Compressing Data via Dimensionality Reduction

Aprendizaje Automático

Ingeniería de Robótica Software

Universidad Rey Juan Carlos

Content

Introduction

- **Different methods** exist for **reducing the dimensionality of datasets**.
- **Feature selection techniques** are one approach to achieve this.
- An alternative to feature selection for dimensionality reduction is **feature extraction**, that involves transforming the dataset into a new feature subspace with lower dimensionality.

Introduction

- **Dimensionality reduction** or data compression is **important** in some cases in **machine learning**.
- **Why**?
  - By transforming high-dimensional data into a **lower-dimensional space**, the **complexity of the dataset is reduced**, which can result in **faster training times** and **less use of computing resources**.
  - By **discarding irrelevant or noisy data** helps to create more **robust models**.
  - By reducing the number of dimensions, it enables a **visualization of the data** (2 or 3 dimensions), which facilitates the identification of patterns and clusters.

Introduction

- The difference between feature selection and feature extraction is that:
  - In **feature selection**, we **select a subset of the original features** from the dataset. The goal is to **identify** and keep the **most relevant features** that contribute to the predictive power of the model.
  - In **feature extraction**, we **transform** or project the original dataset onto a **new feature space**. This involves **creating new features** by **combining the existing ones**, resulting in a reduced dimensionality that can **captures the essential information** in the dataset.

Feature extraction

- **Feature extraction** can be understood as an approach to **data compression** with the goal of **maintaining most of the relevant information**.
- In practice, **feature extraction** is used to:
    - Improve **storage space**.
    - Improve the **computational efficiency** of the learning.
    - Enable a **visualization of the data** (2D or 3D).
    - Improve the predictive performance by **discarding irrelevant or noisy**
    - Reduce the **curse of dimensionality**.

Feature extraction

- What **problems** are caused by a high dimensionality (the **curse of dimensionality**)?
  - In **high dimensions**, data tends to become sparser. This means that **data points are farther apart** (most of the high-dimensional space is empty), **making it difficult to identify patterns** (making clustering and classification tasks challenging).
  - So, in **high dimensions it is required more data** to fill the empty space and obtain meaningful results.
  - If you **don't have more data**, **algorithms are prone to overfitting** (they fail to generalize correctly). In an attempt to capture all the variability in the dataset, the **model** can become **complex, fitting to irrelevant details that do not generalize well to new data**.

Principal Component Analysis (PCA)

- Principal Component Analysis (**PCA**) is an **unsupervised linear transformation technique** widely used across different fields for feature extraction and dimensionality reduction.
- PCA aims to **find the directions of maximum variance in high-dimensional data** and **projects the data onto a new subspace** with equal or **fewer dimensions** than the original one.
- 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.

Principal Component Analysis (PCA)

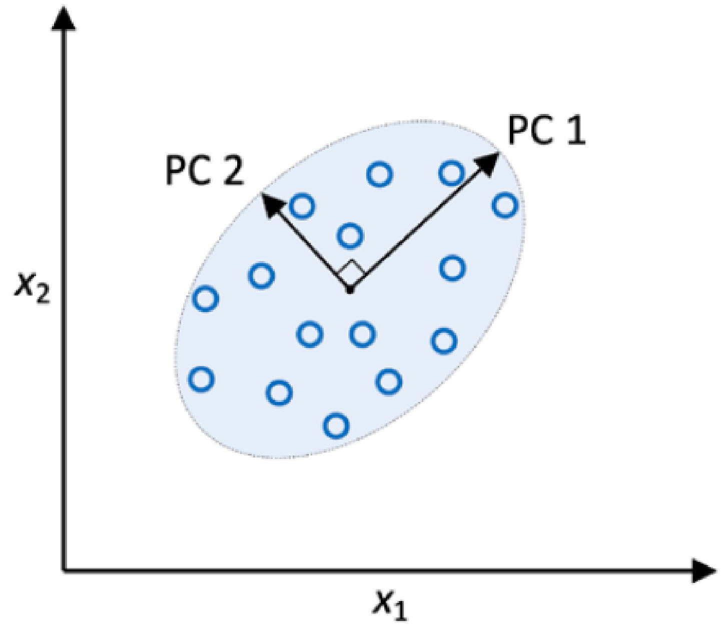


Figure 5.1: Using PCA to find the directions of maximum variance in a dataset

x_1 and x_2 are the original feature axes, and PC 1 and PC 2 are the principal components.

Principal Component Analysis (PCA)

$$\begin{tikzcd} \boldsymbol{x} = [x_1, x_2, \dots, x_d], \quad \boldsymbol{x} \in \mathbb{R}^d \arrow[r] & \boxed{\boldsymbol{W} \in \mathbb{R}^{d\times k}} \arrow[r] & \boldsymbol{z} = [z_1, z_2, \dots, z_k], \quad \boldsymbol{z} \in \mathbb{R}^k \end{tikzcd}$$

$$xW = z$$

***W* *is a d*×*k*-dimensional transformation matrix** that allows us to **map a *d*-dimensional vector of the features of the training example**, *x*, onto a **new *k*-dimensional feature subspace** that has fewer dimensions than the original *d*-dimensional feature space.

Typically, *k* << *d*

Principal Component Analysis (PCA)

- As a result of **transforming the original *d*-dimensional dataset onto this new *k*-dimensional subspace**, the **first principal component** will have **the largest possible variance**.
- All **consequent principal components will have the largest variance given the constraint that these components are uncorrelated (orthogonal)** to the other principal components.
- **Even** if the **input features are correlated**, the **resulting principal components will be mutually orthogonal** (**uncorrelated**).

Principal Component Analysis (PCA)

- **PCA** directions are highly **sensitive to data scaling**, and **we need to standardize the features prior to PCA** if the features were measured on different scales and we want to **assign equal importance to all features**.
- **Standardization** typically involves:
  - **Centering the data**: **Subtracting the mean** of each feature from the dataset so that each feature has a **mean of zero**.
  - **Scaling the data**: **Dividing each feature by its standard deviation** so that each feature has a **standard deviation of one**.

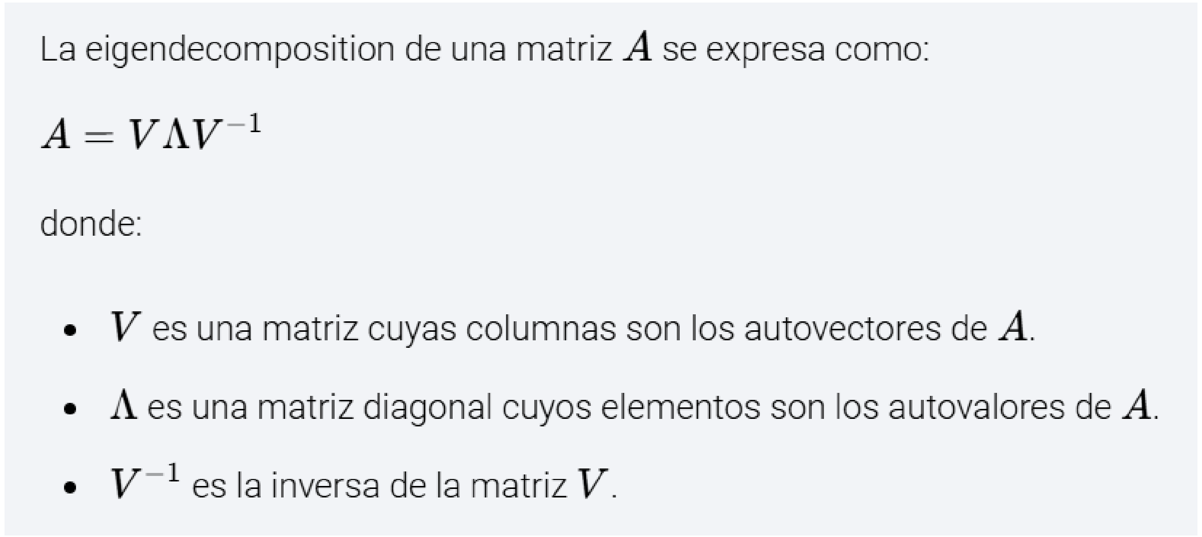
  This process is often referred to as **z-score normalization**.

  By standardizing the data, you ensure that each **feature contributes equally to the calculation of the principal components**, allowing PCA to identify the directions of maximum variance without being biased by the scale of the features.

Principal Component Analysis (PCA)

- Let’s summarize the approach in a few simple steps:
  1. **Standardize** the *d*-dimensional dataset.
  2. Construct the **covariance matrix**.
  3. **Decompose** the **covariance matrix into** its **eigenvectors** and **eigenvalues**.
  4. **Sort the eigenvalues by decreasing order** to rank the corresponding **eigenvectors**.
  5. **Select *k* eigenvectors**, which correspond to the k largest eigenvalues, where *k* is the dimensionality of the new feature subspace ($k<=d$).
  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.

Principal Component Analysis (PCA)



La eigendecomposition de una matriz A se expresa como:

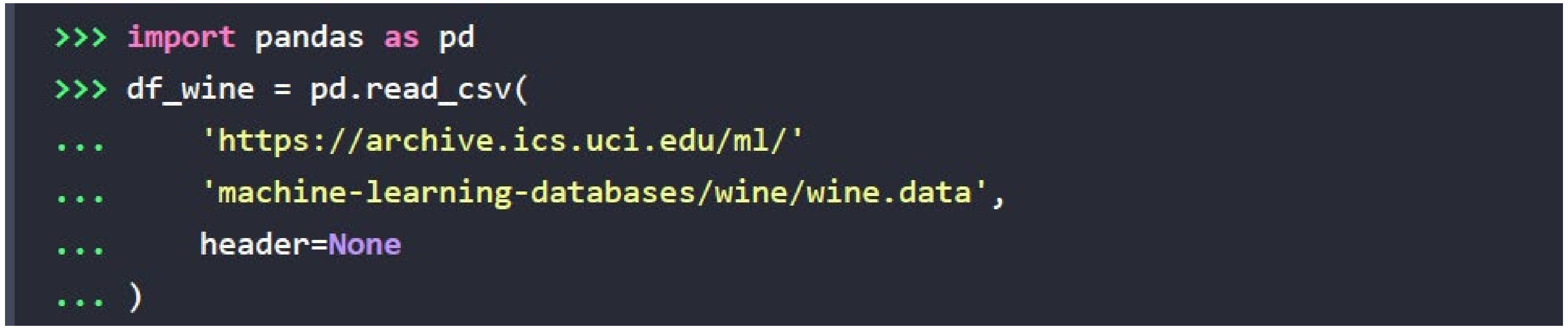
$$A = V\Lambda V^{-1}$$

donde:

- V es una matriz cuyas columnas son los autovectores de A.
- Λ es una matriz diagonal cuyos elementos son los autovalores de A.
- V^{-1} es la inversa de la matriz V.

