PATTERN RECOGNITION USING PYTHON

Dimensionality Reduction

Wen-Yen Hsu

Dept Electrical Engineering

Chang Gung University, Taiwan

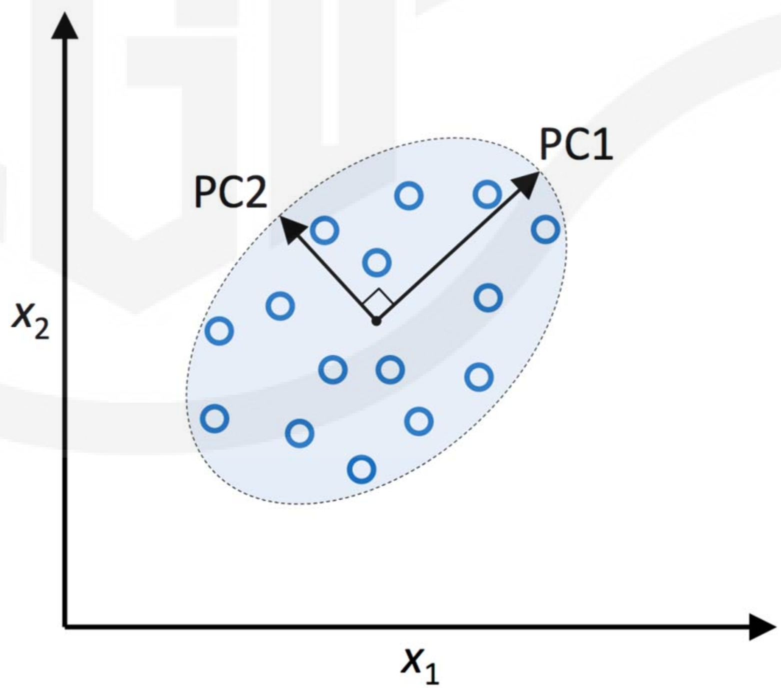
2019-Spring

Approach to Feature Selection

- **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

- PCA find the directions of maximum variance in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions than the original one
- The orthogonal axes (principal components) of the new subspace can be interpreted as the directions of maximum variance
- **PC1** and **PC2** are the principal components.



Transformation Matrix

- Construct a $d \times k$ –dimensional transformation matrix W that to map a sample vector x onto a new k–dimensional feature subspace that has **fewer dimensions** than the original d–dimensional feature space (typically $k << d$)

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

$$\downarrow \boldsymbol{xW}, \quad \boldsymbol{W} \in \mathbb{R}^{d \times k}$$

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

Covariance Matrix

- Covariance between two features, and $\boldsymbol{\mu}$ are the sample means

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

- Covariance matrix Σ of three features

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

- The eigenpairs of the covariance matrix： eigenvalue $\boldsymbol{\lambda}$, eigenvector $\boldsymbol{v}$

