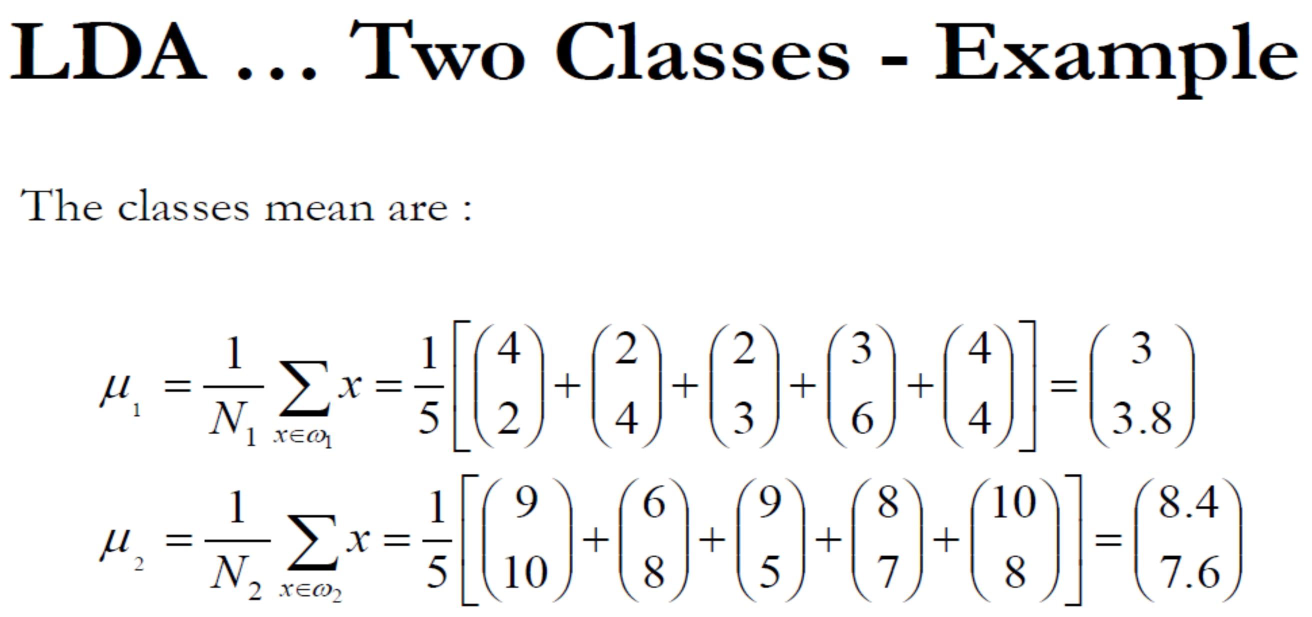


```
% samples for class 1
X1 = [4,2;
      2,4;
      2,3;
      3,6;
      4,4];


% samples for class 2
X2 = [9,10;
      6,8;
      9,5;
      8,7;
      10,8];
```

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf

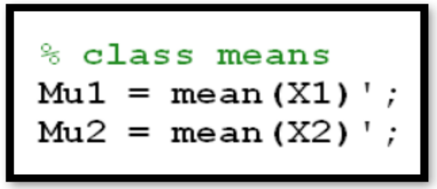


LDA … Two Classes - Example

- The classes mean are :

$$\mu_1 = \frac{1}{N_1}\sum_{x\in\omega_1} x = \frac{1}{5}\left[\begin{pmatrix}4\\2\end{pmatrix}+\begin{pmatrix}2\\4\end{pmatrix}+\begin{pmatrix}2\\3\end{pmatrix}+\begin{pmatrix}3\\6\end{pmatrix}+\begin{pmatrix}4\\4\end{pmatrix}\right]=\begin{pmatrix}3\\3.8\end{pmatrix}$$

$$\mu_2 = \frac{1}{N_2}\sum_{x\in\omega_2} x = \frac{1}{5}\left[\begin{pmatrix}9\\10\end{pmatrix}+\begin{pmatrix}6\\8\end{pmatrix}+\begin{pmatrix}9\\5\end{pmatrix}+\begin{pmatrix}8\\7\end{pmatrix}+\begin{pmatrix}10\\8\end{pmatrix}\right]=\begin{pmatrix}8.4\\7.6\end{pmatrix}$$

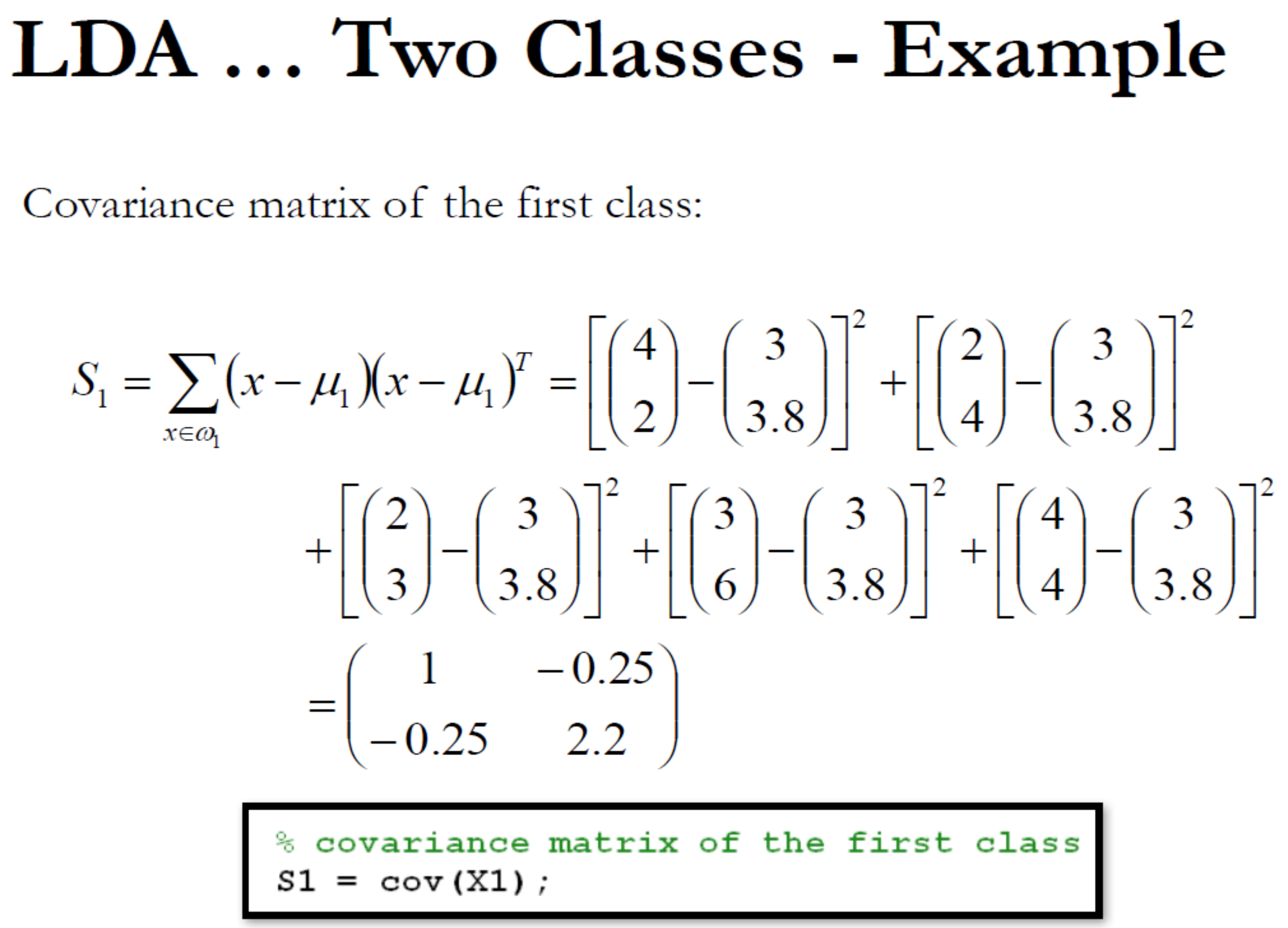


```
% class means
Mu1 = mean(X1)';
Mu2 = mean(X2)';
```

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes - Example

- Covariance matrix of the first class:

$$S_1 = \sum_{x\in\omega_1}(x-\mu_1)(x-\mu_1)^T = \left[\begin{pmatrix}4\\2\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2 + \left[\begin{pmatrix}2\\4\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2$$

$$+\left[\begin{pmatrix}2\\3\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2 + \left[\begin{pmatrix}3\\6\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2 + \left[\begin{pmatrix}4\\4\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2$$

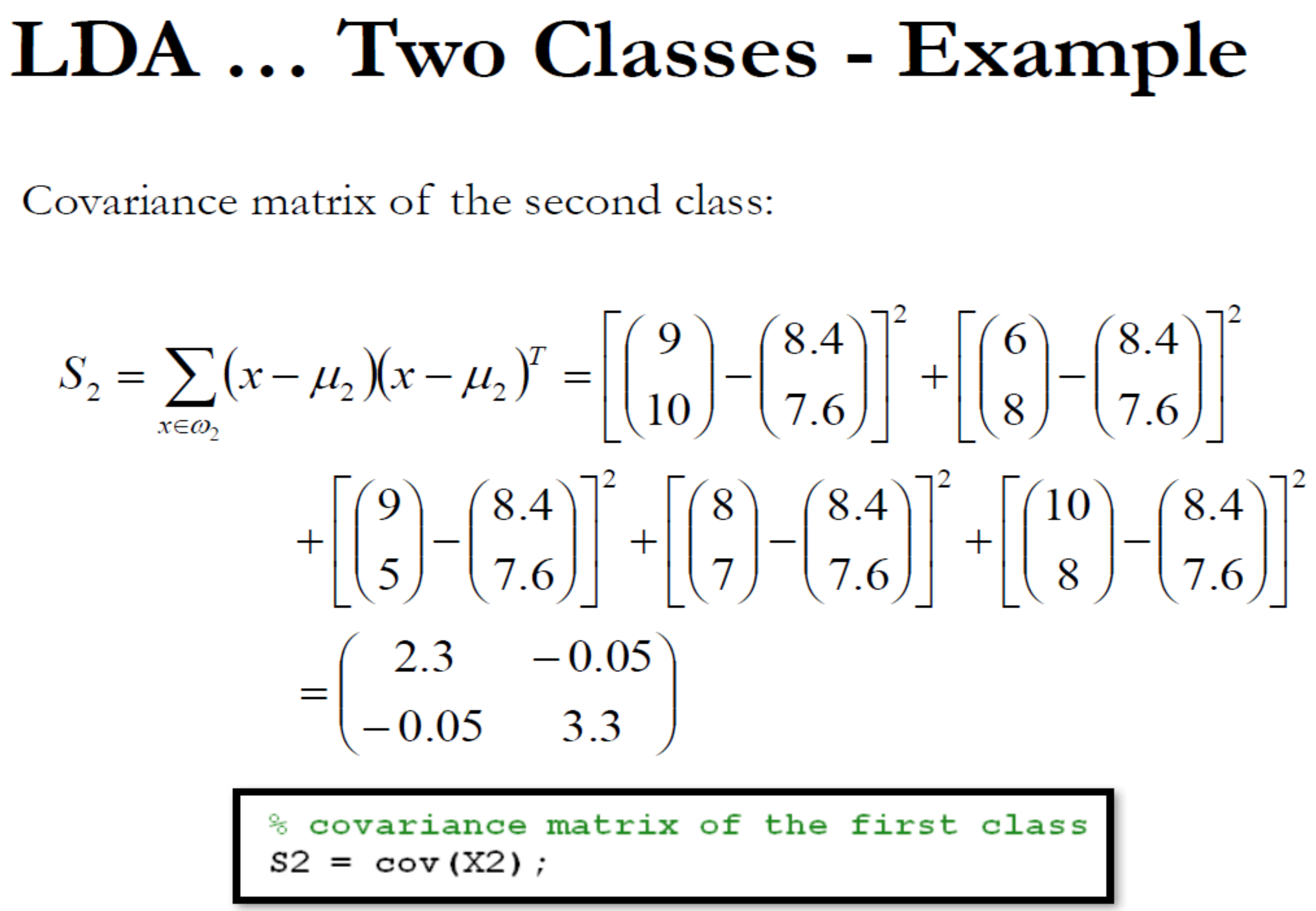
$$= \begin{pmatrix}1 & -0.25\\ -0.25 & 2.2\end{pmatrix}$$

```
% covariance matrix of the first class
S1 = cov(X1);
```

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes - Example

- Covariance matrix of the second class:

$$S_2 = \sum_{x\in\omega_2}(x-\mu_2)(x-\mu_2)^T = \left[\begin{pmatrix}9\\10\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2 + \left[\begin{pmatrix}6\\8\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2$$

$$+\left[\begin{pmatrix}9\\5\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2 + \left[\begin{pmatrix}8\\7\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2 + \left[\begin{pmatrix}10\\8\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2$$