- The **covariance matrix** is a special case of a **square matrix**: it's a **symmetric** matrix, which means that the matrix is equal to its transpose, $\mathbf{A = A^T}$.
- When we decompose (**eigendecomposition**) such a symmetric matrix, the **eigenvalues are real** (rather than complex) **numbers**, and the **eigenvectors are orthogonal** (perpendicular) **to each other**.
- Furthermore, eigenvalues and eigenvectors come in pairs. If we decompose a covariance matrix into its eigenvectors and eigenvalues**, the eigenvectors associated with the highest eigenvalue corresponds to the direction of maximum variance in the dataset**.

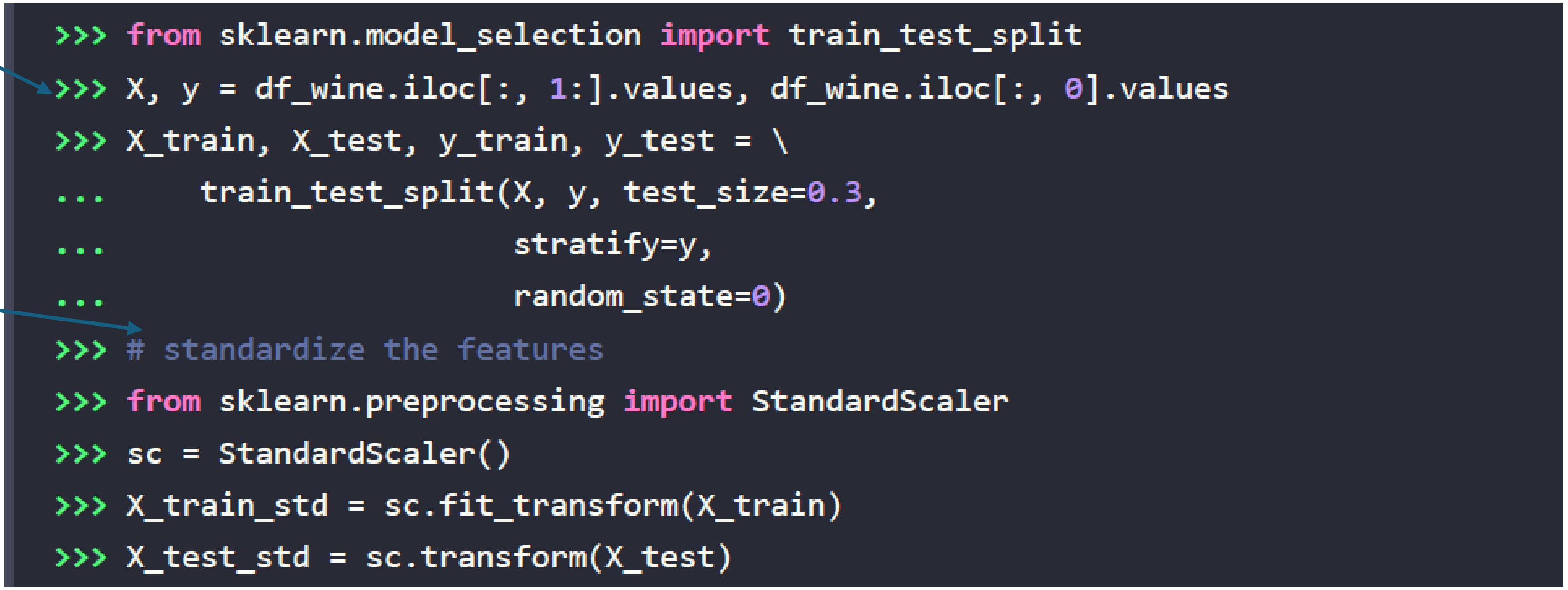
Principal Component Analysis (PCA)



```
>>> 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 in train set and test set

Standardize the *d*-dimensional 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)
>>> # standardize the features
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Principal Component Analysis (PCA)

- **After standardization**, the **covariance matrix** is constructed.
- It is a symmetric *d*×*d*-dimensional matrix, where d is the number of features in the dataset.
- It **stores the pairwise covariances between the** different **features**.
- For example, the covariance between two features can be calculated via the following equation:

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

means of features *j* and *k*

Note that, is our case, the means are zero because we standardized the dataset.

Principal Component Analysis (PCA)

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

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

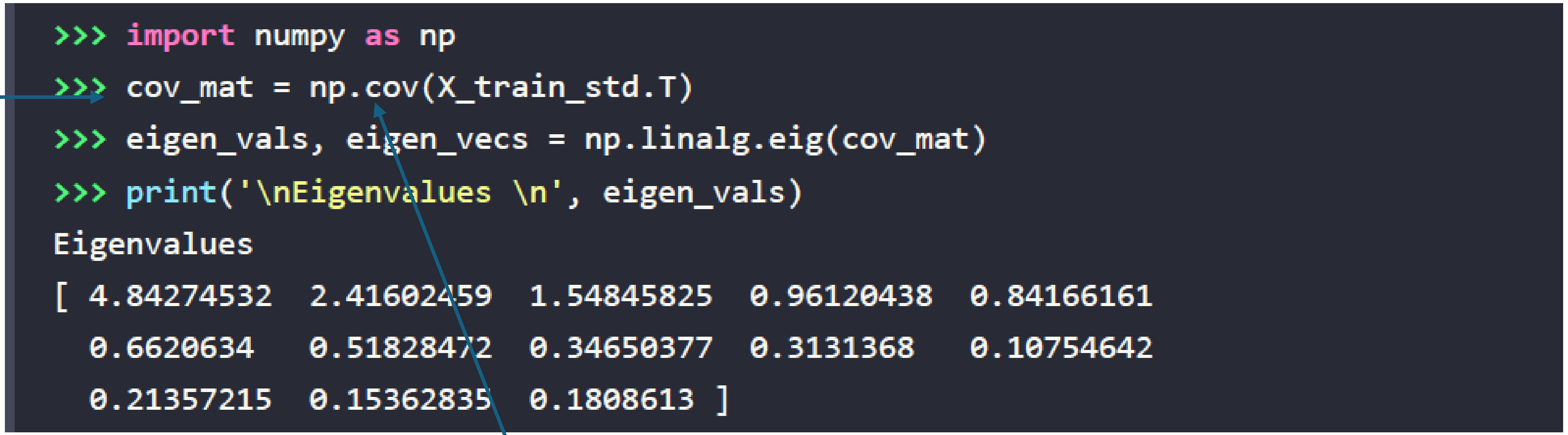
- The eigenvectors of the covariance matrix represent the principal components (the directions of maximum variance), whereas the corresponding eigenvalues will define their magnitude

$$A = V \Lambda V^{-1}$$

$$AV = \Lambda V \longrightarrow \Sigma V = \Lambda V$$

Principal Component Analysis (PCA)

Compute the covariance matrix



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

We use the `linalg.eig` function from NumPy to compute the eigenvectors and eigenvalues.

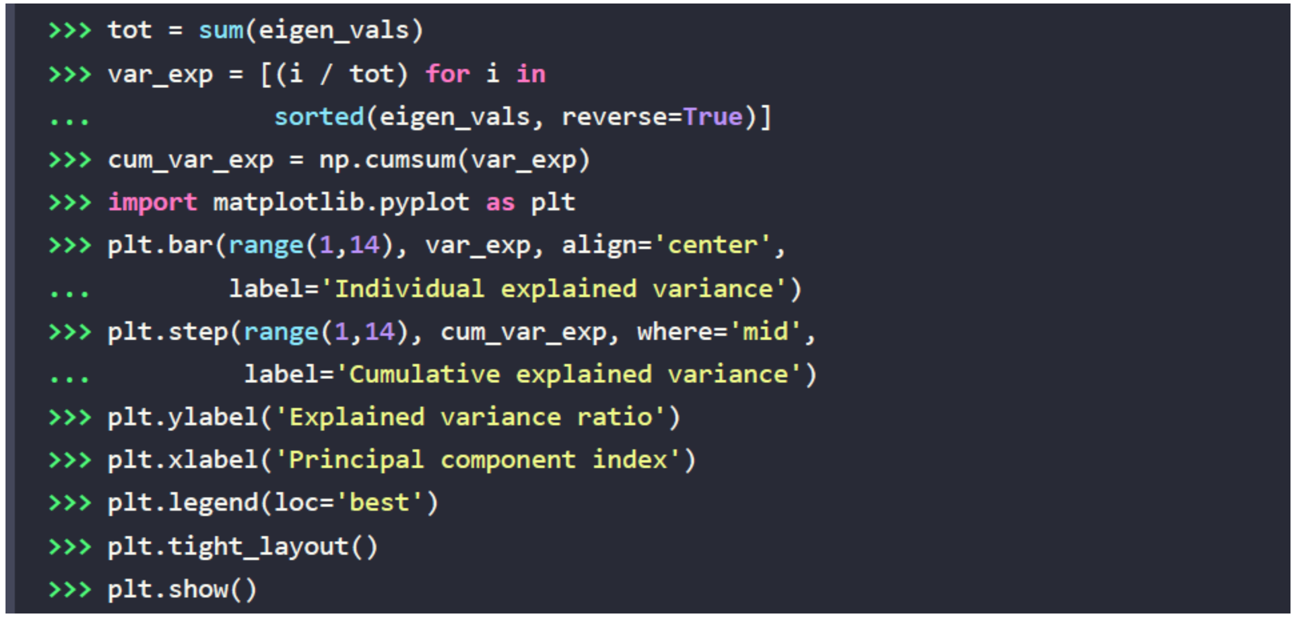
Principal Component Analysis (PCA)

- Since we want to **reduce the dimensionality** of our dataset, we only **select the** subset of the **eigenvectors (principal components)** that contains **most of the information (variance)**.
- The **eigenvalues define the magnitude of the eigenvectors**, so we have to **sort the eigenvalues by decreasing magnitude**.
  - We are interested in the **top *k* eigenvectors** based on the values of their corresponding eigenvalues.

Principal Component Analysis (PCA)

- Let's plot the **explained variance ratios** of the eigenvalues. The variance explained ratio of an eigenvalue is simply the **fraction of an eigenvalue and the total sum of the eigenvalues**:

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



