Lecture 4
Principle Components Analysis

Wonjun Lee

Contents

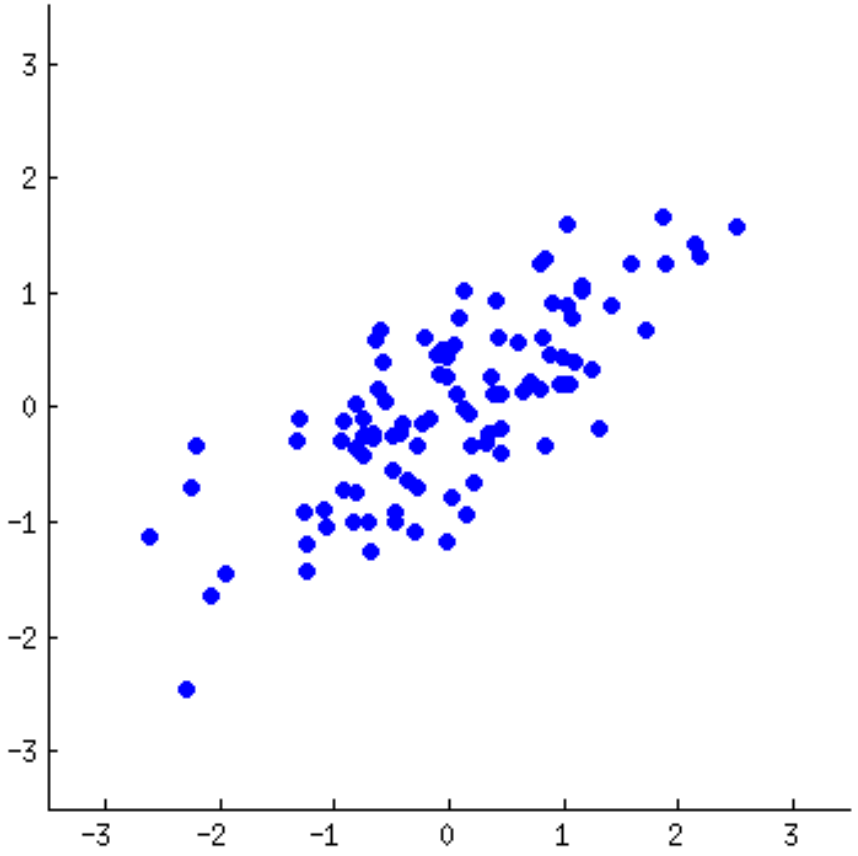
- Principal Component Analysis (PCA)
- Linear Discriminant Analysis (LDA)
- Kernel Principal Component Analysis (KPCA)

Principal Component Analysis

Unsupervised Dimensionality Reduction via PCA

- Feature selection vs. feature extraction
  - ✓Feature selection – maintain the original features
  - ✓Feature extraction – transform or project data onto a new feature space
- Approach to data compression with the goal of maintaining most of the relevant information
- Improve
  - ✓Storage space
  - ✓Computational efficiency of learning algorithm
  - ✓Predictive performance by reducing the curse of dimensionality

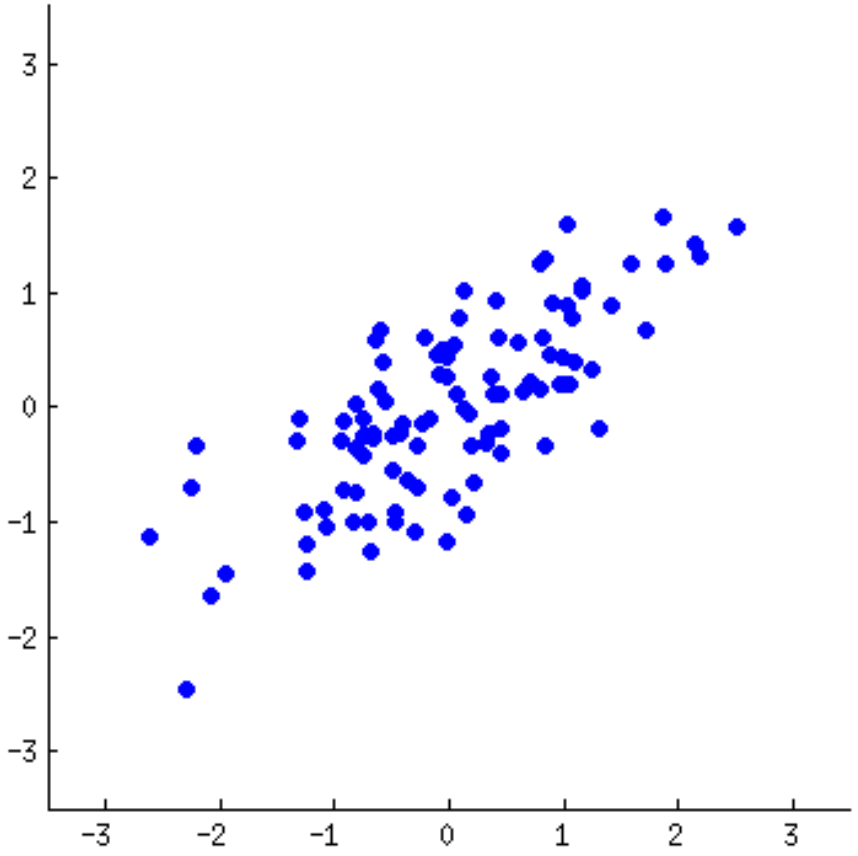
Unsupervised Learning – Principal Component Analysis (PCA)



- A scatter plot of different wines with two characteristics, wine darkness and alcohol content
- The two properties (*x* and *y* on this figure) are correlated

REF: https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues

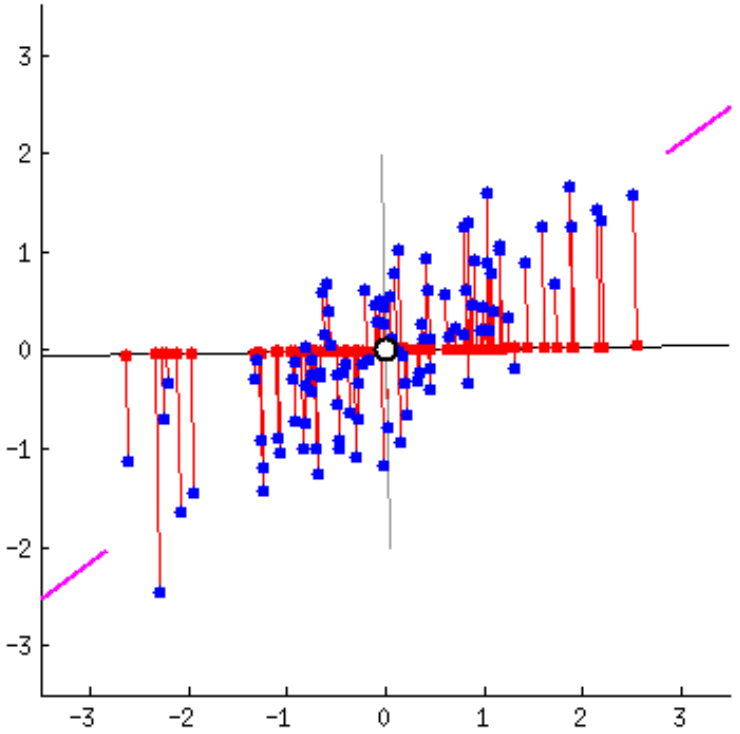
Unsupervised Learning – Principal Component Analysis (PCA)



- A new property can be constructed by drawing a line through the center of this wine cloud and projecting all points onto this line
- This new property will be given by a linear combination w_1x+w_2y where each line corresponds to some particular values of w_1 and w_2

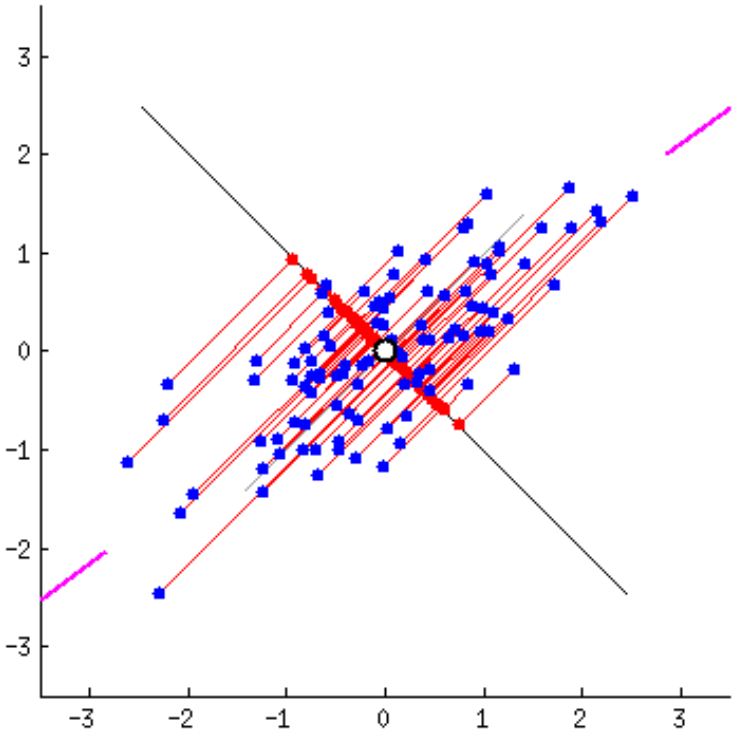
REF: https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues

Unsupervised Learning – PCA



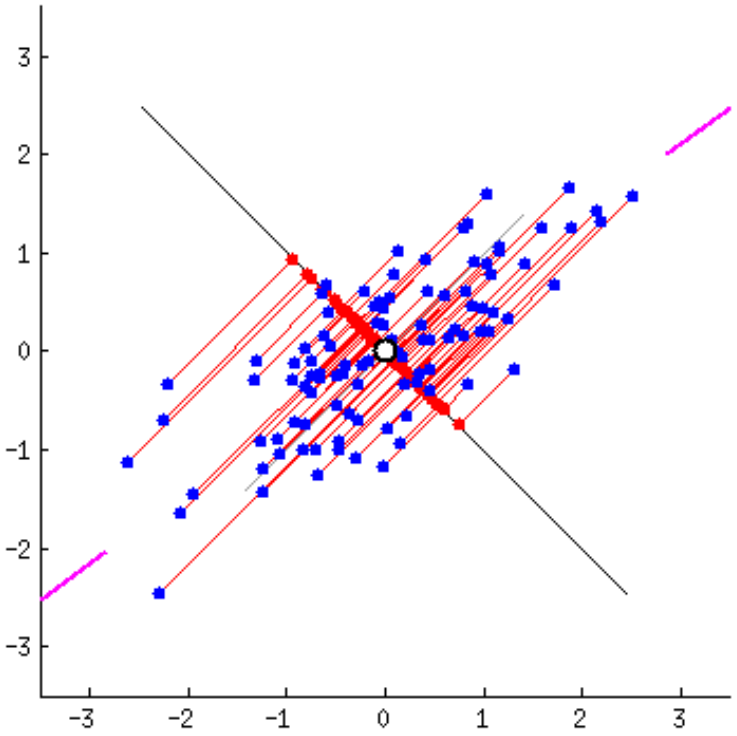
- Red dots are projections of the blue dots
- PCA will find the "best" line according to two different criteria of what is the "best"
  - ✓ First, the **variation** of values along this line should be maximal
  - ✓ Second, the **reconstruction error** (length of the connecting red line) should be minimal

