Dimensionality Reduction

COMP3314
Machine Learning

Outline

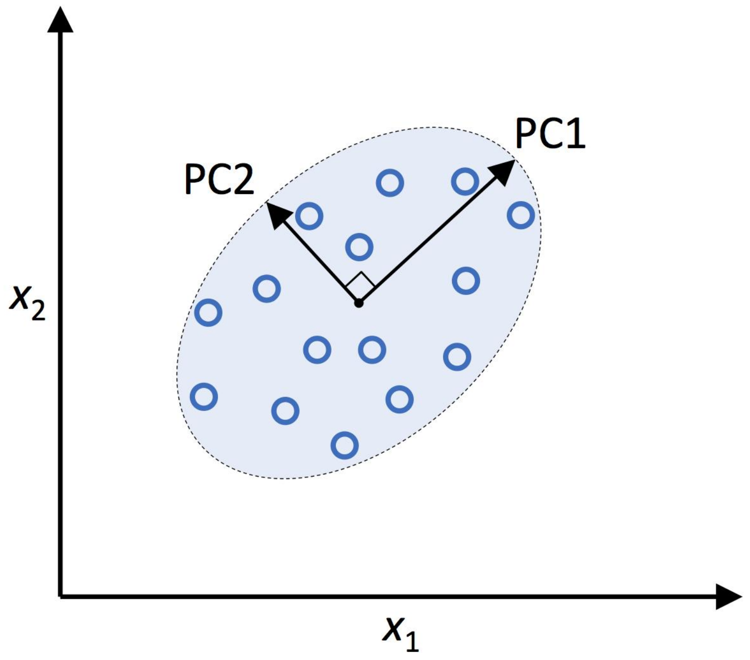
- Principal Component Analysis (PCA) for unsupervised data compression
- Linear Discriminant Analysis (LDA) as a supervised dimensionality reduction technique for maximizing class separability

Feature Extraction

- Transforms (projects) the data onto a new feature space
  - In contrast feature selection maintains the original features
- Approach to data compression with the goal of maintaining most of the relevant information
  - Not just useful to improve storage space or computational efficiency, but can also improve predictive performance by reducing the curse of dimensionality

PCA - Principal Component Analysis

- An unsupervised linear transformation technique
- Aims to find the directions of maximum variance and projects it onto a new subspace with fewer (or equal) dimensions
- 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



Transformation Matrix

- Construct a $d \times k$ transformation matrix **W**
  - Maps a d-dimensional vector **x** to a k-dimensional vector **z**
  - Typical $d >> k$

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

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

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

- The first PC has the largest variance, and all consequent PCs will have the largest variance given the constraint that these components are uncorrelated (orthogonal) to the other PCs
  - Even if the input features are correlated, the resulting PCs will be mutually orthogonal (uncorrelated)

Standardization

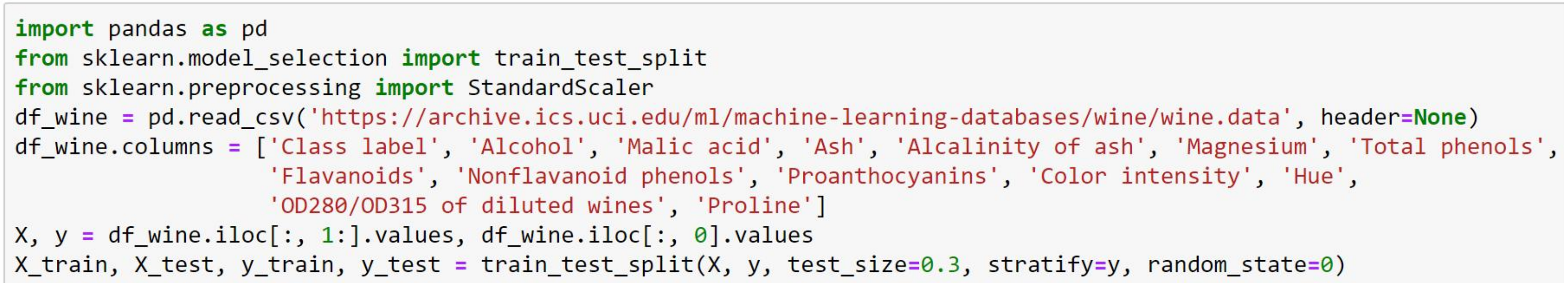
- PCs are highly sensitive to data scaling
- We need to standardize the features prior to PCA if the features were measured on different scales

PCA - Approach

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

Step 1: Standardize the d-dimensional dataset

- Let’s apply the first four steps of the PCA on the wine dataset
- In the following we load the wine dataset and split it into separate train and test sets
- Then we apply step 1: Standardize the *d*-dimensional dataset



```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
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']
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)
```

Step 2: Construct the covariance matrix

- The $d \times d$ covariance matrix stores the pairwise covariances between the different features
- The covariance between two features x_j and x_k on the population level can be calculated via the following equation