```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...            sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)
>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, 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()
```

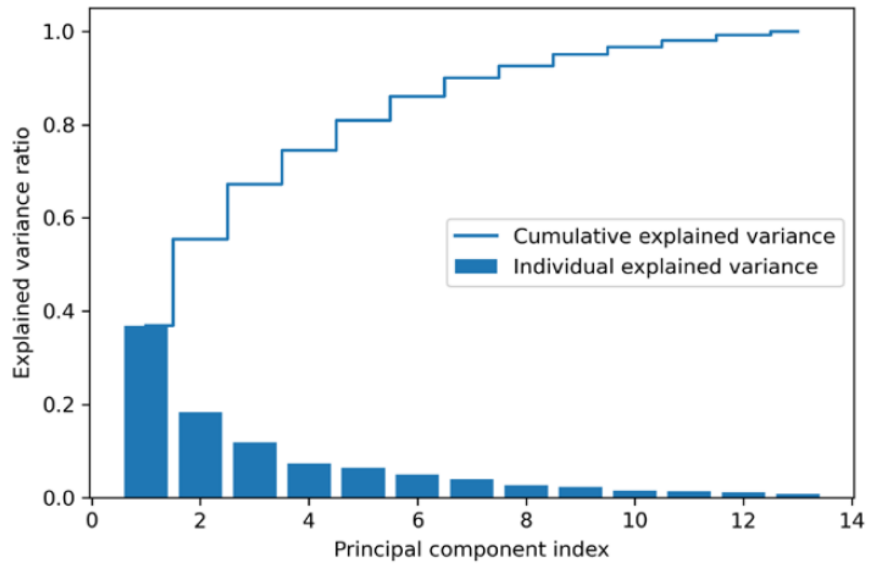


Figure 5.2: The proportion of the total variance captured by the principal components

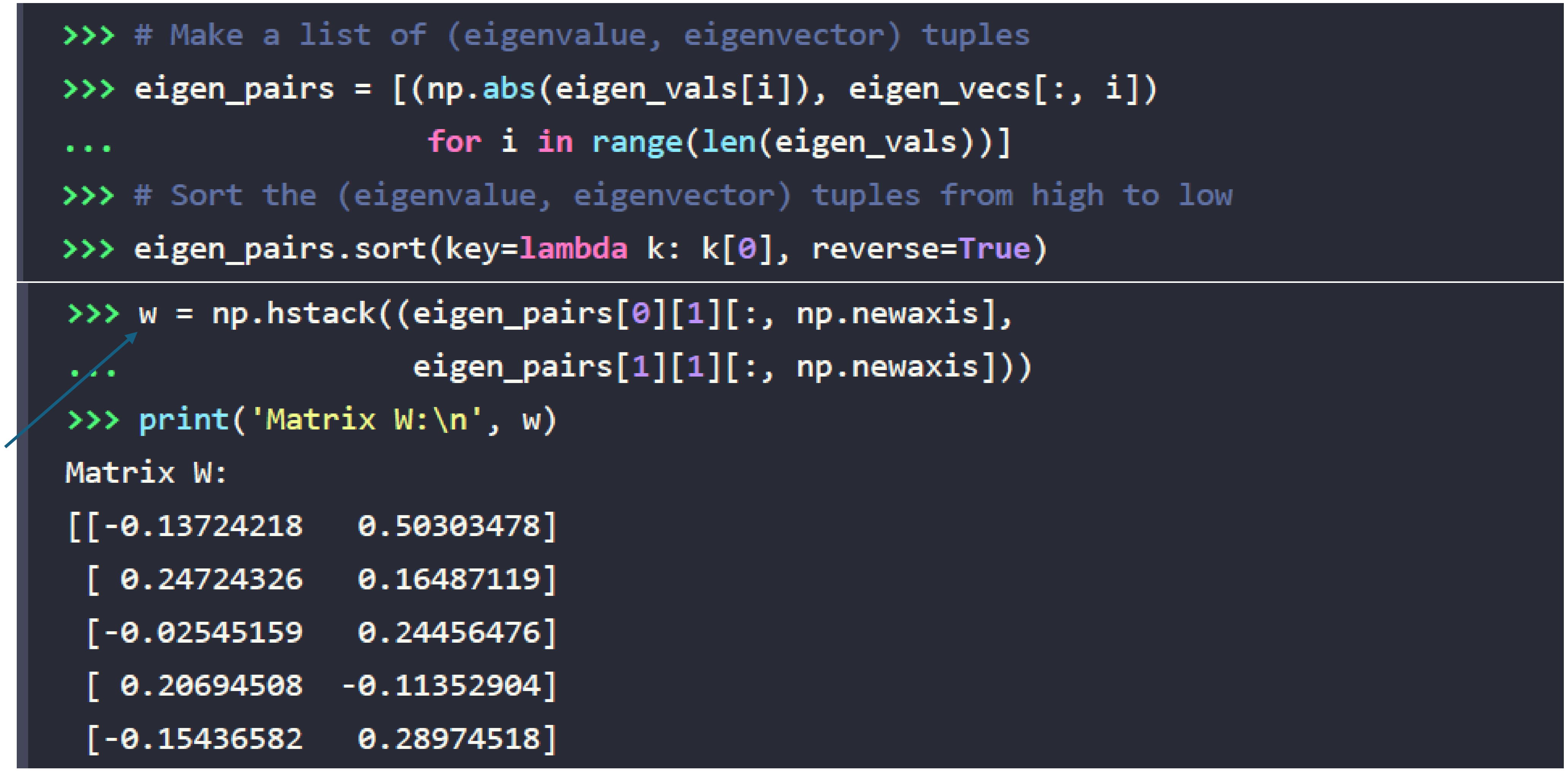
Principal Component Analysis (PCA)

- Using the NumPy `cumsum` function, we can then calculate the cumulative sum of explained variances.
- The resulting plot indicates that the first principal component alone accounts for approximately 40 % of the variance in the dataset.
- Also, we can see that the first two principal components combined explain almost 60 % of the variance in the dataset.

Principal Component Analysis (PCA)

- We have decomposed the covariance matrix into eigenpairs (eigenvectors and eigenvalues).
- **Now**:
  - Sort the eigenvalues by decreasing order.
  - Select *k* eigenvectors, which correspond to the k largest eigenvalues, where *k* is the dimensionality of the new feature subspace ($k<=d$).
  - Construct a projection matrix, *W*, from the k eigenvectors.
  - Transform the *d*-dimensional input dataset, *X*, using the projection matrix, *W*, to obtain the new *k*-dimensional feature subspace.

Principal Component Analysis (PCA)