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

Feature Transformation

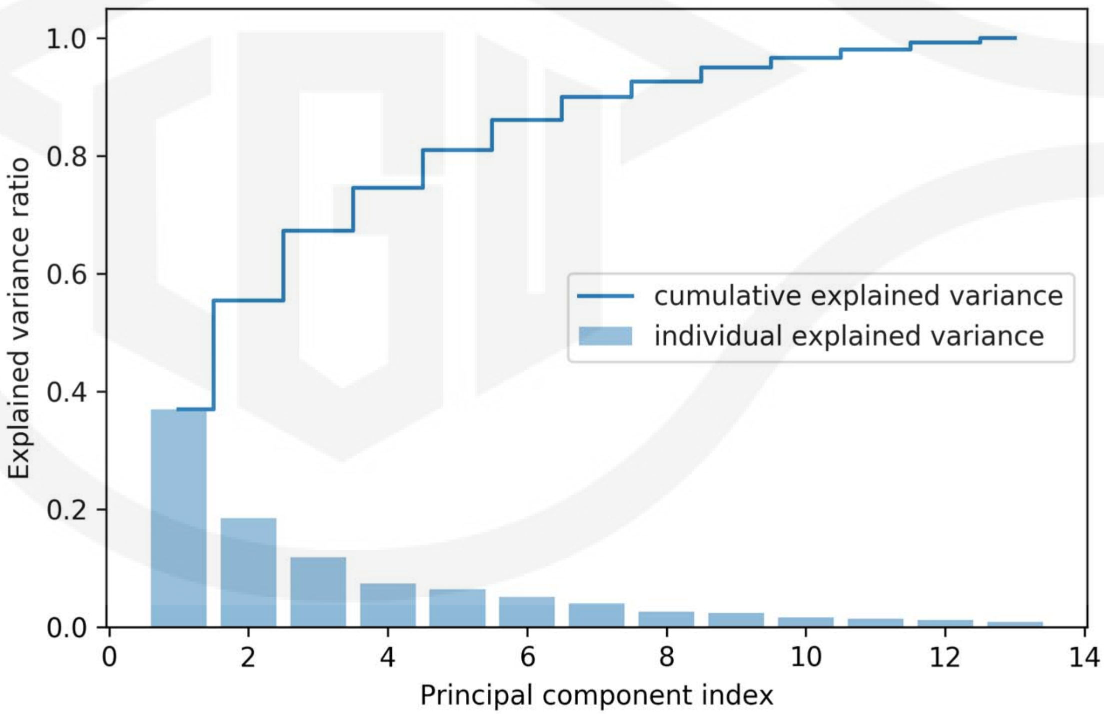
- Select k eigenvectors, which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$)
- Construct a projection matrix W from the "top" k eigenvectors
- Transform the d -dimensional input dataset X using the projection matrix W to obtain the new k -dimensional feature subspace

$$X' = XW$$

Variance Explained Ratios

- The first principal component alone accounts for approximately 40% of the variance, the first two principal components combined explain almost 60% of the variance in this case

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



Principal Component Analysis in Code

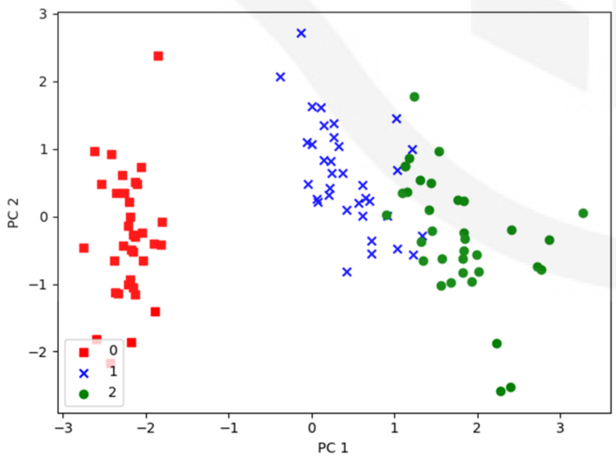
- Steps to perform PCA

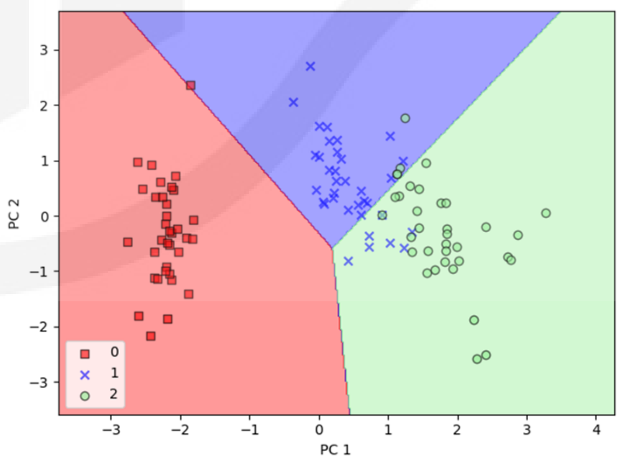
```
# standardize the dataset.
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
# construct the covariance matrix
cov_mat = np.cov(X_train_std.T)
# decompose the covariance matrix
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
print('\nEigenvalues \n%s' % eigen_vals)
print('\nEigenvectors \n%s' %eigen_vecs)
# construct a projection matrix W
w = np.column_stack((eigen_vecs[:, i] for i in range(2)))
print('Matrix W:\n', w)
# transform X' = X · W
X_train_pca = X_train_std.dot(w)
X_test_pca = X_test_std.dot(w)
```

