

Compressing Data via Dimensionality Reduction

Hsi-Pin Ma 馬席彬

<http://lms.nthu.edu.tw/course/40724>

Department of Electrical Engineering
National Tsing Hua University

Outline

- Unsupervised Dimensionality via Principal Component Analysis
- Supervised Data Compression via Linear Discriminant Analysis
- Kernel Principal Component Analysis

Feature Extraction

- To combine existing features to produce more useful ones and has the potential to reduce data dimensionality
- Transformation from the original feature space \mathbb{R}^d to a new lower dimensional feature space \mathbb{R}^k
 - Functionality of data compression ($k < d$)
- Not only reduce required data storage space, improve computational efficiency, but also improve predictive performance by reducing the risk of *curse of dimensionality*, especially with non-regularized learning models

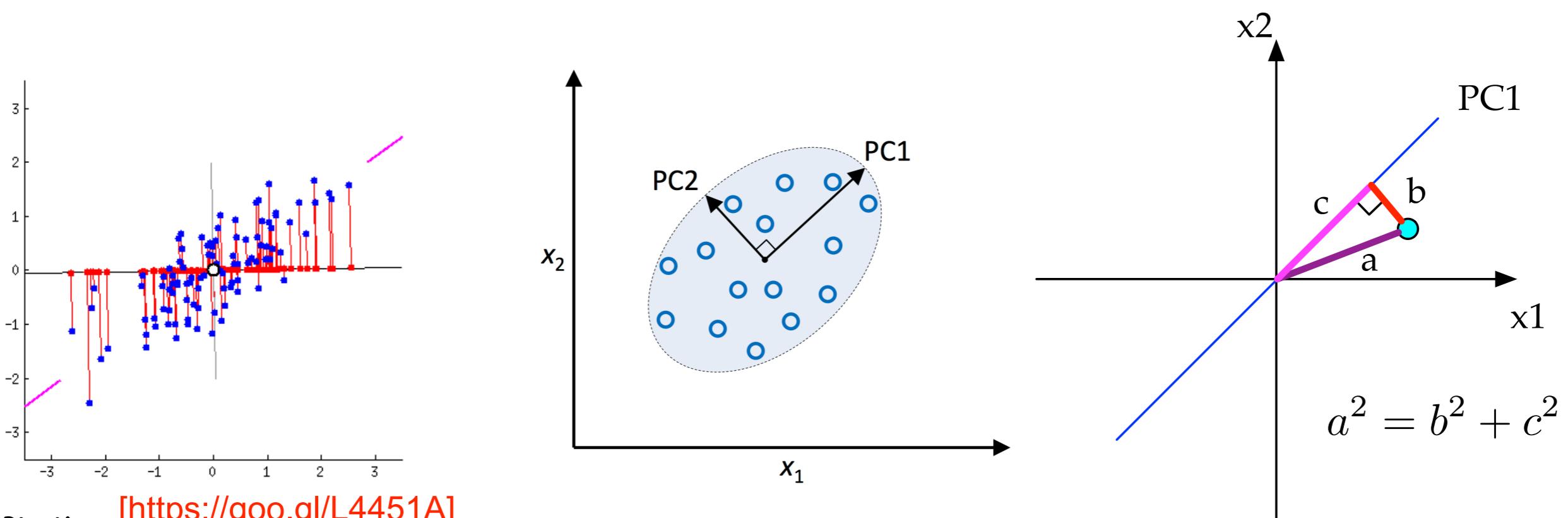
Unsupervised Dimensionality Reduction via Principal Component Analysis

Possible Applications of PCA

- Feature extraction and dimensionality reduction
- Exploratory data analysis such as data visualization
- De-noising of signals in stock market trading
- Analysis of genome data and gene expression level in the field of bioinformatics

Principal Component Analysis

- Find the directions of maximum variance
 - Original features x_1 and x_2
 - Principle components: PC1 and PC2
 - Since a does not change, to find PC1, minimize distance to the PC1 line (b), or maximize the distance from the projected point to the origin (c)



Principal Component Analysis

- Reduce data from d -dimensions to k -dimensions
- Project data onto the lower-dimensional space
- Construct a $d \times k$ transformation matrix \mathbf{W} and map the sample vector \mathbf{x} onto a new k -dimensional feature space ($k < d$)
- PCA is sensitive to data scaling, feature standardization is needed

$$\mathbf{x} = [x_1, x_2, \dots, x_d], \quad \mathbf{x} \in \mathbb{R}^d$$

$$\downarrow \mathbf{x}\mathbf{W}, \quad \mathbf{W} \in \mathbb{R}^{d \times k}$$

$$\mathbf{z} = [z_1, z_2, \dots, z_k], \quad \mathbf{z} \in \mathbb{R}^k$$

Main Steps in Doing PCA

- Standardize the d -dimensional dataset \mathbf{X} to \mathbf{X}_{std}
- Construct the covariance matrix
- Decompose the covariance matrix into its eigenvectors and eigenvalues
- Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors
- Select k eigenvectors which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$)
- Construct a projection matrix \mathbf{W} from the top k eigenvectors
- Transform the d -dimensional input dataset \mathbf{X} using the projection matrix \mathbf{W} to obtain the new k -dimensional feature subspace

Covariance Matrix

- The symmetric $d \times d$ -dimensional (d : dataset dimension) covariance matrix Σ stores the *pairwise covariance* between the different features
- Covariance between two features x_j and x_k

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

sample means of feature j and k

- Sample means are zero if dataset has been standardized

Covariance Matrix

- For three features, covariance matrix looks like

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

- The eigenvectors of Σ represent the principle components
- The corresponding eigenvalues represent their magnitude
 - Principle components: the directions of maximum variance

Extract the Principal Components Step-by-Step

- Standardize the data
- Construct the covariance matrix
- Obtain the eigenvalues and eigenvectors of the covariance matrix
- Sort the eigenvalues by decreasing order to rank the eigenvectors

Extract the Principal Components Step-by-Step

Load 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)

df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue',
                   'OD280/OD315 of diluted wines', 'Proline']
```

```
df_wine.head()
```

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyan
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1

Extract the Principal Components Step-by-Step

- Split training and test, and standardize the dataset

```
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)
```

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

Extract the Principal Components Step-by-Step

- Construct the covariance matrix and do eigendecomposition

```
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 ]
```

Extract the Principal Components Step-by-Step

- Total and explained variance
- To reduce the dimensionality, only select the subset of eigenvectors (PCs) that contain most of the information (variance)
- variance explained ratios of an eigenvalue λ_j is defined as

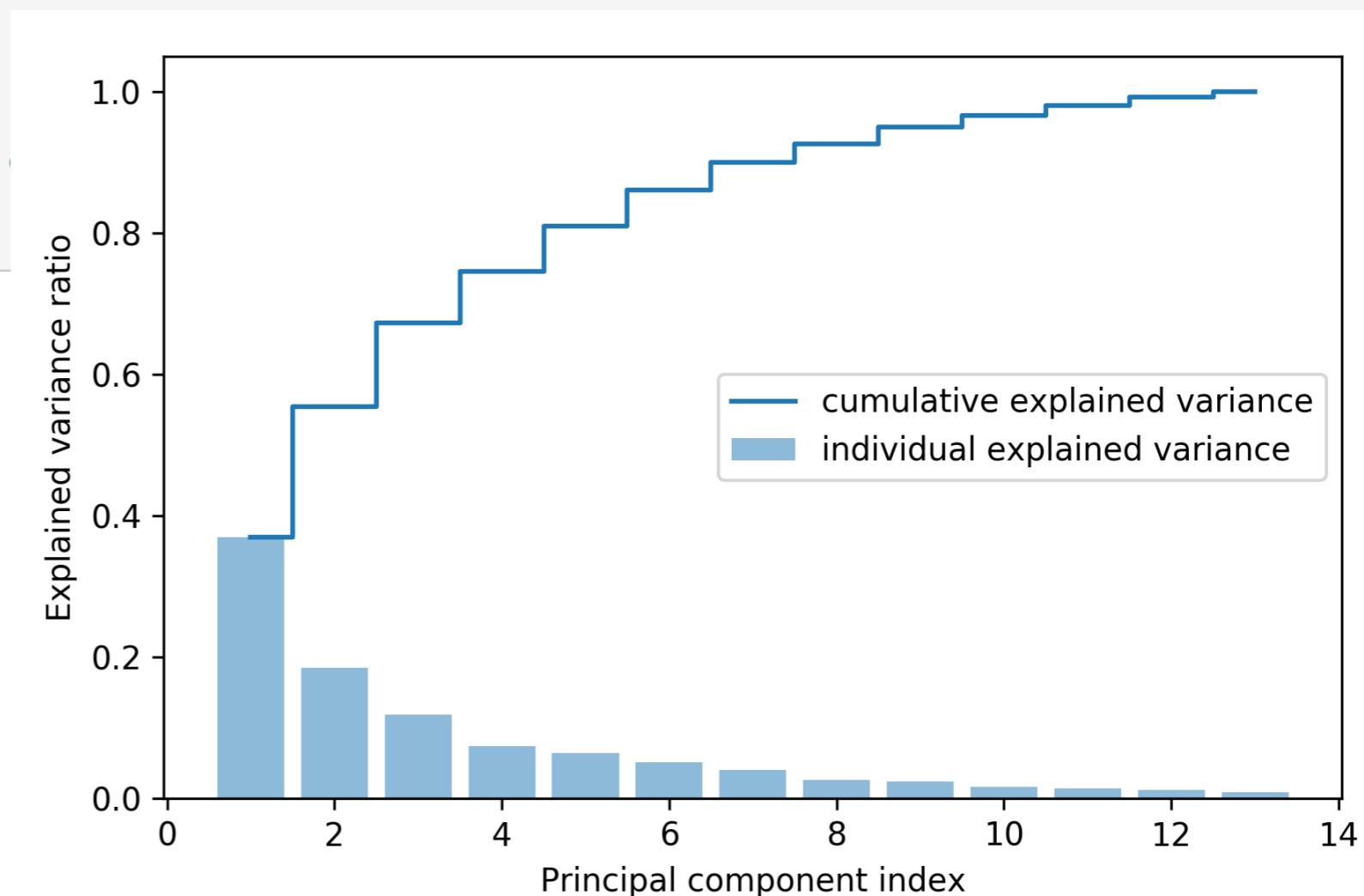
$$\frac{\lambda_j}{\sum_{j=1}^d \lambda_j}$$