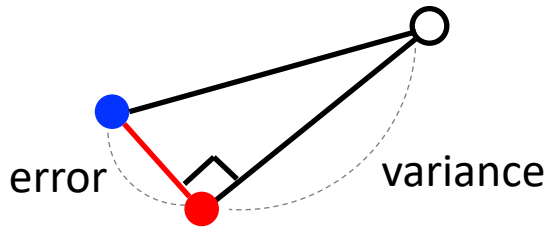
Unsupervised Learning – PCA



- The spread of the red dots is measured as the average squared distance from the center of the wine cloud to each red dot; *variance*
- The total reconstruction error is measured as the average squared length of the corresponding red lines

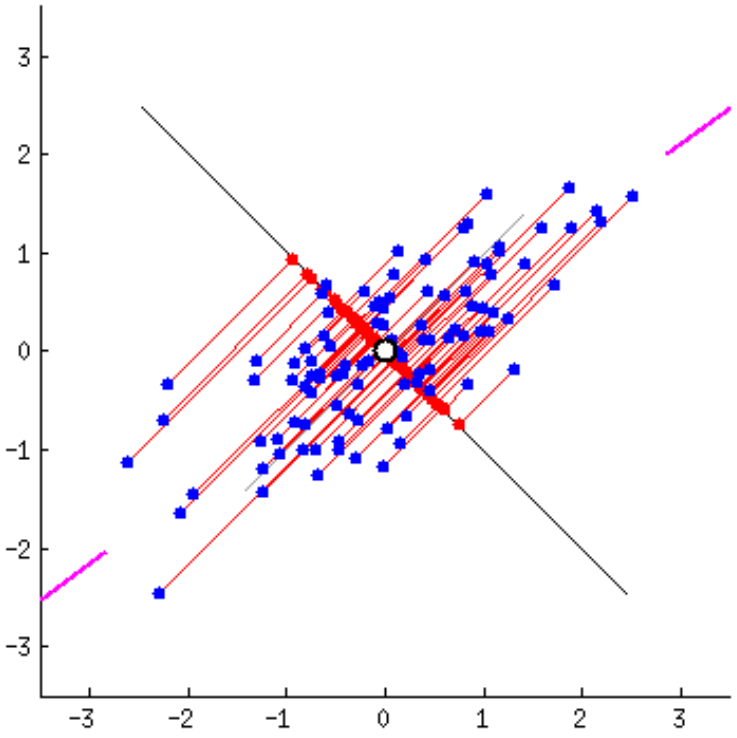
Unsupervised Learning – PCA





- But as the angle between red lines and the black line is always 90◦, the sum of these two quantities is equal to the average squared distance between the center of the wine cloud and each blue dot; Pythagoras theorem
- So the higher the variance the lower the error (because their sum is constant)

Unsupervised Learning – PCA



- Black line is a solid rod and each red line is a spring
- The energy of the spring is proportional to its squared length
- So the rod will orient itself such as to minimize the sum of these squared distances

Main Steps behind PCA

- W = $d \times k$-dim transformation matrix
  - ✓Map a vector x (features) onto a new k-dim feature subspace ($d >> k$)

$$x = [x_1, x_2, \ldots, x_d], \; x \in \mathbb{R}^d$$

$$\mathbf{xW = z}$$

$$\rightarrow z = [z_1, z_2, \ldots, z_k], \; z \in \mathbb{R}^k$$

Main Steps behind PCA

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

Main Steps behind PCA

1. Standardize the *d*-dimensional dataset

- Obtain 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)
```

- Split dataset into training and test datasets using 70% and 30% of data respectively
- Standardize it

Main Steps behind PCA

2. Construct the covariance matrix

- Covariance matrix (d × d)
- Covariance between two features x_j and x_k :

$$\sigma_{jk} = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

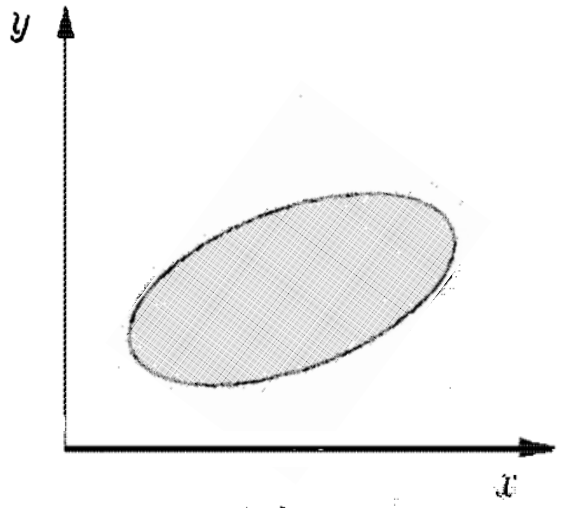
- μ_j, μ_j are sample means of feature j and k

Covariance and Correlation

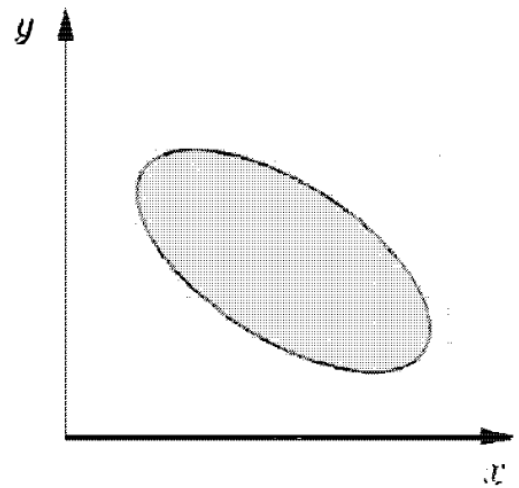
- Quantitative measure of **strength** and **direction** of relationship between two RVs
- Covariance of RV X and Y, cov(X,Y):

$$\operatorname{cov}(X,Y) = \mathbf{E}\Big[\big(X - \mathbf{E}[X]\big)\big(Y - \mathbf{E}[Y]\big)\Big]$$

- Positive → values of X-E[X] and Y-E[Y] obtained in a single experiment "tend" to have the <u>same</u> sign
  → (a)
- Negative → values of X-E[X] and Y-E[Y] obtained in a single experiment "tend" to have the <u>opposite</u> sign
  → (b)



(a)



(b)

Main Steps behind PCA

- Covariance matrix of three features:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

- Covariance matrix of Wine example: 13×13

$$\sum = \begin{bmatrix} \sigma_1^2 & \cdots & \sigma_{1\ 13} \\ \vdots & \ddots & \vdots \\ \sigma_{13\ 1} & \cdots & \sigma_{13}^2 \end{bmatrix}$$

Main Steps behind PCA

3. Decompose the covariance matrix into its eigenvectors and eigenvalues

- Eigenvectors of covariance matrix = principal components (**direction** of maximum variance)
- Eigenvalues of covariance matrix = **magnitude** (squared sum of distance from center to each projected point)
- Ex) Wine example: 13 eigenvectors and eigenvalues from 13×13 cov. matrix:

$$\sum v = \lambda v \rightarrow \begin{bmatrix} \sigma_1^2 & \cdots & \sigma_{1\,13} \\ \vdots & \ddots & \vdots \\ \sigma_{13\,1} & \cdots & \sigma_{13}^2 \end{bmatrix} \underbrace{\begin{bmatrix} v_{11} & \cdots & v_{1\,13} \\ \vdots & \ddots & \vdots \\ v_{13\,1} & \cdots & v_{13\,13} \end{bmatrix}}_{\text{Eigenvectors}} = \lambda \underbrace{\begin{bmatrix} v_{11} & \cdots & v_{1\,13} \\ \vdots & \ddots & \vdots \\ v_{13\,1} & \cdots & v_{13\,13} \end{bmatrix}}_{\text{Eigenvectors}}$$

Eigenvalues

Main Steps behind PCA

3. Decompose the covariance matrix into its eigenvectors and eigenvalues