Classification After PCA

- Using the result of PCA to classifier

```
lr = LogisticRegression()
lr = lr.fit(X_train_pca, y_train)
# F_1 score
y_hat = lr.predict(X_test_pca)
f1 = f1_score(y_test, y_hat, average='micro')
print('f1 score (PCA) =', "%.2f" % f1)
```





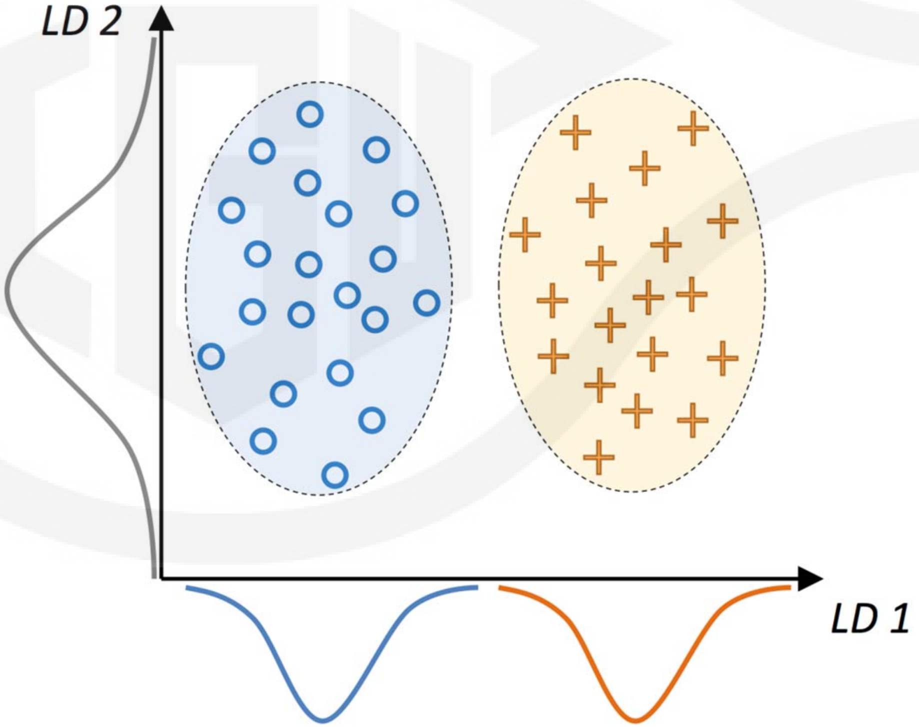
PCA via sklearn

- Using sklearn to perform PCA

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2, svd_solver='randomized')
# pca = PCA(n_components='mle', svd_solver='auto')
X_train_pca_sk = pca.fit_transform(X_train_std)
X_test_pca_sk = pca.transform(X_test_std)
print('X'+"'"+' Dimension=', X_train_pca_sk.shape)
lr_sk = LogisticRegression()
lr_sk = lr_sk.fit(X_train_pca_sk, y_train)