```
>>> # 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)
```

```
>>> 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]
```

2 dimensions

Principal Component Analysis (PCA)

1 sample

$$x' = xW$$

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

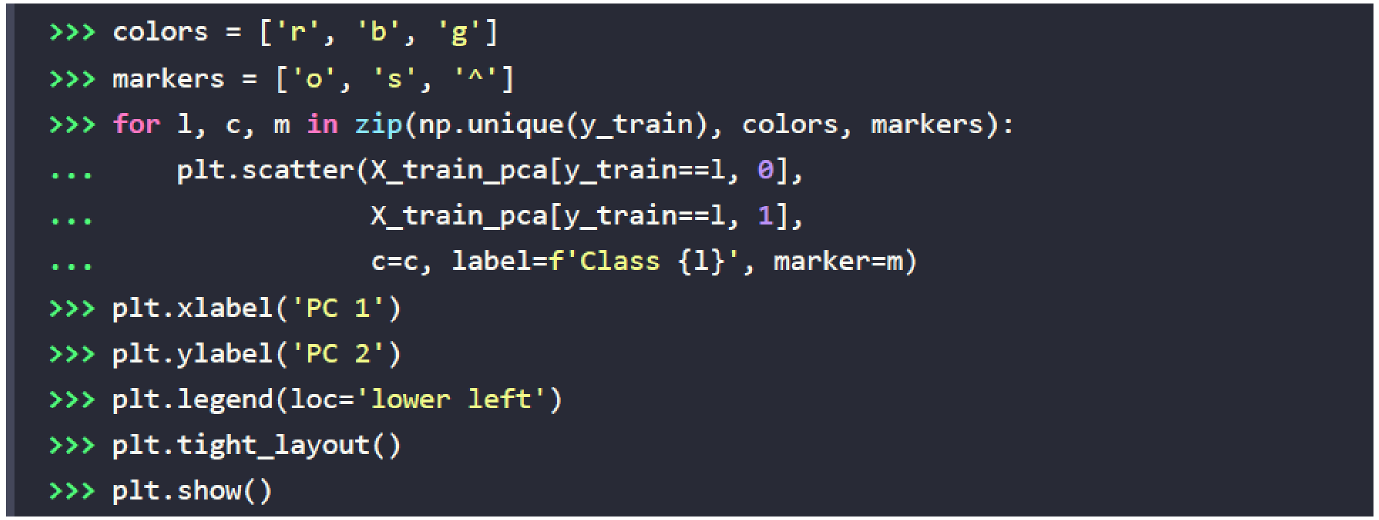
The whole dataset

$$X' = XW$$

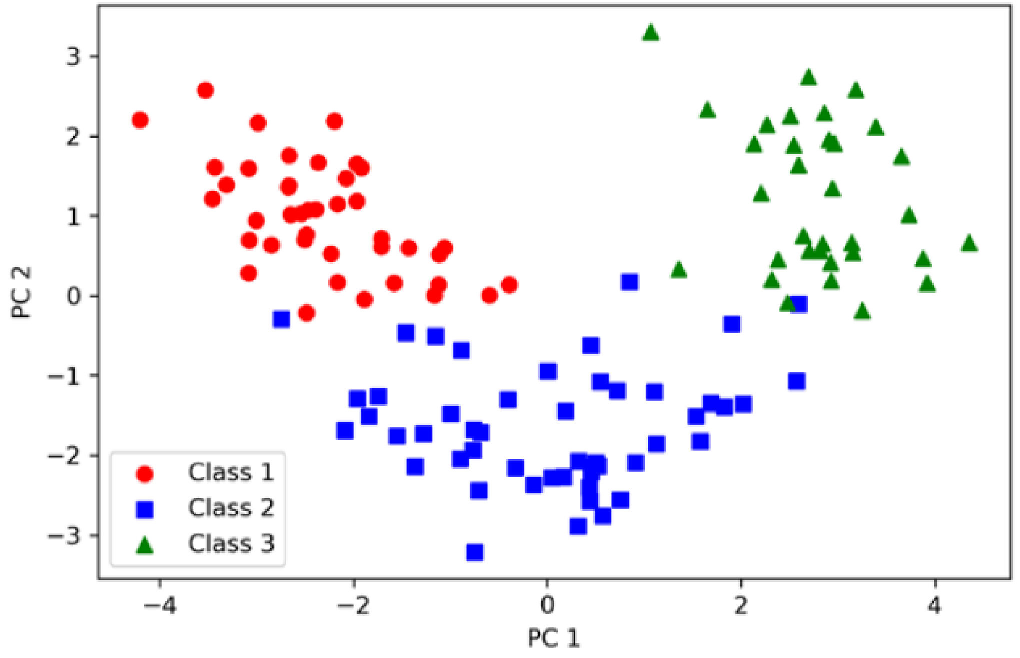
```
>>> X_train_pca = X_train_std.dot(w)
```

Principal Component Analysis (PCA)

- Let’s visualize the transformed training dataset in a 2-dimensional scatterplot.



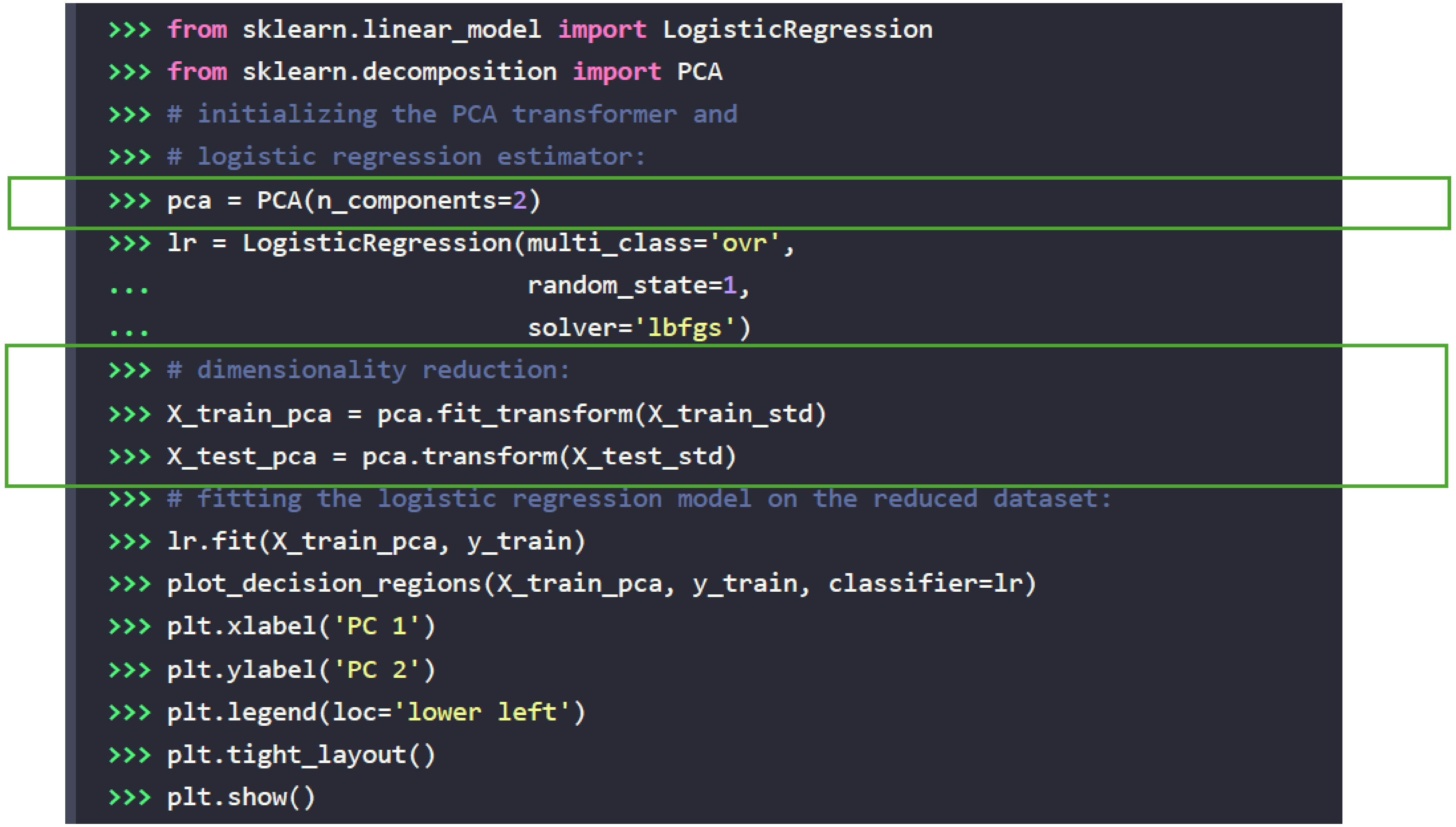
```
>>> colors = ['r', 'b', 'g']
>>> markers = ['o', 's', '^']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=f'Class {l}', marker=m)
>>> plt.xlabel('PC 1')
>>>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



Although we encoded the class label information for the purpose of illustration in the preceding scatterplot, we have to keep in mind **that PCA is an unsupervised technique** that doesn’t use any class label information.

Principal Component Analysis (PCA)

- Principal component analysis in scikit-learn



```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> # initializing the PCA transformer and
>>> # logistic regression estimator:
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression(multi_class='ovr',
...                         random_state=1,
...                         solver='lbfgs')
>>> # dimensionality reduction:
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> # fitting the logistic regression model on the reduced dataset:
>>> 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()
```

Principal Component Analysis (PCA)

- Principal component analysis in scikit-learn

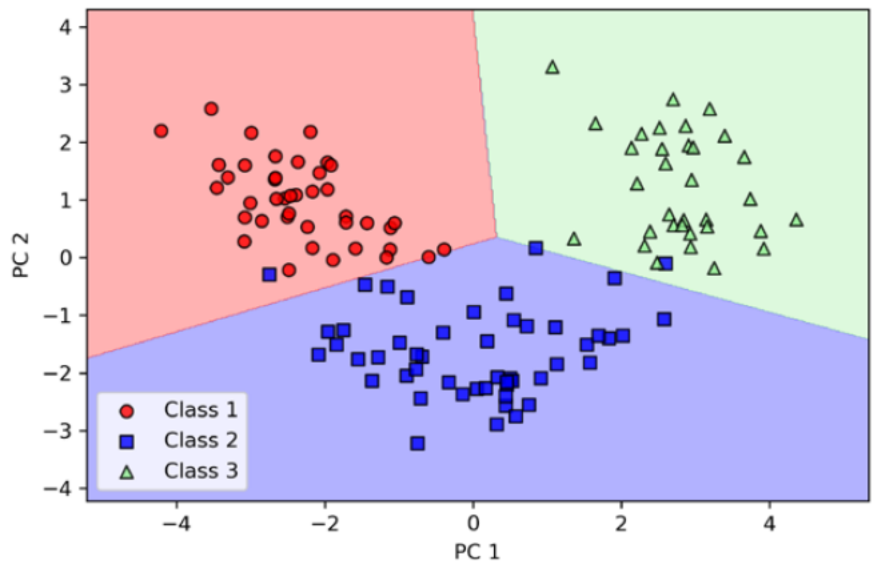


Figure 5.4: Training examples and logistic regression decision regions after using scikit-learn's PCA for dimensionality reduction

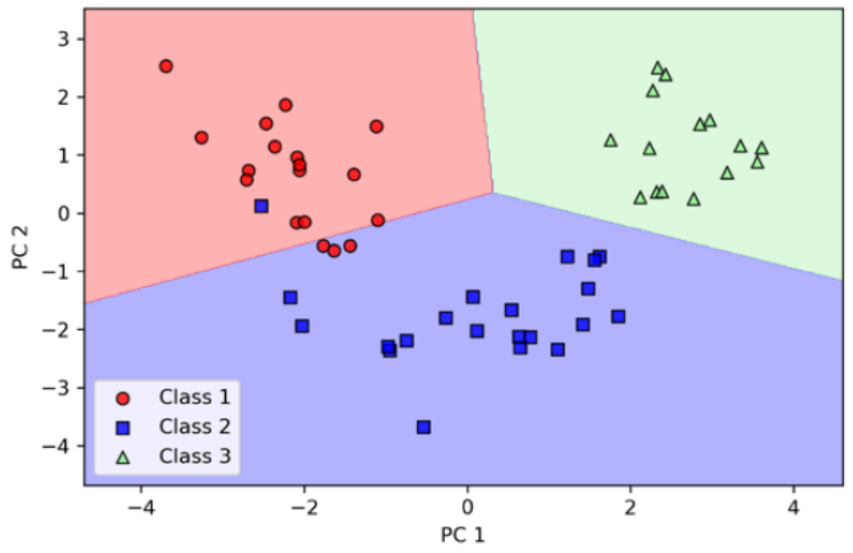
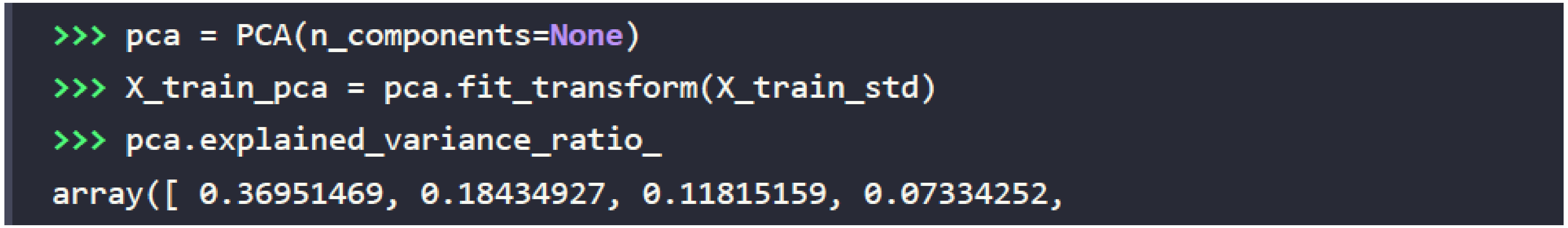