```
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 ]
```

Total and Explained Variance

4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors

- *Variance Explained Ratios*: To get *k* most informative eigenvectors
- The variance explained ratio of an eigenvalue, λ_j, is simply the fraction of an eigenvalue, λ_j, and the total sum of the eigenvalues:

$$\text{Explained variance ratio} = \frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

Total and Explained Variance

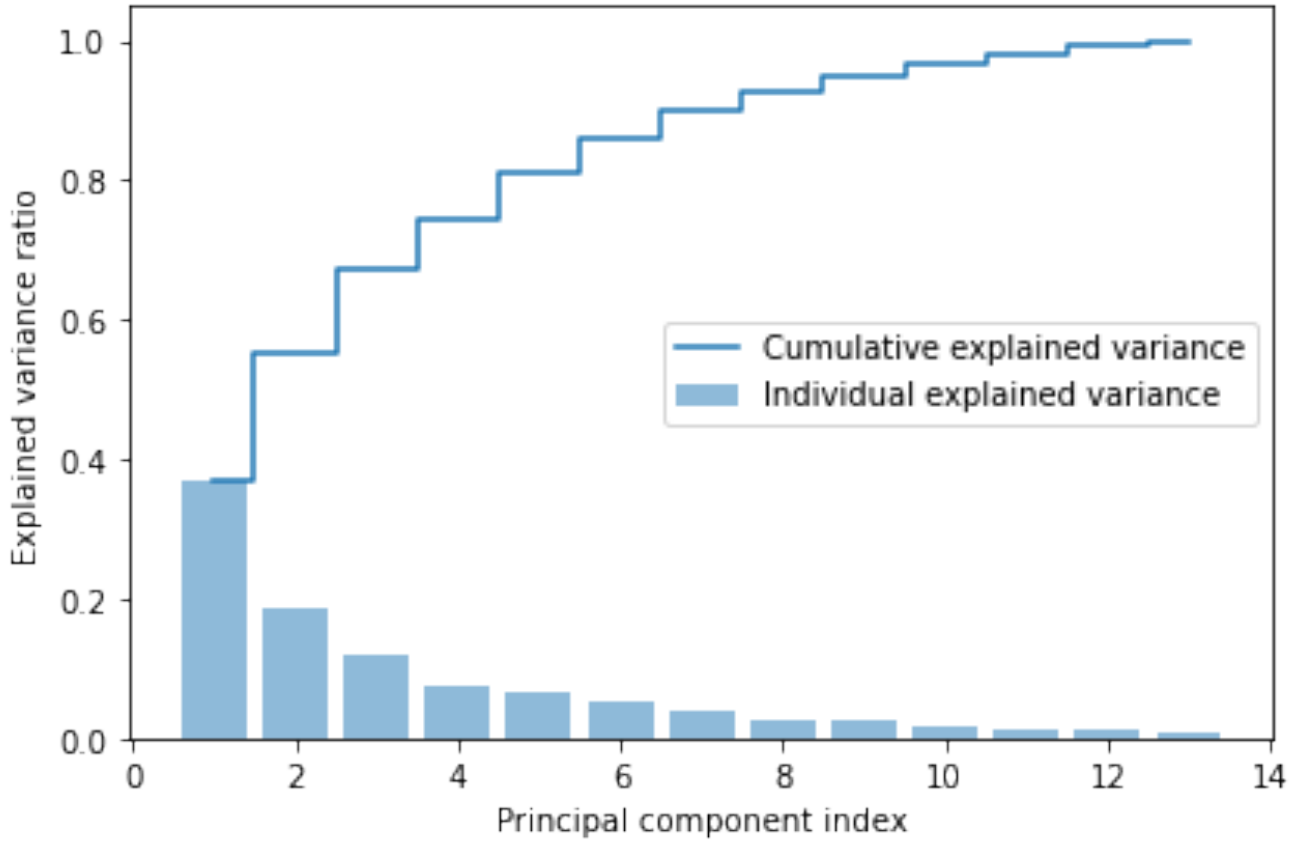
4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors

```
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.savefig('images/05_02.png', dpi=300)
plt.show()
```

Total and Explained Variance

4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors



Feature Transformation

5. Select *k* eigenvectors, which correspond to the *k* largest eigenvalues, where *k* is the dimensionality of the new feature subspace ($k \leq d$)

- Transform the Wine dataset onto the new principal component axes
- Sort eigenvalues in decreasing order

```
# 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)
```

Feature Transformation

6. Construct a projection matrix, ***W***, from the "top" *k* eigenvectors

- Collect the two eigenvectors that correspond to the two largest eigenvalues, to capture about 60 percent of the variance in this dataset
- In practice, # of principal components has to be determined by a tradeoff between computational efficiency and the performance of the classifier
- We have created a 13 × 2-dimensional projection matrix, ***W***, from the top two eigenvectors

```
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]]
```

Feature Transformation

7. Transform the *d*-dimensional input dataset, ***X***, using the projection matrix, ***W***, to obtain the new *k*-dimensional feature subspace

- $x' = xW$

```
print(X_train_std[0])
X_train_std[0].dot(w)
```

```
[ 0.71225893  2.22048673 -0.13025864  0.05962872 -0.50432733 -0.52831584
 -1.24000033  0.84118003 -1.05215112 -0.29218864 -0.20017028 -0.82164144
 -0.62946362]

array([2.38299011, 0.45458499])
```

- $X' = XW$

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

PCA in scikit-learn

```
from sklearn.decomposition import PCA

pca = PCA()
X_train_pca = pca.fit_transform(X_train_std)
pca.explained_variance_ratio_
```

```
pca = PCA(n_components=None)
X_train_pca = pca.fit_transform(X_train_std)
pca.explained_variance_ratio_
```

→ All principal components are kept (default is none)