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

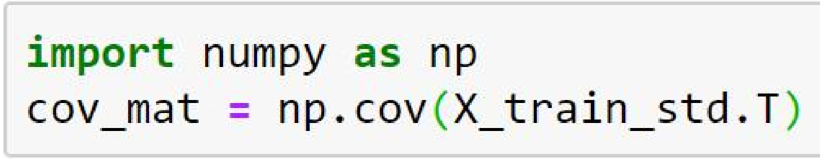
- Here, μ_j and μ_k are the sample means of features *j* and *k*, respectively
  - Note that the sample means are zero if we standardized the dataset
- A positive covariance between two features indicates that the features increase or decrease together, whereas a negative covariance indicates that the features vary in opposite directions

Step 2: Construct the covariance matrix

- For example, the covariance matrix of three features can then be written as follows

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

- The eigenvectors of 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 x 13-dimensional covariance matrix



```python
import numpy as np
cov_mat = np.cov(X_train_std.T)
```

Step 3: Covariance matrix decomposition

- In step 3 we obtain the eigenpairs of the covariance matrix
- Recall from your introductory linear algebra classes, an eigenvector **v** satisfies the following condition

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

- Here, λ is a scalar: the eigenvalue

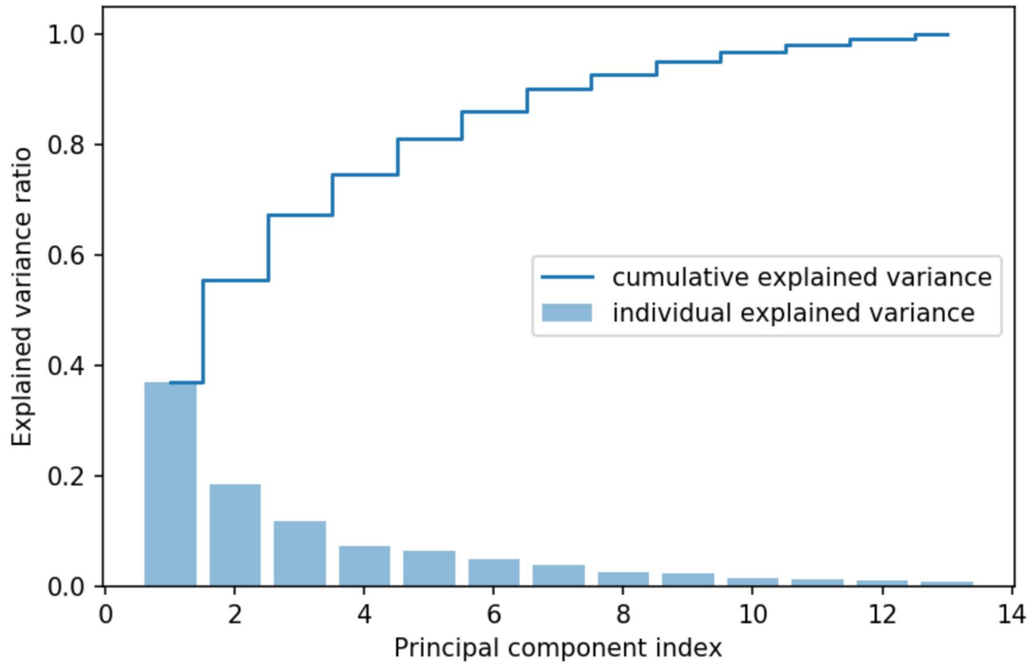
```
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
```

Explained variance

- We only select the subset of the eigenvectors (principal components) that contains most of the information (variance)
  - Top *k* eigenvectors based on the values of their corresponding eigenvalues
- Before we collect those *k* most informative eigenvectors, let us plot the explained variance of the eigenvalues
- The explained variance of an eigenvalue λ_j is simply the fraction of an eigenvalue λ_j and the total sum of the eigenvalues

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

```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 150
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
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()
```



Step 4: Sort the eigenvalues

- Sort eigenpairs by decreasing order of the eigenvalues

```
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i]) for i in range(len(eigen_vals))]
eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

Step 5 & 6: Select k eigenvectors and construct $\mathbf{W}$

- Select k eigenvectors which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$)
  - We collect the $k = 2$ eigenvectors that correspond to the two largest eigenvalues, to capture about 60 percent of the variance in this dataset

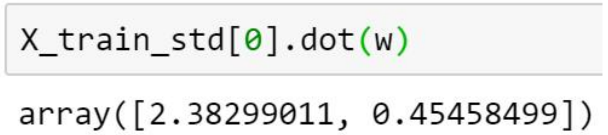
```
w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
               eigen_pairs[1][1][:, np.newaxis]))
w
```

```
array([[-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]])
```

Step 7: Use the projection matrix

- Using the projection matrix, we can now transform a sample 1 x 13 row vector **x** onto the PCA subspace obtaining **x'**

$$\boldsymbol{x'} = \boldsymbol{xW}$$

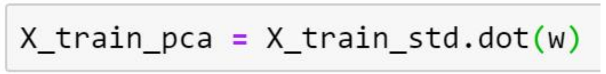