y_hat_sk = lr_sk.predict(X_test_pca_sk)
f1_sk = f1_score(y_test, y_hat_sk, average='micro')
print('f1 score (SK PCA)=', "%.2f" % f1_sk)
```

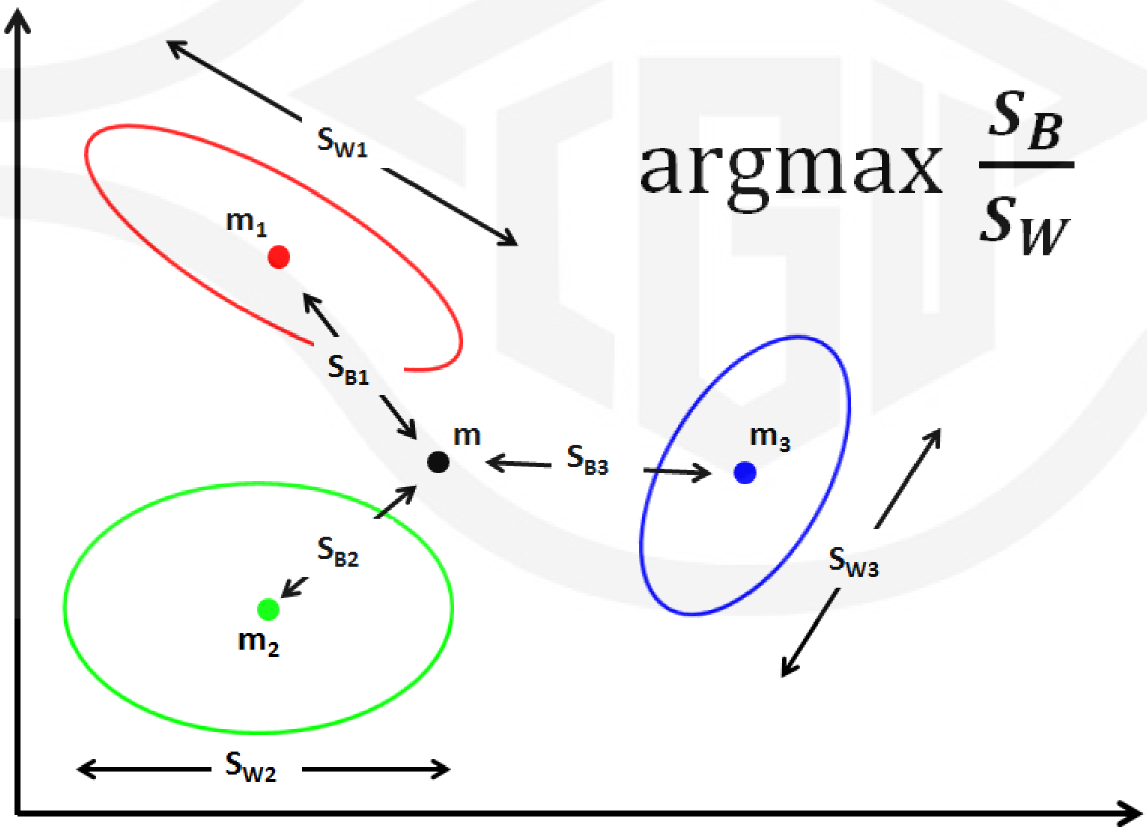
Linear Discriminant

- LDA is to find the feature subspace that optimizes class separability
- LDA takes **class label information** into account



Optimizes Class Separability

- Minimize the variance within class (S_W), maximum the distance between class (S_B)
- Solve the eigenpairs of the matrix $S_W^{-1}S_B$



$$\operatorname{argmax} \frac{S_B}{S_W}$$

Within-class Scatter Matrix

- Compute the within-class scatter matrix

$$\mathbf{S}_W = \sum_{i=1}^{c} \mathbf{S}_i$$