```
pca = PCA(n_components=2)
```

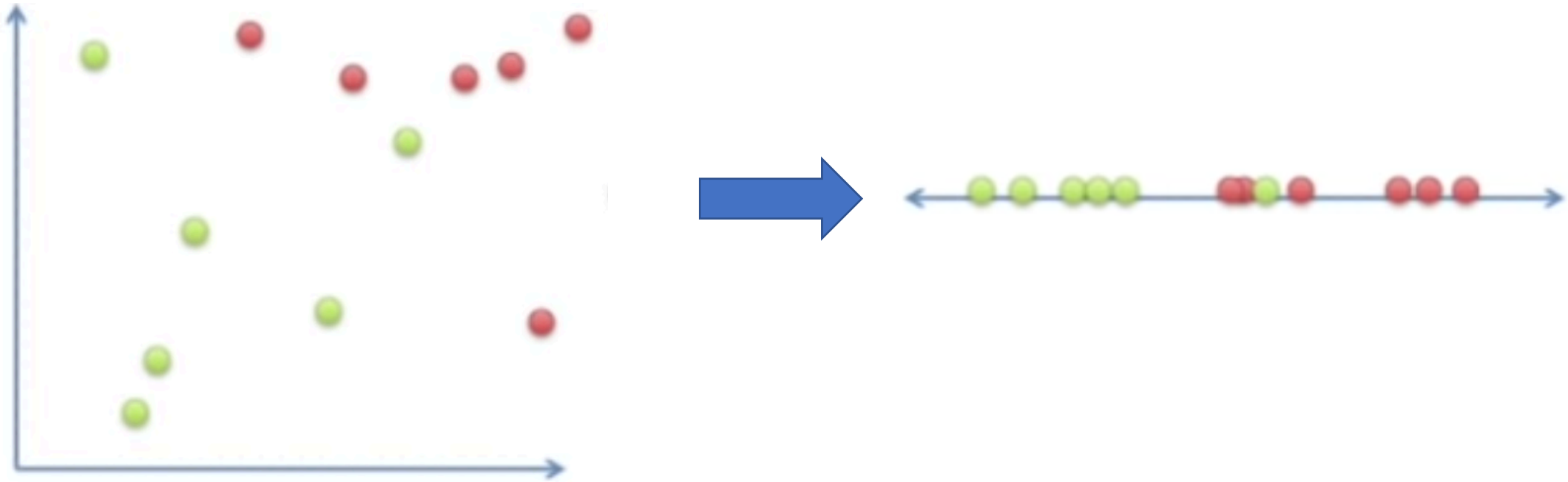
Linear Discriminant Analysis

Supervised Data Compression via Linear Discriminant Analysis (LDA)

- The goal in LDA – find the feature subspace that optimizes class separability
  - ✓PCA – find the orthogonal component axes of maximum variance in a dataset
- LDA takes class label into account (represented in the form of the mean vectors)

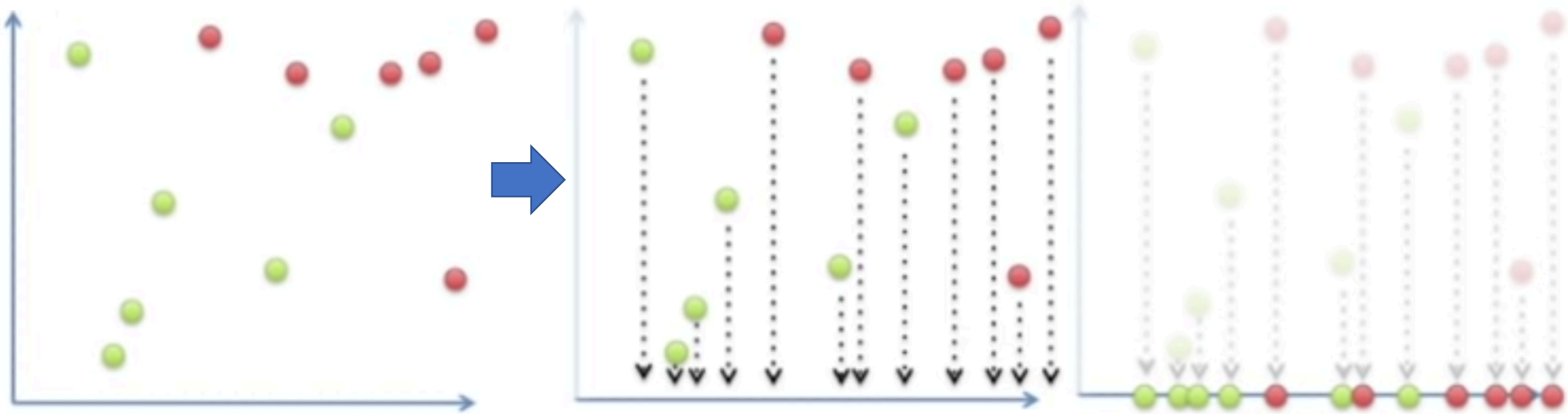
Supervised Data Compression via Linear Discriminant Analysis (LDA)

- Reducing a 2-D to a 1-D – What is the best way to reduce the dimension?



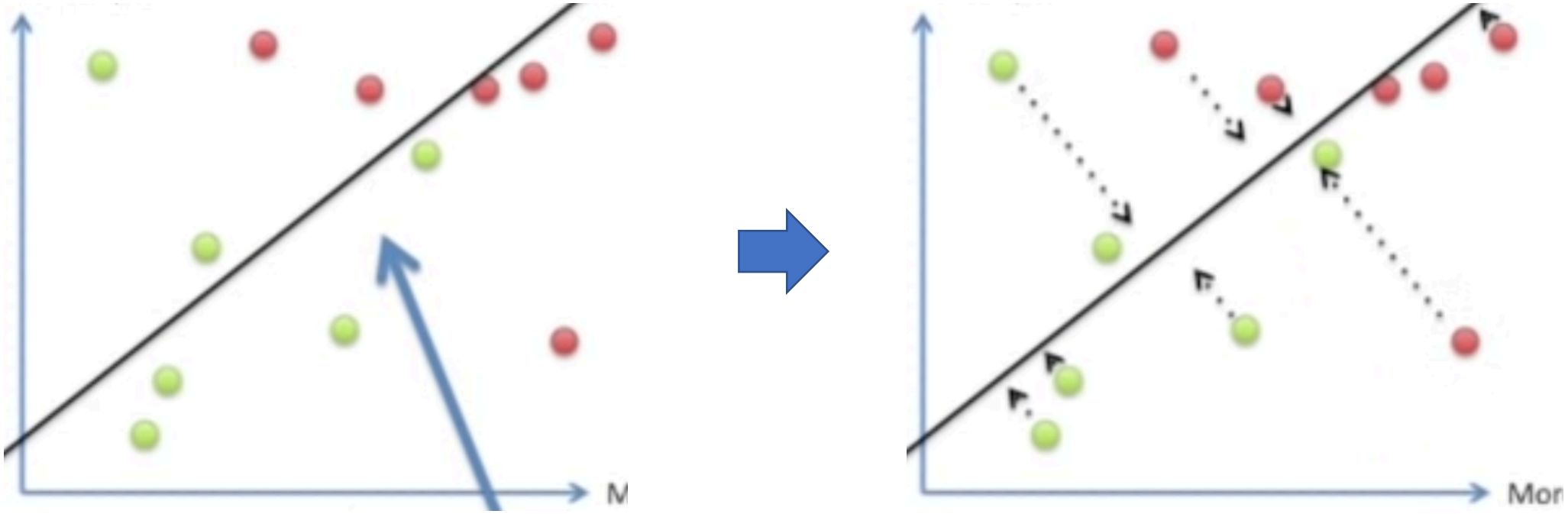
Supervised Data Compression via Linear Discriminant Analysis (LDA)

- One bad option would be to ignore Y axis



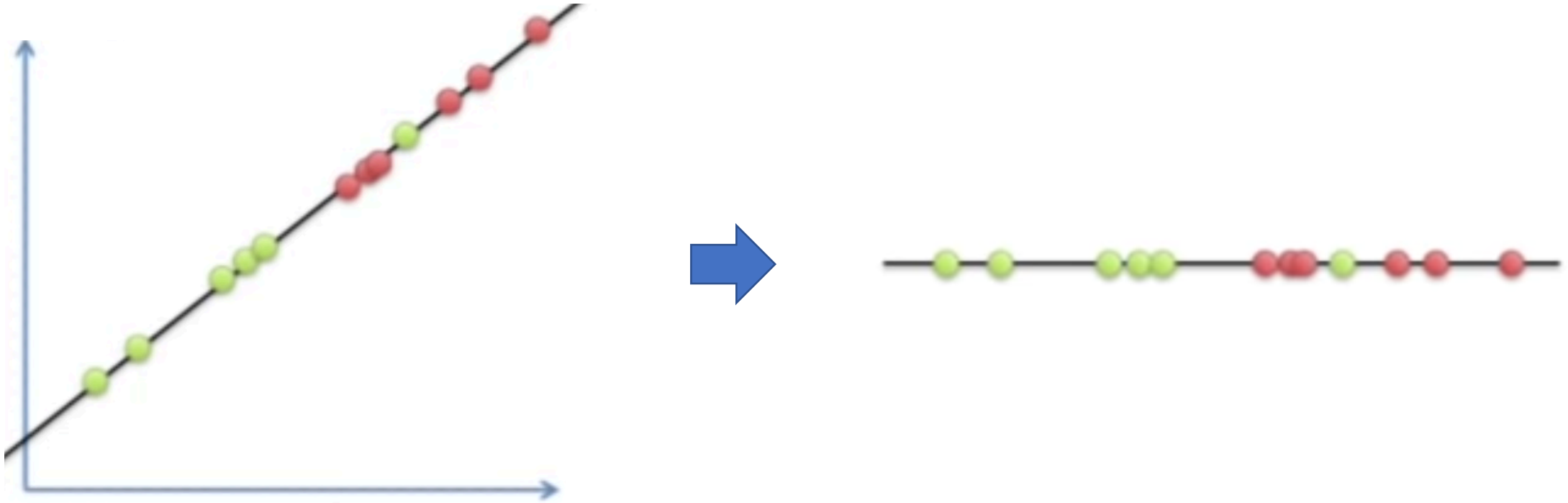
Supervised Data Compression via Linear Discriminant Analysis (LDA)

- LDA uses both axes to create a new axis and projects the data onto this new axis in a way to maximize the separation of the two categories



Supervised Data Compression via Linear Discriminant Analysis (LDA)

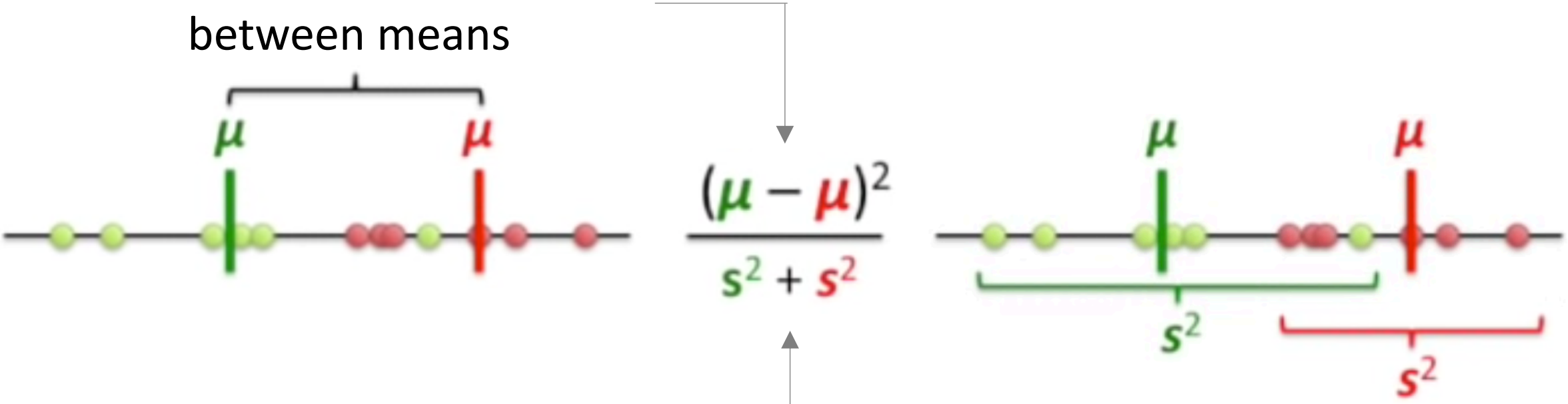
- LDA uses both axes to create a new axis and projects the data onto this new axis in a way to maximize the separation of the two categories



Supervised Data Compression via Linear Discriminant Analysis (LDA)

- New axis is created accordingly to two criteria

1) Maximize the distance between means



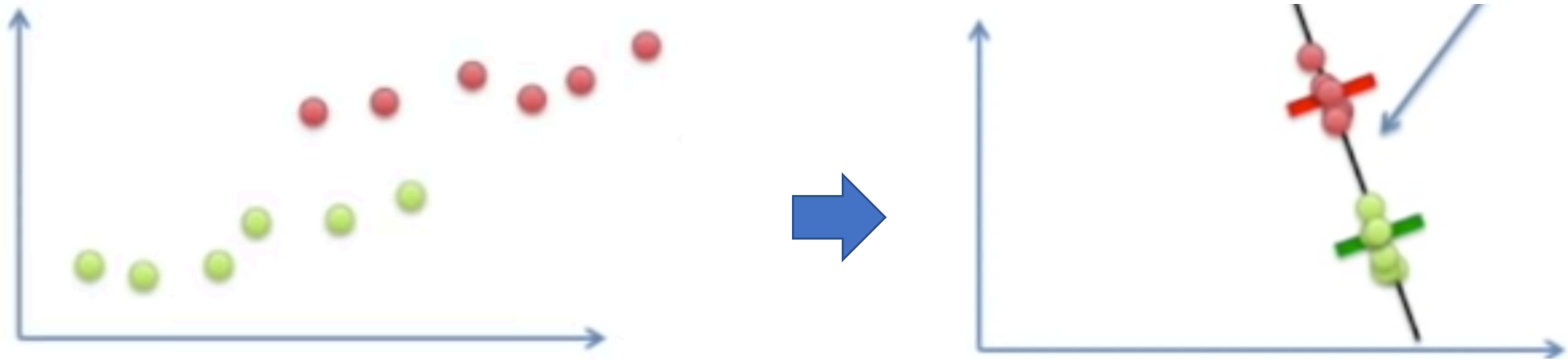
2) Minimize the variation (scatter) within each category

Supervised Data Compression via Linear Discriminant Analysis (LDA)

- If we only maximize the distance between means: separation is not great

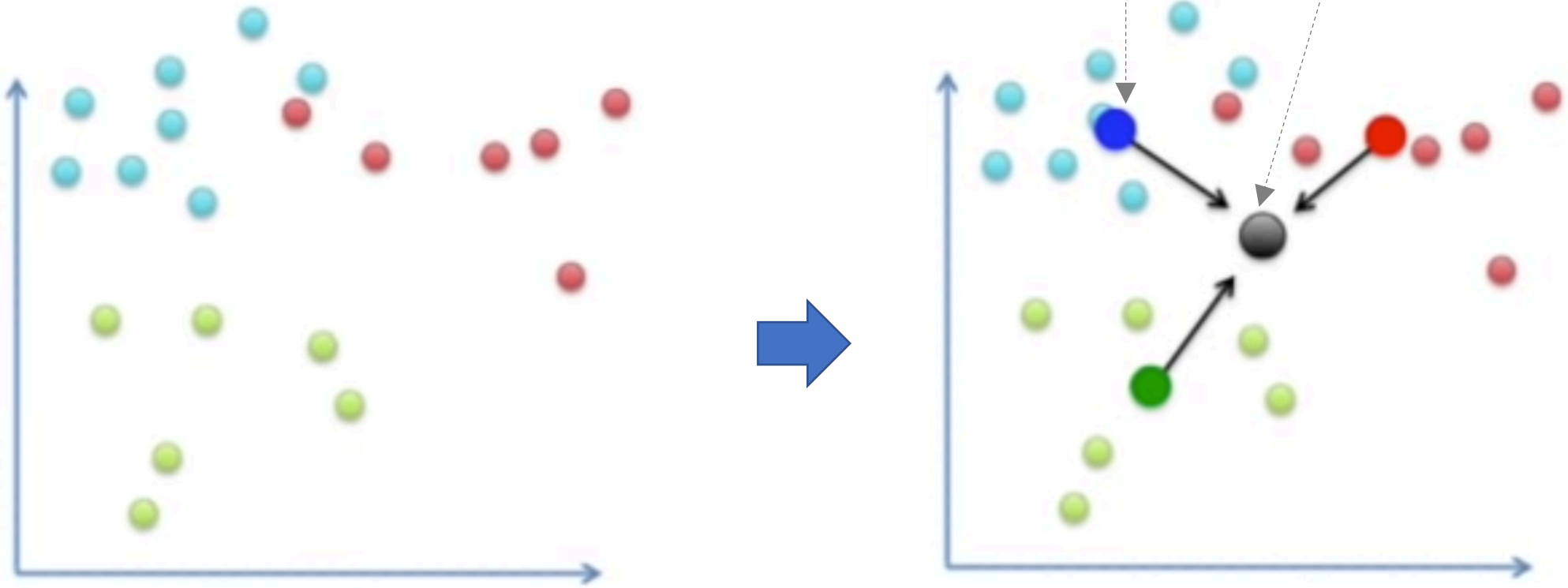
Supervised Data Compression via Linear Discriminant Analysis (LDA)

- If we optimize the distance between means and scatter: we get nice separation



Supervised Data Compression via Linear Discriminant Analysis (LDA)

- LDA for 3 classes



Supervised Data Compression via Linear Discriminant Analysis

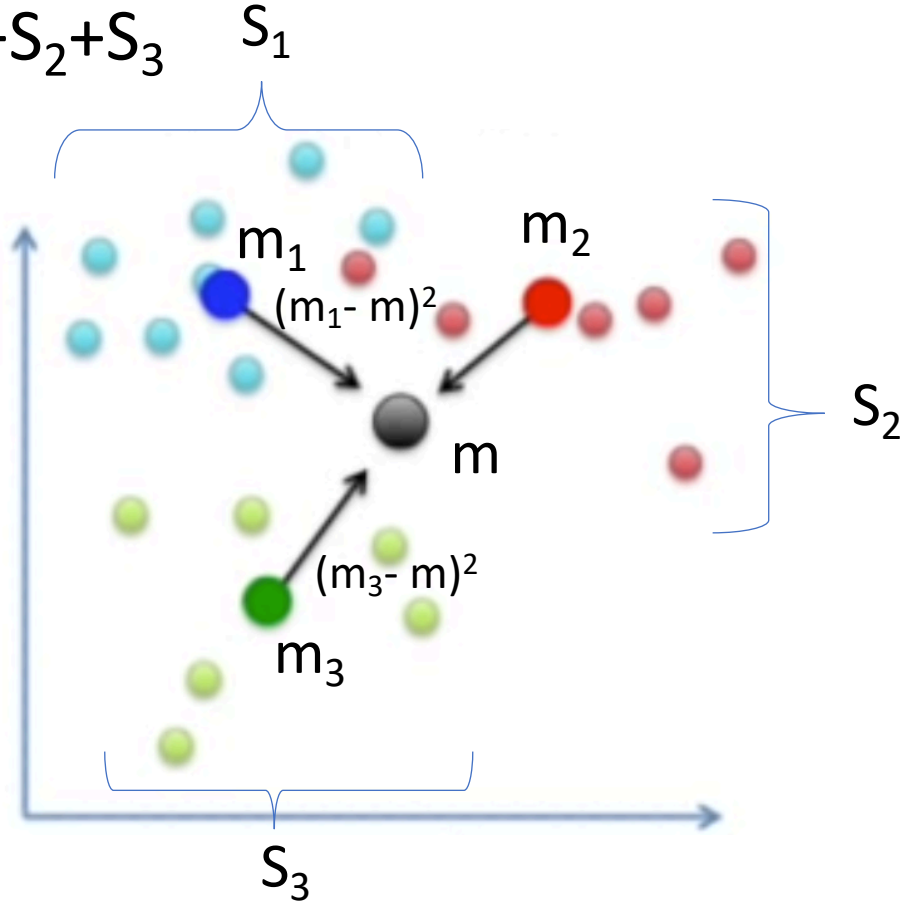
- The goal in LDA – find the feature subspace that optimizes class separability
  - ✓ PCA – find the orthogonal component axes of maximum variance in a dataset

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, $\boldsymbol{S}_{\mathrm{w}}^{-1}\boldsymbol{S}_{\mathrm{B}}$
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, $\boldsymbol{W}$; the eigenvectors are the columns of this matrix
7. Project the examples onto the new feature subspace using the transformation matrix, $\boldsymbol{W}$

Supervised Data Compression via Linear Discriminant Analysis

Step 2. 3:

- Within-class scatter matrix (S_W) = $\sum_{i=1}^{c} S_i$ = S_1+S_2+S_3
  - ✓Mean vector (m_i) = $\frac{1}{n_i}\sum_{x \in D_i} x_m$
  - ✓S_i = $\sum_{x \in D_i}(x - m_i)(x - m_i)^T$
- Between-class scatter matrix (S_B) = $\sum_{i=1}^{c} n_i(m_i - m)(m_i - m)^T$
  - ✓m is overall mean



Supervised Data Compression via Linear Discriminant Analysis

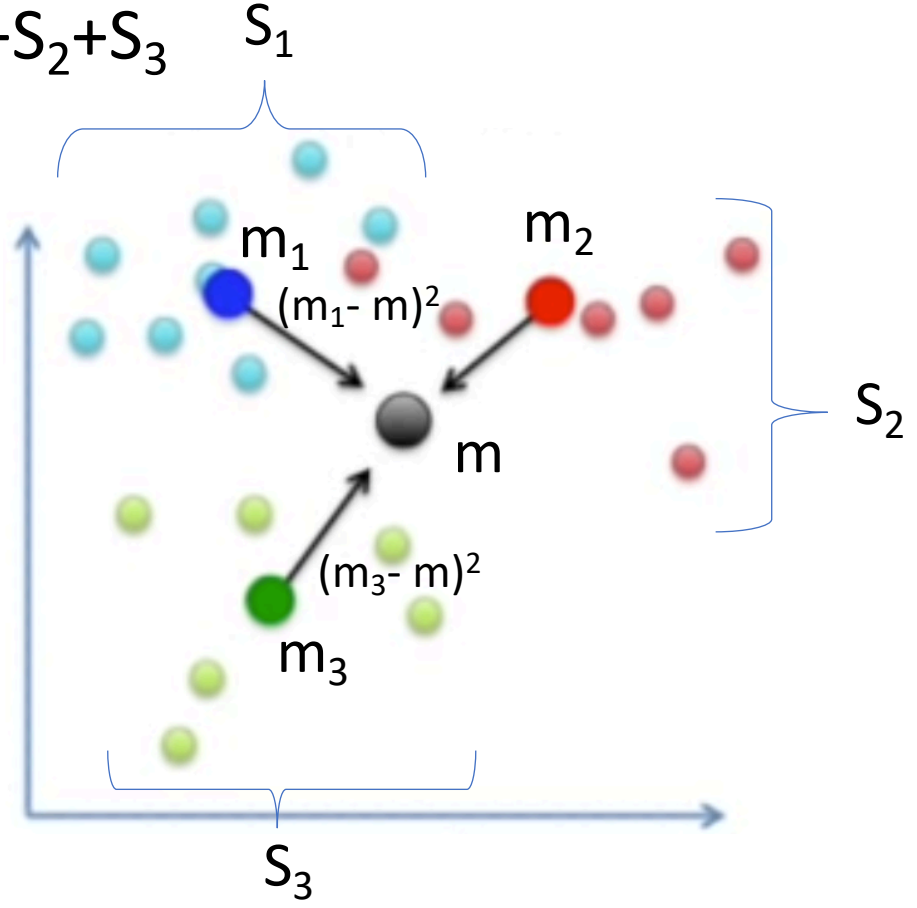
Step 2. 3:

- Within-class scatter matrix (S_W) = $\sum_{i=1}^{c} S_i$ = S_1+S_2+S_3
  - ✓ Mean vector (m_i) = $\frac{1}{n_i}\sum_{x \in D_i} x_m$
  - ✓ $S_i = \sum_{x \in D_i}(x - m_i)(x - m_i)^T$