```
X_train_std[0].dot(w)
```

```
array([2.38299011, 0.45458499])
```

- We can transform the entire 124 x 13-dimensional training dataset onto the two principal components by calculating

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

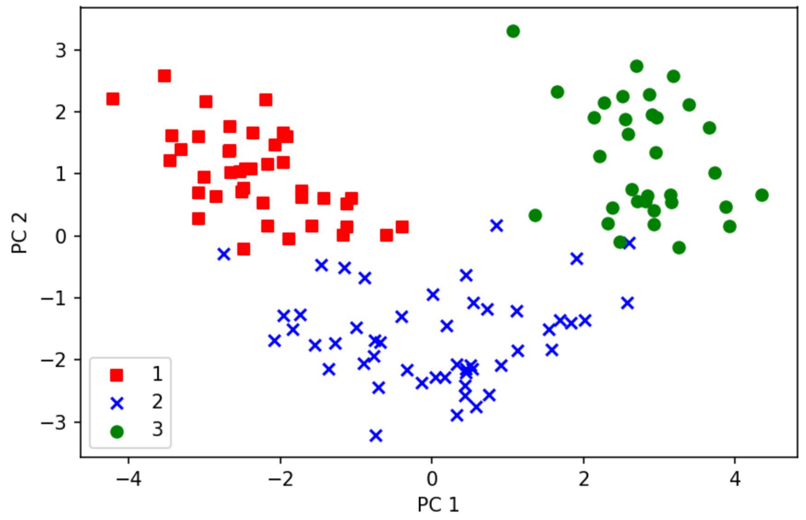


```
X_train_pca = X_train_std.dot(w)
```

Visualization

- Let us visualize the transformed Wine training set, now stored as an 124 x 2-dimensional matrix

```
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()
```



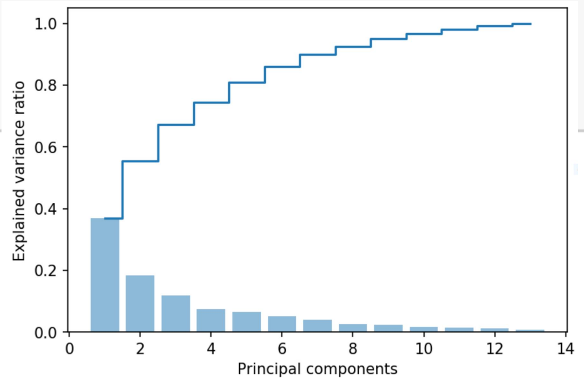
PCA in scikit-learn

- The PCA class from scikit-learn is a transformer class
  - First fit the model using the training data then transform both the training data and the test dataset using the same model parameters
- Let’s replicate the results from our own PCA implementation in scikit-learn

```
from sklearn.decomposition import PCA
pca = PCA()
X_train_pca = pca.fit_transform(X_train_std)
pca.explained_variance_ratio_
```

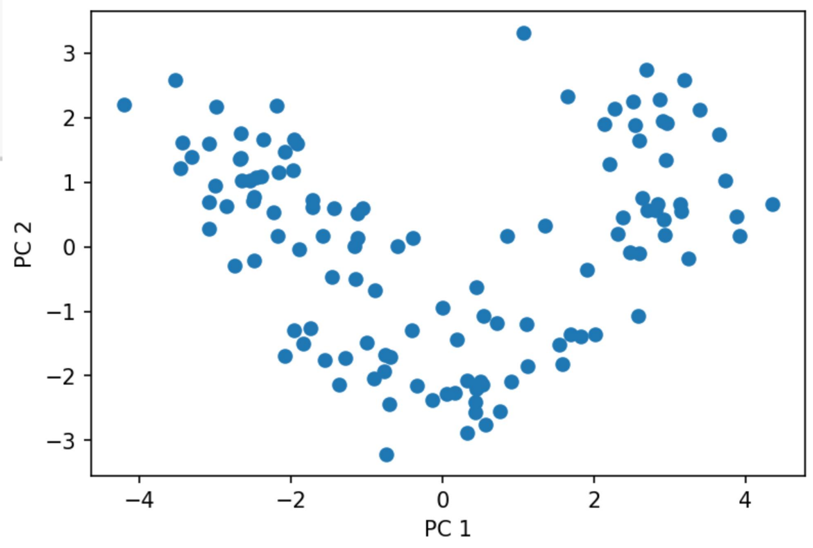
```
array([0.36951469, 0.18434927, 0.11815159, 0.07334252, 0.06422108,
       0.05051724, 0.03954654, 0.02643918, 0.02389319, 0.01629614,
       0.01380021, 0.01172226, 0.00820609])
```