- Divide the scatter matrices by the number of class-samples $\boldsymbol{n}_i$, computing the scatter matrix is in fact the same as computing the covariance matrix

$$\Sigma_i = \frac{1}{n_i}\mathbf{S}_i = \frac{1}{n_i}\sum_{x \in D_i}^{c}\left(\mathbf{x}-\mathbf{m}_i\right)\left(\mathbf{x}-\mathbf{m}_i\right)^T$$

Between-class Scatter Matrix

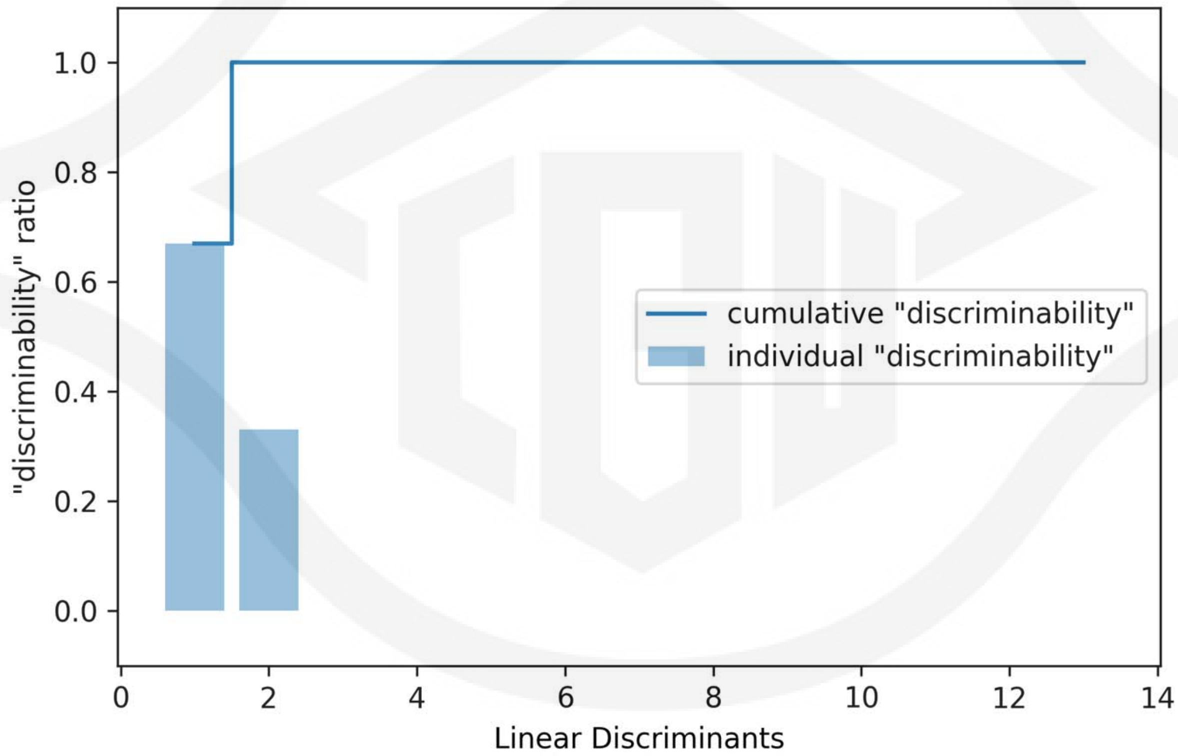
- Compute the between-class scatter matrix

$$\boldsymbol{S}_B = \sum_{i=1}^{c} n_i \left(\boldsymbol{m}_i - \boldsymbol{m}\right)\left(\boldsymbol{m}_i - \boldsymbol{m}\right)^T$$

- $\boldsymbol{m}$ is the overall mean including samples from all classes

Discriminability

- The content of class-discriminatory information



Linear Discriminant Analysis in Code

- Compute the scatter matrix