```
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]))
```

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix}^T \quad i \in \{1, 2, 3\}$$

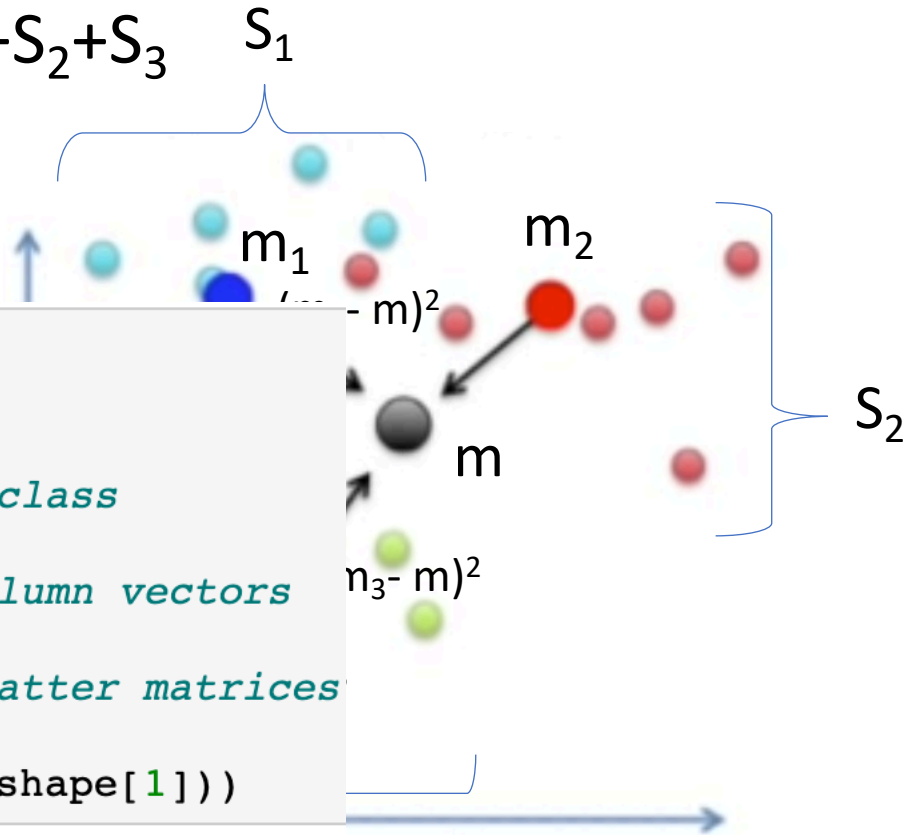
```
MV 1: [ 0.9066 -0.3497  0.3201 -0.7189  0.5056  0.8807  0.9589 -0.5516  0.5416
  0.2338  0.5897  0.6563  1.2075]

MV 2: [-0.8749 -0.2848 -0.3735  0.3157 -0.3848 -0.0433  0.0635 -0.0946  0.0703
 -0.8286  0.3144  0.3608 -0.7253]