```
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
```

Extract the Principal Components Step-by-Step

```
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',
plt.show()
```



First two principal
components explained about
60% of the variance of the data

Feature Transformation

- Select k eigenvectors, which correspond to the largest k eigenvalues (k : dimensionality of new feature subspace)
- Construct a projection matrix \mathbf{W} from the “top” k eigenvectors
- Transform the d -dimensional input dataset \mathbf{X} using the projection matrix \mathbf{W} to obtain the new k -dimensional feature subspace

Feature Transformation

- Sort the eigenpairs by decreasing order of 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)
```

Feature Transformation

- Collect the two largest to capture about 60% of the variance to form 13×2 projection matrix \mathbf{W}
 - The number of principal components has to be determined by a trade-off between computational efficiency and the classifier performance

```
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]]
```

Feature Transformation

- Transform a sample x onto PCA subspace
obtaining x' $x' = xW$

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

- Transform the entire dataset X' $X' = XW$

```
x_train_pca = x_train_std.dot(w)
```

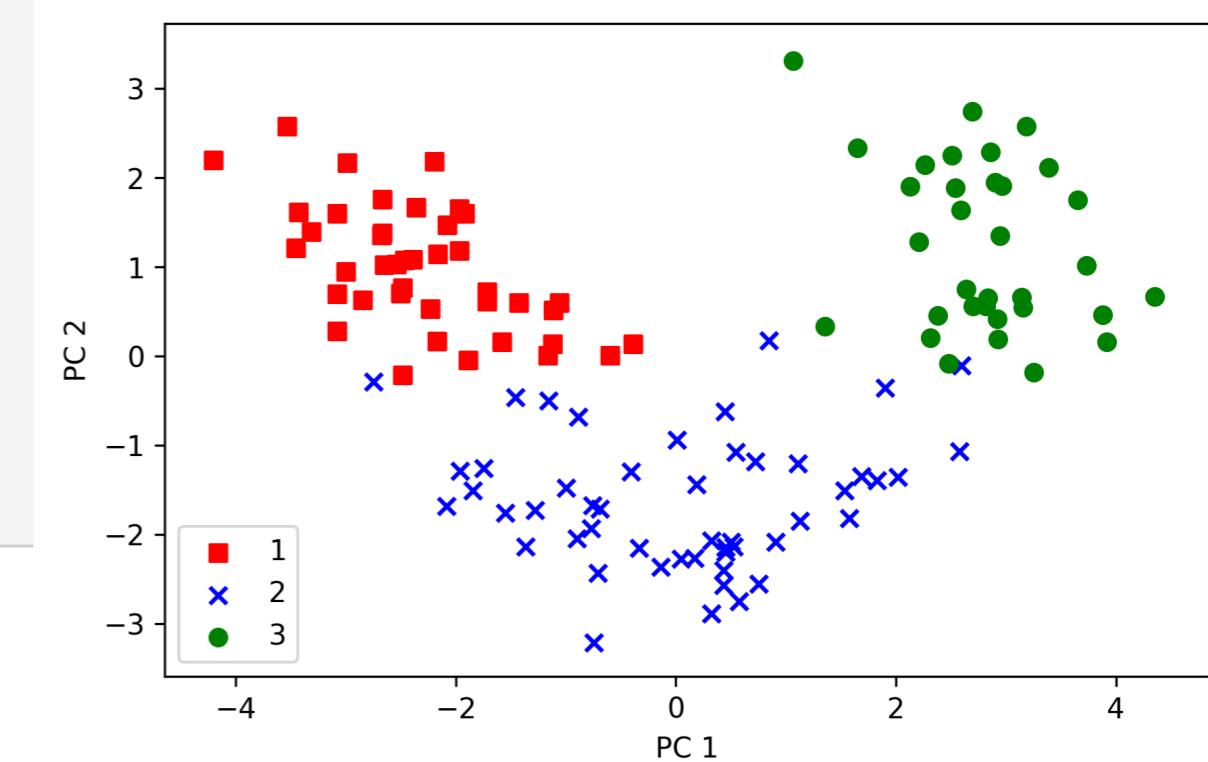
Feature Transformation

- Visualize the transformed training set

```
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.savefig('images/05_03.png', dpi=300)
plt.show()
```



Principal Component Analysis in Scikit-learn

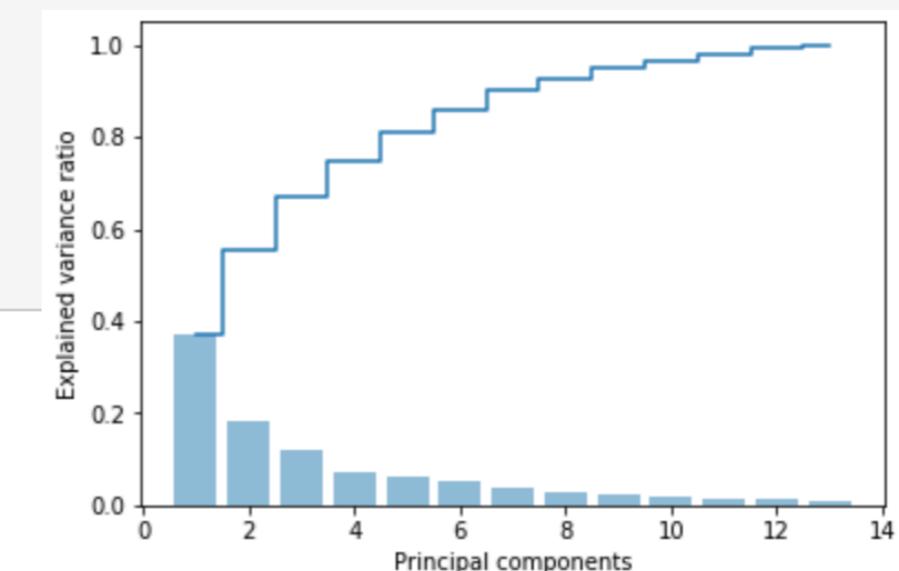
- In scikit-learn, **PCA** class in the **decomposition** module

```
from sklearn.decomposition import PCA

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

```
array([ 0.36951469,  0.18434927,  0.11815159,  0.07334252,  0.06422108,
       0.05051724,  0.03954654,  0.02643918,  0.02389319,  0.01629614,
       0.01380021,  0.01172226,  0.00820609])
```

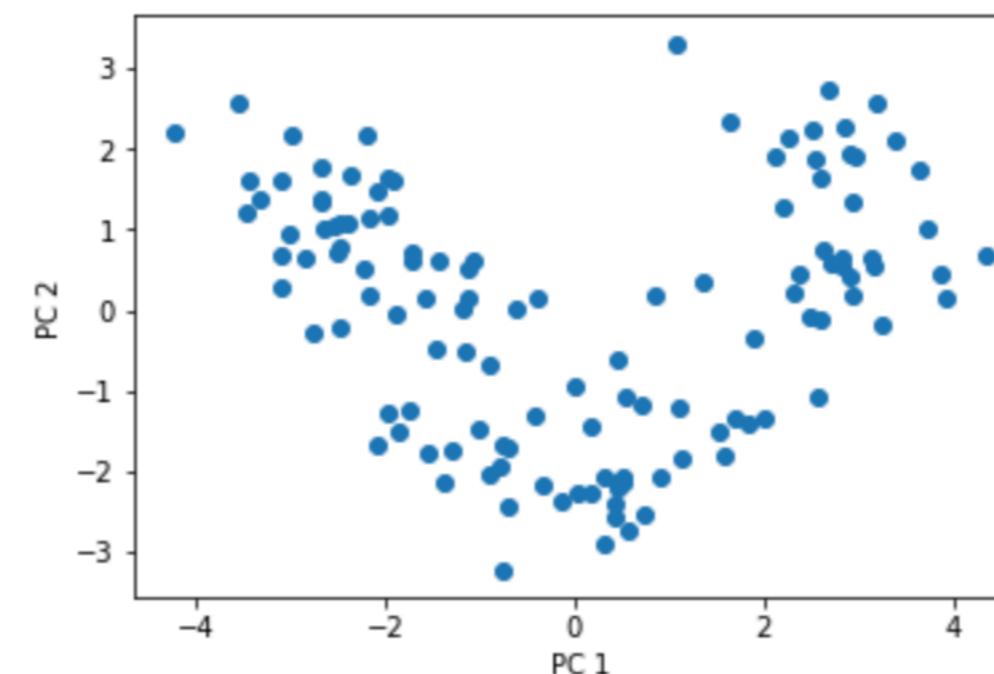
```
plt.bar(range(1, 14), pca.explained_variance_ratio_, alpha=0.5, align='center')
plt.step(range(1, 14), np.cumsum(pca.explained_variance_ratio_), where='mid')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.show()
```



Principal Component Analysis in Scikit-learn