```
# Compute the within-class scatter matrix SW
d = X.shape[1]  # number of features
S_W = np.zeros((d, d))
# for label, mv in zip(np.unique(y), mean_vecs):
for label in np.unique(y):
    class_scatter = np.cov(X_train_std[y_train == label].T)
    S_W += class_scatter
# Compute the between-class scatter matrix SB
mean_overall = np.mean(X_train_std, axis=0)
d = X.shape[1]  # number of features
S_B = np.zeros((d, d))
for i, mean_vec in enumerate(mean_vecs):
    n = X_train[y_train == i, :].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)
```

Linear Discriminant Analysis in Code (Cont.)

- Solve the eigenpairs

```
# Solve the generalized eigenvalue
eigen_vals, eigen_vecs =
np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
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)
print('Eigenvalues in descending order:\n')
for eigen_val in eigen_pairs:
    print(eigen_val[0])
w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
              eigen_pairs[1][1][:, np.newaxis].real))
print('Matrix W:\n', w)
X_train_lda = X_train_std.dot(w)
```

Linear Discriminant Analysis via sklearn

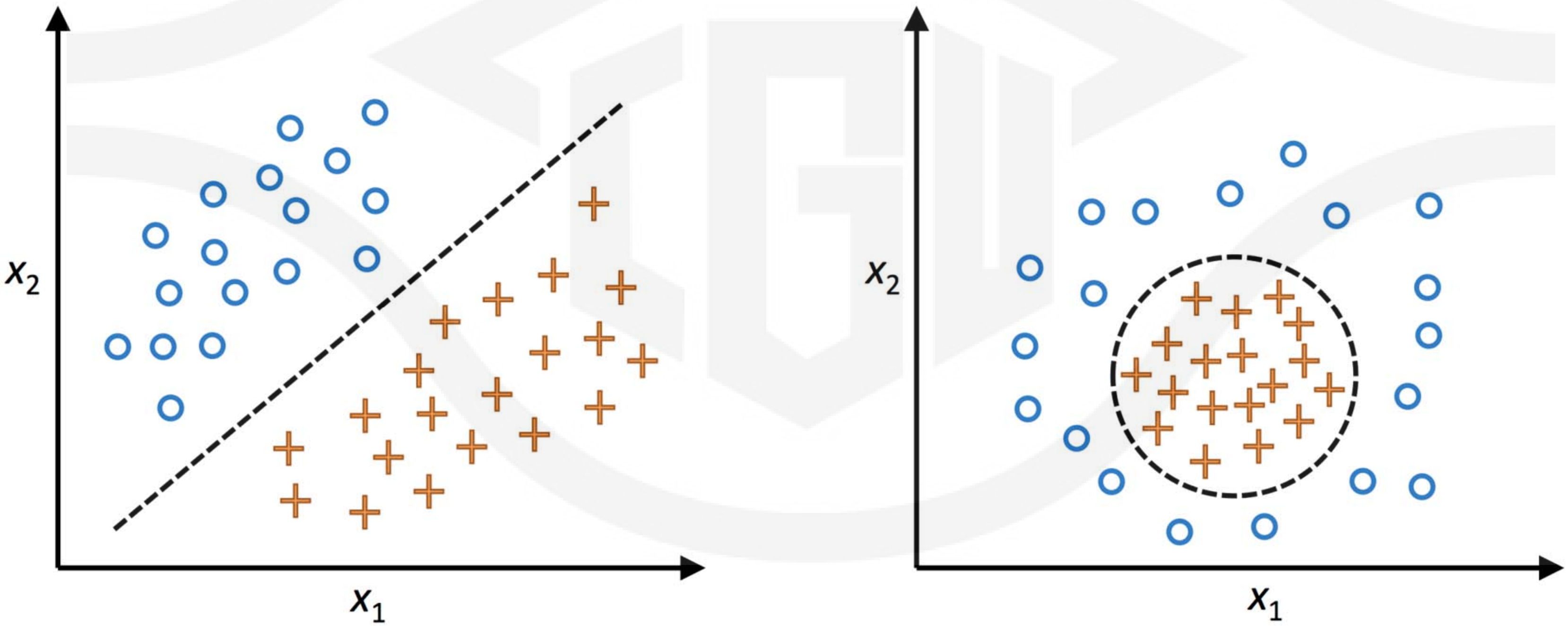
- Using sklearn to perform LDA