MV 3: [ 0.1992  0.866   0.1682  0.4148 -0.0451 -1.0286 -1.2876  0.8287 -0.7795
  0.9649 -1.209  -1.3622 -0.4013]
```



Supervised Data Compression via Linear Discriminant Analysis

Step 2. 3:

- Within-class scatter matrix (S_W) = $\sum_{i=1}^{c} S_i$ = S_1+S_2+S_3
  - ✓Mean vector (m_i) = $\frac{1}{n_i}\sum_{x \in D_i} x_m$
  - ✓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))  # scatter matrix for each class
    for row in X_train_std[y_train == label]:
        row, mv = row.reshape(d, 1), mv.reshape(d, 1)  # make column vectors
        class_scatter += (row - mv).dot((row - mv).T)
    S_W += class_scatter                             # sum class scatter matrices

print('Within-class scatter matrix: %sx%s' % (S_W.shape[0], S_W.shape[1]))
```

```
Within-class scatter matrix: 13x13
```

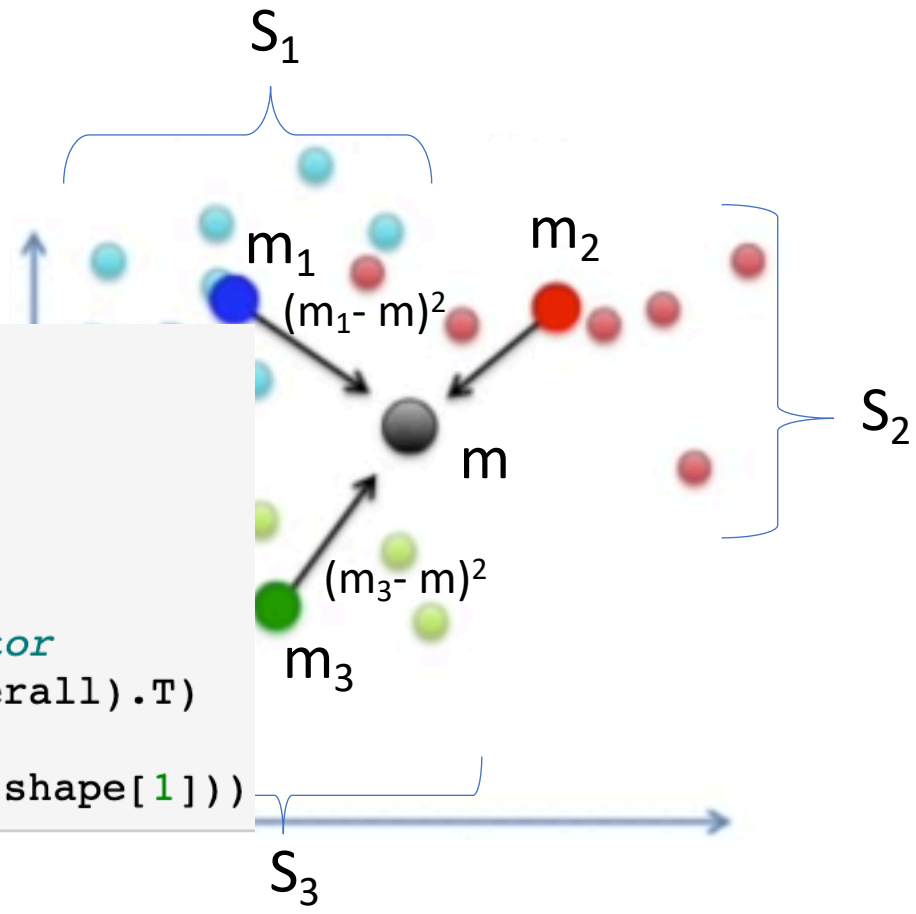
Supervised Data Compression via Linear Discriminant Analysis

Step 2. 3:

- Between-class scatter matrix (S_B) =

$$\sum_{i=1}^{c} n_i (m_i - m)(m_i - m)^T$$

✓ m is overall mean




```
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)  # make column vector
    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
```

Selecting Linear Discriminants For the New Feature Subspace

- Step 4: Solve the generalized eigenvalue problem of $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$

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

Supervised Data Compression via Linear Discriminant Analysis

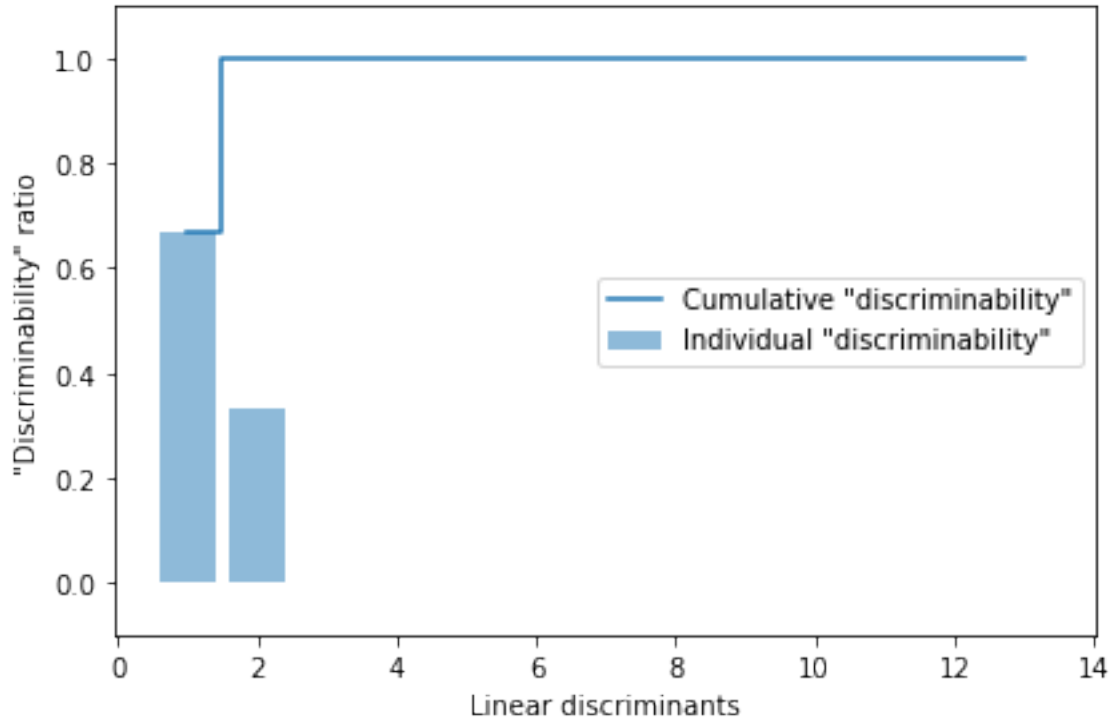
- Step 5: Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors

```
# 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 = sorted(eigen_pairs, key=lambda k: k[0], reverse=True)

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

print('Eigenvalues in descending order:\n')
for eigen_val in eigen_pairs:
    print(eigen_val[0])
```

Projecting Examples onto the New Feature Space

- Step 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

```
Eigenvalues in descending order:

349.61780890599397
172.7615221897938
3.3428382148413644e-14
2.842170943040401e-14
2.5545786180111397e-14
1.753393918073425e-14
1.753393918073425e-14
1.657919399596089e-14
1.657919399596089e-14
8.242524002707208e-15
8.242524002707208e-15
6.36835506006027e-15
2.97463437554573e-15
```



Projecting Examples onto the New Feature Space

- Step 7: Project the examples onto the new feature subspace using the transformation matrix, ***W*** → stack the two most discriminative eigenvector columns to create, ***W***

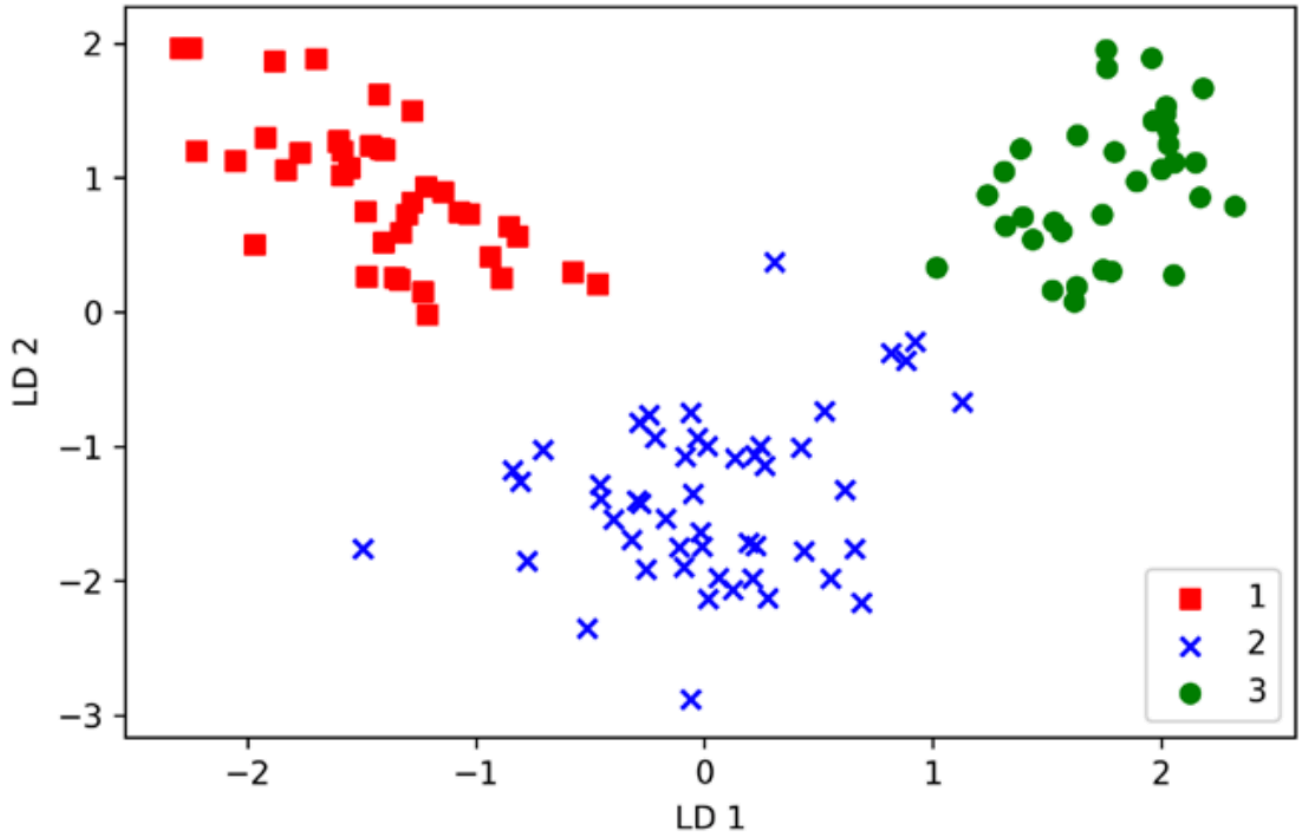
```
w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
               eigen_pairs[1][1][:, np.newaxis].real))
print('Matrix W:\n', w)
```

```
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]]
```