```
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
```

```
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1])
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.show()
```



Principal Component Analysis in Scikit-learn

- Use **PCA** class implemented in scikit-learn
 - One of transformer classes
 - First fit the model using training data before transforming both the training and test data using the same model parameters
- Example
 - Use PCA class on Wine training dataset
 - Classify the transformed samples via logistic regression
 - Visualize the decision regions via the `plot_decision_region` function

Principal Component Analysis in Scikit-learn

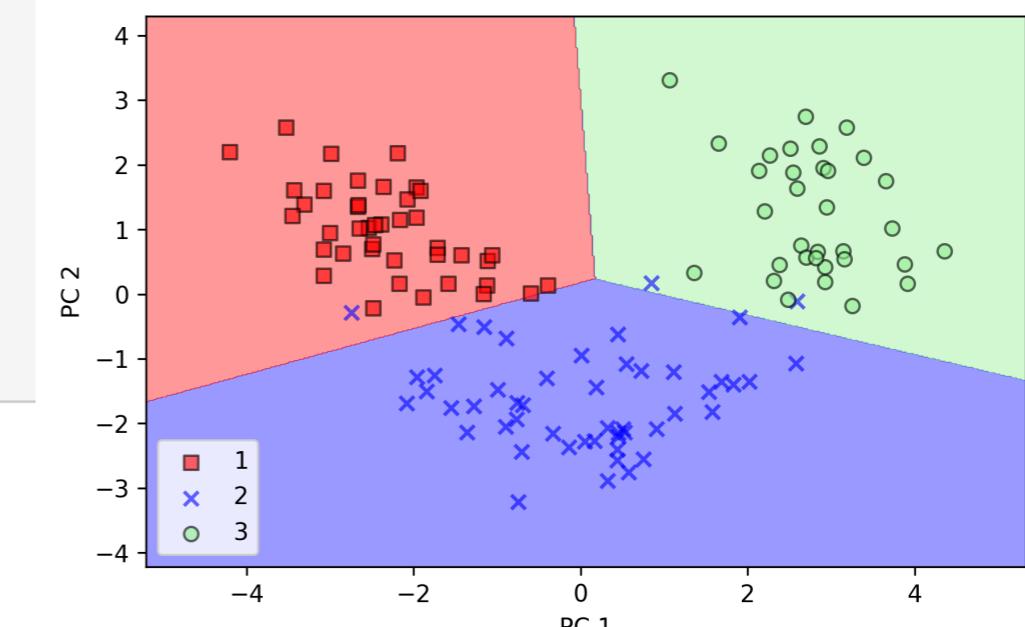
Training dataset

```
from sklearn.linear_model import LogisticRegression

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)

lr = LogisticRegression()
lr = lr.fit(X_train_pca, y_train)
```

```
plot_decision_regions(X_train_pca, y_train, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
# plt.savefig('images/05_04.png', dpi=300)
plt.show()
```



Principal Component Analysis in Scikit-learn

```
In [20]: y_pred = lr.predict(X_train_pca)
         print('Misclassified instances for training dataset: %d' % (y_train != y_pred).sum())
```

Misclassified instances for training dataset: 3

```
In [21]: from sklearn.metrics import accuracy_score
         print('Accuracy for training dataset: %.2f' % accuracy_score(y_train, y_pred))
```

Accuracy for training dataset: 0.98

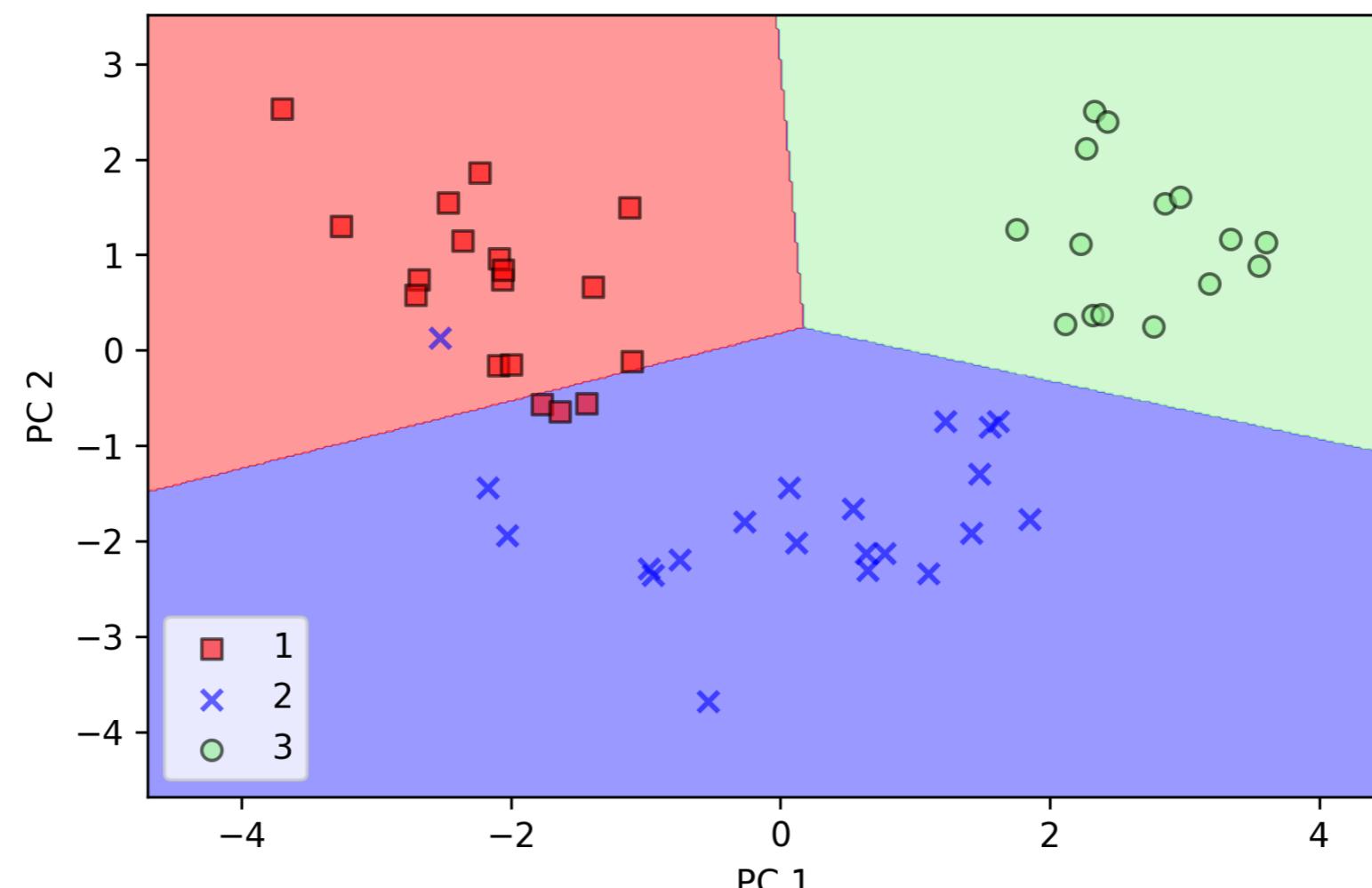
```
In [22]: print('Accuracy for training dataset: %.2f' % lr.score(X_train_pca, y_train))
```

Accuracy for training dataset: 0.98

Principal Component Analysis in Scikit-learn

Test dataset

```
plot_decision_regions(X_test_pca, y_test, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
# plt.savefig('images/05_05.png', dpi=300)
plt.show()
```



Principal Component Analysis in Scikit-learn

```
In [24]: y_pred = lr.predict(X_test_pca)
         print('Misclassified instances for test dataset: %d' % (y_test != y_pred).sum())
```

Misclassified instances for test dataset: 4

```
In [25]: print('Accuracy for test dataset: %.2f' % accuracy_score(y_test, y_pred))
```

Accuracy for test dataset: 0.93

```
In [26]: print('Accuracy for test dataset: %.2f' % lr.score(X_test_pca, y_test))
```

Accuracy for test dataset: 0.93

Principal Component Analysis in Scikit-learn

- By initializing the *n_components* parameter in the PCA class to be *None*, all PCs will be kept, and the explained variance ratios can be accessed via the *explained_variance_ratio_* attribute.

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

array([ 0.36951469,  0.18434927,  0.11815159,  0.07334252,  0.06422108,
       0.05051724,  0.03954654,  0.02643918,  0.02389319,  0.01629614,
       0.01380021,  0.01172226,  0.00820609])
```

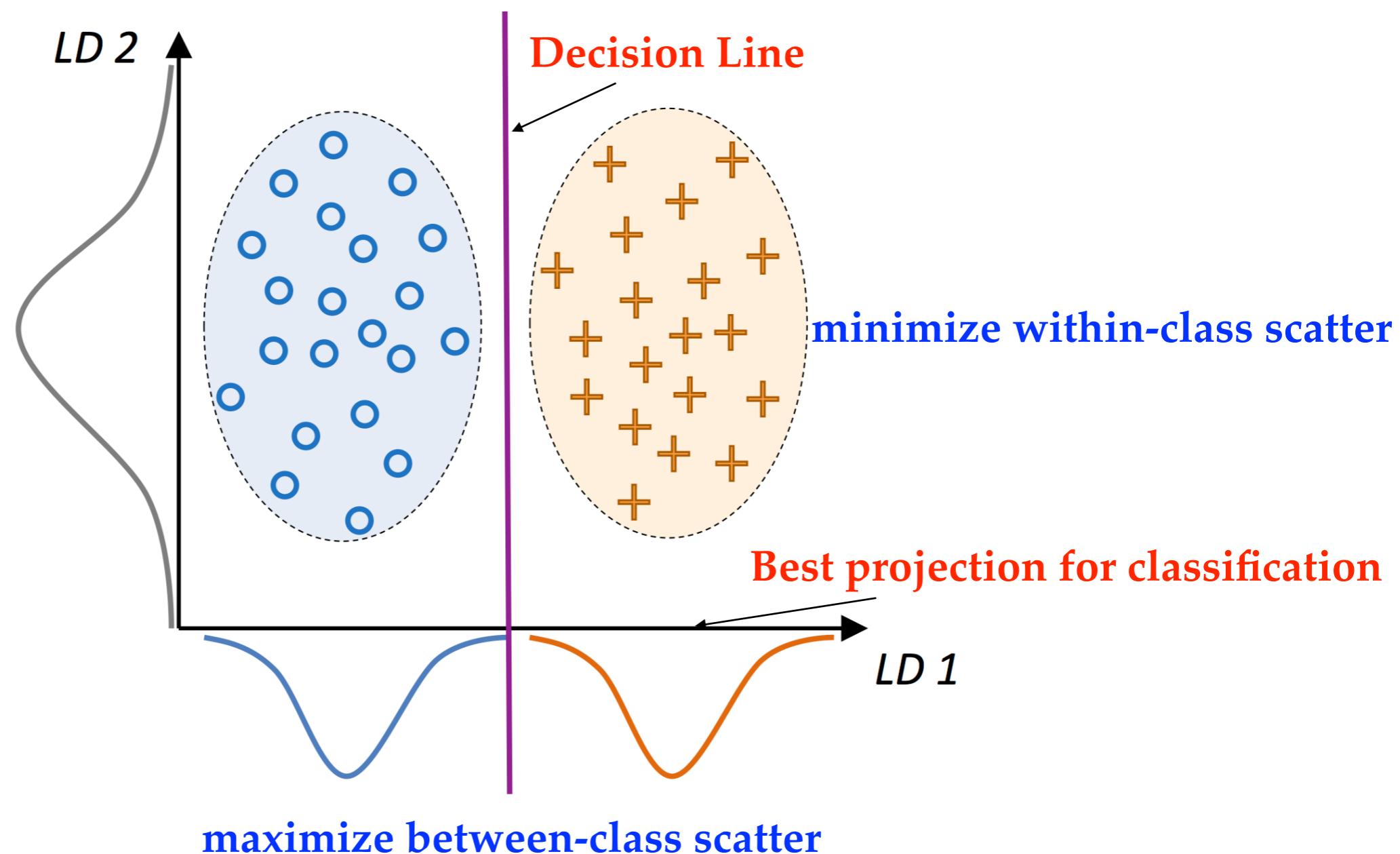
Supervised Data Compression Via Linear Discriminant Analysis