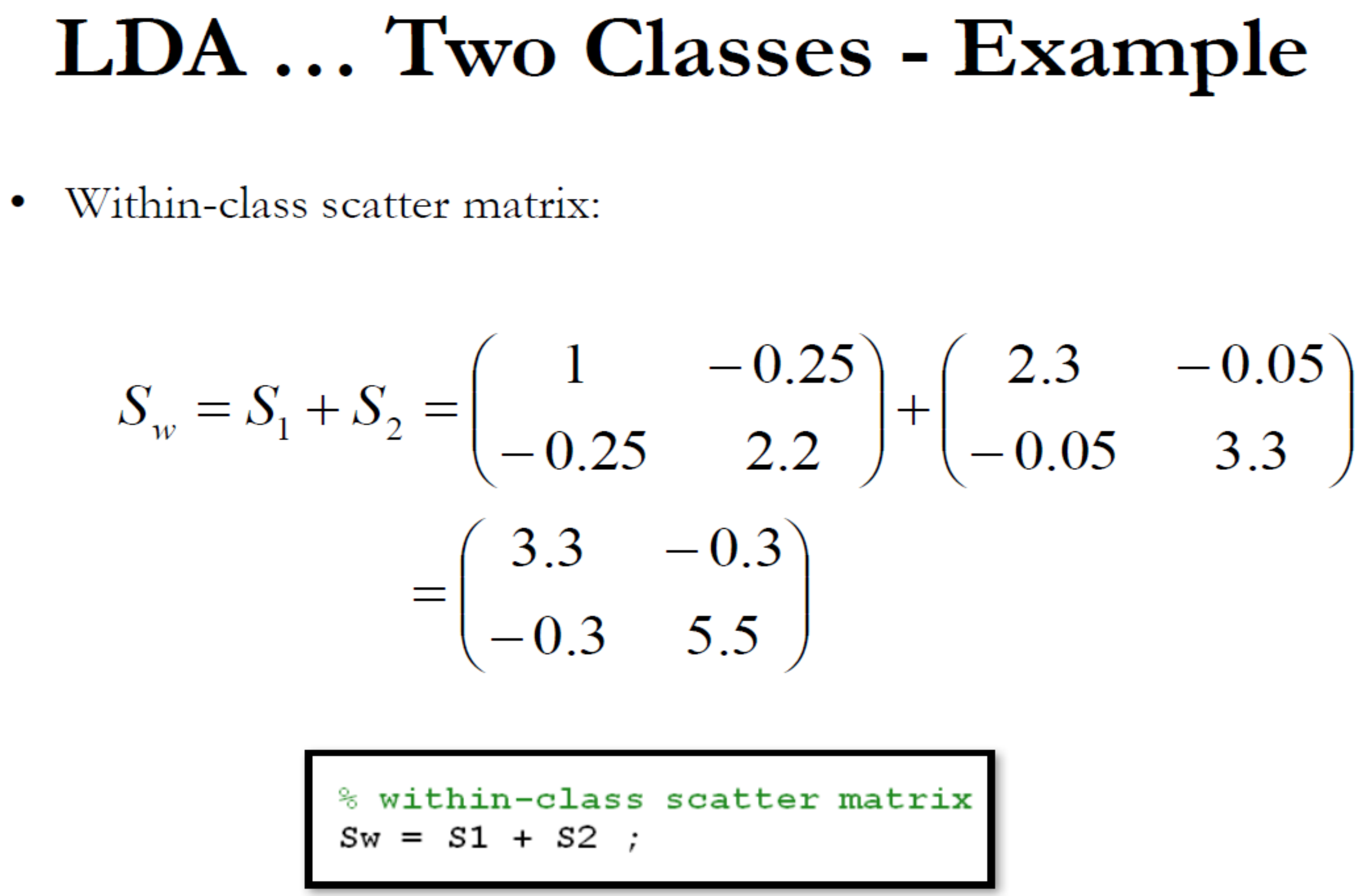
$$=\begin{pmatrix}2.3 & -0.05\\ -0.05 & 3.3\end{pmatrix}$$

```
% covariance matrix of the first class
S2 = cov(X2);
```

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes - Example

- Within-class scatter matrix:

$$S_w = S_1 + S_2 = \begin{pmatrix} 1 & -0.25 \\ -0.25 & 2.2 \end{pmatrix} + \begin{pmatrix} 2.3 & -0.05 \\ -0.05 & 3.3 \end{pmatrix}$$

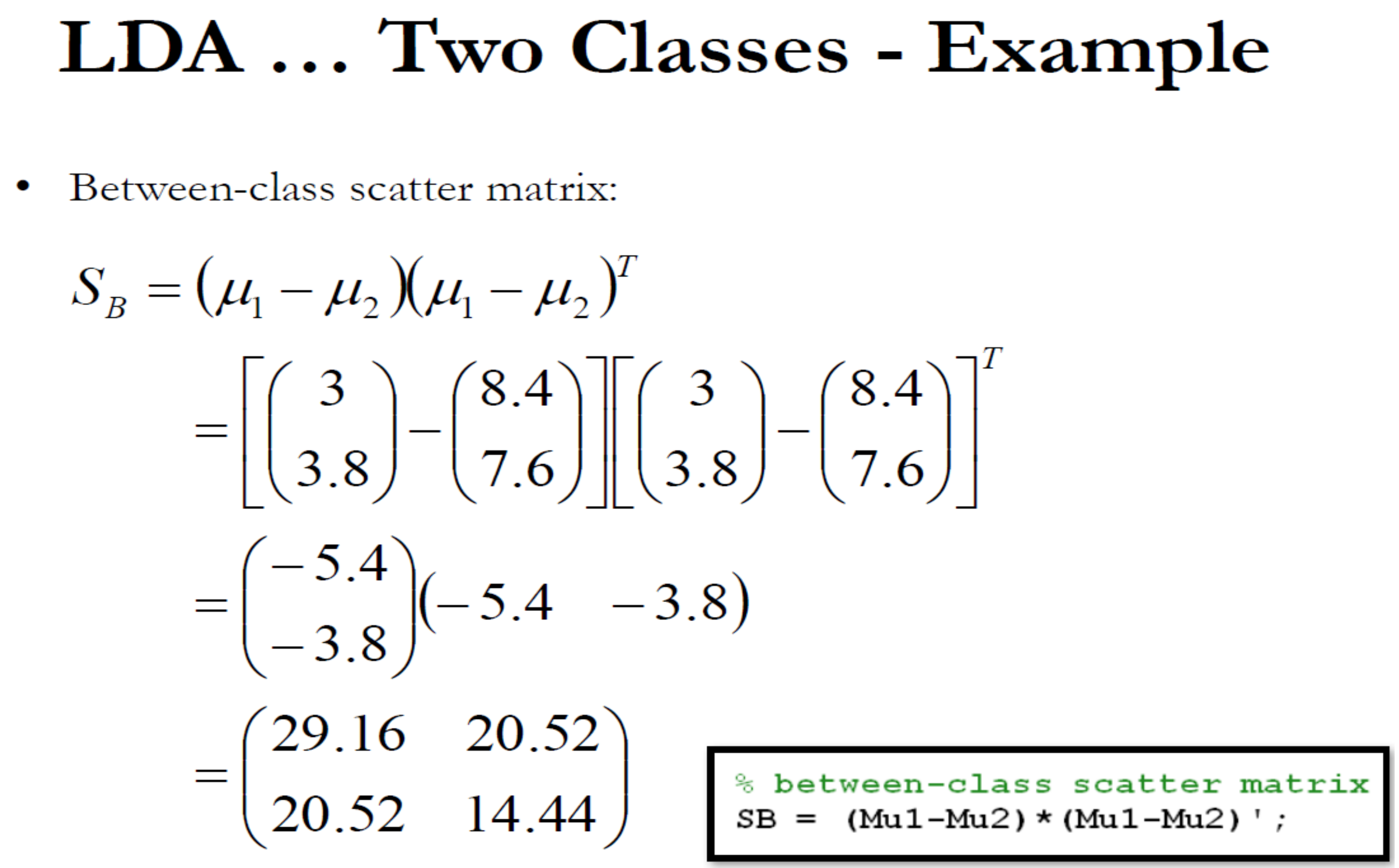
$$= \begin{pmatrix} 3.3 & -0.3 \\ -0.3 & 5.5 \end{pmatrix}$$

```
% within-class scatter matrix
Sw = S1 + S2 ;
```

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes - Example

- Between-class scatter matrix:

$$S_B = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$$

$$= \left[\begin{pmatrix} 3 \\ 3.8 \end{pmatrix} - \begin{pmatrix} 8.4 \\ 7.6 \end{pmatrix}\right]\left[\begin{pmatrix} 3 \\ 3.8 \end{pmatrix} - \begin{pmatrix} 8.4 \\ 7.6 \end{pmatrix}\right]^T$$

$$= \begin{pmatrix} -5.4 \\ -3.8 \end{pmatrix}\begin{pmatrix} -5.4 & -3.8 \end{pmatrix}$$

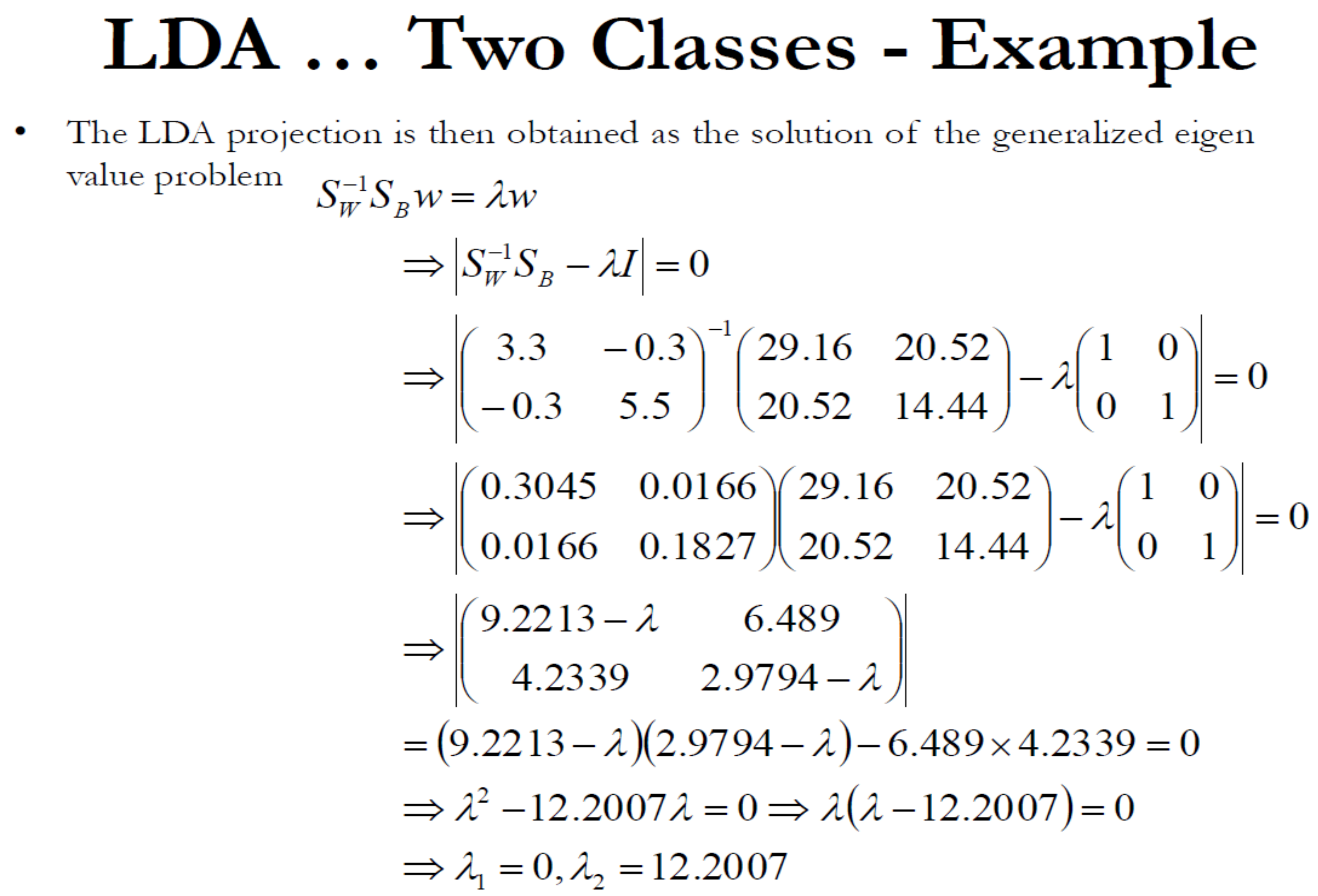
$$= \begin{pmatrix} 29.16 & 20.52 \\ 20.52 & 14.44 \end{pmatrix}$$

```
% between-class scatter matrix
SB = (Mu1-Mu2)*(Mu1-Mu2)';
```

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes - Example

- The LDA projection is then obtained as the solution of the generalized eigen value problem $S_W^{-1} S_B w = \lambda w$

$$\Rightarrow \left| S_W^{-1} S_B - \lambda I \right| = 0$$

$$\Rightarrow \left| \begin{pmatrix} 3.3 & -0.3 \\ -0.3 & 5.5 \end{pmatrix}^{-1} \begin{pmatrix} 29.16 & 20.52 \\ 20.52 & 14.44 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right| = 0$$

$$\Rightarrow \left| \begin{pmatrix} 0.3045 & 0.0166 \\ 0.0166 & 0.1827 \end{pmatrix} \begin{pmatrix} 29.16 & 20.52 \\ 20.52 & 14.44 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right| = 0$$

$$\Rightarrow \left| \begin{pmatrix} 9.2213 - \lambda & 6.489 \\ 4.2339 & 2.9794 - \lambda \end{pmatrix} \right|$$