Projecting Examples onto the New Feature Space

- Step 7: Project the examples onto the new feature subspace using the transformation matrix, $\boldsymbol{W}$

$$\boldsymbol{X' = XW}$$



LDA via scikit-learn

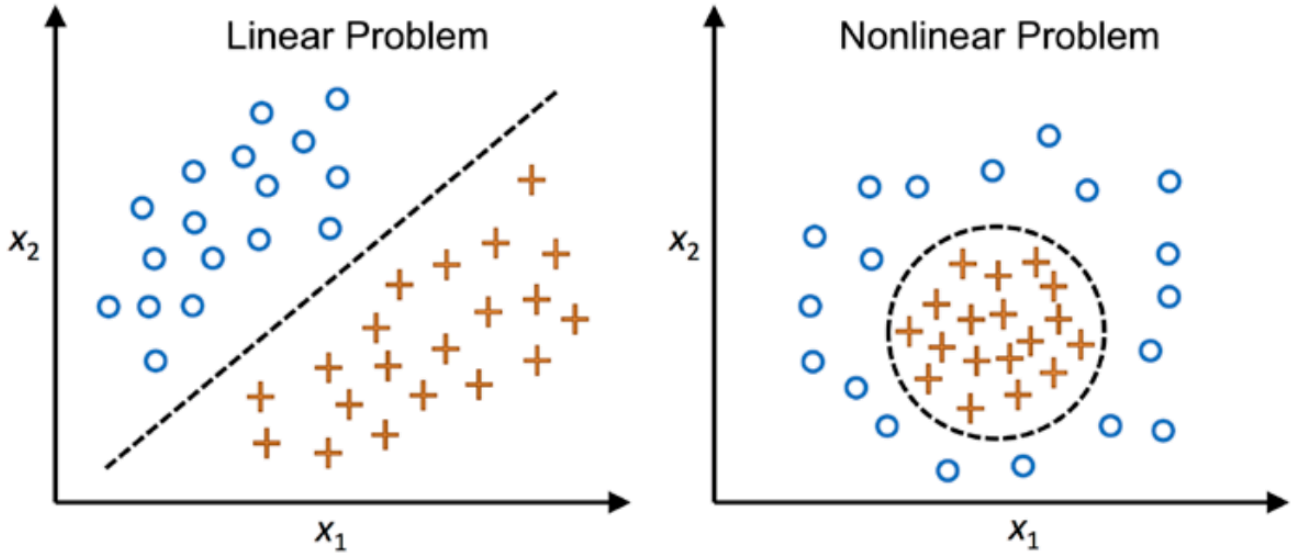
- LDA class implemented in scikit-learn:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

lda = LDA(n_components=2)
X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Kernel Principal Component Analysis

Using Kernel PCA for Nonlinear Mappings

- PCA, LDA – linear transformation techniques
- Many Nonlinear problems in real-world – linear transformation for dimensionality reduction may not be the best choice
- KPCA - transforms data that is not linearly separable, onto a new, lower-dimensional subspace that is suitable for linear classifiers



Kernel Functions and Kernel Trick

- Nonlinear problems – project onto a new feature space of higher dimensionality where the classes become linearly separable
- Mapping function, ϕ: $\mathbb{R}^d \longrightarrow \mathbb{R}^k \qquad (k \gg d)$
- x $\in \mathbb{R}^d$ (x is a column vector consisting of d features)

$$\boldsymbol{x} = [x_1, x_2]^T$$

$$\downarrow \phi$$

$$\boldsymbol{z} = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]^T$$

- 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

Kernel Functions and Kernel Trick

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n} x_j^{(i)} x_k^{(i)}$$

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}^{(i)}\, \boldsymbol{x}^{(i)^T}$$

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^T$$

Deriving the Kernel Matrix

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

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^T \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow v = \frac{1}{n\lambda}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^T \boldsymbol{v} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{a}^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right) = \phi(\boldsymbol{X})^T \boldsymbol{a}$$

Since $\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$, we get:

$$\frac{1}{n}\phi(\boldsymbol{X})^T \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T \boldsymbol{a} = \lambda \phi(\boldsymbol{X})^T \boldsymbol{a}$$

Multiplying it by $\phi(\boldsymbol{X})$ on both sides

$$\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T \boldsymbol{a} = \lambda \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T \boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T \boldsymbol{a} = \lambda \boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\boldsymbol{K}\boldsymbol{a} = \lambda \boldsymbol{a}$$

$$\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T$$

Kernel Functions and Kernel Trick

- Use the kernel trick to avoid calculating the pairwise dot products of the examples, $\boldsymbol{x}$, under ϕ explicitly by using a kernel function, κ, so that we don't need to calculate the eigenvectors explicitly:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$$

- Basically, the kernel function (or simply kernel) can be understood as a function that calculates a dot product between two vectors—a measure of similarity
  - ✓Polynomial kernel
  - ✓Hyperbolic tangent
  - ✓Radial Basis Function (RBF)

Kernel Functions and Kernel Trick

- Radial Basis Function (RBF)

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2}{2\sigma^2}\right)$$

$$\gamma = \frac{1}{2\sigma^2}$$

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2\right)$$

Three Steps to Implement an RBF KPCA

1. Compute the kernel (similarity) matrix, K, where we need to calculate :

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma \left\| \boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)} \right\|^2\right)$$

   ✓Do this for each pair of examples:

$$\boldsymbol{K} = \begin{bmatrix} \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(n)}\right) \\ \kappa\left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right) & \left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(n)}\right) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(n)}\right) \end{bmatrix}$$

2. Center the kernel matrix, K, using following equation:

$$\boldsymbol{K}' = \boldsymbol{K} - \mathbf{1}_n \boldsymbol{K} - \boldsymbol{K}\mathbf{1}_n + \mathbf{1}_n \boldsymbol{K} \mathbf{1}_n$$

3. Collect top k eigenvectors of the centered kernel matrix based on their corresponding eigenvalues (decreasing order)

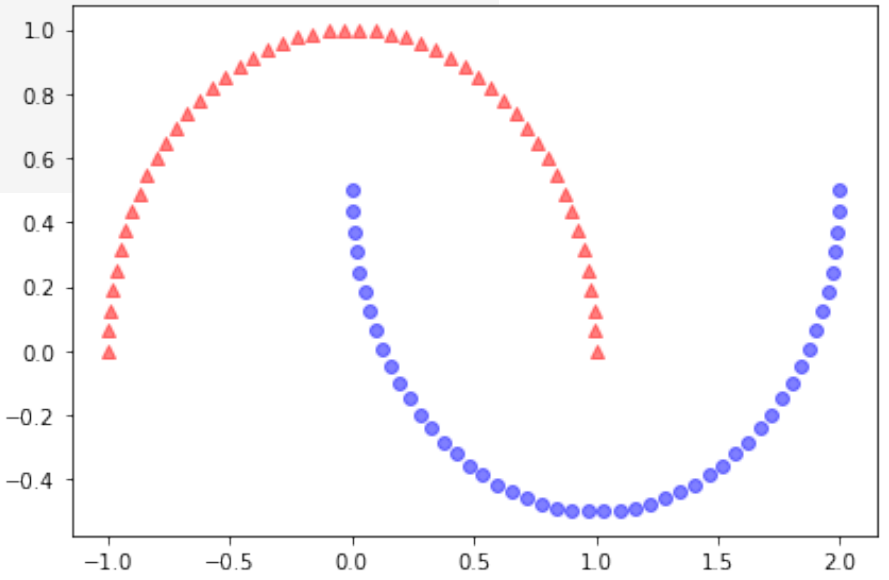
Implementing a KPCA

- http://127.0.0.1:8888/notebooks/work/python-machine-learning-book-3rd-edition-master/ch05/ch05.ipynb#Implementing-a-kernel-principal-component-analysis-in-Python

Example 1 – Separating Half-Moon Shapes

- Create a two-dimensional dataset of 100 example points

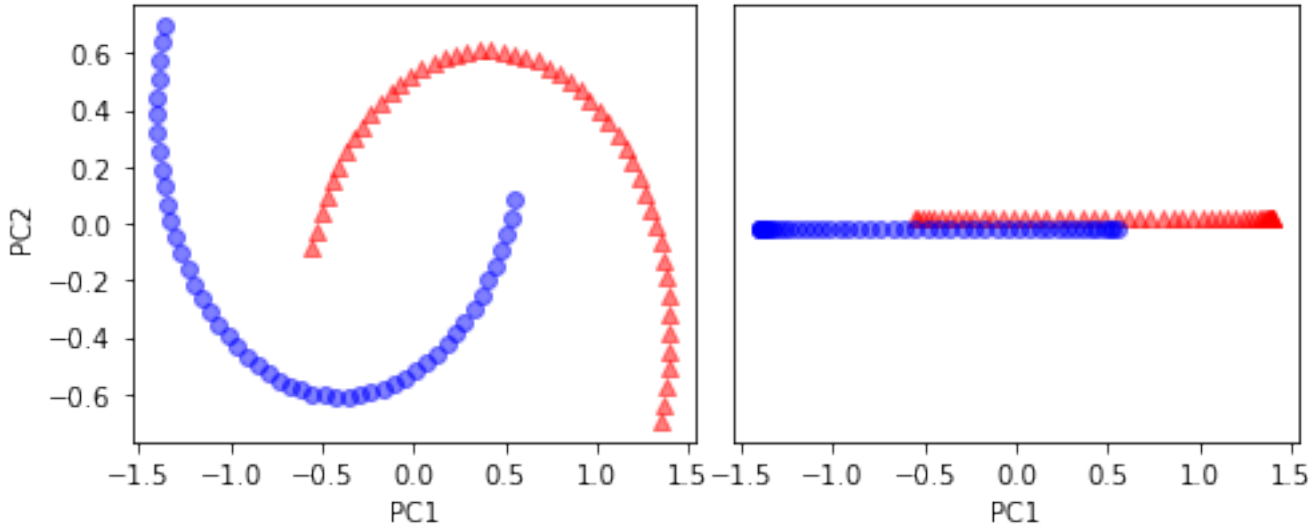
```
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons

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.savefig('images/05_12.png', dpi=300)
plt.show()
```