PCA vs. LDA

- Both PCA and LDA are linear transformation techniques for feature extraction
- PCA is unsupervised while LDA is supervised
 - LDA uses the class label information of instances
- PCA attempts to find the orthogonal component axes of maximum variance in a dataset while LDA is to find the feature subspace that optimizes class separability

Linear Discriminant Analysis

- In the figure, a linear discriminant on the x-axis (LD1) would separate the two classes well.



Main Steps to Perform LDA

- Standardize the d -dimensional dataset
- For each class, compute the d -dimensional mean vector
- Construct the between-class scatter matrix \mathbf{S}_B and the within-class scatter \mathbf{S}_w
- Compute the eigenvectors and corresponding eigenvalues of the matrix $\mathbf{S}_w^{-1} \mathbf{S}_B$
- Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors
- Choose the k eigenvectors that correspond to the k largest eigenvalues to construct a $d \times k$ -dimensional transformation matrix \mathbf{W} ; the eigenvectors are the columns of this matrix
- Project the samples onto the new feature subspace using \mathbf{W} .

Inner Workings of LDA

- Basic assumption

- The feature vectors of instance from different class are approximately normally distributed with different mean and variance

- Class sample mean

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{x \in D_i}^c \mathbf{x}_m$$

- Each mean vector \mathbf{m}_i stores the mean feature values μ_m with respect to the samples of class i
- For Wine dataset

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

Inner Workings of LDA

- mean vectors for each class in Wine dataset

```
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 -0.3497  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 ]
```

Within-class Scatter Matrix

- Individual scatter matrix S_i of each individual class i

$$S_i = \sum_{x \in D_i}^c (x - m_i)(x - m_i)^T$$

- Within-class scatter matrix S_W

$$S_W = \sum_{i=1}^c S_i$$

```

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

Between-Class Scatter Matrix

- Between-class matrix $S_B = \sum_{i=1}^c n_i (\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$
 - \mathbf{m} is the overall mean that is computed, including samples from all 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[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 Space

- Solve the eigenvalue problem of the matrix $S_w^{-1}S_B$

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

- Sort eigenvectors in descending order of 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 = sorted(eigen_pairs, key=lambda k: k[0], reverse=True)

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

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

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

Selecting Linear Discriminants for the New Feature Space

- In LDA, the number of linear discriminants is at most $c-1$ (c : number of class labels)
 - \mathbf{S}_B is the sum of c matrices with rank 1 or less.
- Indeed only two nonzero eigenvalues
 - eigenvalues 3-13 are not exactly zero due to 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

Selecting Linear Discriminants for the New Feature Space

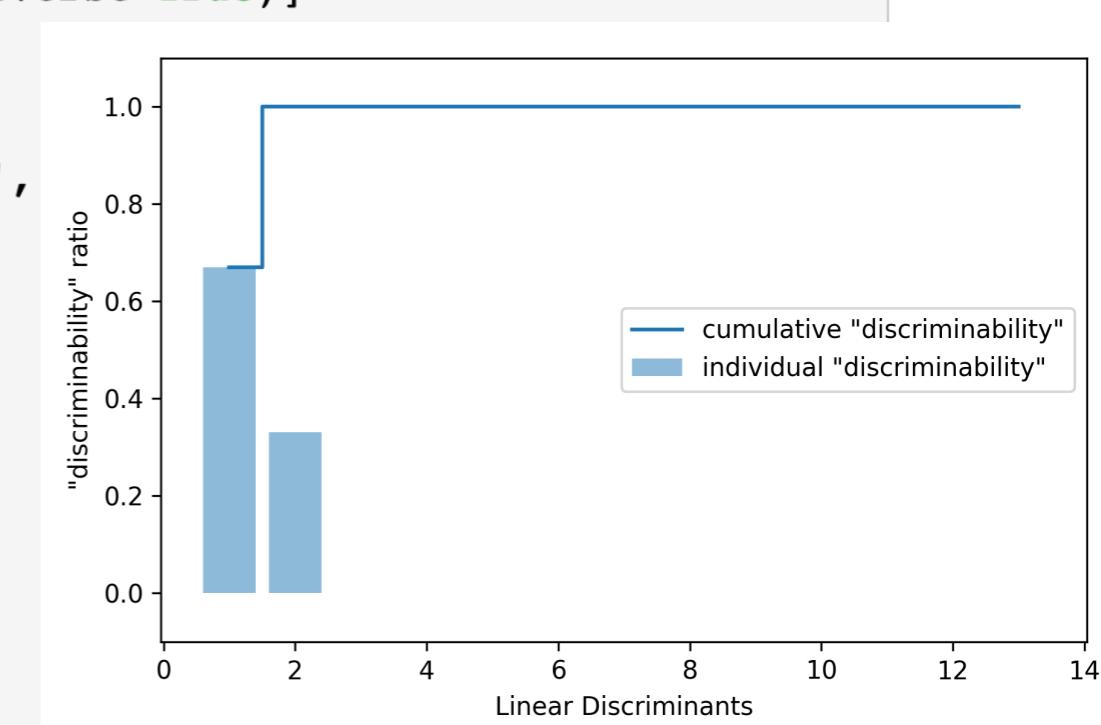
- Discriminability of a linear discriminant
 - To measure how much the class-discriminatory information is captured by the linear discriminants (the eigenvectors)
 - Ratio of its corresponding eigenvalues to the sum of all eigenvalues

```

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.savefig('images/05_07.png', dpi=300)
plt.show()

```



Selecting Linear Discriminants for the New Feature Space

- Stack the two most discriminative eigenvector columns to create the transformation matrix \mathbf{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 Samples onto the New Feature Space

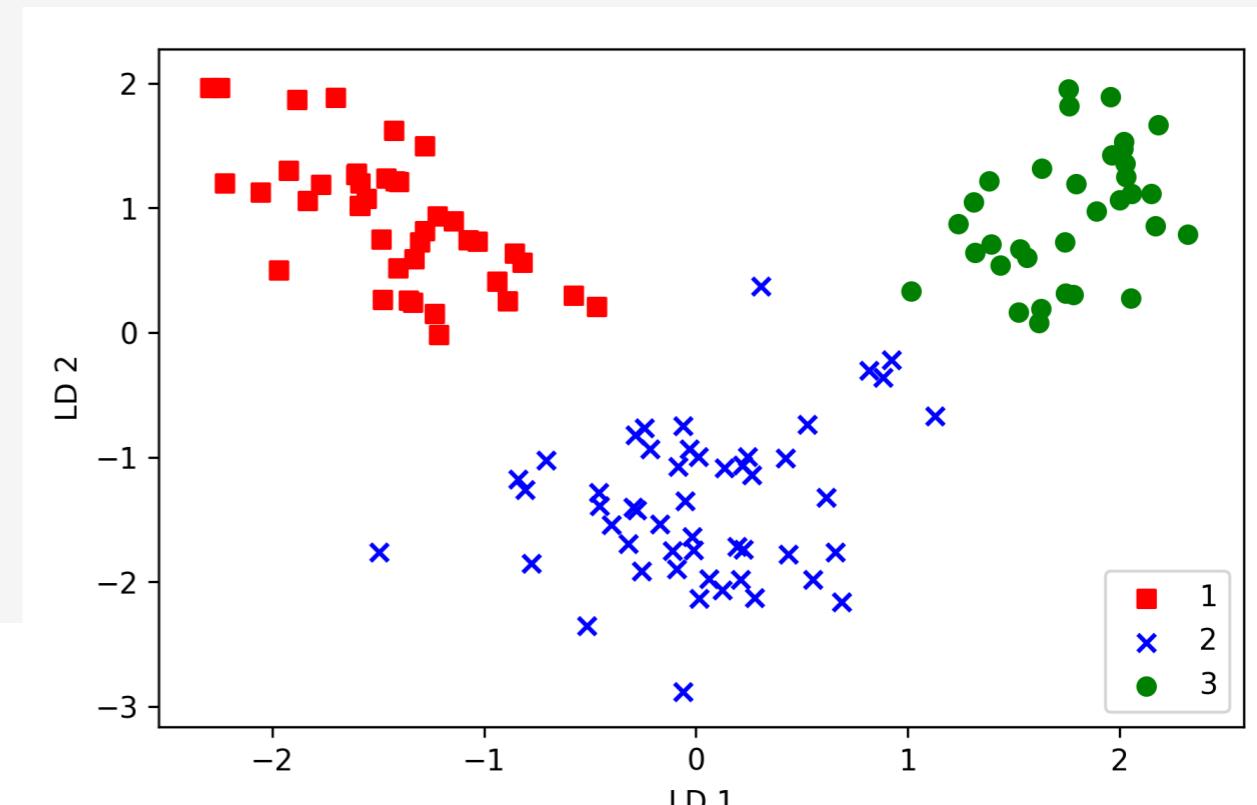
- Transformation of training set $X' = XW$