$$= (9.2213 - \lambda)(2.9794 - \lambda) - 6.489 \times 4.2339 = 0$$

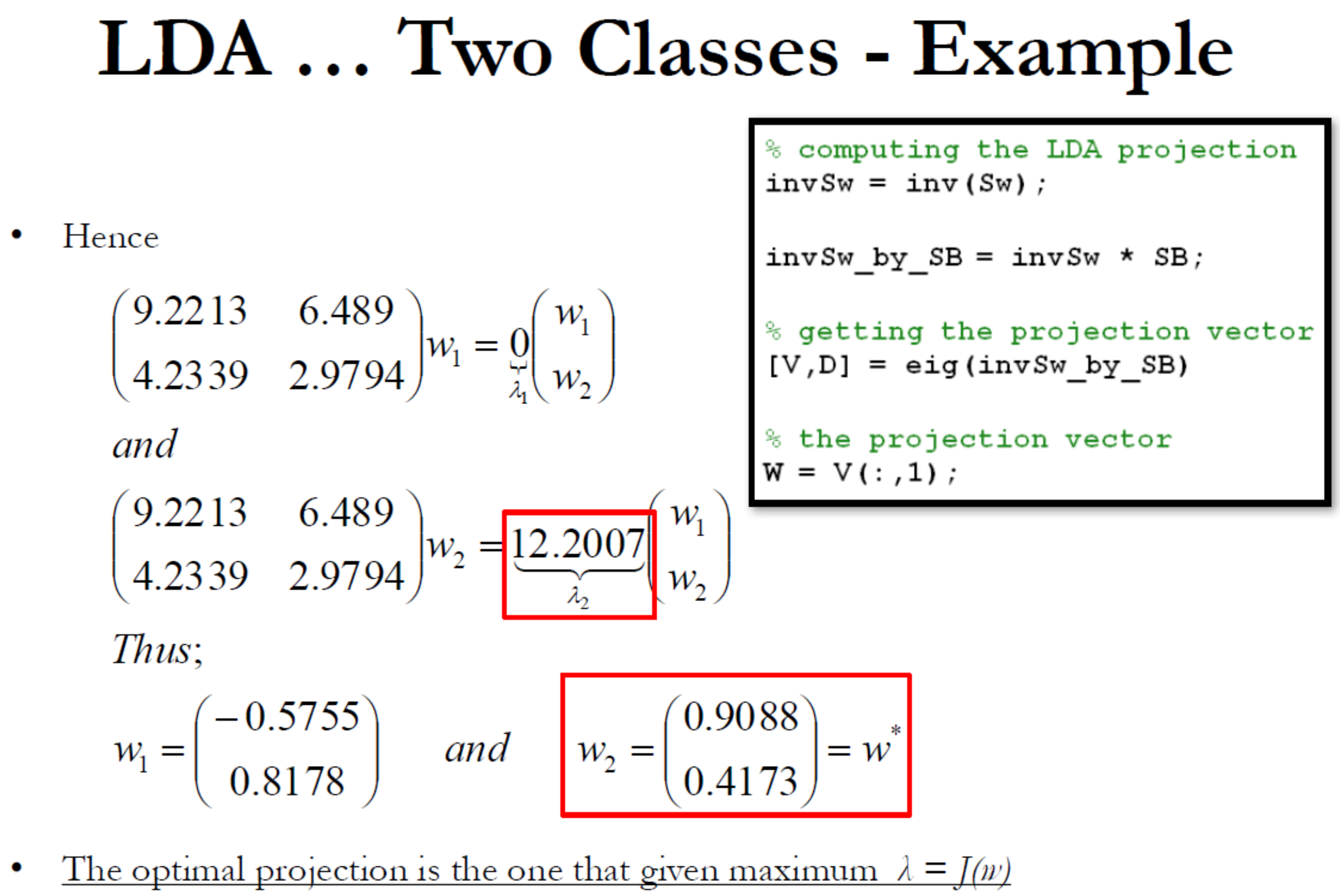
$$\Rightarrow \lambda^2 - 12.2007\lambda = 0 \Rightarrow \lambda(\lambda - 12.2007) = 0$$

$$\Rightarrow \lambda_1 = 0, \lambda_2 = 12.2007$$

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes - Example

- Hence

$$\begin{pmatrix} 9.2213 & 6.489 \\ 4.2339 & 2.9794 \end{pmatrix} w_1 = \underbrace{0}_{\lambda_1} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

and

$$\begin{pmatrix} 9.2213 & 6.489 \\ 4.2339 & 2.9794 \end{pmatrix} w_2 = \boxed{\underbrace{12.2007}_{\lambda_2}} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

Thus;

$$w_1 = \begin{pmatrix} -0.5755 \\ 0.8178 \end{pmatrix} \quad and \quad \boxed{w_2 = \begin{pmatrix} 0.9088 \\ 0.4173 \end{pmatrix} = w^*}$$

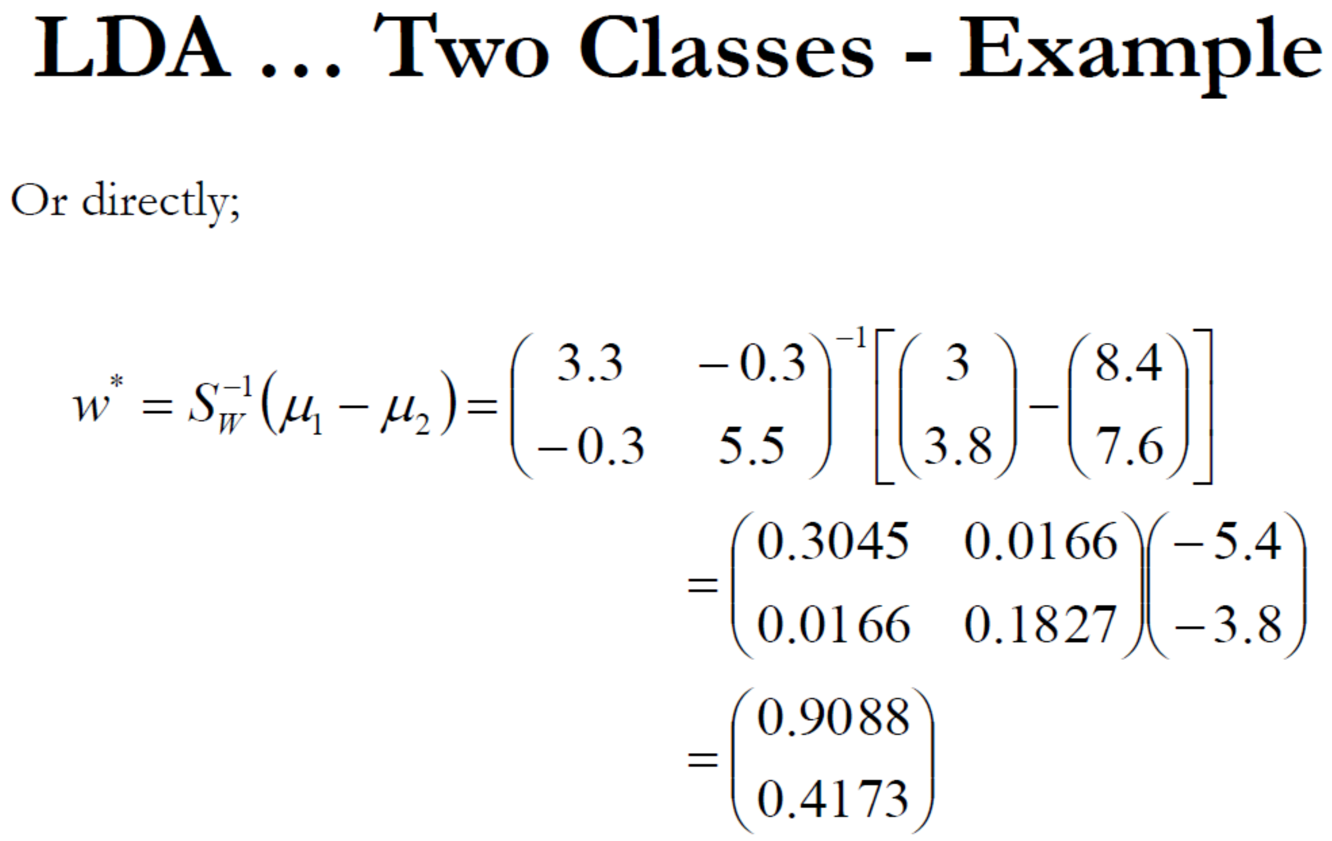
```
% computing the LDA projection
invSw = inv(Sw);

invSw_by_SB = invSw * SB;

% getting the projection vector
[V,D] = eig(invSw_by_SB)

% the projection vector
W = V(:,1);
```

- The optimal projection is the one that given maximum $\lambda = J(w)$

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



LDA … Two Classes - Example

Or directly;

$$w^* = S_W^{-1}(\mu_1 - \mu_2) = \begin{pmatrix} 3.3 & -0.3 \\ -0.3 & 5.5 \end{pmatrix}^{-1} \left[\begin{pmatrix} 3 \\ 3.8 \end{pmatrix} - \begin{pmatrix} 8.4 \\ 7.6 \end{pmatrix} \right]$$

$$= \begin{pmatrix} 0.3045 & 0.0166 \\ 0.0166 & 0.1827 \end{pmatrix} \begin{pmatrix} -5.4 \\ -3.8 \end{pmatrix}$$

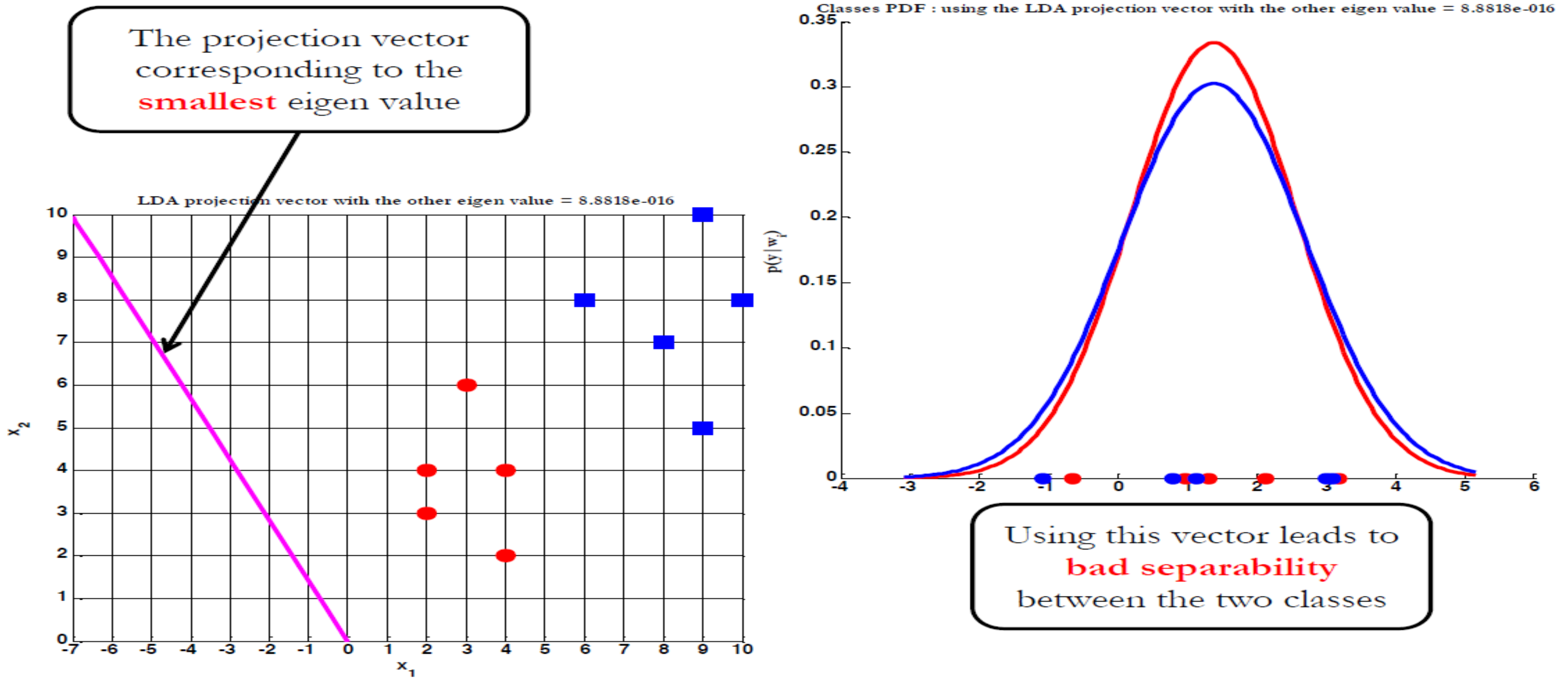
$$= \begin{pmatrix} 0.9088 \\ 0.4173 \end{pmatrix}$$

Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf

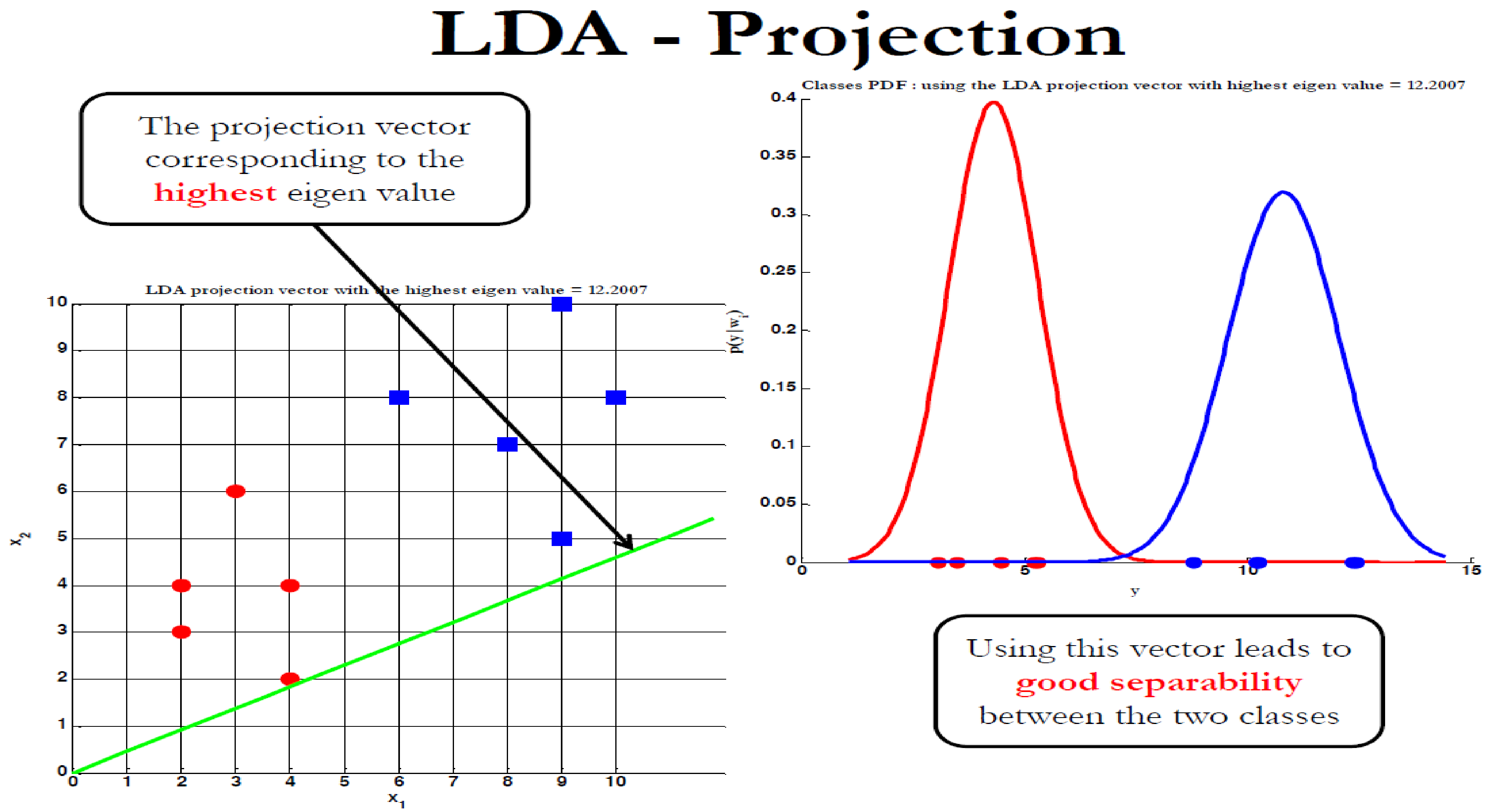
LDA - Projection



Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf



Linear Discriminant Analysis(LDA)