Example 1 – Separating Half-Moon Shapes

- Project data onto the principal components via standard PCA

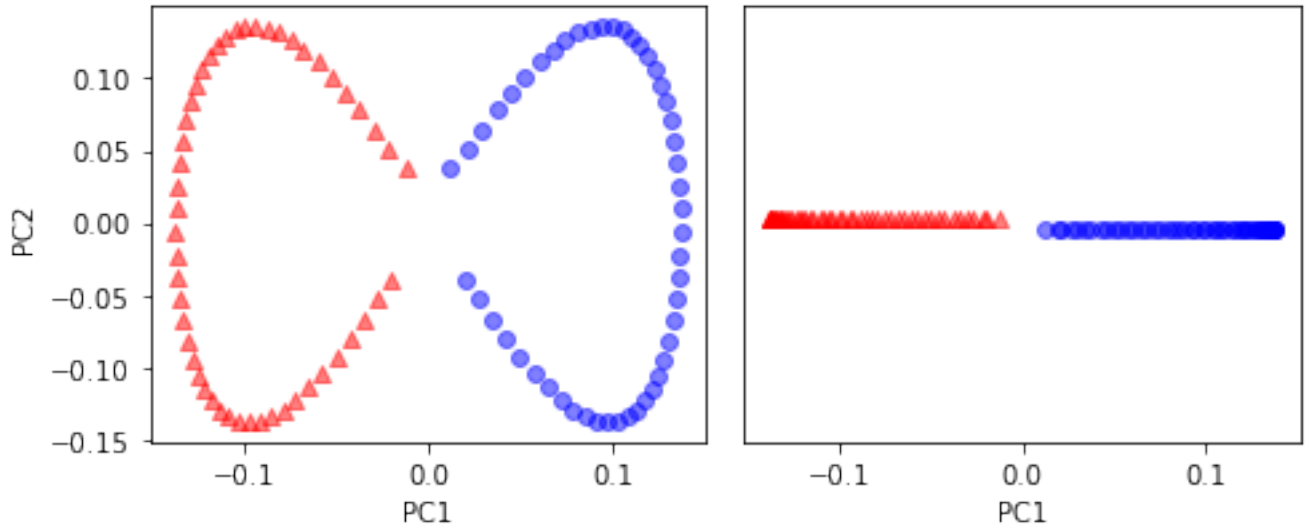
```
from sklearn.decomposition import PCA

scikit_pca = PCA(n_components=2)
X_spca = scikit_pca.fit_transform(X)
```



✓ A linear classifier would be unable to perform well on the dataset transformed via standard PCA

Example 1 – Separating Half-Moon Shapes

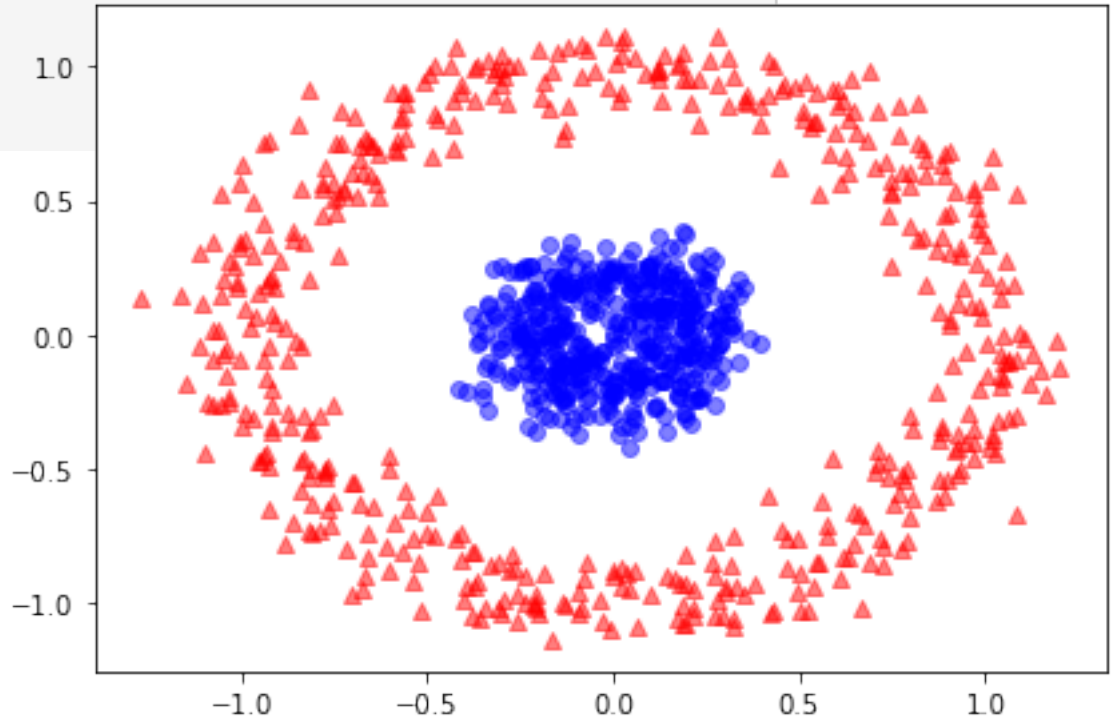
```
X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
```



- ✓ 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

Example 2 – Separating 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.savefig('images/05_15.png', dpi=300)
plt.show()
```

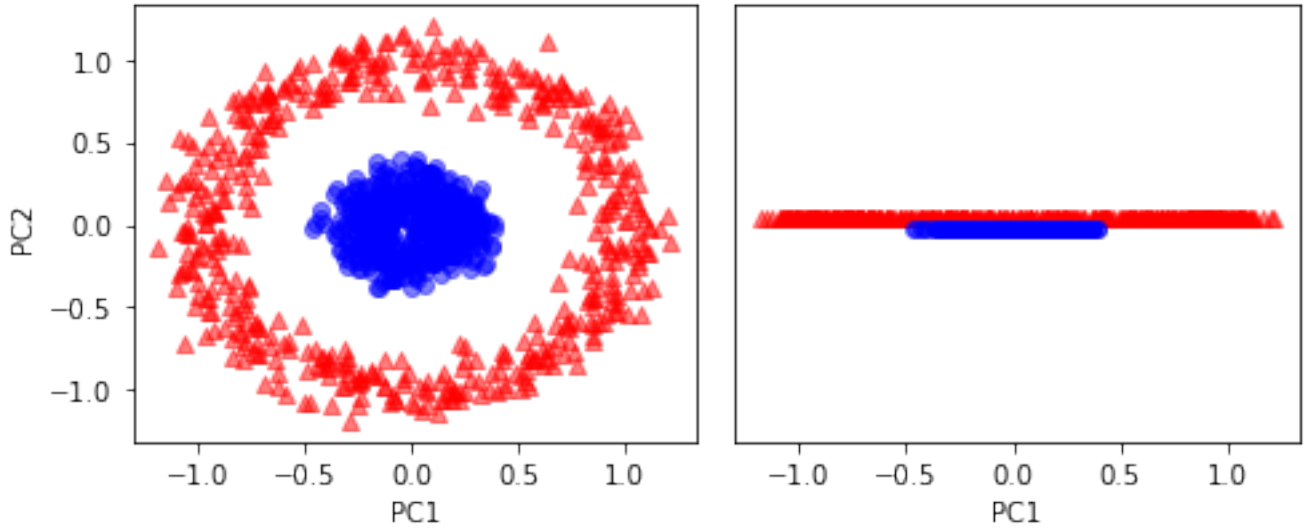


triangle shapes represent one class, and the circle shapes represent another class

Example 2 – Separating Concentric Circles

- Standard PCA approach

```
scikit_pca = PCA(n_components=2)
X_spca = scikit_pca.fit_transform(X)
```

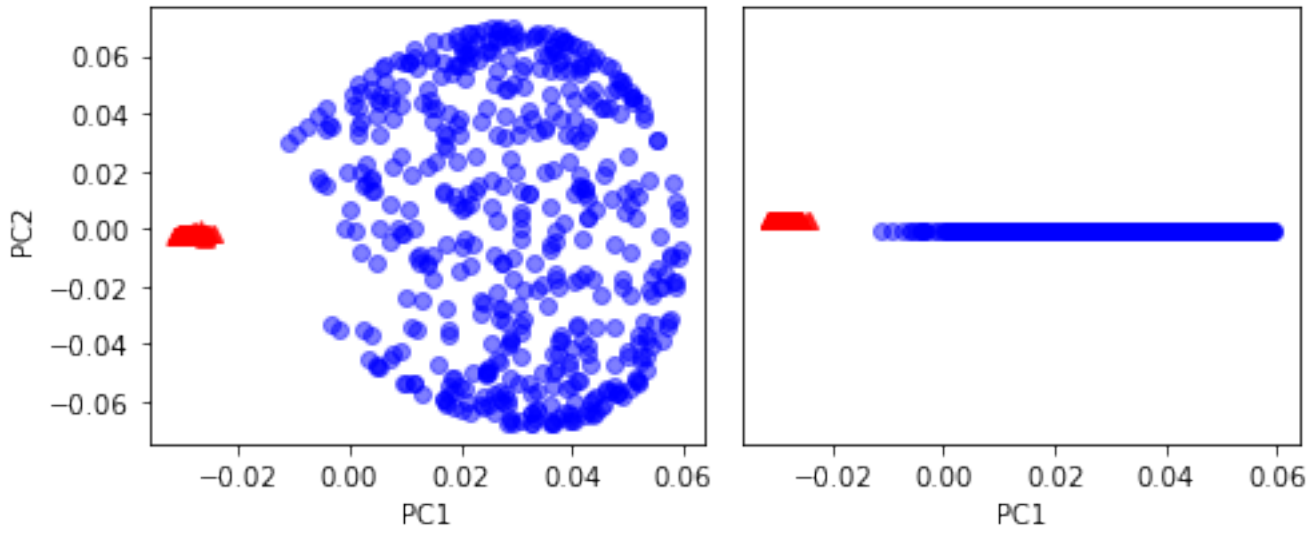


✓ Standard PCA is not able to produce results suitable for training a linear classifier

Example 2 – Separating Concentric Circles

- Using the RBF KPCA implementation

```
X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
```



✓ The RBF KPCA projected the data onto a new subspace where the two classes become linearly separable

KPCA in Scikit-Learn

- Use *sklearn.decomposition* submodule

```
from sklearn.decomposition import KernelPCA

X, y = make_moons(n_samples=100, random_state=123)
scikit_kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_skernpca = scikit_kpca.fit_transform(X)
```