```

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.savefig('images/05_08.png', dpi=300)
plt.show()
  
```



LDA via Scikit-learn

- In scikit-learn, LDA is implemented in the **discriminant_analysis** module as the **LinearDiscriminantAnalysis** class

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
```

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

```
from sklearn.linear_model import LogisticRegression
```

```
lr = LogisticRegression()
lr = lr.fit(X_train_lda, y_train)
```

```
plot_decision_regions(X_train_lda, y_train, classifier=lr)
```

```
plt.xlabel('LD 1')
```

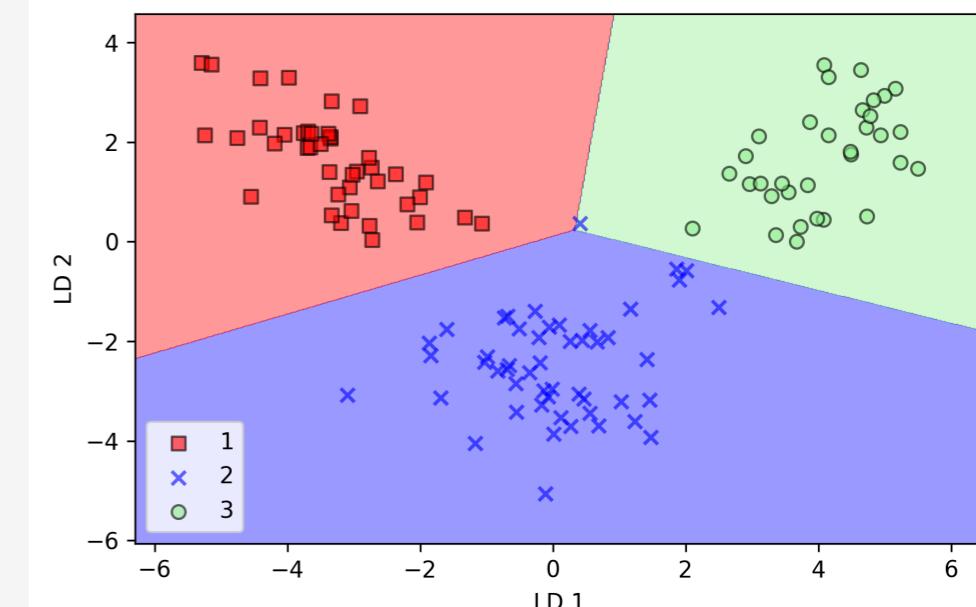
```
plt.ylabel('LD 2')
```

```
plt.legend(loc='lower left')
```

```
plt.tight_layout()
```

```
# plt.savefig('images/05_09.png', dpi=300)
```

```
plt.show()
```

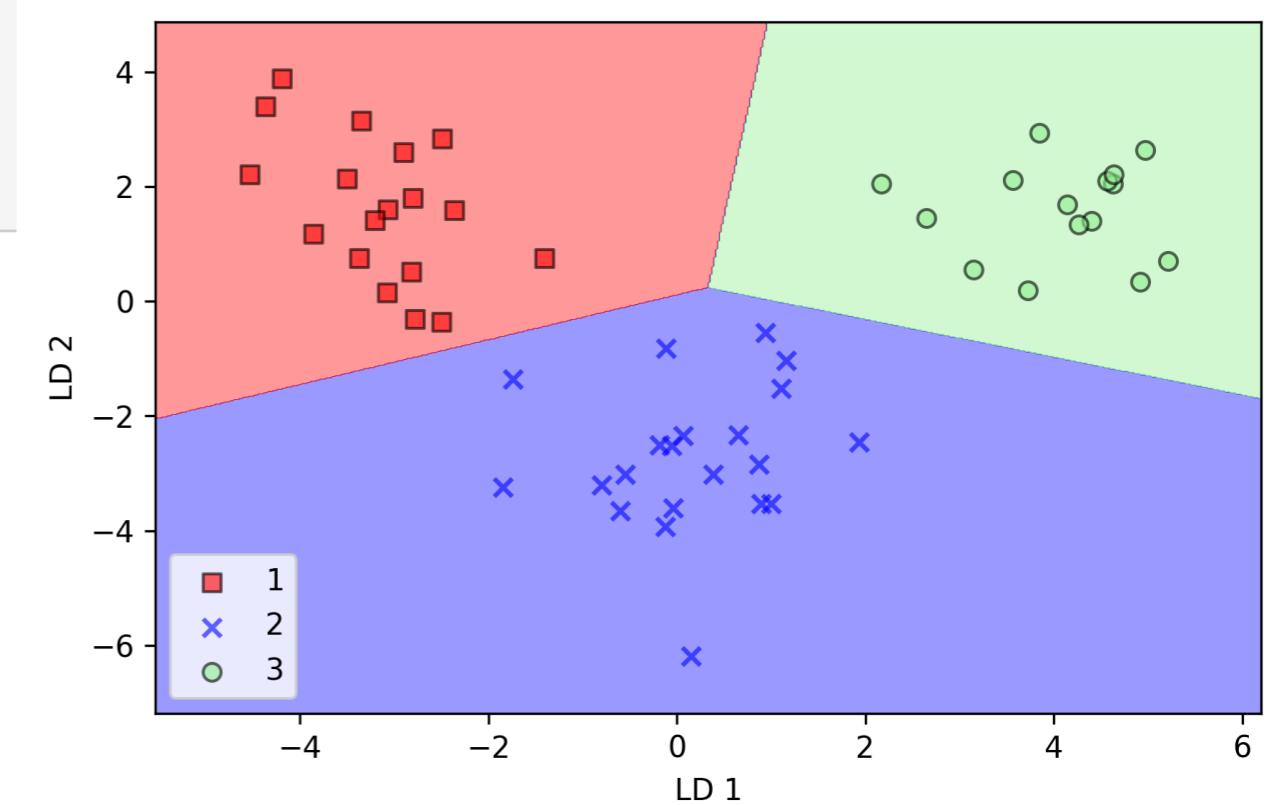


LDA via Scikit-learn

- Apply test dataset

```
x_test_lda = lda.transform(X_test_std)

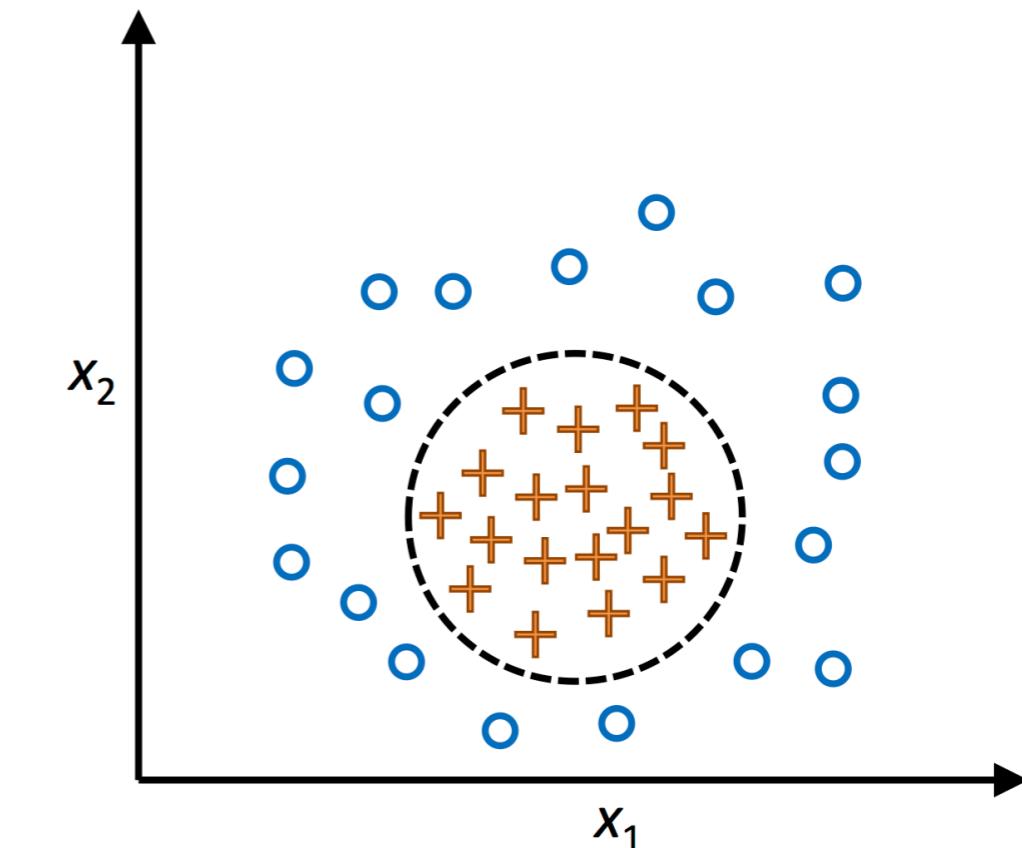
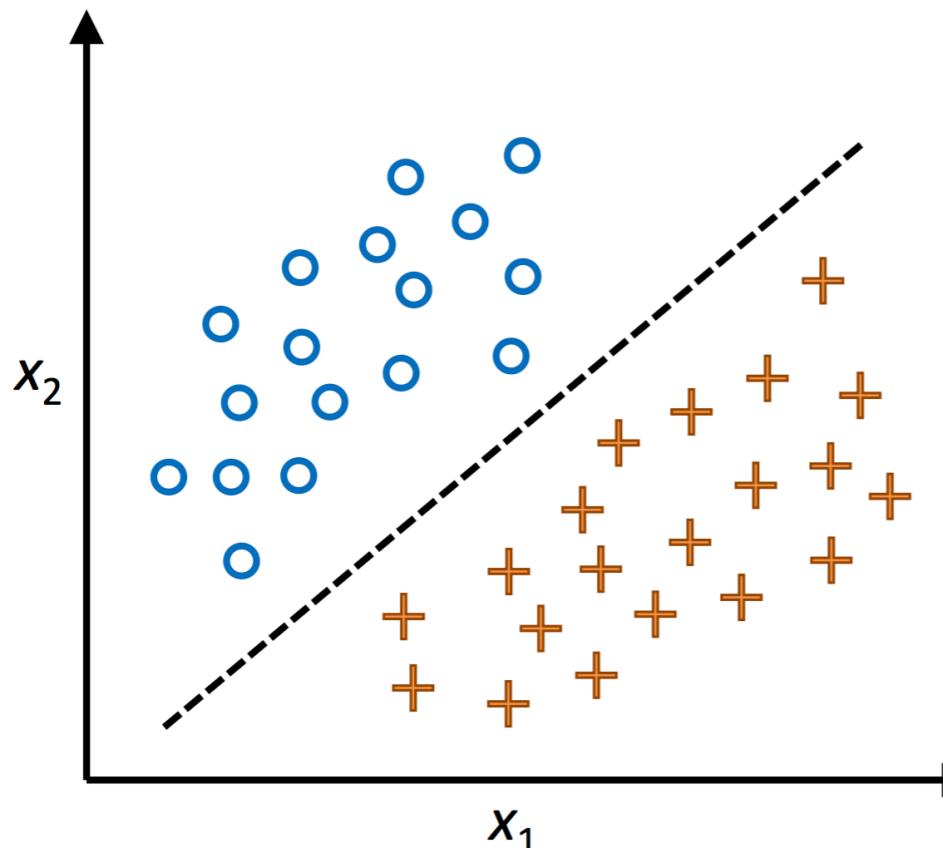
plot_decision_regions(X_test_lda, y_test, classifier=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.tight_layout()
# plt.savefig('images/05_10.png', dpi=300)
plt.show()
```