```
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()
```



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

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

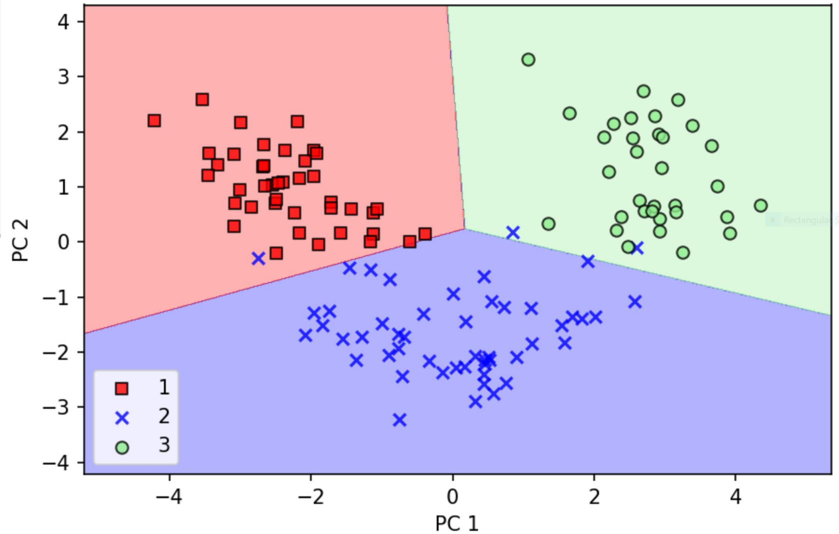


PCA in scikit-learn

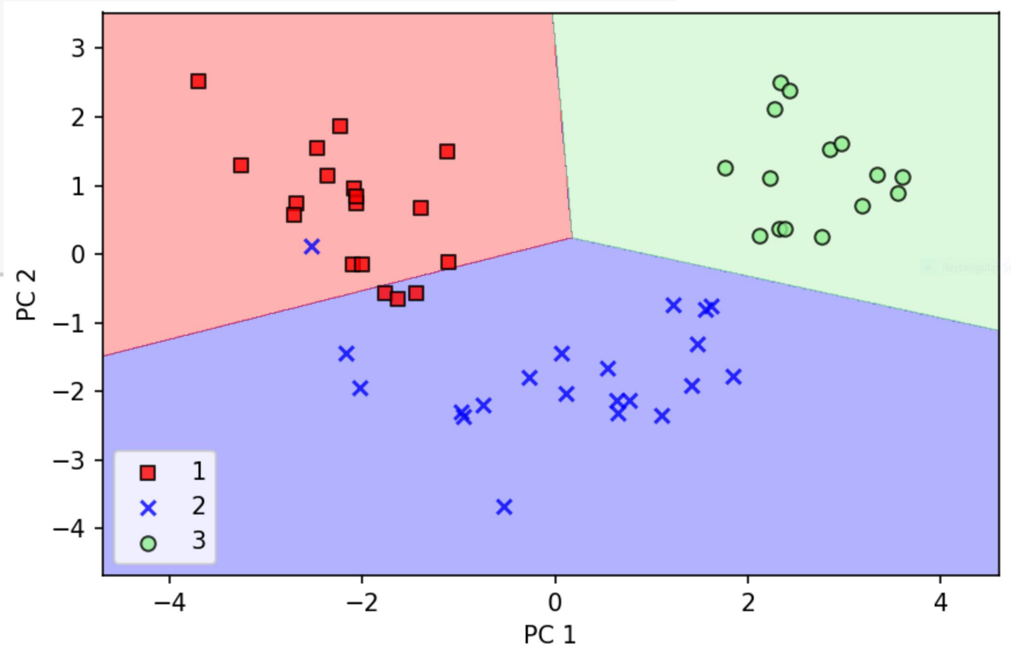
- Let's now apply the PCA class to the wine training dataset, classify the transformed samples via logistic regression, and visualize the decision regions via the plot_decision_regions function defined in Chapter 2

```
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(solver='liblinear', multi_class='ovr')
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.show()
```