- LDA

http://www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_LDA09.pdf

LDA – C-Classes

- Recall the two classes case, the *between-class scatter* was computed as:

$$S_B = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$$

- For *C*-classes case, we will measure the between-class scatter with respect to the mean of all classes as follows:

$$S_B = \sum_{i=1}^{C} N_i (\mu_i - \mu)(\mu_i - \mu)^T$$

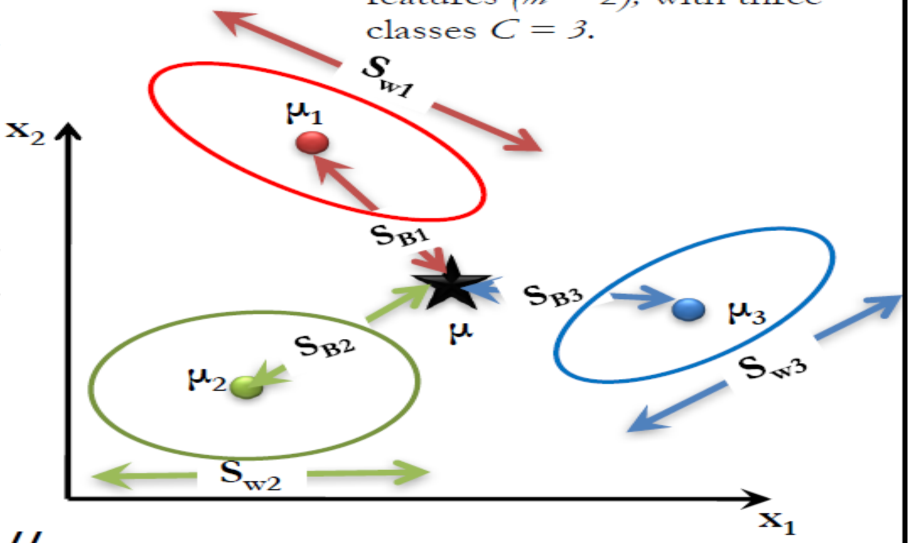
$$\textit{where} \qquad \mu = \frac{1}{N}\sum_{\forall x} x = \frac{1}{N}\sum_{\forall x} N_i \mu_i$$

$$\textit{and} \qquad \mu_i = \frac{1}{N_i}\sum_{x \in \omega_i} x$$

N: number of all data .

N_i : number of data samples in class ω_i.

Example of two-dimensional features (*m* = *2*), with three classes *C* = *3*.



Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- **Before we dive into the code implementation, let's briefly summarize the main steps that are required to perform LDA:**
- 1. Standardize the d-dimensional dataset (d is the number of features).
- 2. For each class, compute the d-dimensional mean vector.
- 3. Construct the between-class scatter matrix, $\boldsymbol{S_B}$, and the within-class scatter matrix, $\boldsymbol{S_w}$.
- 4. Compute the eigenvectors and corresponding eigenvalues of the matrix,
- $\boldsymbol{S_w^{-1}\ S_B}$.
- 5. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.
- 6. Choose the k eigenvectors that correspond to the k largest eigenvalues to construct a $d{\times}k$-dimensional transformation matrix, **W**; the eigenvectors are the columns of this matrix.
- 7. Project the examples onto the new feature subspace using the transformation matrix, **W**.

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**

Preparing the sample data set

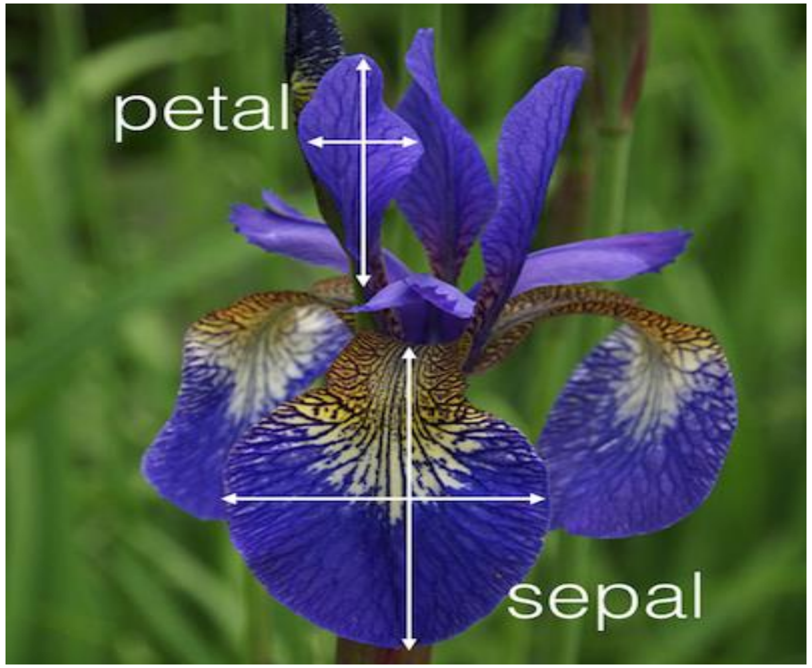
About the Iris dataset
For the following tutorial, we will be working with the famous “Iris” dataset that has been deposited on the UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/Iris).

Reference: Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.
The iris dataset contains measurements for 150 iris flowers from three different species.

The three classes in the Iris dataset:

Iris-setosa (n=50)
Iris-versicolor (n=50)
Iris-virginica (n=50)
The four features of the Iris dataset:

sepal length in cm
sepal width in cm
petal length in cm
petal width in cm



Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- **Preparing dataset**

$$X = \begin{bmatrix} x_{1f1} & x_{1f2} & x_{1f3} & x_{1f4} \\ x_{2f1} & x_{2f2} & x_{2f3} & x_{2f4} \\ x_{150f1} & x_{150f2} & x_{150f3} & x_{150f4} \end{bmatrix}$$

4x150(4 features)

$$y = \begin{bmatrix} w_{setosa} \\ w_{setosa} \\ w_{setosa} \\ \cdots \\ w_{virgincia} \end{bmatrix} \rightarrow \quad y = \begin{bmatrix} 1 \\ 1 \\ 2 \\ \cdots \\ 3 \end{bmatrix}$$

150x1 (50 samples for each class)

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- Normality assumptions
- It should be mentioned that LDA assumes normal distributed data, features that are statistically independent, and identical covariance matrices for every class. However, this only applies for LDA as classifier and LDA for dimensionality reduction can also work reasonably well if those assumptions are violated. And even for classification tasks LDA seems can be quite robust to the distribution of the data

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- LDA in 5 steps
- Step 1: Computing the d-dimensional mean vectors
- In this first step, we will start off with a simple computation of the mean vectors m_i, (i=1,2,3) of the 3 different flower classes:

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{w_i}(\text{sepal length}) \\ \mu_{w_i}(\text{sepal } width) \\ \mu_{w_i}(\text{petal } hight) \\ \mu_{w_i}(\text{petal } width) \end{bmatrix} \qquad i = 1,2,3$$

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- LDA in 5 steps
- Step 1: Computing the d-dimensional mean vectors
- In this first step, we will start off with a simple computation of the mean vectors m_i, (i=1,2,3) of the 3 different flower classes:

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{w_i}(\text{sepal length}) \\ \mu_{w_i}(\text{sepal } width) \\ \mu_{w_i}(\text{petal } hight) \\ \mu_{w_i}(\text{petal } width) \end{bmatrix} \qquad i = 1,2,3$$

```
np.set_printoptions(precision=4)

mean_vectors = []
for cl in range(1,4):
    mean_vectors.append(np.mean(X[y==cl], axis=0))
    print('Mean Vector class %s: %s\n' %(cl, mean_vectors[cl-1]))
```

```
Mean Vector class 1: [ 5.006  3.418  1.464  0.244]

Mean Vector class 2: [ 5.936  2.77   4.26   1.326]

Mean Vector class 3: [ 6.588  2.974  5.552  2.026]
```

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- LDA in 5 steps
- Step 2: Computing the Scatter Matrices
- Now, we will compute the two 4x4-dimensional matrices: **The within-class and the between-class scatter matrix.**
- 2.1 Within-class scatter matrix $\boldsymbol{S_w}$
- $\boldsymbol{S_w}=\sum_{i=1}^{c} S_i$
- This is calculated by summing up the individual scatter matrices, S_i, of each individual class i:
- $S_i=\sum_{x\in D_i}(x-m_i)(x-m_i)^T \ where$
- $m_i=\frac{1}{n_i}\sum_{x\in D_i} x_m$ m_i is the mean vector

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- Step 2: Computing the Scatter Matrices
- Now, we will compute the two 4x4-dimensional matrices: **The within-class and the between-class scatter matrix.**
- 2.1 Within-class scatter matrix $\boldsymbol{S_w}$
- $\boldsymbol{S_w}=\sum_{i=1}^{c} S_i$
- This is calculated by summing up the individual scatter matrices, S_i, of each individual class i:
- $S_i=\sum_{x\in D_i}(x-m_i)(x-m_i)^T \; where$
- $m_i=\frac{1}{n_i}\sum_{x\in D_i} x_m$ $\qquad m_i$ is the mean vector

```
S_W = np.zeros((4,4))
for cl,mv in zip(range(1,4), mean_vectors):
    class_sc_mat = np.zeros((4,4))                  # scatter matrix for every class
    for row in X[y == cl]:
        row, mv = row.reshape(4,1), mv.reshape(4,1) # make column vectors
        class_sc_mat += (row-mv).dot((row-mv).T)
    S_W += class_sc_mat                             # sum class scatter matrices
print('within-class Scatter Matrix:\n', S_W)
```

```
within-class Scatter Matrix:
 [[ 38.9562  13.683   24.614    5.6556]
 [ 13.683   17.035    8.12     4.9132]
 [ 24.614    8.12    27.22     6.2536]
 [  5.6556   4.9132   6.2536   6.1756]]
```

where NiNi is the sample size of the respective class (here: 50), and in this particular case, we can drop the term (Ni−1)Ni−1) since all classes have the same sample size.

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- Step 2 1.: Computing the Scatter Matrices
- Alternatively, we could also compute the class-covariance matrices by adding the scaling factor 1/N−1 to the within-class scatter matrix, so that our equation becomes
- $\Sigma_i = \frac{1}{N_i - 1} S_i = \frac{1}{N_i - 1} \sum_{x \in D_i} (x - m_i)(x - m_i)^T$
- $\boldsymbol{S_w} = \sum_i^c (N_i - 1)\, \Sigma_i$
- where N_i is the sample size of the respective class (here: 50), and in this particular case, we can drop the term (N_i −1) since all classes have the same sample size.

However, the resulting eigenspaces will be identical (identical eigenvectors, only the eigenvalues are scaled differently by a constant factor).

where NiNi is the sample size of the respective class (here: 50), and in this particular case, we can drop the term (Ni−1)Ni−1) since all classes have the same sample size.

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- Step 2 1.: Computing the Scatter Matrices
- Alternatively, we could also compute the class-covariance matrices by adding the scaling factor 1/N−1 to the within-class scatter matrix, so that our equation becomes
- $\Sigma_i = \frac{1}{N_i - 1} S_i = \frac{1}{N_i - 1} \sum_{x \in D_i} (x - m_i)(x - m_i)^T$
- $\boldsymbol{S_w} = \sum_i^c (N_i - 1)\, \Sigma_i$
- where N_i is the sample size of the respective class (here: 50), and in this particular case, we can drop the term (N_i −1) since all classes have the same sample size.

However, the resulting eigenspaces will be identical (identical eigenvectors, only the eigenvalues are scaled differently by a constant factor).

where NiNi is the sample size of the respective class (here: 50), and in this particular case, we can drop the term (Ni−1)Ni−1) since all classes have the same sample size.

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- 2.2 Between-class scatter matrix S_B
- compute the between-class scatter matrix $\boldsymbol{S_B}$:
- $S_B = \sum_{i=1}^{c} N_i\,(m_i - m)(m_i - m)^T$
- Where m is the overall mean, and m_i and N_i are the sample mean and sizes of the respective classes.

```
overall_mean = np.mean(X, axis=0)

S_B = np.zeros((4,4))
for i,mean_vec in enumerate(mean_vectors):
   n = X[y==i+1,:].shape[0]
   mean_vec = mean_vec.reshape(4,1) # make column vector
   overall_mean = overall_mean.reshape(4,1) # make column vector
   S_B += n * (mean_vec - overall_mean).dot((mean_vec - overall_mean).T)

print('between-class Scatter Matrix:\n', S_B)
```

```
between-class Scatter Matrix:
 [[  63.2121  -19.534   165.1647   71.3631]
 [ -19.534    10.9776  -56.0552  -22.4924]
 [ 165.1647  -56.0552  436.6437  186.9081]
 [  71.3631  -22.4924  186.9081   80.6041]]
```

where NiNi is the sample size of the respective class (here: 50), and in this particular case, we can drop the term (Ni−1)Ni−1) since all classes have the same sample size.