Kernel Principal Component Analysis

Nonlinear Dimension Reduction

- When linear separation is infeasible in the original feature space, a nonlinear transformation from the original feature space to a higher (possibly infinite) dimensional feature space is desired
 - Linear transformation techniques for dimensionality reduction, such as PCA or LDA, may not be the best.



Kernel Trick

- Using kernel trick, we can compute the similarity between two high-dimension feature vectors in the original feature space

covariance between two features

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

zero mean

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

covariance matrix

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



define the kernel function

$$K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$$

nonlinear feature mapping

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

Most Commonly Used Kernels

- Polynomial kernel

$$\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (\mathbf{x}^{(i)T} \mathbf{x}^{(j)} + \theta)^p$$

- Hyperbolic tangent (sigmoid) kernel

$$\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \tanh(\eta \mathbf{x}^{(i)T} \mathbf{x}^{(j)} + \theta)$$

- Radial Basis Function (RBF) or Gaussian kernel

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

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

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

Three Steps to Implement an RBF Kernel PCA

- Compute the kernel (similarity) matrix K

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

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

- Center the kernel matrix K using

$$K' = K - \mathbf{1}_n K - K \mathbf{1}_n + \mathbf{1}_n \mathbf{1}_n^T$$

– $\mathbf{1}_n$ is an $n \times n$ -dimensional matrix with all values $1/n$

- Collect the top k eigenvectors of the centered kernel matrix based on their corresponding eigenvalues, ranked by decreasing magnitude

RBF Kernel PCA Implementation in Python

```
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_samples, 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_samples, k_features]
        Projected dataset
    """

    """
```

RBF Kernel PCA Implementation in Python

```

# 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 samples)
X_pc = np.column_stack((eigvecs[:, i]
                        for i in range(n_components)))
return X_pc

```

have to tune **gamma** in advance

Separating Half-moon Shapes

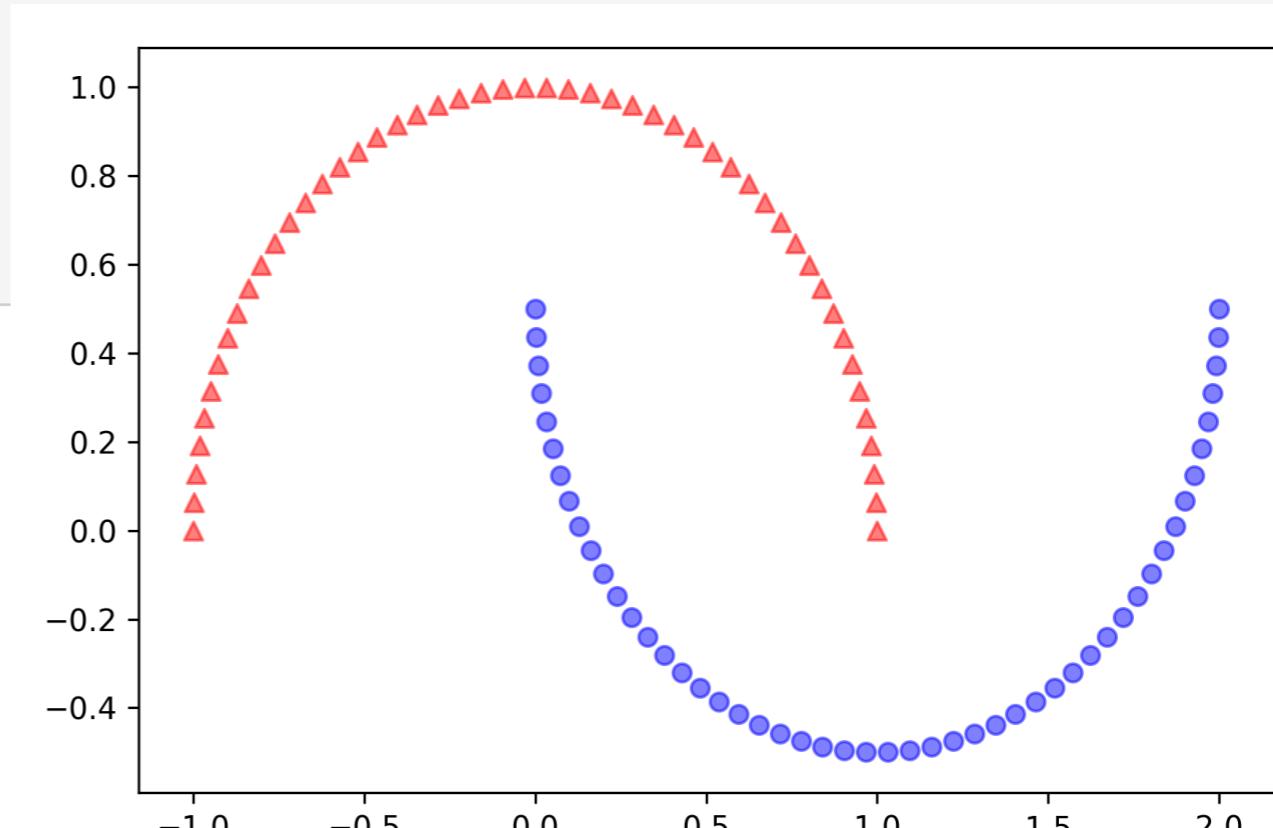
- Create a 2D dataset of 100 samples representing two half-moon shapes (for binary classification)

```
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()
```



Separating Half-moon Shapes

- Try standard PCA

```
from sklearn.decomposition import PCA

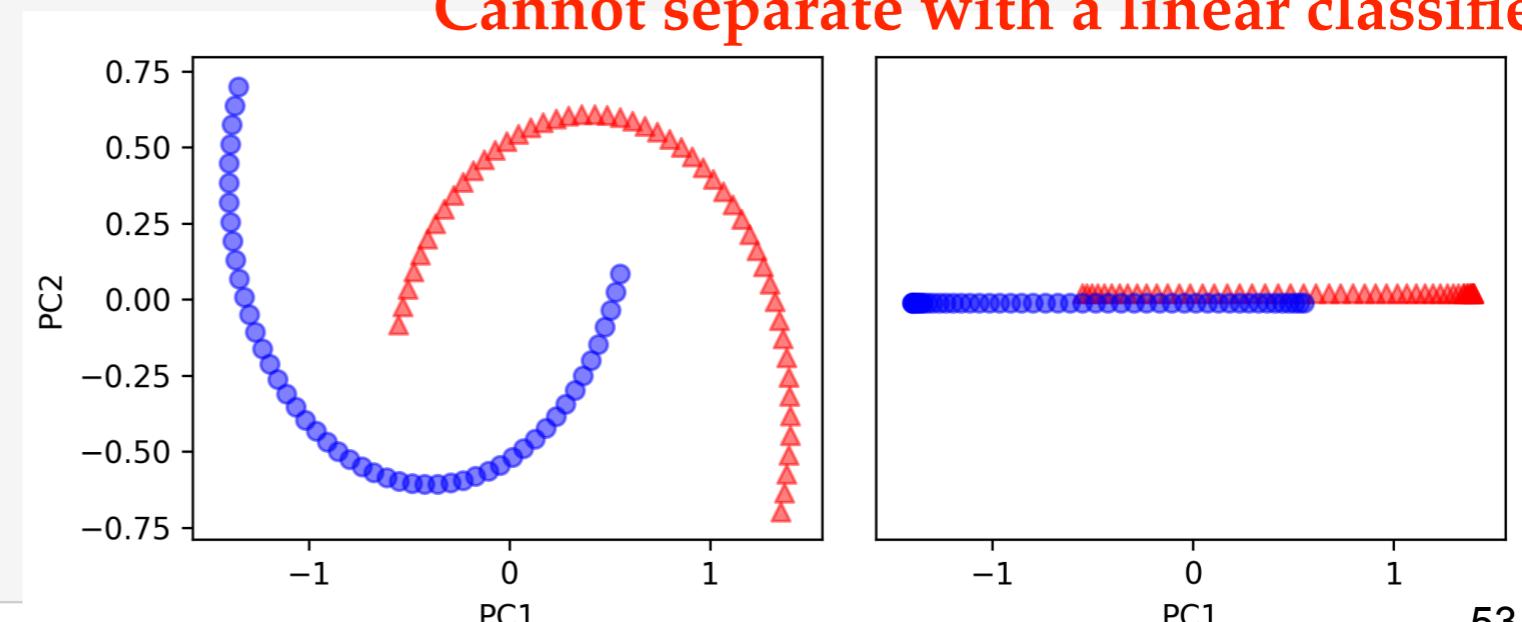
scikit_pca = PCA(n_components=2)
X_spca = scikit_pca.fit_transform(X)

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.savefig('images/05_13.png', dpi=300)
plt.show()
```