```
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.show()
```

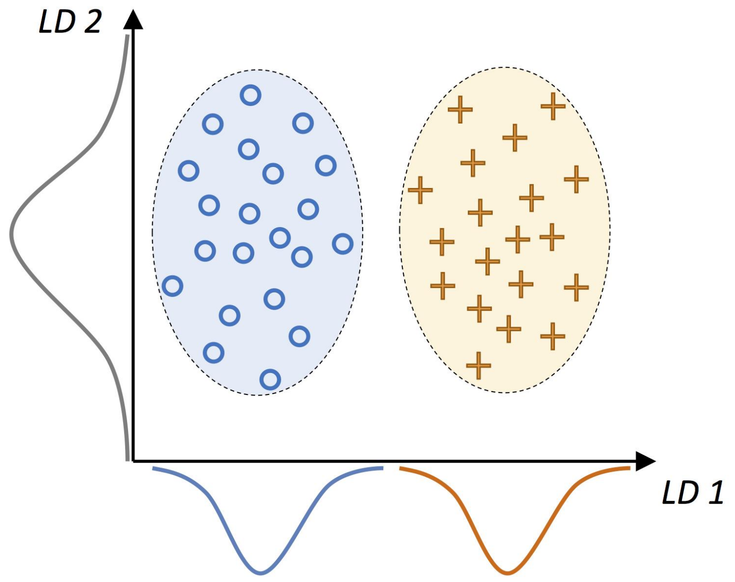


Outline

- Principal Component Analysis (PCA) for unsupervised data compression
- Linear Discriminant Analysis (LDA) as a supervised dimensionality reduction technique for maximizing class separability

LDA - Linear Discriminant Analysis

- The linear discriminant LD1 would separate the two normal distributed classes well
- The linear discriminant LD2 would fail as a good linear discriminant since it does not capture any of the class-discriminatory information
- LDA assumes that the data is normally distributed and that the classes have identical covariance matrices and that the samples are statistically independent of each other
  - If one or more of those assumptions are (slightly) violated, LDA can still work reasonably well



LDA vs. PCA

- The general concept behind LDA is very similar to PCA
  - Both PCA and LDA are linear transformation techniques that can be used to reduce the number of dimensions in a dataset
  - PCA is unsupervised, LDA is supervised
  - PCA attempts to find the orthogonal component axes of maximum variance in a dataset
  - In LDA the goal is to find the feature subspace that optimizes class separability
- We might intuitively think that LDA is a superior feature extraction technique for classification tasks compared to PCA.
  - However, it has been 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 samples

LDA - Approach

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 $\mathbf{S_B}$ and the within-class scatter matrix $\mathbf{S_W}$
4. Compute the eigenvectors and corresponding eigenvalues of the matrix $\mathbf{S_W}^{-1}\mathbf{S_B}$
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 $\mathbf{W}$
   - The eigenvectors are the columns of this matrix
7. Project the samples onto the new feature subspace using the transformation matrix $\mathbf{W}$

Step 1: Standardize the d-dimensional dataset

- Since we already standardized the features of the Wine dataset in the PCA section at the beginning of this chapter, we can skip the first step and proceed with the next step

Step 2: Compute Mean Vector

- Each mean vector **m$_i$** stores the mean feature value μ_m with respect to the samples of class i

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

- This results is three mean vectors

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

```
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' % (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]
```

Step 3: Construct the scatter matrix

- In step 3, we construct the between-class scatter matrix $\mathbf{S_B}$ and the within-class scatter matrix $\mathbf{S_W}$
- Using the mean vectors, we can compute the within-class scatter matrix

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

- This is calculated by summing up the individual scatter matrices $\mathbf{S_i}$ of each individual class i

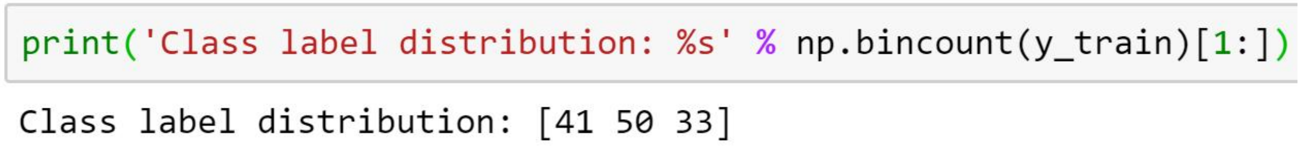
$$\boldsymbol{S}_i = \sum_{\boldsymbol{x} \in D_i}^{c} \left(\boldsymbol{x} - \boldsymbol{m}_i\right)\left(\boldsymbol{x} - \boldsymbol{m}_i\right)^T$$

```
d = 13
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
```

Step 3: Construct the scatter matrix

- The assumption that we are making when we are computing the scatter matrices is that the class labels in the training set 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]
```

Step 3: Construct the scatter matrix

- Thus, we scale the individual scatter matrices $\mathbf{S_i}$ by dividing it by the number of class samples n_i before we sum them up as scatter matrix $\mathbf{S_W}$
- When we scale the scatter matrices, we can see that computing the scatter matrix is in fact the same as computing the covariance matrix $\mathbf{\Sigma_i}$
  - The covariance matrix is a normalized version of the scatter matrix

$$\boldsymbol{S}_i = \sum_{\boldsymbol{x} \in D_i}^{c} (\boldsymbol{x} - \boldsymbol{m}_i)(\boldsymbol{x} - \boldsymbol{m}_i)^T$$

```
d = 13
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]))
```

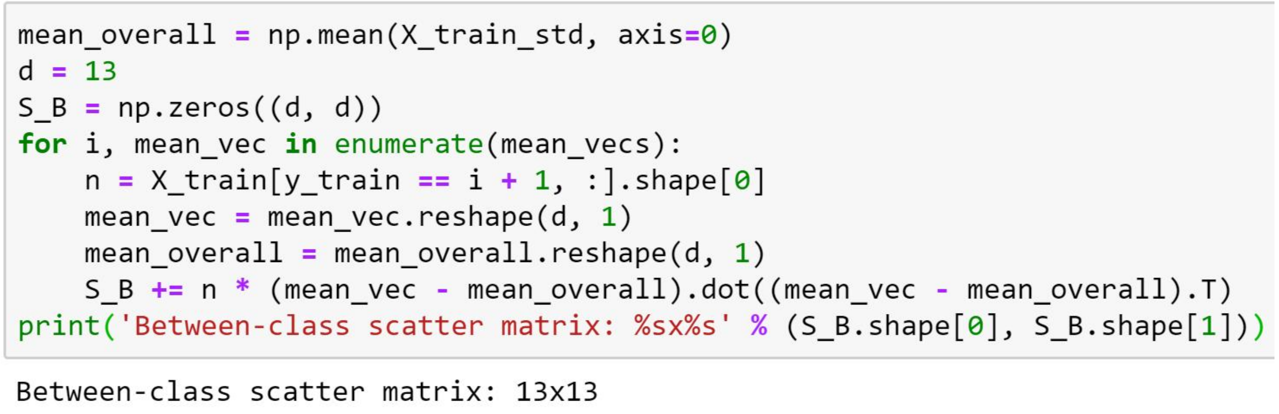
```
Scaled within-class scatter matrix: 13x13
```