Figure 5.5: Test datapoints with logistic regression decision regions in the PCA-based feature space

```
>>> 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()
```

Principal Component Analysis (PCA)

- If we are interested in the **explained variance ratios** of the different principal components, we can simply initialize the PCA class with the `n_components` parameter set to None, so all principal components are kept and the explained variance ratio can then 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,
```

Principal Component Analysis (PCA)

- The new features represent linear combinations of the original features with the principal components.

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \qquad W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$$

$$X' = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \cdot \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} =$$

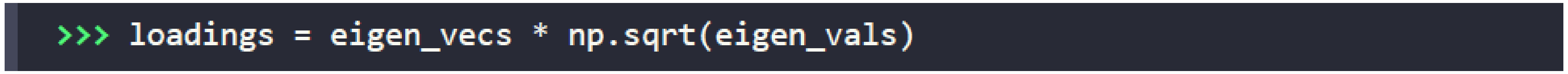
Linear combination

$$\begin{bmatrix} x_{11} \cdot w_{11} + x_{12} \cdot w_{21} & x_{11} \cdot w_{12} + x_{12} \cdot w_{22} \\ x_{21} \cdot w_{11} + x_{22} \cdot w_{21} & x_{21} \cdot w_{12} + x_{22} \cdot w_{22} \end{bmatrix}$$

Each w_{ij} is the **contributions of the original features to the new feature**.

Principal Component Analysis (PCA)

- How ever, to measure the **contributions of the original features to the new feature**, we use a scaled versions of *W*, where each eigenvector is multiplied by the square root of its eigenvalue.
- The result is often called **loadings**.
- Why to scale?
  - The **resulting values can then be interpreted as the correlation between the original features and the new features**.



```
>>> loadings = eigen_vecs * np.sqrt(eigen_vals)
```

Principal Component Analysis (PCA)

```
>>> fig, ax = plt.subplots()
>>> ax.bar(range(13), loadings[:, 0], align='center')
>>> ax.set_ylabel('Loadings for PC 1')
>>> ax.set_xticks(range(13))
>>> ax.set_xticklabels(df_wine.columns[1:], rotation=90)
>>> plt.ylim([-1, 1])
>>> plt.tight_layout()
>>> plt.show()
```

Plot the loadings for the first principal component, `loadings[:, 0]`, which is the first column in this matrix.

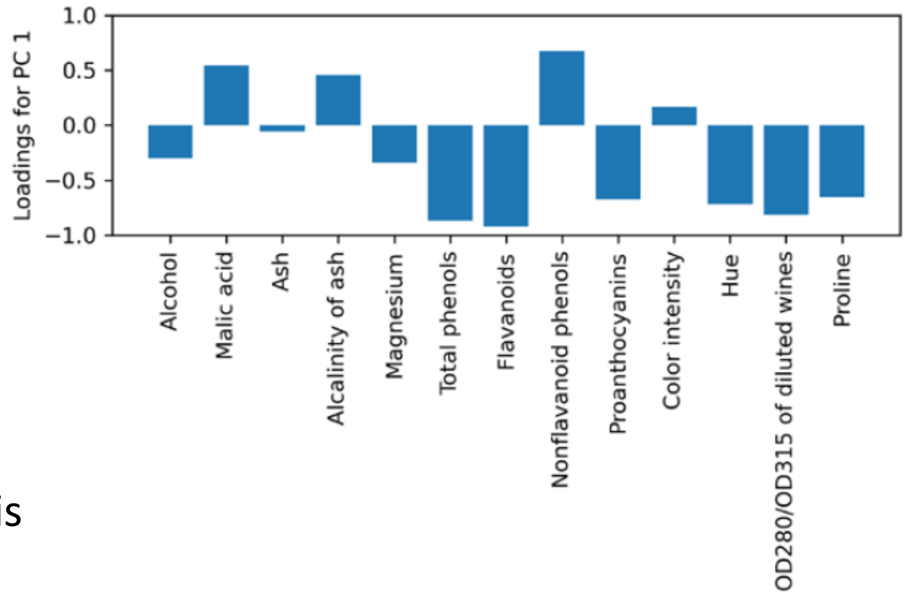


Figure 5.6: Feature correlations with the first principal component

For example, Alcohol has a negative correlation with the first new feature (approximately –0.3), whereas Malic acid has a positive correlation (approximately 0.54).

Principal Component Analysis (PCA)

We can obtain the loadings from a fitted scikit-learn PCA object in a similar manner, where `pca.components_` represents the eigenvectors and `pca.explained_variance_` represents the eigenvalues.

```
>>> sklearn_loadings = pca.components_.T * np.sqrt(pca.explained_variance_)
```