Separating Half-moon Shapes

- Try RBF-kernel PCA

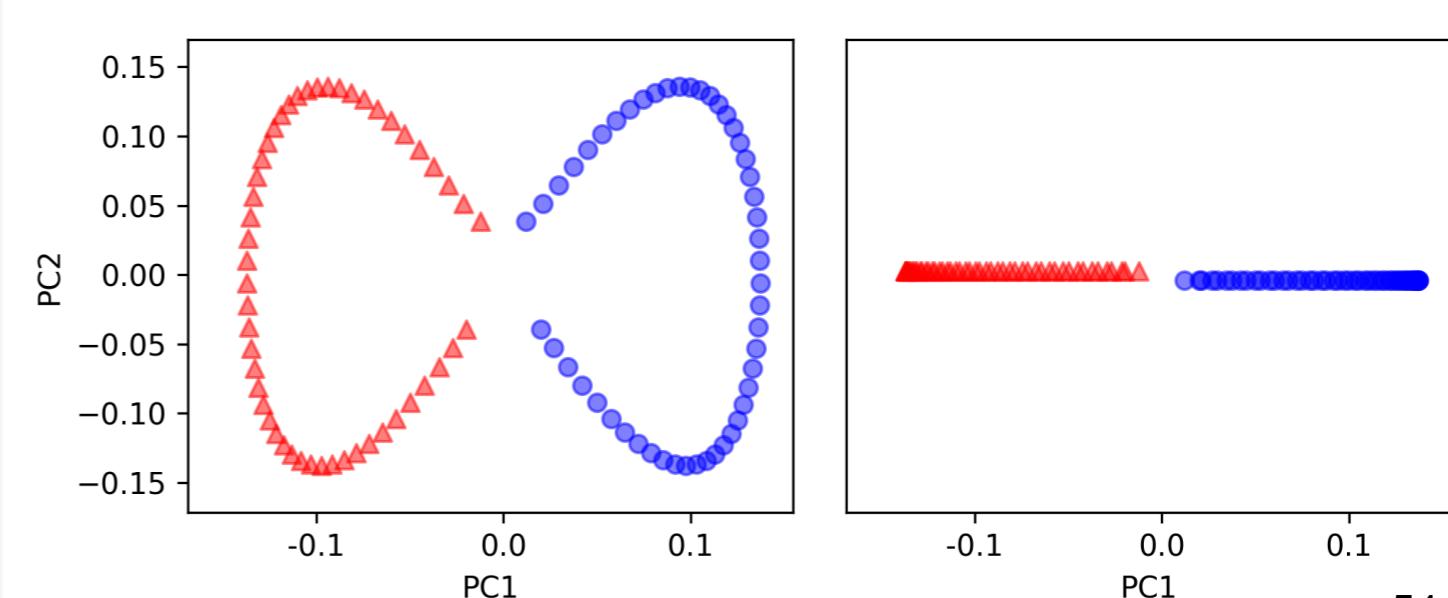
```
x_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)

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.savefig('images/05_14.png', dpi=300)
plt.show()
```



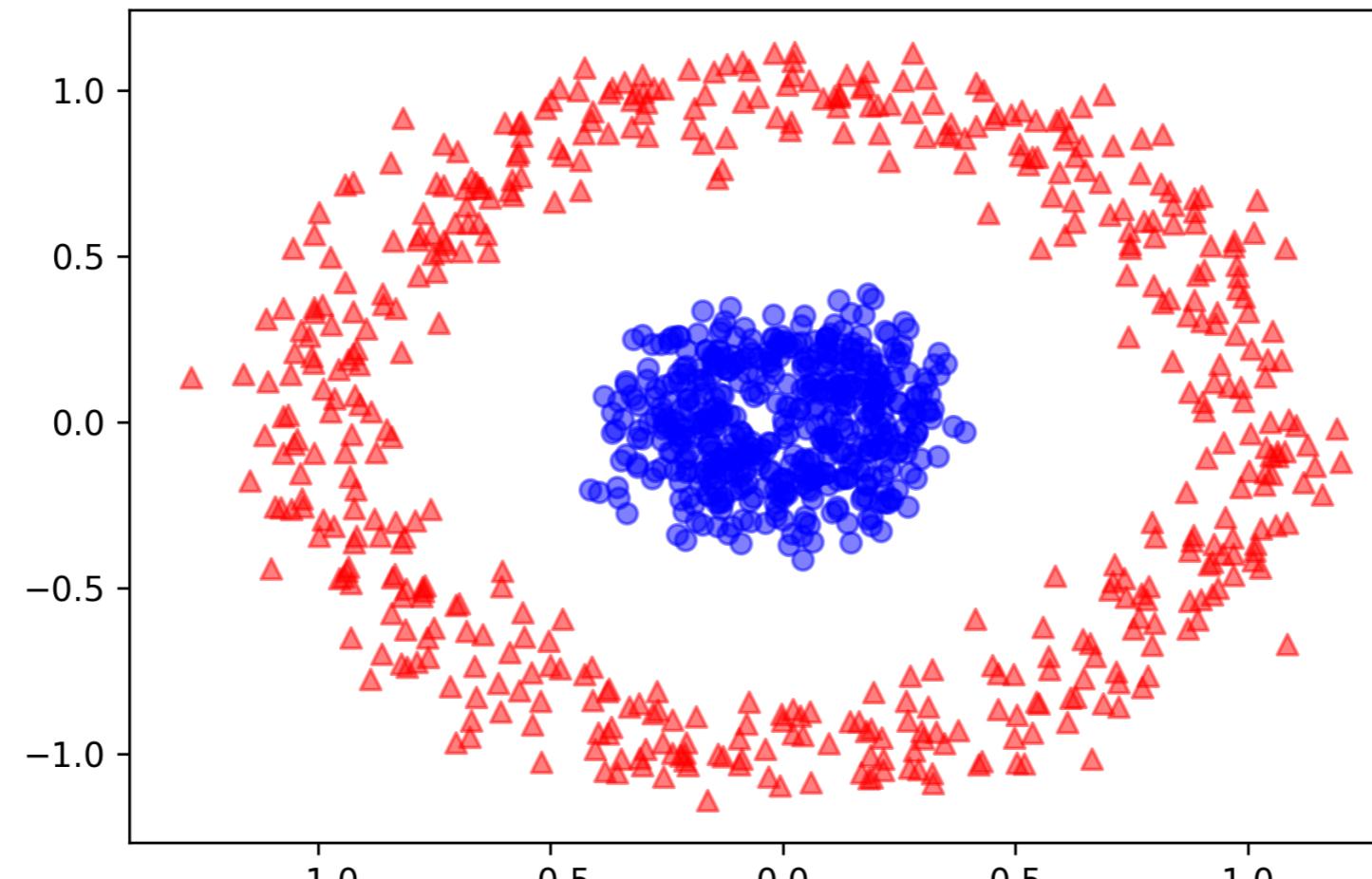
Separating Concentric Circles

```
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()
```



Separating Concentric Circles

- Standard PCA

```

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

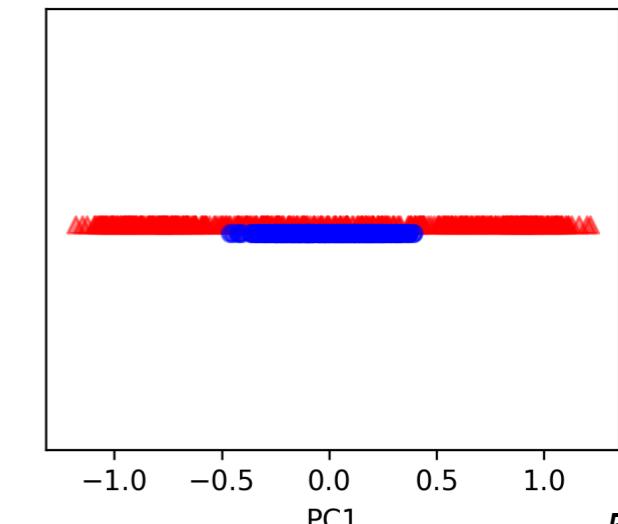
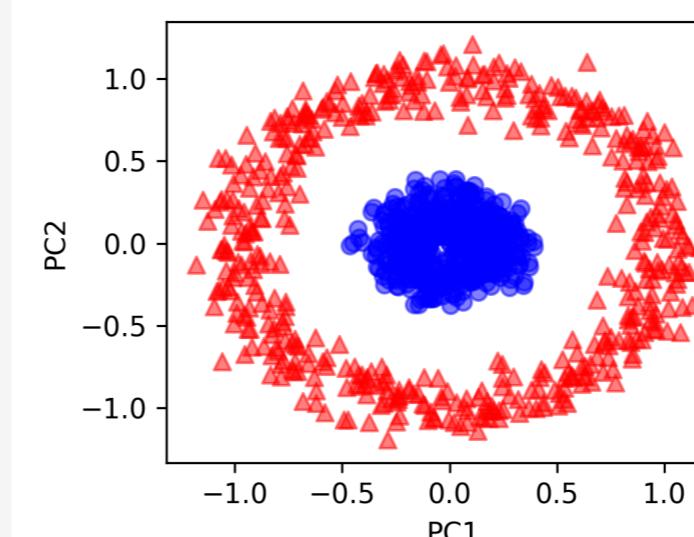
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7, 3))

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.savefig('images/05_16.png', dpi=300)
plt.show()
  