Step 3: Construct the scatter matrix

- After we computed the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute the between-class scatter matrix $\mathbf{S_B}$

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

- Here, **m** is the overall mean that is computed, including samples from all classes

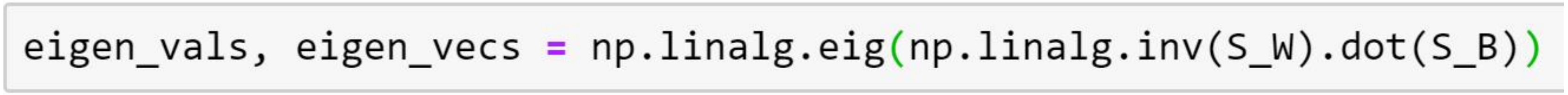


```
mean_overall = np.mean(X_train_std, axis=0)
d = 13
S_B = np.zeros((d, d))
for i, mean_vec in enumerate(mean_vecs):
    n = X_train[y_train == i + 1, :].shape[0]
    mean_vec = mean_vec.reshape(d, 1)
    mean_overall = mean_overall.reshape(d, 1)
    S_B += n * (mean_vec - mean_overall).dot((mean_vec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

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

Step 4: Eigenvectors and Eigenvalues Computations

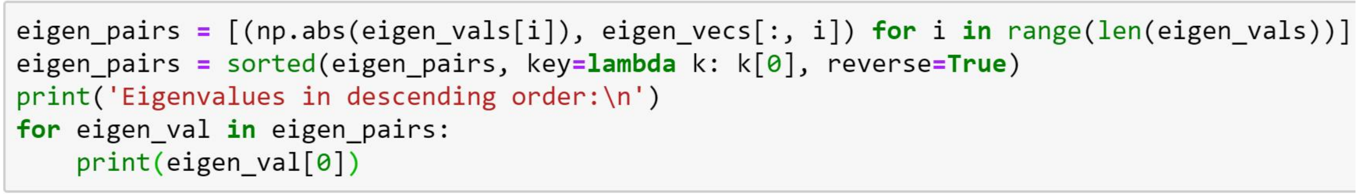
- Instead of performing the eigendecomposition on the covariance matrix, we solve the generalized eigenvalue problem of the matrix $\mathbf{S_W^{-1}S_B}$



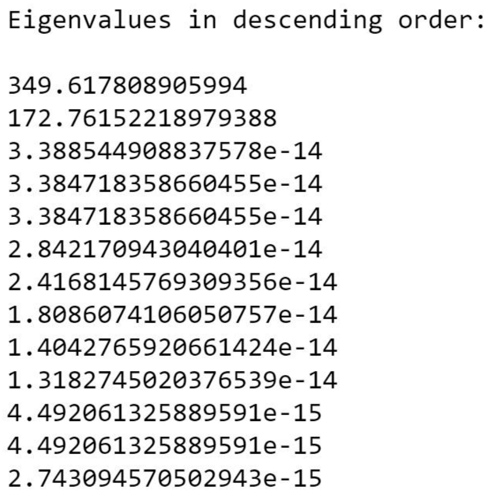
```
eigen_vals, eigen_vecs = np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

Step 5: Sort the eigenvalues

- After we computed the eigenpairs, we can now sort the eigenvalues in descending order



```python
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])
```



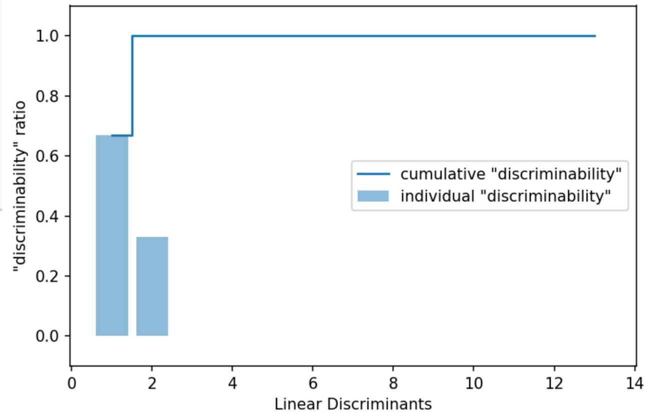
```
Eigenvalues in descending order:

349.617808905994
172.76152218979388
3.3885449088375780e-14
3.384718358660455e-14
3.384718358660455e-14
2.842170943040401e-14
2.4168145769309356e-14
1.8086074106050757e-14
1.4042765920661424e-14
1.3182745020376539e-14
4.492061325889591e-15
4.492061325889591e-15
2.743094570502943e-15
```