```
>>> fig, ax = plt.subplots()
>>> ax.bar(range(13), sklearn_loadings[:, 0], align='center')
>>> ax.set_ylabel('Loadings for PC 1')
>>> ax.set_xticks(range(13))
>>> ax.set_xticklabels(df_wine.columns[1:], rotation=90)
>>> plt.ylim([-1, 1])
>>> plt.tight_layout()
>>> plt.show()
```

Principal Component Analysis (PCA)

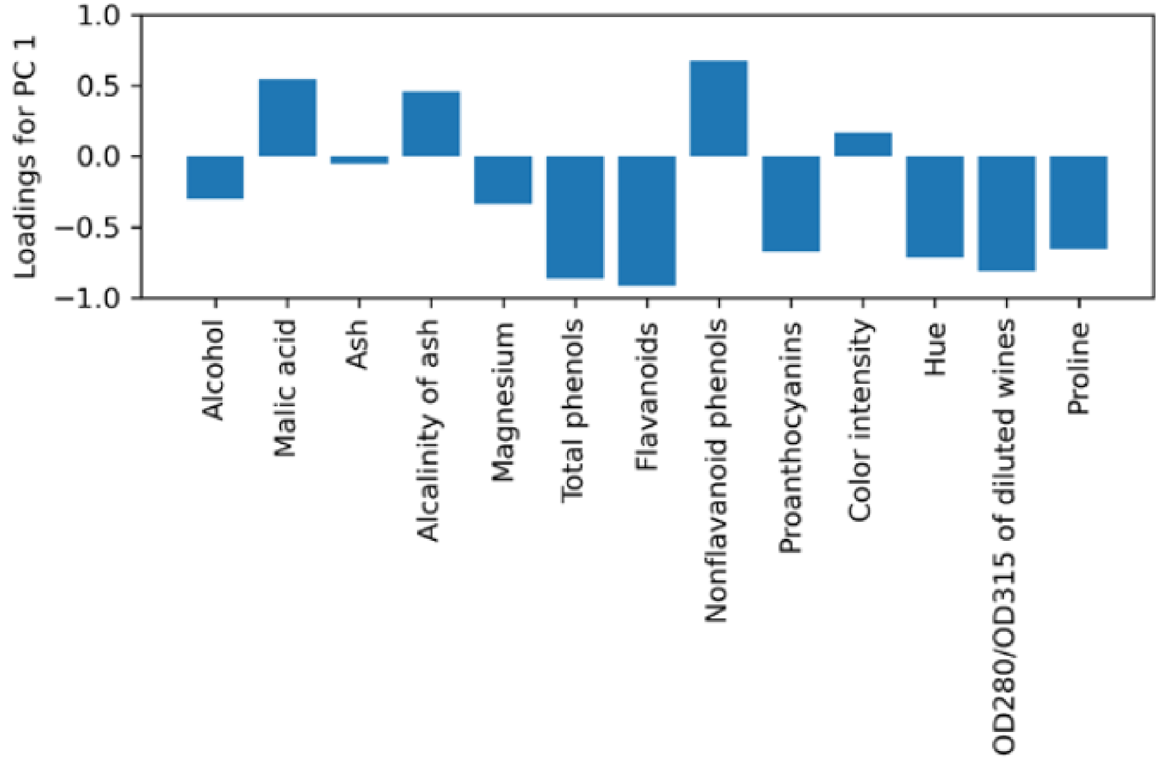


Figure 5.7: Feature correlations to the first principal component using scikit-learn

Principal Component Analysis (PCA)

- **Ejercicio**

Linear Discriminant Analysis (LDA)

- After exploring PCA as an unsupervised feature extraction technique, we introduce **Linear Discriminant Analysis (LDA).**
- LDA is a **linear transformation technique** that takes **class label information** into account (**supervised** algorithm).
  - It is a linear transformation technique that can be **used to reduce the number of dimensions**.
- LDA is sometimes also called **Fisher's LDA**.

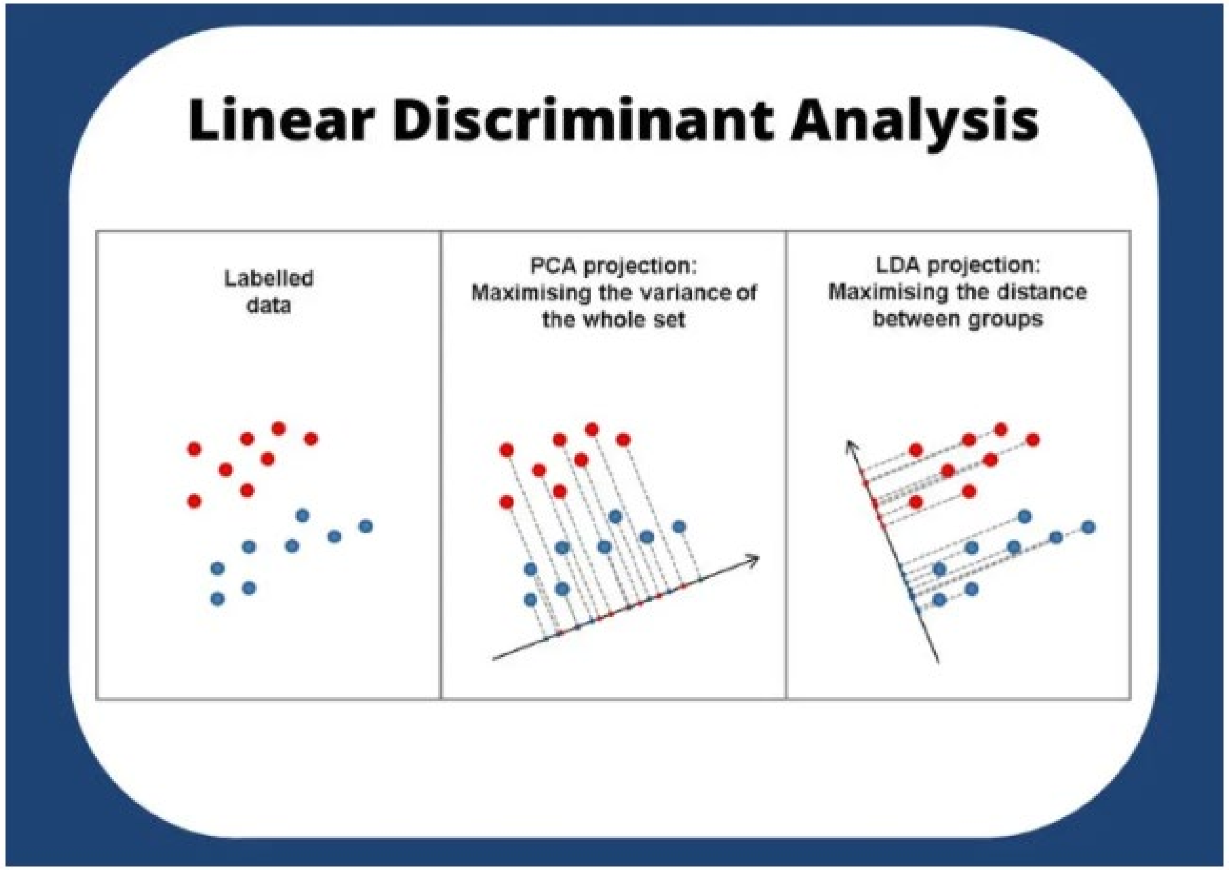
Linear Discriminant Analysis (LDA)

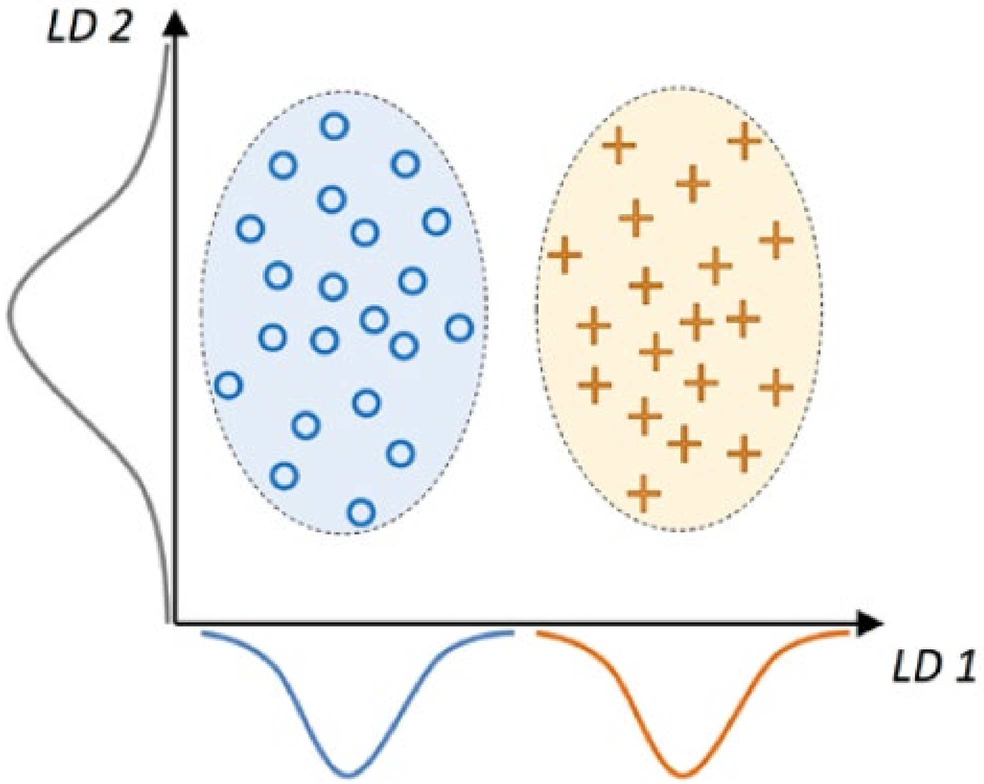
- LDA seeks to **find a linear projection** of the data that **maximizes the separability between** different **classes**.
- In other words, LDA attempts to **identify one or more directions in the feature space** (linear discriminats) that **maximize the distance between class means**.

Linear Discriminant Analysis (LDA)

- The **maximum number of linear discriminants** that it can be obtained with LDA is determined by the number of classes and the number of features in your data set.
- Specifically, the maximum number of linear discriminants is the smaller of:
    - C-1, where **C is the number of classes** in the data set.
    - d, where **d is the number of original features**.
- Therefore, the maximum number of linear discriminants is:
    - **min(C-1,d).**

Linear Discriminant Analysis (LDA)


Linear Discriminant Analysis
Labelled data
PCA projection: Maximising the variance of the whole set
LDA projection: Maximising the distance between groups



Linear Discriminant Analysis (LDA)

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

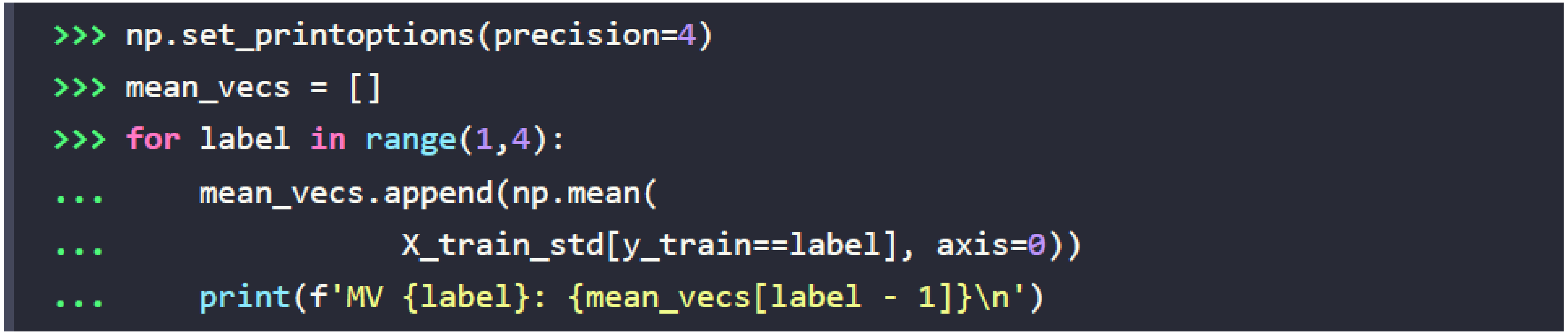
Linear Discriminant Analysis (LDA)

- We construct the within-class scatter matrix and between-class scatter matrix.
- Each **mean vector**, m_i, stores the mean feature value, μ_m , **with respect to the examples of class i**:

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

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

Linear Discriminant Analysis (LDA)