```
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis as LDA

lda = LDA(n_components=2)
X_train_lda = lda.fit_transform(X_train_std, y_train)
X_test_lda = lda.transform(X_test_std)
lr = LogisticRegression()
lr = lr.fit(X_train_lda, y_train)
y_hat = lr.predict(X_test_lda)
f1 = f1_score(y_test, y_hat, average='micro')
print('f1 score (LDA) =', "%.2f" % f1)
```

Kernel Principal Component Analysis

- Using kernel PCA learn how to transform data that is not linearly separable onto a new, lower-dimensional subspace that is suitable for linear classifiers



Nonlinear Mapping

- Nonlinear mapping via kernel PCA that transforms the data into a higher-dimensional space (Reproducing Kernel Hilbert Space, RKHS), then use standard PCA in RKHS to project the data back onto a lower-dimensional space where the samples can be separated by a linear classifier

$$\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k >> d)$$

Kernel Method and Kernel Trick

- Can we find a function $\kappa(x^{(i)}, x^{(j)})$ in the original space let $\kappa(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^{\mathrm{T}} \phi(x^{(j)})$?
- If this function exists, then we **only need** to calculate the value of the function κ in the low-dimensional space, **without** mapping the data to the high-dimensional space, and then solving the mapped inner product through complex calculations
- Radial Basis Function (RBF) or Gaussian kernel $\gamma = \dfrac{1}{2\sigma}$

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

Implement an RBF kernel PCA

- Obtain the eigenvectors—the principal components—from covariance matrix Σ by extracting the eigenvectors of the kernel (similarity) matrix $\boldsymbol{K}$

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

- Cannot guarantee that the new feature space is also centered at zero, need to center the kernel matrix $\boldsymbol{K}$

$$\boldsymbol{K}' = \boldsymbol{K} - 1_{\boldsymbol{n}}\boldsymbol{K} - \boldsymbol{K}1_{\boldsymbol{n}} + 1_{\boldsymbol{n}}\boldsymbol{K}1_{\boldsymbol{n}}$$

Kernel Principal Component Analysis in Code

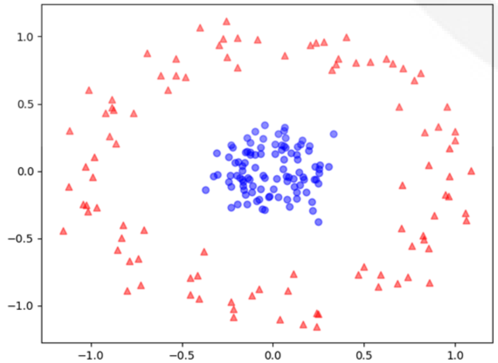
- Implementing a kernel principal component analysis

```
def rbf_kernel_pca(X, gamma, n_components):
    # 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
```

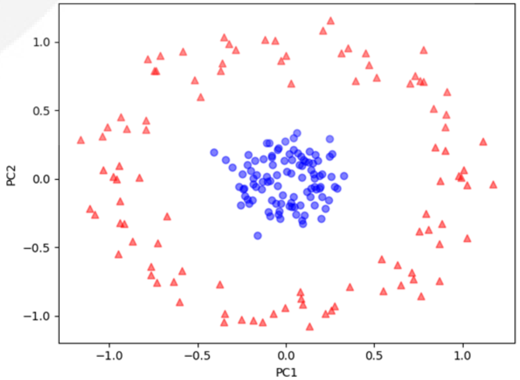
Nonlinear Mappings via PCA

- Concentric circles