```



Separating Concentric Circles

- RBF-kernel PCA

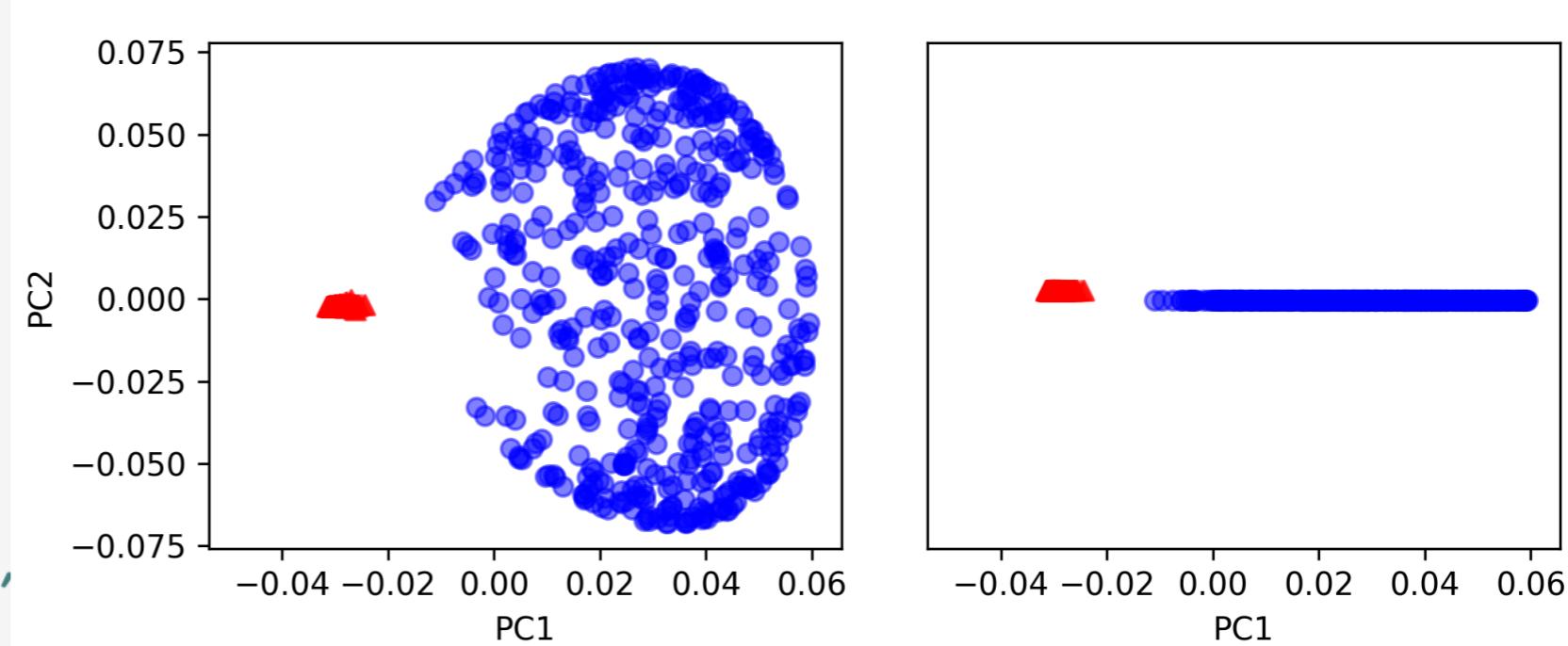
```
x_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)

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.savefig('images/05_17.png',
plt.show()
```



Projecting New Data Points

- For kernel-PCA, we obtain an eigenvector a of the centered kernel matrix (not the covariance matrix)
 - a are samples that are already projected onto the principal component axis v .
 - For new data sample x' , the projection computes $\phi(x')^T v$

$$\phi(x')^T v = \sum_i a^{(i)} K(x', x^{(i)})$$

- Kernel PCA is a memory-based method, because we need original training set $x^{(i)}$ each time to project new samples
- Have to normalize the eigenvector a by its eigenvalue

Modified rbf_kernel_pca Function

```
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_samples, 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_samples, k_features]
        Projected dataset

    lambdas: list
        Eigenvalues
    """

    """
```

Modified rbf_kernel_pca Function

```
# 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 samples)
alphas = np.column_stack((eigvecs[:, i]
                           for i in range(n_components)))

# Collect the corresponding eigenvalues
lambdas = [eigvals[i] for i in range(n_components)]

return alphas, lambdas
```

Half-moon Dataset Example

- A new half-moon dataset and project onto 1D subspace

```
x, y = make_moons(n_samples=100, random_state=123)
alphas, lambdas = rbf_kernel_pca(X, gamma=15, n_components=1)
```

- Assume the 26th point is a new data \mathbf{x}' and project it onto new subspace

```
x_new = X[25]
x_new
array([ 1.8713,  0.0093])
```

```
x_proj = alphas[25] # original projection
x_proj
```

```
array([ 0.0788])
```

```
def project_x(x_new, X, gamma, alphas, lambdas):
    pair_dist = np.array([np.sum((x_new - row)**2) for row in X])
    k = np.exp(-gamma * pair_dist)
    return k.dot(alphas / lambdas)

# projection of the "new" datapoint
x_reproj = project_x(x_new, X, gamma=15, alphas=alphas, lambdas=lambdas)
x_reproj
```

Half-moon Dataset Example

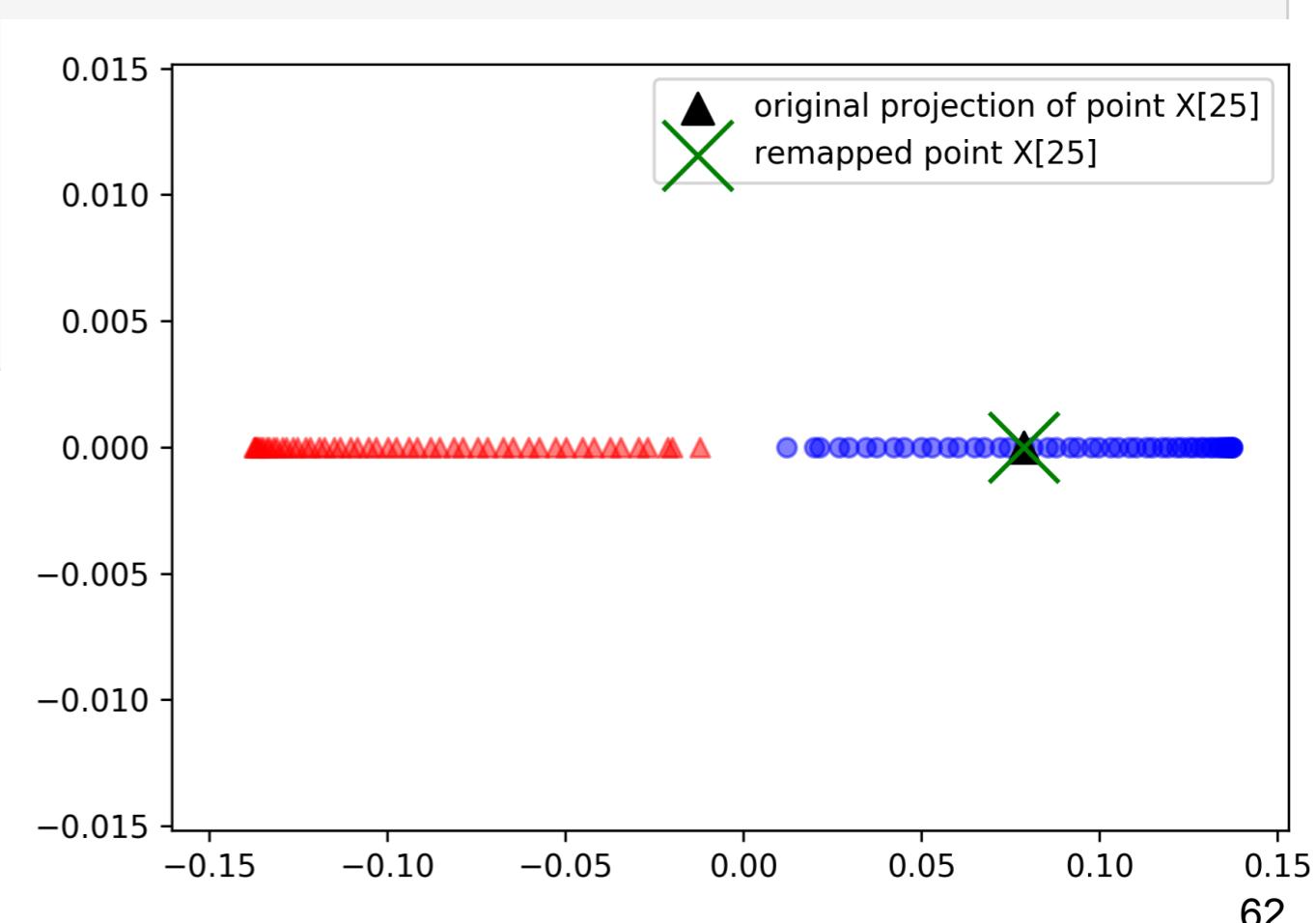
- Visualize the projection

```

plt.scatter(alphas[y == 0, 0], np.zeros((50)),
           color='red', marker='^', alpha=0.5)
plt.scatter(alphas[y == 1, 0], np.zeros((50)),
           color='blue', marker='o', alpha=0.5)
plt.scatter(x_proj, 0, color='black',
            label='original projection of point X[25]', marker='^', s=100)
plt.scatter(x_reproj, 0, color='green',
            label='remapped point X[25]', marker='x', s=500)
plt.legend(scatterpoints=1)

plt.tight_layout()
# plt.savefig('images/05_18.png', dpi=300)
plt.show()

```



Kernel PCA in Scikit-learn

- A kernel PCA class in the **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)

plt.scatter(X_skernpca[y == 0, 0], X_skernpca[y == 0, 1],
            color='red', marker='^', alpha=0.5)
plt.scatter(X_skernpca[y == 1, 0], X_skernpca[y == 1, 1],
            color='blue', marker='o', alpha=0.5)

plt.xlabel('PC1')
plt.ylabel('PC2')
plt.tight_layout()
# plt.savefig('images/05_19.png', dpi=300)
plt.show()
```