```
>>> 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(f'MV {label}: {mean_vecs[label - 1]}\n')
```

Linear Discriminant Analysis (LDA)

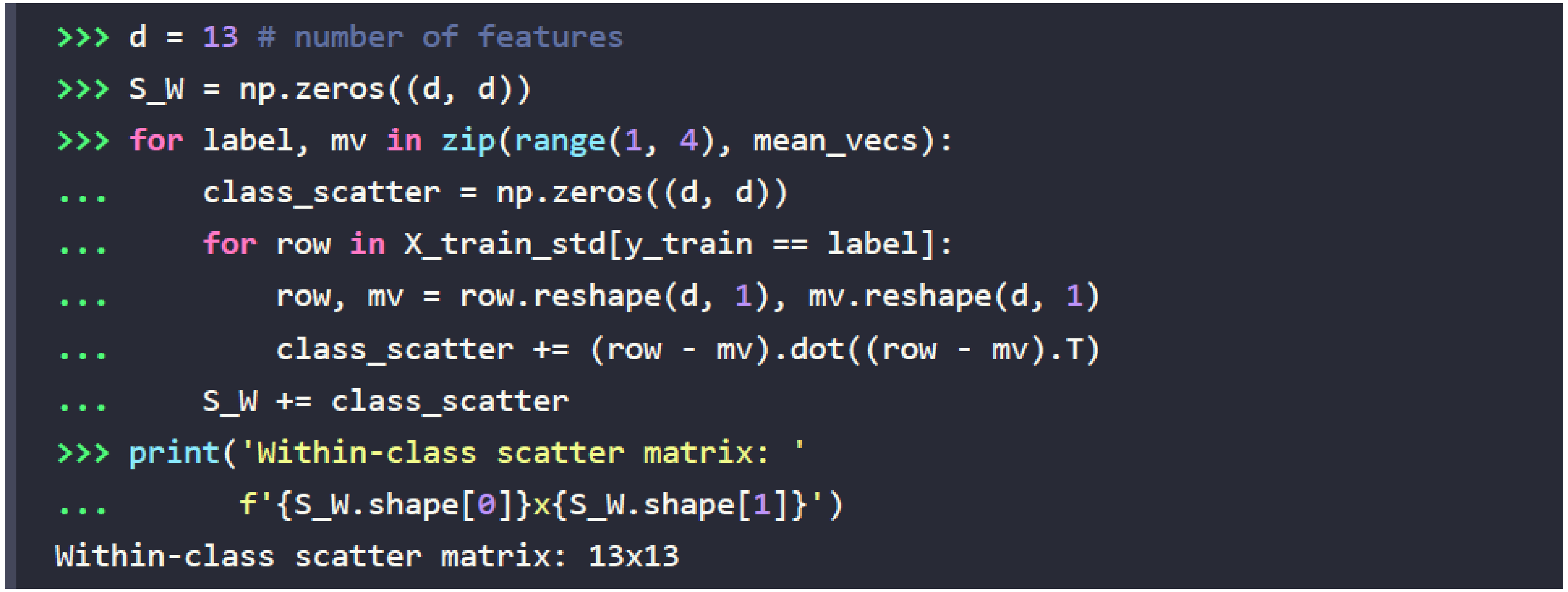
- We can now compute the **within-class scatter matrix**, **S_W**:

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

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

Individual scatter matrices, S_i, of each individual class *i*.

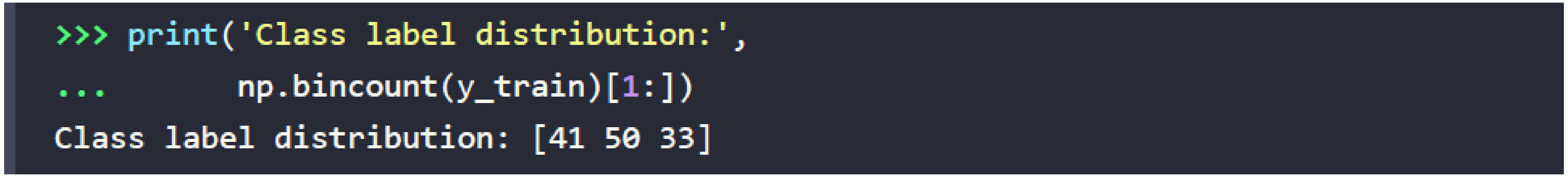
Linear Discriminant Analysis (LDA)



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

Linear Discriminant Analysis (LDA)

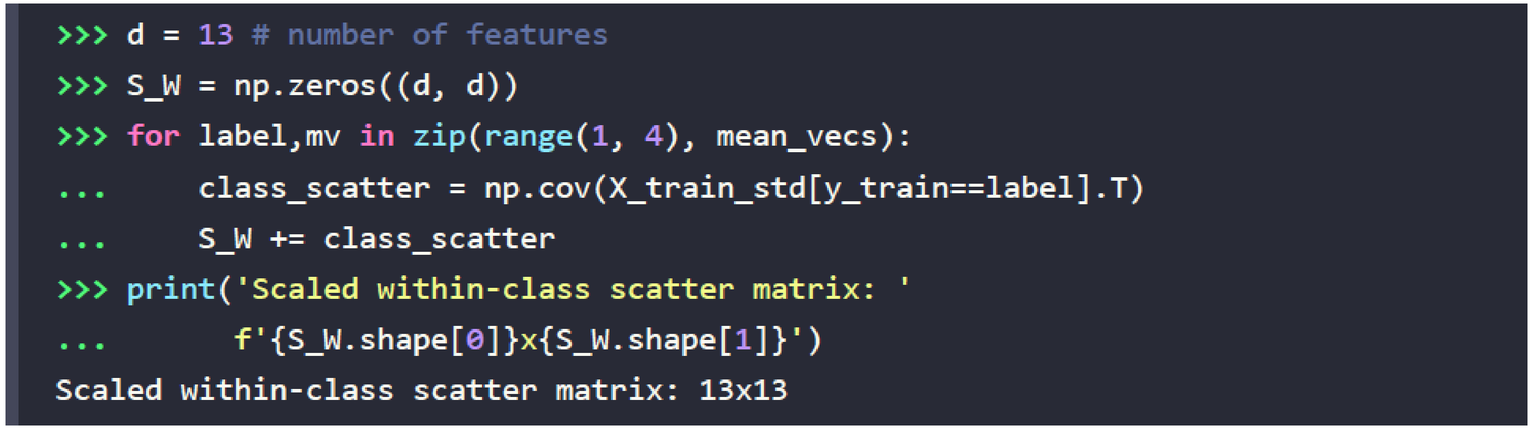
- The **assumption** that we are making when we are computing the scatter matrices is that the **class labels in the training dataset are uniformly distributed**.
- **However**, if we print the number of class labels, we see that **this assumption is violated**:



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

Linear Discriminant Analysis (LDA)

- We want to **scale the individual scatter matrices**, S_i, before we sum them up as the scatter matrix, S_W.
- When we divide the scatter matrices by the number of class-examples, n_i, we can see that **computing the scatter matrix is in fact the same as computing the covariance matrix,** Σ_i (the covariance matrix is a normalized version of the scatter matrix):



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

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

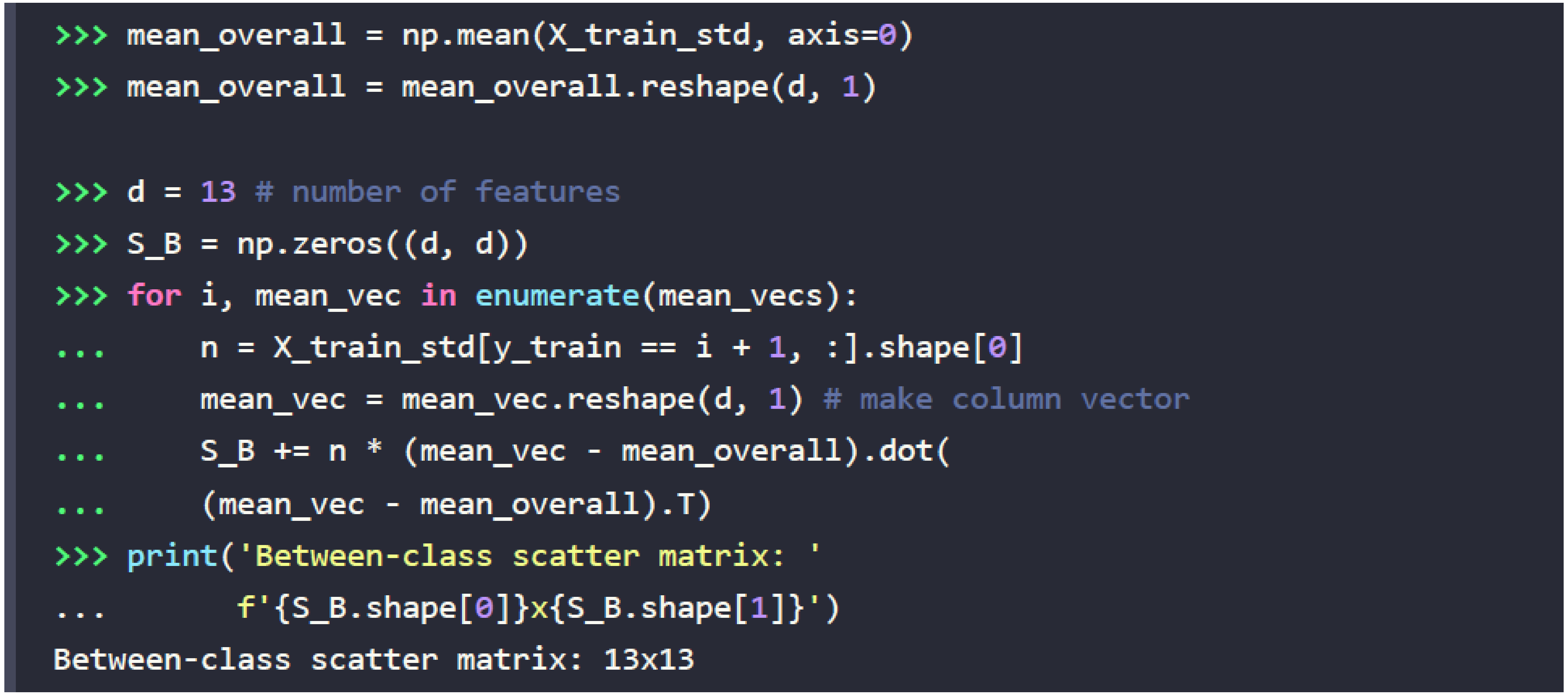
Linear Discriminant Analysis (LDA)

- After we compute the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute **the between-class scatter matrix** S_B:

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

m is the overall mean that is computed including examples from all *c* classes.

Linear Discriminant Analysis (LDA)

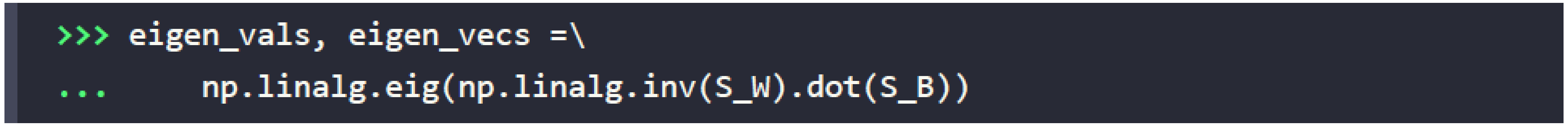


```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> mean_overall = mean_overall.reshape(d, 1)