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- Step 3: Solving the generalized eigenvalue problem for the matrix $\boldsymbol{S_w^{-1}\ S_B}$:
- Next, we will solve the generalized eigenvalue problem for the matrix $\boldsymbol{S_w^{-1}\ S_B}$ to obtain the linear discriminants.

```
eig_vals, eig_vecs = np.linalg.eig(np.linalg.inv(S_W).dot(S_B))

for i in range(len(eig_vals)):
   eigvec_sc = eig_vecs[:,i].reshape(4,1)
   print('\nEigenvector {}: \n{}'.format(i+1, eigvec_sc.real))
   print('Eigenvalue {:}: {:.2e}'.format(i+1, eig_vals[i].real))
```

```
Eigenvector 1:
[[-0.2049]
 [-0.3871]
 [ 0.5465]
 [ 0.7138]]
Eigenvalue 1: 3.23e+01

Eigenvector 2:
[[-0.009 ]
 [-0.589 ]
 [ 0.2543]
 [-0.767 ]]
Eigenvalue 2: 2.78e-01

Eigenvector 3:
[[ 0.179 ]
 [-0.3178]
 [-0.3658]
 [ 0.6011]]
Eigenvalue 3: -4.02e-17

Eigenvector 4:
[[ 0.179 ]
 [-0.3178]
 [-0.3658]
 [ 0.6011]]
Eigenvalue 4: -4.02e-17
```

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- **Step 4: Selecting linear discriminants for the new feature subspace**
- **4.1. Sorting the eigenvectors by decreasing eigenvalues**
- Remember from the introduction that we are not only interested in merely projecting the data into a subspace that improves the class separability, but also reduces the dimensionality of our feature space, (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.
- 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.**

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- Step 4: Selecting linear discriminants for the new feature subspace
- 4.1. Sorting the eigenvectors by decreasing eigenvalues

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

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

# Visually confirm that the list is correctly sorted by decreasing eigenvalues

print('Eigenvalues in decreasing order:\n')
for i in eig_pairs:
   print(i[0])
```

```
Eigenvalues in decreasing order:

32.2719577997
0.27756686384
5.71450476746e-15
5.71450476746e-15
```

Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**
- If we take a look at the eigenvalues, we can already see that 2 eigenvalues are close to 0. The reason why these are close to 0 is not that they are not informative but it's due to floating-point imprecision. In fact, these two last eigenvalues should be exactly zero: In LDA, the number of linear discriminants **is at most c−1 where c is the number of class** labels, since the in-between scatter matrix SB is the sum of c matrices with rank 1 or less. Note that in the rare case of perfect collinearity (all aligned sample points fall on a straight line), the covariance matrix would have rank one, which would result in only one eigenvector with a nonzero eigenvalue.
- Now, let's express **the "explained variance" as percentage:**

```
print('Variance explained:\n')
eigv_sum = sum(eig_vals)
for i,j in enumerate(eig_pairs):
   print('eigenvalue {0:}: {1:.2%}'.format(i+1,
(j[0]/eigv_sum).real))
```

```
Variance explained:

eigenvalue 1: 99.15%
eigenvalue 2: 0.85%
eigenvalue 3: 0.00%
eigenvalue 4: 0.00%
```

Linear Discriminant Analysis(LDA)

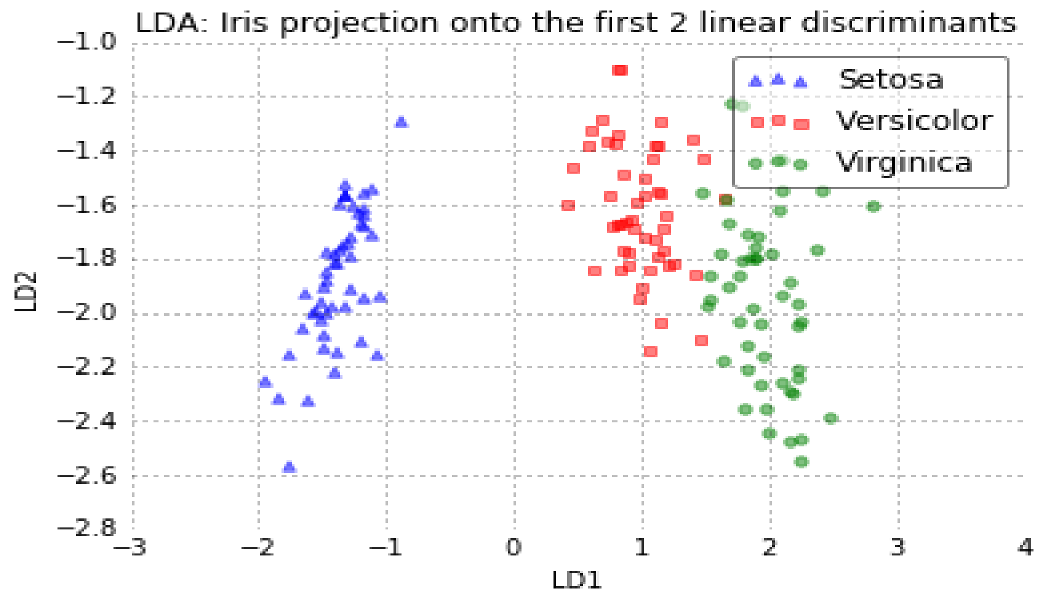
- **Linear discriminant analysais**
- **4.2. Choosing k eigenvectors with the largest eigenvalues**
- After sorting the eigenpairs by decreasing eigenvalues, it is now time to construct our k×d-dimensional eigenvector matrix W(here 4×2: based on the 2 most informative eigenpairs) and thereby reducing the initial 4-dimensional feature space into a 2-dimensional feature subspace.

```
W = np.hstack((eig_pairs[0][1].reshape(4,1),
eig_pairs[1][1].reshape(4,1)))
print('Matrix W:\n', W.real)
```

```
Matrix W:
 [[-0.2049 -0.009 ]
 [-0.3871 -0.589 ]
 [ 0.5465  0.2543]
 [ 0.7138 -0.767 ]]
```

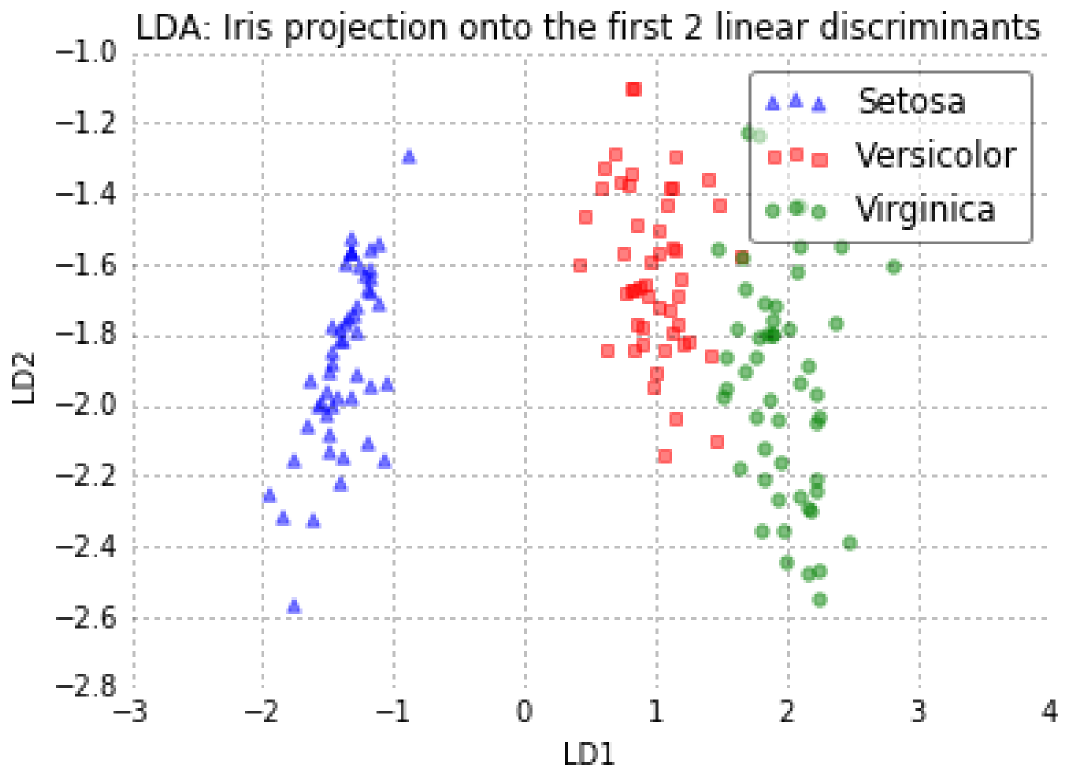
Linear Discriminant Analysis(LDA)

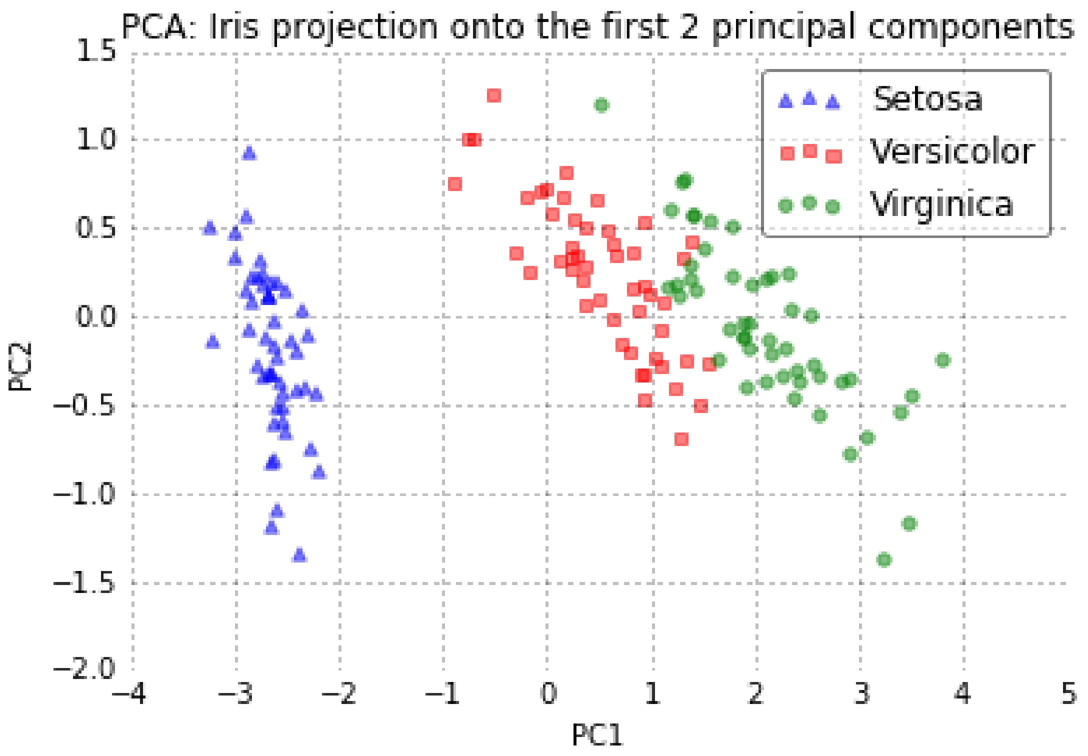
- **Linear discriminant analysais**
- **Step 5: Transforming the samples onto the new subspace**
- In the last step, we use the 4×2-dimensional matrix W that we just computed to transform our samples onto the new subspace via the equation
- Y=X×W.
- (where X is a n×d-dimensional matrix representing the n samples, and Y are the transformed n×k-dimensional samples in the new subspace).



Linear Discriminant Analysis(LDA)

- **Linear discriminant analysais**





Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- As we can see, LDA is quite similar to PCA in the sense that we are decomposing matrices into eigenvalues and eigenvectors, which will form the new lower- dimensional feature space.
- **However, as mentioned before, LDA takes class label information into account, which is represented in the form of the mean vectors computed in step 2**
- **Computing the scatter matrices**
- Since we already standardized the features of the Wine dataset in the PCA section, we can skip the first step and proceed with the calculation of the mean vectors, which we will use to construct the within-class scatter matrix and between-class scatter matrix, respectively. Each mean vector, m_i, stores the mean feature value, μ_m, with respect to the examples of class i:
- $m_i = \frac{1}{n_i} \sum_{x \in D_i} x_m$
- This results in three mean vectors:

Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- Since we already standardized the features of the Wine dataset in the PCA section, we can skip the first step and proceed with the calculation of the mean vectors, which we will use to construct the within-class scatter matrix and between-class scatter matrix, respectively. Each mean vector, m_i, stores the mean feature value, μ_m, with respect to the examples of class i:

- $m_i = \frac{1}{n_i} \sum_{x \in D_i} x_m$

- This results in three mean vectors:
- $m_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \dots \\ \mu_{i,proline} \end{bmatrix}^T \quad i \in \{1,2,3\}$

Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- $m_i = \frac{1}{n_i} \sum_{x \in D_i} x_m$

- This results in three mean vectors:
- $m_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \dots \\ \mu_{i,proline} \end{bmatrix}^T \quad i \in \{1,2,3\}$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...             mean_vecs.append(np.mean(

...             X_train_std[y_train==label], axis=0))
...             print('MV %s: %s\n' %(label, mean_vecs[label-1]))
MV 1: [ 0.9066
-0.5516     -0.3497     0.3201      -0.7189     0.5056      0.8807      0.9589
0.5416      0.2338      0.5897      0.6563      1.2075]
MV 2: [-0.8749
-0.0946     -0.2848     -0.3735     0.3157      -0.3848     -0.0433     0.0635
0.0703      -0.8286     0.3144      0.3608      -0.7253]
MV 3: [
0.8287      0.1992      0.866       0.1682      0.4148      -0.0451     -1.0286     -
1.2876
-0.7795     0.9649      -1.209      -1.3622     -0.4013]
```

Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- $m_i = \frac{1}{n_i} \sum_{x \in D_i} x_m$

- This results in three mean vectors:
- $m_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \dots \\ \mu_{i,proline} \end{bmatrix}^T \quad i \in \{1,2,3\}$
- Using the mean vectors, we can now compute the within-class scatter matrix, $\boldsymbol{S_w}$:
- $\boldsymbol{S_w} = \sum_{i=1}^{c} S_i$
- This is calculated by summing up the individual scatter matrices, S_i, of each individual class i:
- $S_i = \sum_{x \in D_i} (x - m_i)(x - m_i)^T$

Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- $m_i = \frac{1}{n_i} \sum_{x \in D_i} x_m$
- Using the mean vectors, we can now compute the within-class scatter matrix, $\boldsymbol{S_w}$:
- $\boldsymbol{S_w} = \sum_{i=1}^{c} S_i$
- This is calculated by summing up the individual scatter matrices, S_i, of each individual class i:
- $S_i = \sum_{x \in D_i} (x - m_i)(x - m_i)^T$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...         class_scatter = np.zeros((d, d))
>>> for row in X_train_std[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row - mv).dot((row - mv).T)
...         S_W += class_scatter
>>> print('Within-class scatter matrix: %sx%s' % (
...         S_W.shape[0], S_W.shape[1])) Within-class scatter matrix: 13x13
```

Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- $m_i = \frac{1}{n_i} \sum_{x \in D_i} x_m$
- Using the mean vectors, we can now compute the within-class scatter matrix, $\boldsymbol{S_w}$:
- $\boldsymbol{S_w} = \sum_{i=1}^{c} S_i$
- This is calculated by summing up the individual scatter matrices, S_i, of each individual class i:
- $S_i = \sum_{x \in D_i} (x - m_i)(x - m_i)^T$
- The assumption that we are making when we are computing the scatter matrices is that the class labels in the training dataset are uniformly distributed. However, if we print the number of class labels, we see that this assumption is violated:

```
>>> print('Class label distribution: %s'
...            % np.bincount(y_train)[1:]) Class label distribution:
[41 50 33]
```

Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- Thus, we want to scale the individual scatter matrices, S_i, before we sum them up as scatter matrix S_w. When we divide the scatter matrices by the number of class- examples, n_i, we can see that computing the scatter matrix is in fact the same as computing the covariance matrix, Σi—the covariance matrix is a normalized version of the scatter matrix:
- $\Sigma_i = \frac{1}{n_i} S_i = \frac{1}{n_i} \sum_{x \in D_i} (x - m_i)(x - m_i)^T$
- The code for computing the scaled within-class scatter matrix is as follows:

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1, 4), mean_vecs):
...         class_scatter = np.cov(X_train_std[y_train==label].T)
...         S_W += class_scatter
>>> print('Scaled within-class scatter matrix: %sx%s'
...         % (S_W.shape[0], S_W.shape[1])) Scaled within-class scatter
matrix: 13x13
```

Linear Discriminant Analysis(LDA)

- **The inner workings of linear discriminant analysis**
- After we compute the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute the between-class scatter matrix $\boldsymbol{S_B}$:
- $S_B = \sum_{i=1}^{c} n_i \, (m_i - m)(m_i - m)^T$

Here, m is the overall mean that is computed, including examples from all c classes:

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # number of features
>>> S_B = np.zeros((d, d))
>>> for i, mean_vec in enumerate(mean_vecs):
...         n = X_train_std[y_train == i + 1, :].shape[0]
...         mean_vec = mean_vec.reshape(d, 1) # make column vector
...         mean_overall = mean_overall.reshape(d, 1)
...         S_B += n * (mean_vec - mean_overall).dot(
...         (mean_vec - mean_overall).T)
>>> print('Between-class scatter matrix: %sx%s' % (
...         S_B.shape[0], S_B.shape[1])) Between-class scatter matrix: 13x13
```

Linear Discriminant Analysis(LDA)

- **Selecting linear discriminants for the new feature subspace**
- The remaining steps of the LDA are similar to the steps of the PCA. However, instead of performing the eigendecomposition on the covariance matrix, we solve the generalized eigenvalue problem of the matrix, $\boldsymbol{S_W^{-1}\ S_B}$:

```
>>> eigen_vals, eigen_vecs =\
...         np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

After we compute the eigenpairs, we can sort the eigenvalues in descending order:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...         for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...         key=lambda k: k[0], reverse=True)
>>> print('Eigenvalues in descending order:\n')
>>> for eigen_val in eigen_pairs:
...         print(eigen_val[0])
```

$\rightarrow$

```
Eigenvalues in descending order:
349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

Linear Discriminant Analysis(LDA)

- **Selecting linear discriminants for the new feature subspace**
- In LDA, the number of linear discriminants is at most c−1, where c is the number of class labels, since the in-between scatter matrix, $\boldsymbol{S_B}$, is the sum of c matrices with rank one or less. We can indeed see that we only have two nonzero eigenvalues (the eigenvalues 3-13 are not exactly zero, but this is due to the floating-point arithmetic in NumPy).

```
Eigenvalues in descending order: 349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

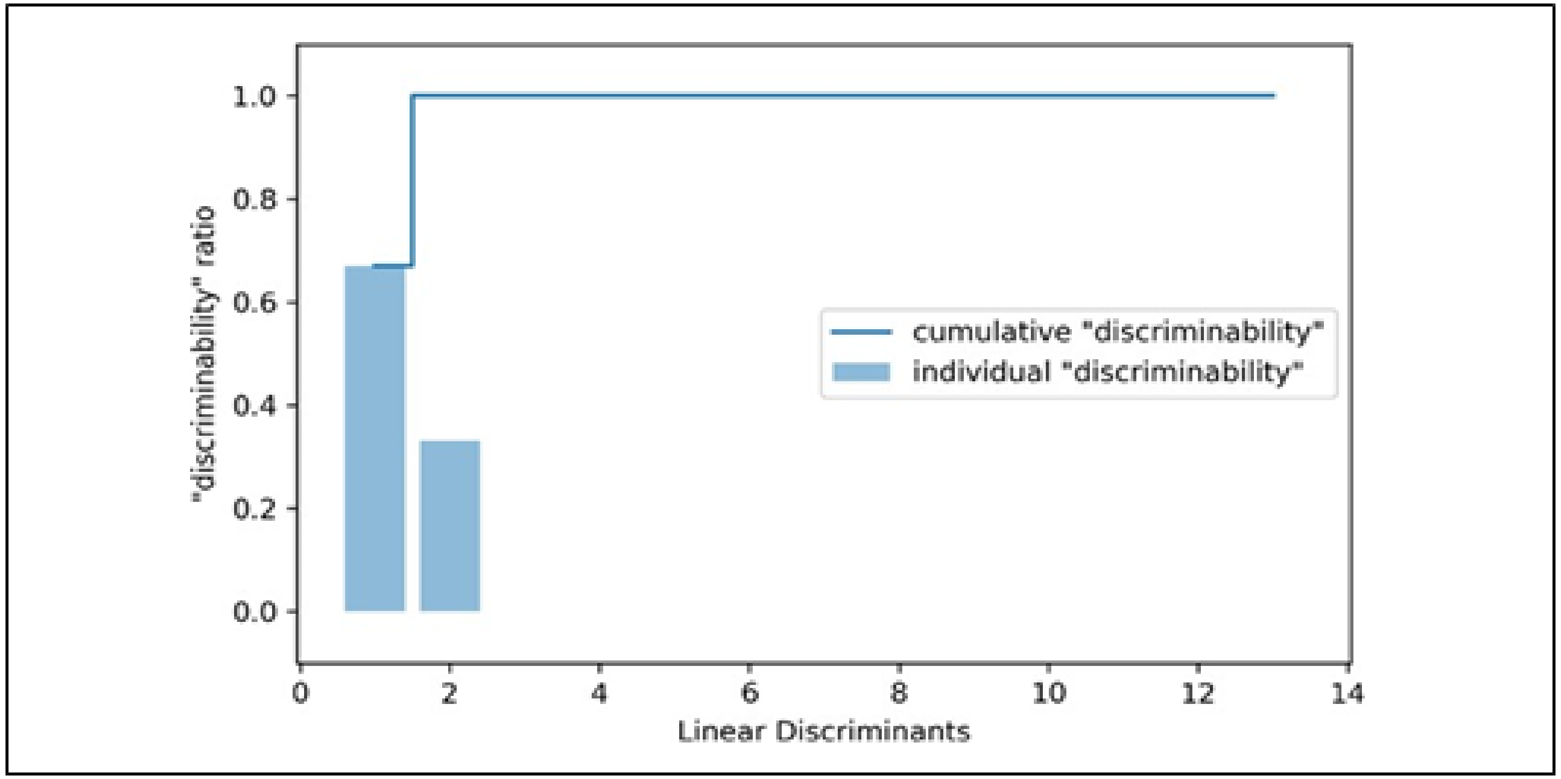
Linear Discriminant Analysis(LDA)

- **Selecting linear discriminants for the new feature subspace**
- To measure how much of the class-discriminatory information is captured by the linear discriminants (eigenvectors), let's plot the linear discriminants by decreasing eigenvalues, similar to the explained variance plot that we created in the PCA section.
- For simplicity, we will call the content of class-discriminatory information discriminability:

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...             label='Individual "discriminability"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...             label='Cumulative "discriminability"')
>>> plt.ylabel('"Discriminability" ratio')
>>> plt.xlabel('Linear Discriminants')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

Linear Discriminant Analysis(LDA)

- **Selecting linear discriminants for the new feature subspace**
- As we can see in the resulting figure, the first two linear discriminants alone capture
- 100 percent of the useful information in the Wine training dataset:



Linear Discriminant Analysis(LDA)

- **Selecting linear discriminants for the new feature subspace**
- Let's now stack the two most discriminative eigenvector columns to create the transformation matrix, W:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...            eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w) Matrix W:
[[-0.1481 -0.4092]
[ 0.0908  -0.1577]
[-0.0168  -0.3537]
[ 0.1484  0.3223]
[-0.0163  -0.0817]
[ 0.1913  0.0842]
[-0.7338  0.2823]
[-0.075    -0.0102]
[ 0.0018  0.0907]
[ 0.294    -0.2152]
[-0.0328  0.2747]
[-0.3547  -0.0124]
[-0.3915  -0.5958]]
```

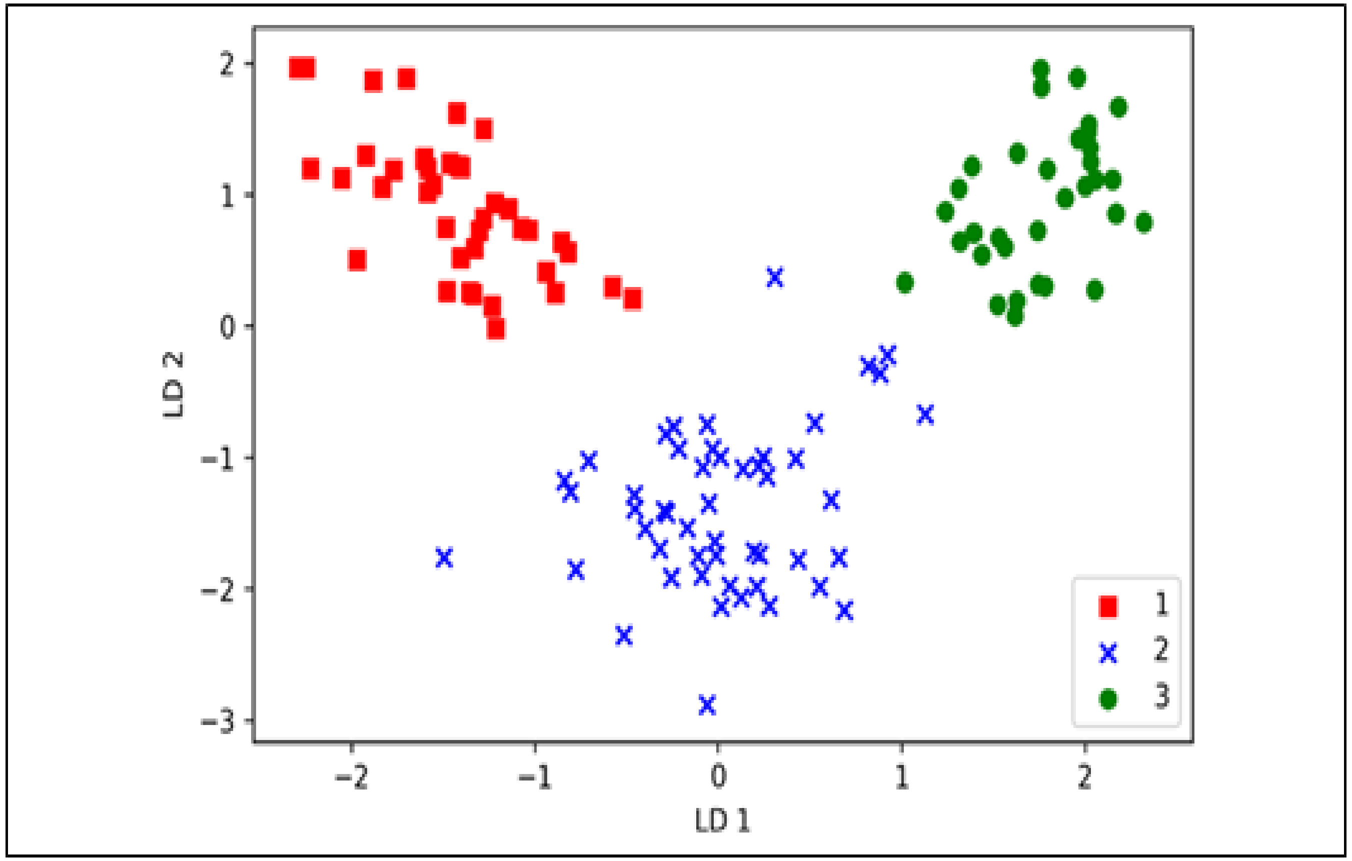
Linear Discriminant Analysis(LDA)

- **Projecting examples onto the new feature space**
- Using the transformation matrix, W, that we created in the previous subsection, we can now transform the training dataset by multiplying the matrices:
- X'=XW

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...            plt.scatter(X_train_lda[y_train==l, 0],
...            X_train_lda[y_train==l, 1] * (-1),
...            c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

Linear Discriminant Analysis(LDA)

- **Projecting examples onto the new feature space**
- As we can see in the resulting plot, the three Wine classes are now perfectly linearly separable in the new feature subspace:

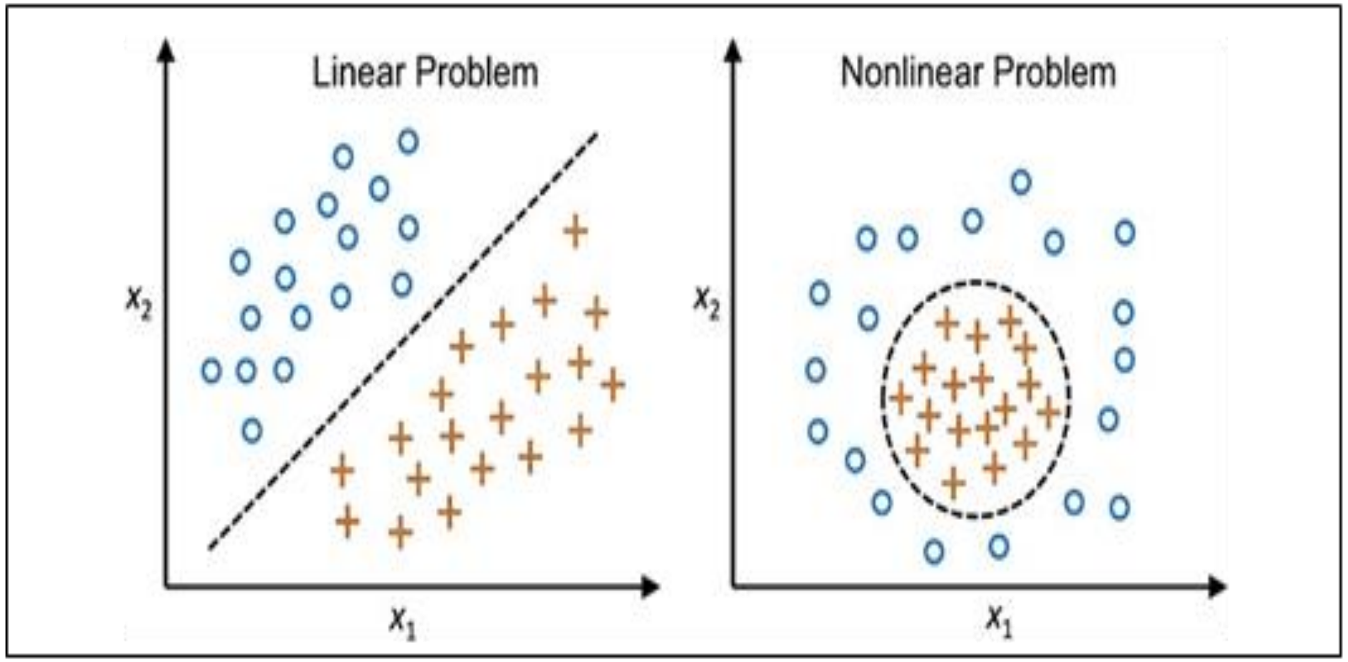


Kernelized PCA

- **Using kernel principal component analysis for nonlinear mappings**
- Many machine learning algorithms make assumptions about the linear separability of the input data. You have learned that the perceptron even requires perfectly linearly separable training data to converge. Other algorithms that we have covered so far assume that the lack of perfect linear separability is due to noise: Adaline, logistic regression, and the (standard) SVM to just name a few.
- However, if we are dealing with nonlinear problems, which we may encounter rather frequently in real-world applications, linear transformation techniques for dimensionality reduction, such as PCA and LDA, may not be the best choice.
- In this section, we will take a look at a kernelized version of PCA, or KPCA, which relates to the concepts of kernel SVM.
- Using KPCA, we will learn how to transform data that is not linearly separable onto a new, lower-dimensional subspace that is suitable for linear classifiers.

Kernelized PCA

- **Using kernel principal component analysis for nonlinear mappings**
- Many machine learning algorithms make assumptions about the linear separability of the input data. You have learned that the perceptron even requires perfectly linearly separable training data to converge. Other algorithms that we have covered so far assume that the lack of perfect linear separability is due to noise: Adaline, logistic regression, and the (standard) SVM to just name a few.



Kernelized PCA

- **Using kernel principal component analysis for nonlinear mappings**
- **Kernel functions and the kernel trick**
- We can tackle nonlinear problems by projecting them onto a new feature space of higher dimensionality where the classes become linearly separable. To transform the examples $\boldsymbol{x} \in R^d$ onto this higher k-dimensional subspace, we defined a nonlinear mapping function, ϕ:
- $\phi: R^d \rightarrow R^k$

- We can think of ϕ as a function that creates nonlinear combinations of the original features to map the original d-dimensional dataset onto a larger, k-dimensional feature space.

- For example, if we had a feature vector $\boldsymbol{x} \in R^d$(x is a column vector consisting of d
- features) with two dimensions (d = 2), a potential mapping onto a 3D-space could be:
- $x = [x_1 \ \ x_2]^T$
- $\downarrow \ \phi$
- $x = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]^T$

Kernelized PCA

- **Using kernel principal component analysis for nonlinear mappings**
- In other words, we perform a nonlinear mapping via KPCA that transforms the data onto a higher-dimensional space. We then use standard PCA in this higher- dimensional space to project the data back onto a lower-dimensional space where the examples can be separated by a linear classifier (under the condition that the examples can be separated by density in the input space). However, one downside of this approach is that it is computationally very expensive, and this is where we use the kernel trick. Using the kernel trick, we can compute the similarity between two high-dimension feature vectors in the original feature space.

Kernelized PCA

- **Implementing a kernel principal component analysis in Python**
- Now, we are going to implement an RBF KPCA in Python following the three steps that summarized the KPCA approach. Using some SciPy and NumPy helper functions, we will see that implementing a KPCA is actually really simple:

```
from scipy.spatial.distance import pdist, squareform from scipy import exp
from scipy.linalg import eigh import numpy as np

def rbf_kernel_pca(X, gamma, n_components): """
RBF kernel PCA implementation.

Parameters
------------
X: {NumPy ndarray}, shape = [n_examples, n_features]

gamma: float
Tuning parameter of the RBF kernel

n_components: int
Number of principal components to return

Returns
------------
X_pc: {NumPy ndarray}, shape = [n_examples, k_features]
Projected dataset

"""
```

```
# Calculate pairwise squared Euclidean distances # in the MxN dimensional dataset.
sq_dists = pdist(X, 'sqeuclidean')

# Convert pairwise distances into a square matrix. mat_sq_dists = squareform(sq_dists)

# Compute the symmetric kernel matrix. K = exp(-gamma * mat_sq_dists)

# Center the kernel matrix. N = K.shape[0]
one_n = np.ones((N,N)) / N
K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

# Obtaining eigenpairs from the centered kernel matrix # scipy.linalg.eigh returns them
in ascending order eigvals, eigvecs = eigh(K)
eigvals, eigvecs = eigvals[::-1], eigvecs[:, ::-1]

# Collect the top k eigenvectors (projected examples) X_pc = np.column_stack([eigvecs[:,
i]
for i in range(n_components)])

return X_pc
```

Kernelized PCA

- **Implementing a kernel principal component analysis in Python**
- One downside of using an RBF KPCA for dimensionality reduction is that we have to specify the γ parameter a priori. Finding an appropriate value for γ requires experimentation and is best done using algorithms for parameter tuning, for example, performing a grid search,

Kernelized PCA

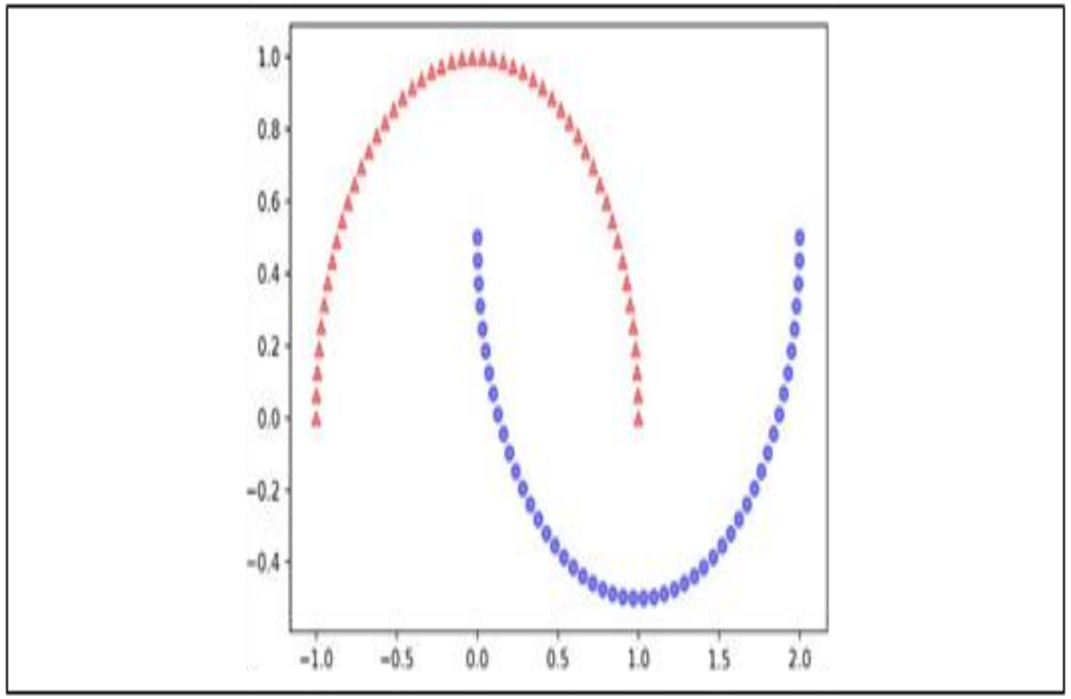
- **Example 1 – separating half-moon shapes**
- Now, let us apply our rbf_kernel_pca on some nonlinear example datasets. We will start by creating a two-dimensional dataset of 100 example points representing two half-moon shapes:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

- For the purposes of illustration, the half-moon of triangle symbols will represent one class, and the half-moon depicted by the circle symbols will represent the examples from another class:

Kernelized PCA

- **Example 1 – separating half-moon shapes**
- For the purposes of illustration, the half-moon of triangle symbols will represent one class, and the half-moon depicted by the circle symbols will represent the examples from another class:



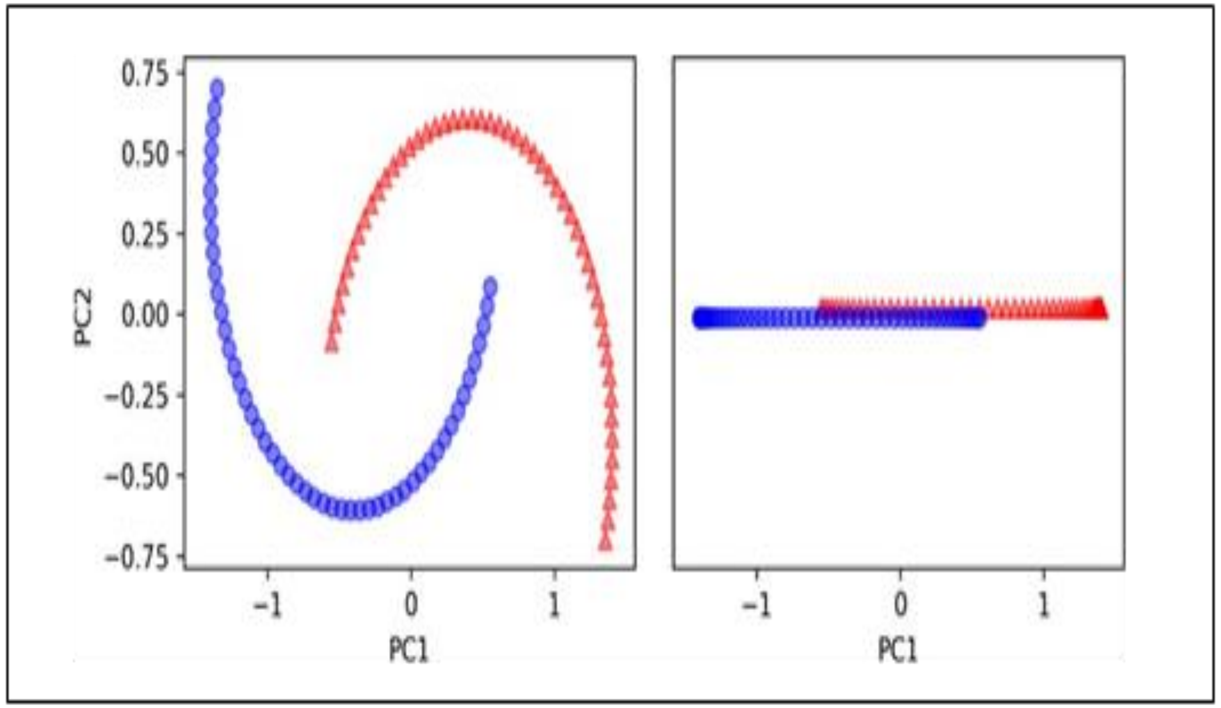
Kernelized PCA

- **Example 1 – separating half-moon shapes**
- Clearly, these two half-moon shapes are not linearly separable, and our goal is to unfold the half-moons via KPCA so that the dataset can serve as a suitable input for a linear classifier. But first, let's see how the dataset looks if we project it onto the principal components via standard PCA:

```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...             color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...             color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

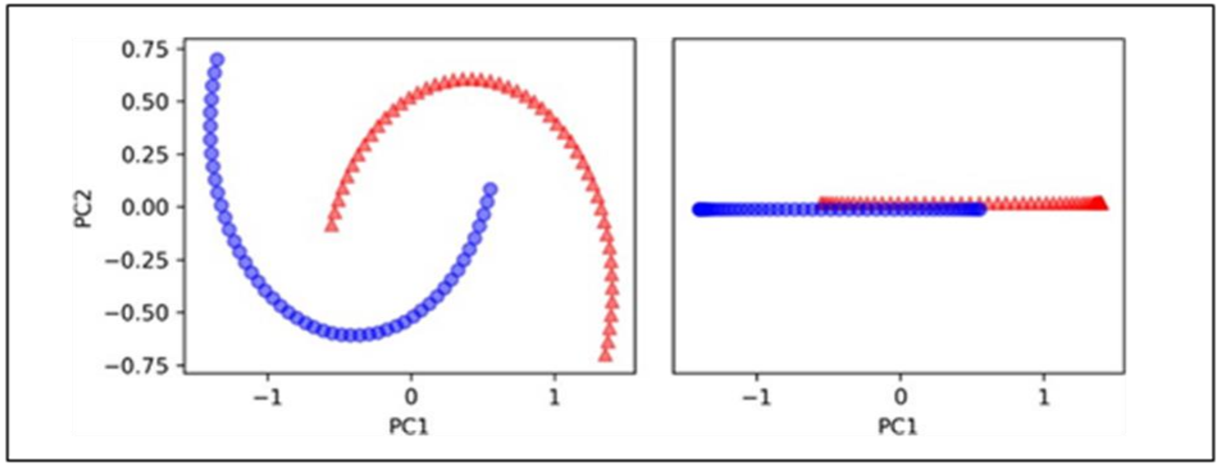
Kernelized PCA

- **Example 1 – separating half-moon shapes**
- Clearly, we can see in the resulting figure that a linear classifier would be unable to
- perform well on the dataset transformed via standard PCA:



Kernelized PCA

- **Example 1 – separating half-moon shapes**
- Note that when we plotted the first principal component only (right subplot), we shifted the triangular examples slightly upward and the circular examples slightly downward to better visualize the class overlap. As the left subplot shows, the original half-moon shapes are only slightly sheared and flipped across the vertical center—this transformation would not help a linear classifier in discriminating between circles and triangles. Similarly, the circles and triangles corresponding to the two half-moon shapes are not linearly separable if we project the dataset onto a one-dimensional feature axis, as shown in the right subplot.



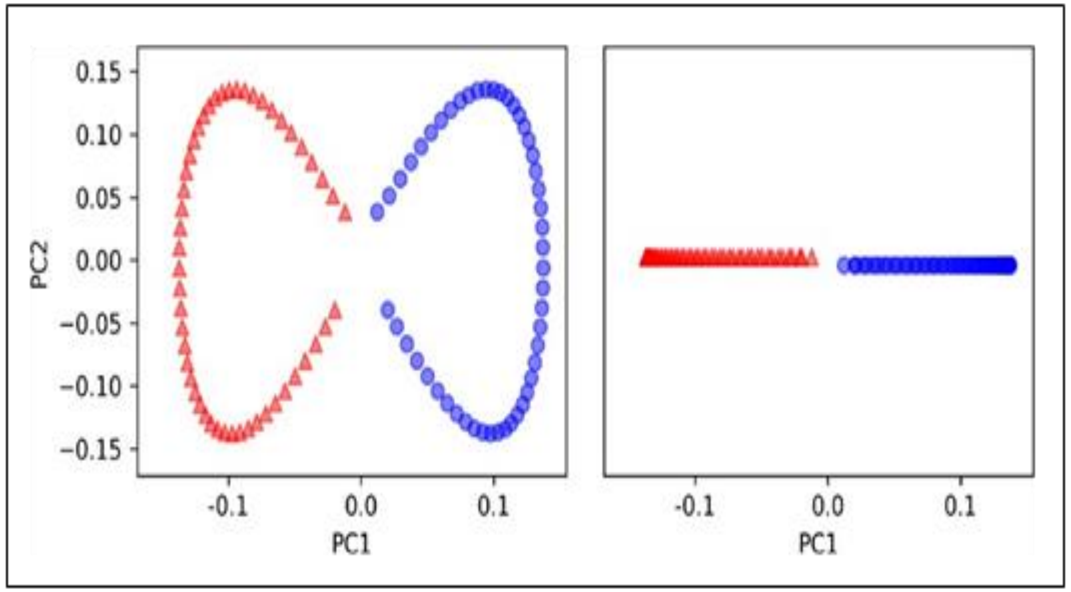
Kernelized PCA

- **Example 1 – separating half-moon shapes**
- Now, let's try out our kernel PCA function, rbf_kernel_pca, which we implemented in the previous subsection:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7, 3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...             color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...             color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

Kernelized PCA

- **Example 1 – separating half-moon shapes**
- We can now see that the two classes (circles and triangles) are linearly well separated so that we have a suitable training dataset for linear classifiers:
- Unfortunately, there is no universal value for the tuning parameter, $\gamma\gamma$, that works well for different datasets. Finding a γ value that is appropriate for a given problem requires experimentation. Here, we will use values for γ that have been found to produce good results.



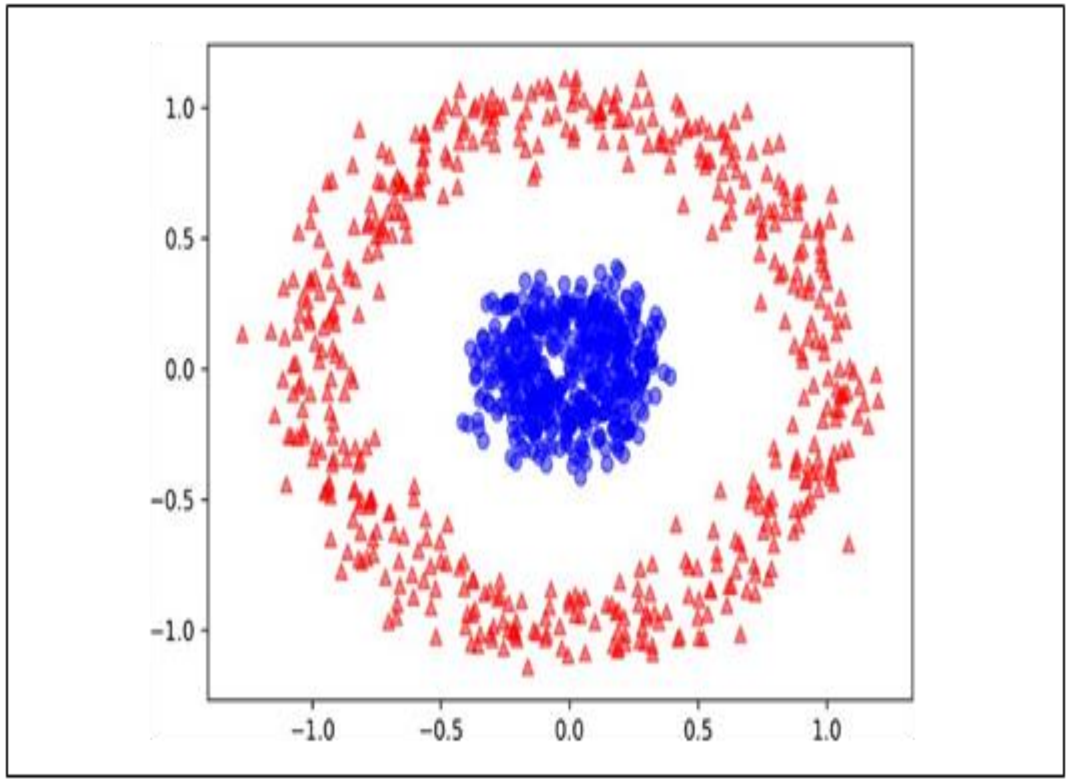
Kernelized PCA

- **Example 2 – separating concentric circles**
- In the previous subsection, we saw how to separate half-moon shapes via KPCA. Since we put so much effort into understanding the concepts of KPCA, let's take a look at another interesting example of a nonlinear problem, concentric circles:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...             random_state=123, noise=0.1,
...             factor=0.2)
>>> plt.scatter(X[y == 0, 0], X[y == 0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y == 1, 0], X[y == 1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

Kernelized PCA

- **Example 2 – separating concentric circles**
- Again, we assume a two-class problem where the triangle shapes represent one class, and the circle shapes represent another class:



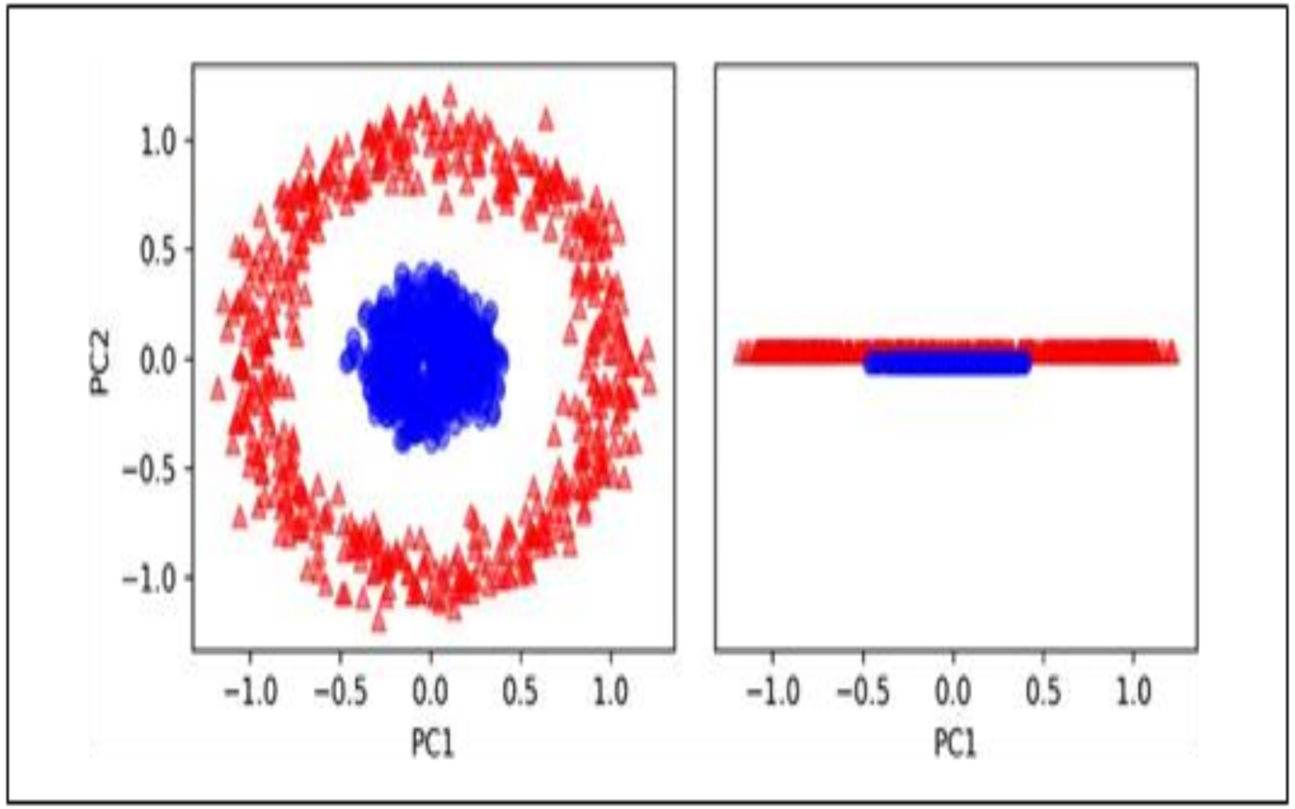
Kernelized PCA

- **Example 2 – separating concentric circles**
- Let's start with the standard PCA approach to compare it to the results of the RBF kernel PCA:

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...             color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...             color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

Kernelized PCA

- **Example 2 – separating concentric circles**
- Again, we can see that standard PCA is not able to produce results suitable for training a linear classifier:



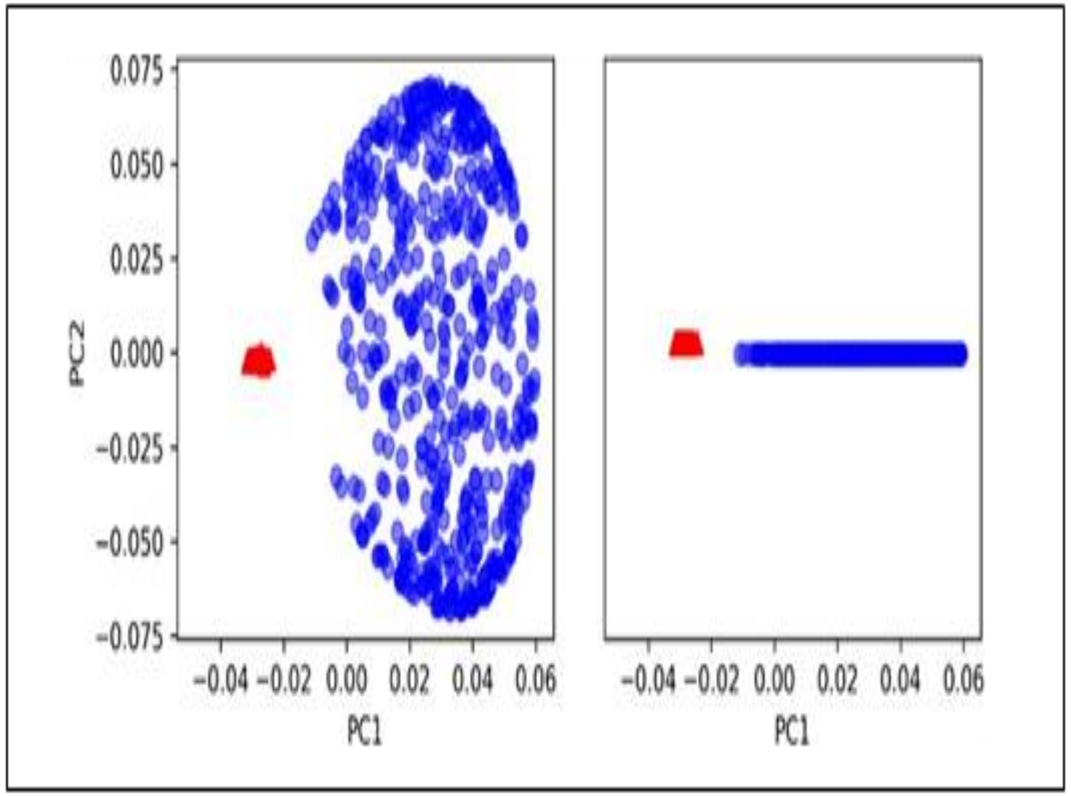
Kernelized PCA

- **Example 2 – separating concentric circles**
- Given an appropriate value for γ, let's see if we are luckier using the RBF KPCA implementation

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...             color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...             color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

Kernelized PCA

- **Example 2 – separating concentric circles**
- Again, the RBF KPCA projected the data onto a new subspace where the two classes become linearly separable:



Q&A