```
from sklearn.datasets import make_circles
## plot non-linear picture
X, y = make_circles(n_samples=200, 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()
scikit_pca = PCA(n_components=2)
X_spca = scikit_pca.fit_transform(X)
```



PCA



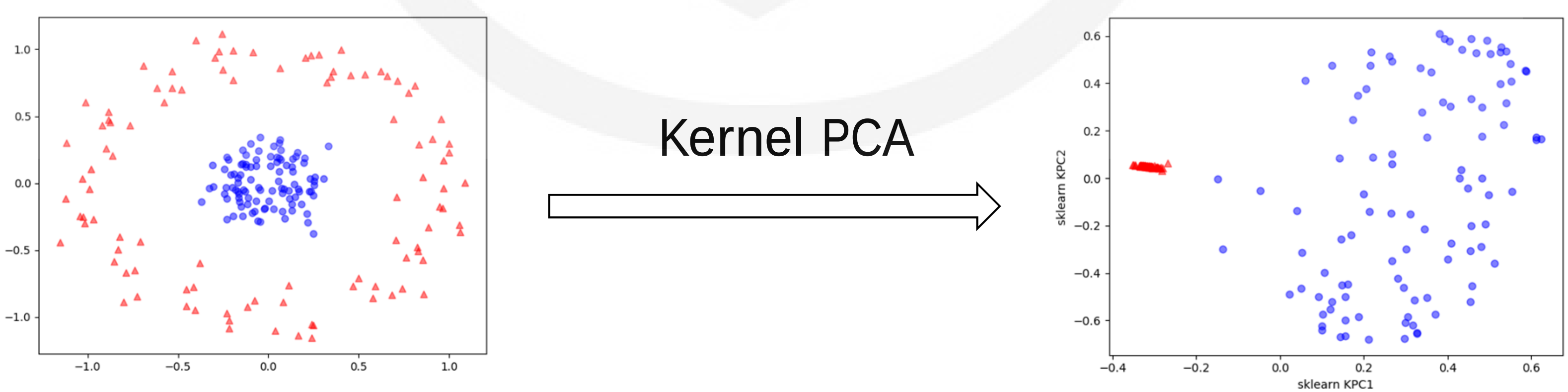
Nonlinear Mappings via Kernel PCA

- Call `rbf_kernel_pca(X, gamma, n_components)`

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

- Using Sklearn to perform kernel PCA

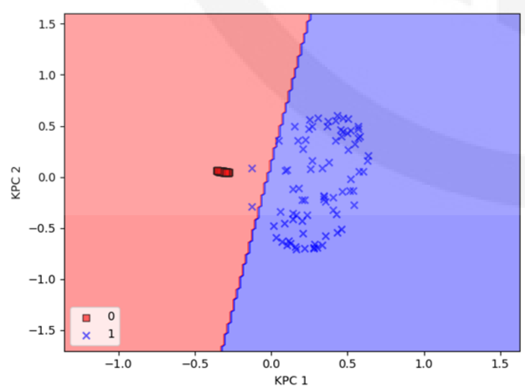
```
from sklearn.decomposition import KernelPCA
scikit_kpca = KernelPCA(n_components=2, kernel='rbf',
gamma=15)
X_skernpca = scikit_kpca.fit_transform(X)
```

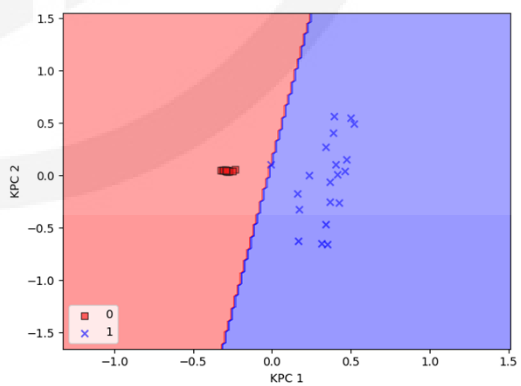


Classification

- Using the result of KPCA to classifier

```
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_train_skkpca = kpca.fit_transform(X_train)
X_test_skkpca = kpca.transform(X_test)
lr = LogisticRegression()
lr = lr.fit(X_train_skkpca, y_train)
y_hat = lr.predict(X_test_skkpca)
f1 = f1_score(y_test, y_hat, average='micro')
print('f1 score =', "%.2f" % f1)
```





Reference

- Sebastian Raschka, Vahid Mirjalili. Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow. Second Edition. Packt Publishing, 2017.