- The number of linear discriminants is at most c−1 (c is the number of class labels)
    - This is because the between-class scatter matrix S_B is the sum of c matrices with rank 1 or less

Discriminability

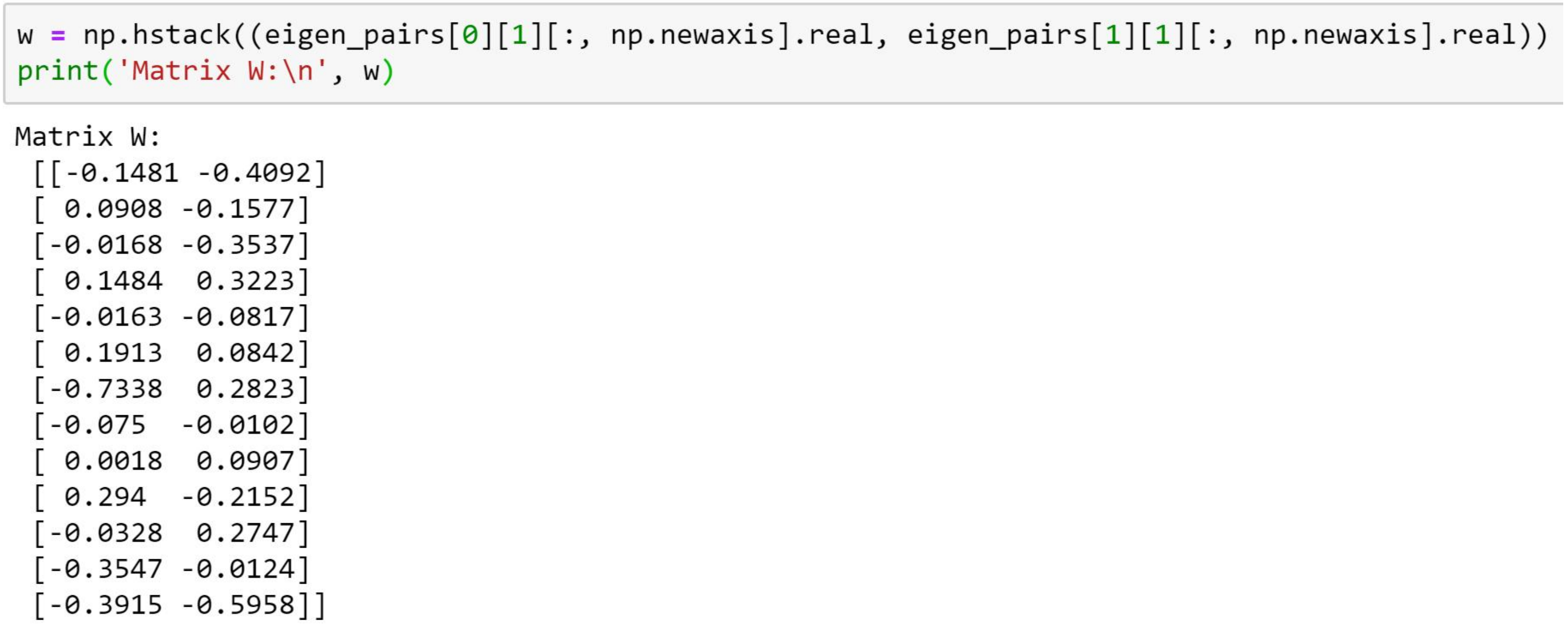
- To measure how much of the class-discriminatory information is captured by the linear discriminants (eigenvectors), let's plot the linear discriminants by decreasing eigenvalues similar to the explained variance plot that we created in the PCA section

```
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()
```



Step 6: Construct **W**

- 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)
```

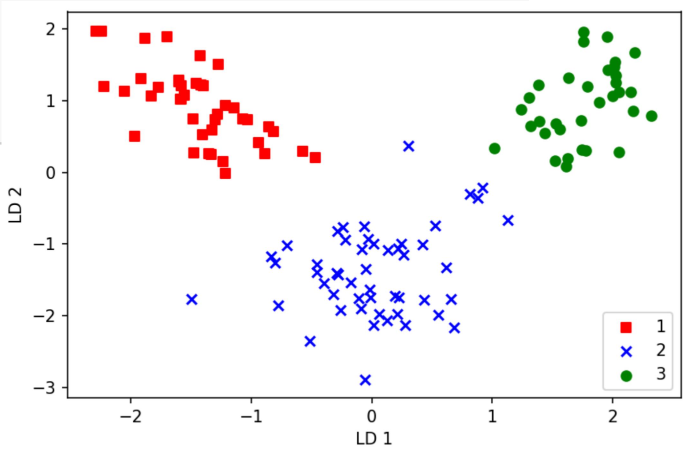
```
Matrix W:
 [[-0.1481 -0.4092]
 [ 0.0908 -0.1577]
 [-0.0168 -0.3537]
 [ 0.1484  0.3223]
 [-0.0163 -0.0817]
 [ 0.1913  0.0842]
 [-0.7338  0.2823]
 [-0.075  -0.0102]
 [ 0.0018  0.0907]
 [ 0.294  -0.2152]
 [-0.0328  0.2747]
 [-0.3547 -0.0124]
 [-0.3915 -0.5958]]
```

Step 7: Project samples onto the new space

- Using the transformation matrix W 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()
```

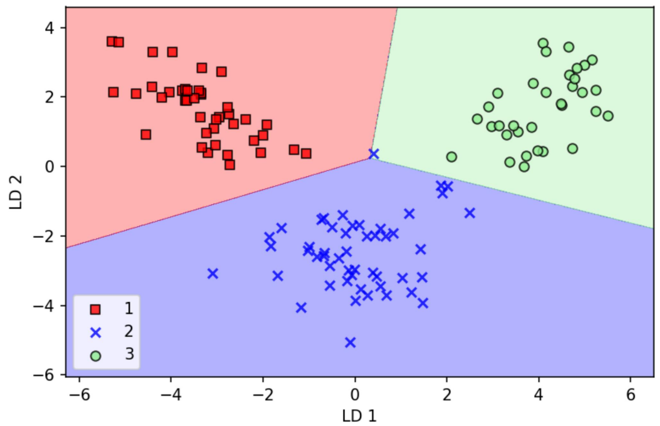


LDA via scikit-learn

- Now, let's look at the 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)
```

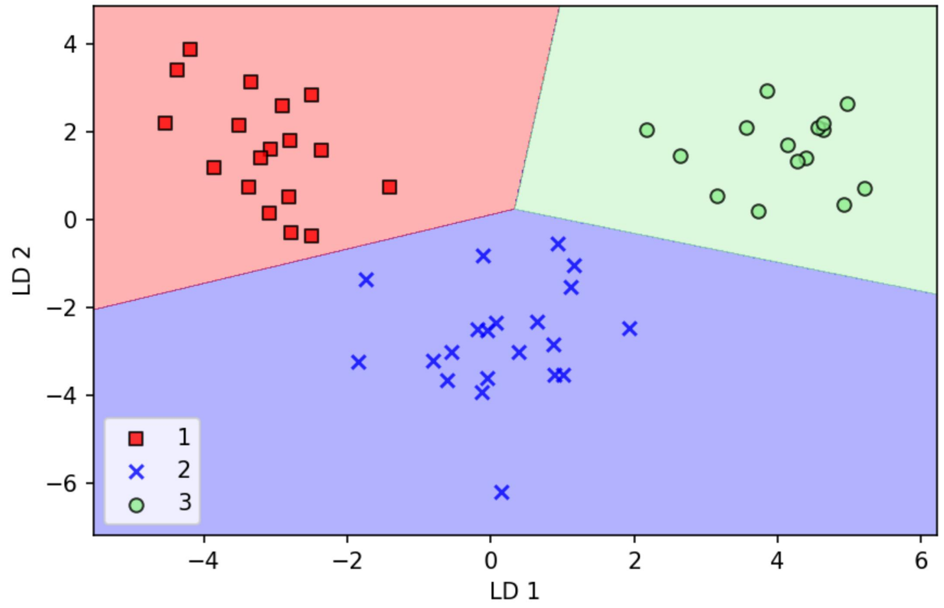
```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(solver='liblinear', multi_class='ovr')
lr = lr.fit(X_train_lda, y_train)
plot_decision_regions(X_train_lda, y_train, classifier=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```



LDA via scikit-learn

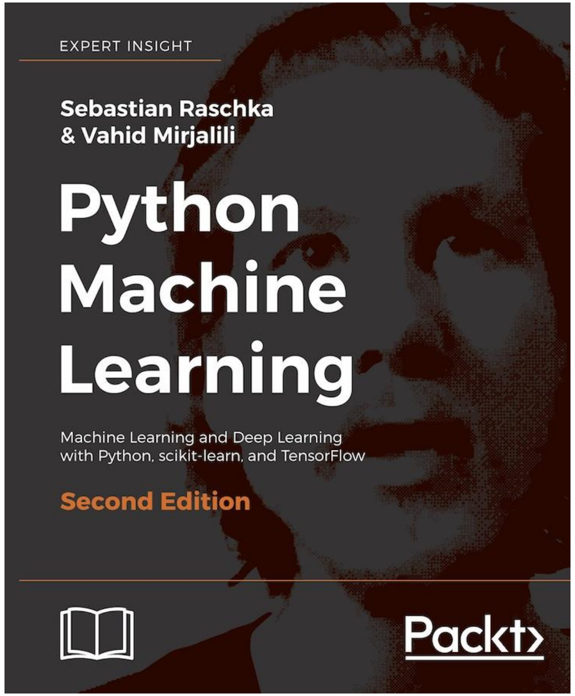
- Let us take a look at the results on the test set

```
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.show()
```



References

- Most materials in this chapter are based on
  - Book
  - Code


EXPERT INSIGHT
Sebastian Raschka & Vahid Mirjalili
Python Machine Learning
Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow
Second Edition
Packt>

Task

- Let **v** be an an eigenvector, show that **-v** is also an eigenvector