>>> 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
...     S_B += n * (mean_vec - mean_overall).dot(
...     (mean_vec - mean_overall).T)
>>> print('Between-class scatter matrix: '
...       f'{S_B.shape[0]}x{S_B.shape[1]}')
Between-class scatter matrix: 13x13
```

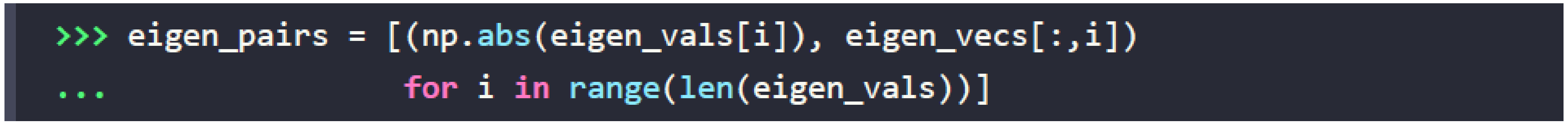
Linear Discriminant Analysis (LDA)

- We perform the **eigendecomposition on the matrix $S_W^{-1}S_B$** :



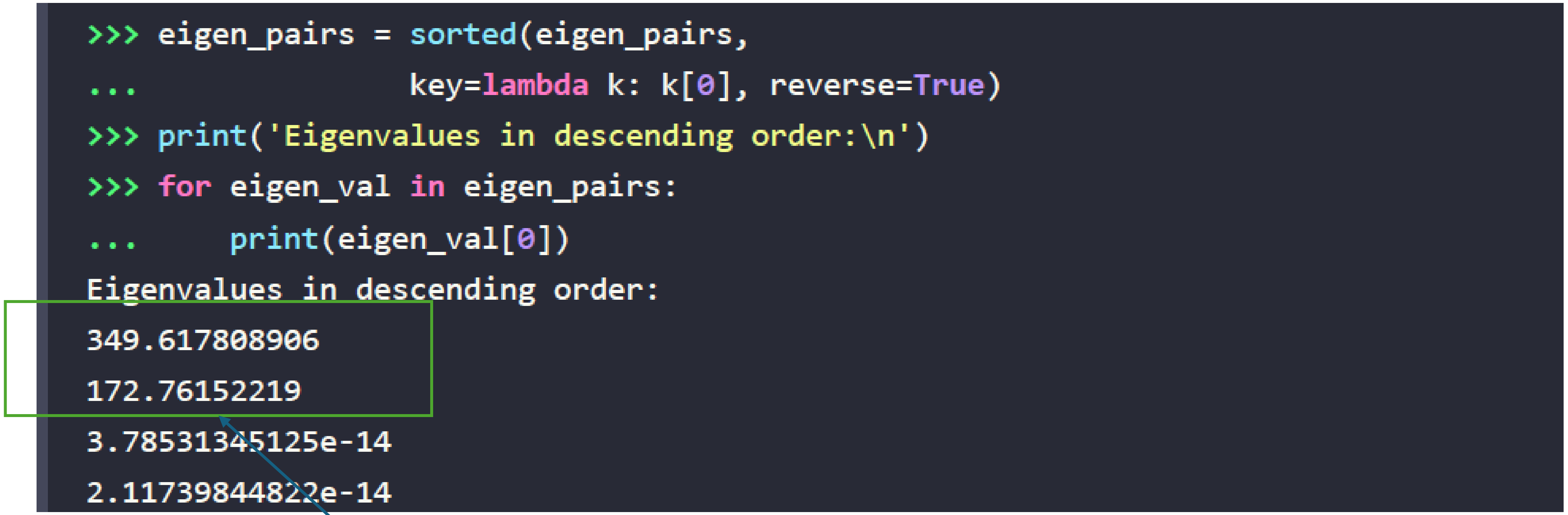
```
>>> eigen_vals, eigen_vecs =\
...     np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

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



```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                for i in range(len(eigen_vals))]
```

Linear Discriminant Analysis (LDA)

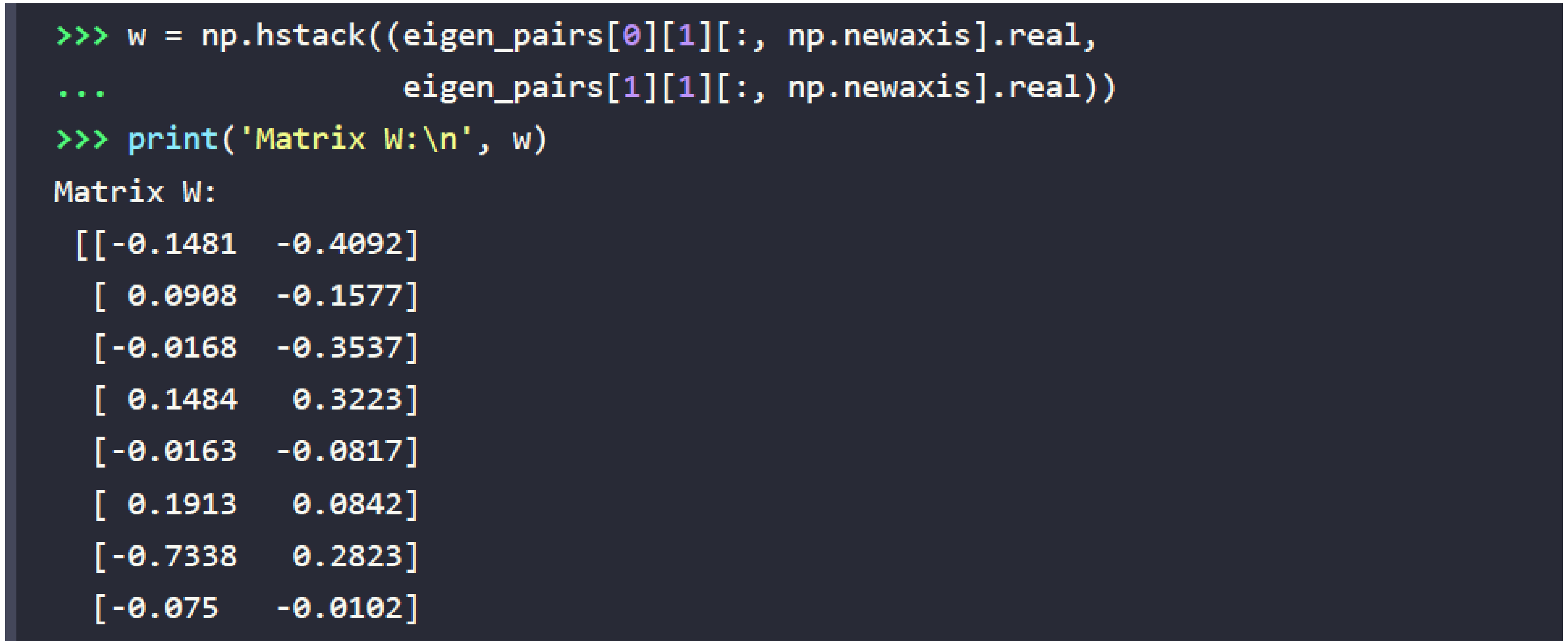


```
>>> eigen_pairs = sorted(eigen_pairs,
...                      key=lambda k: k[0], reverse=True)
>>> print('Eigenvalues in descending order:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])
Eigenvalues in descending order:
349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
```

The number of linear discriminants is min(3-1,13) = 2

Linear Discriminant Analysis (LDA)

- Let’s now stack the two most discriminative eigenvector columns to **create the transformation matrix,** *W*:

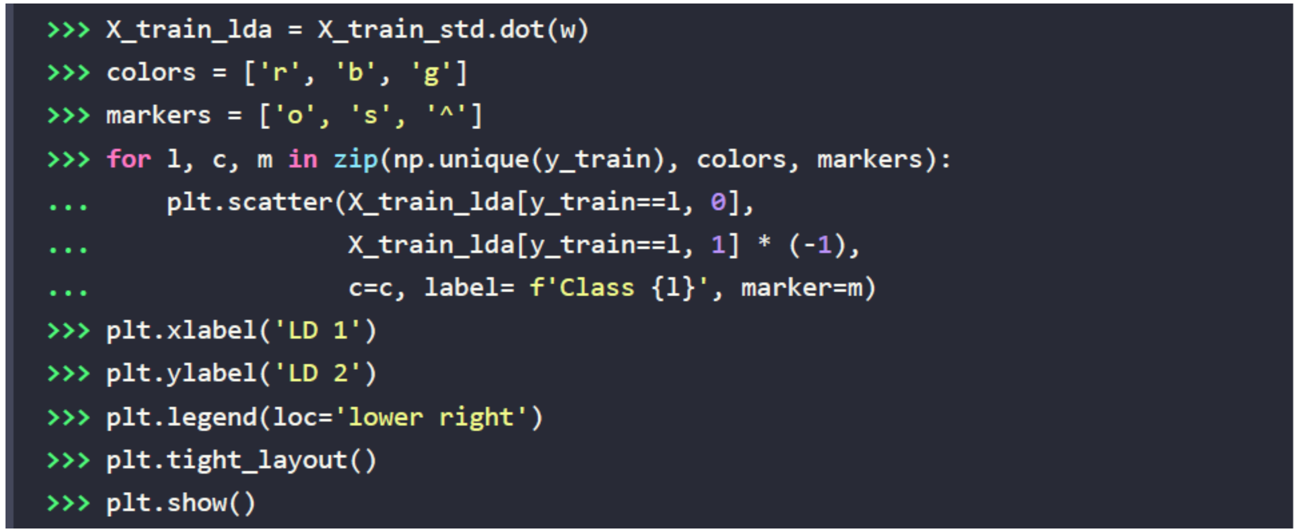


```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
 [[-0.1481  -0.4092]
  [ 0.0908  -0.1577]
  [-0.0168  -0.3537]
  [ 0.1484   0.3223]
  [-0.0163  -0.0817]
  [ 0.1913   0.0842]
  [-0.7338   0.2823]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

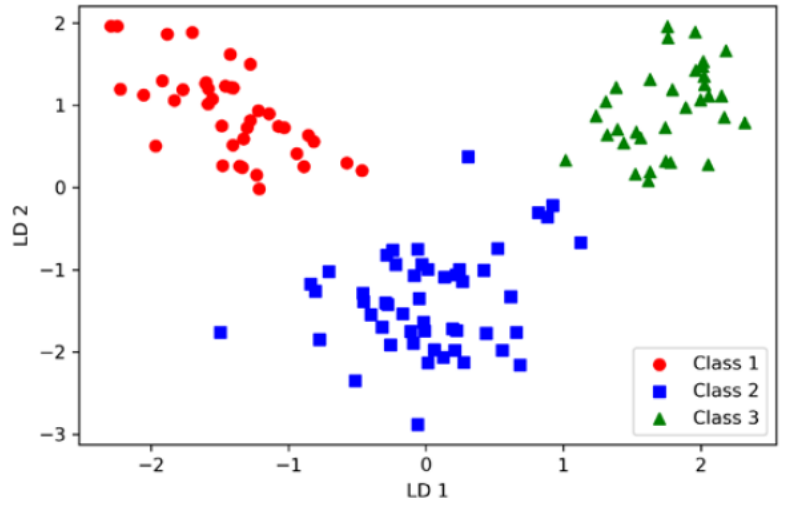
Linear Discriminant Analysis (LDA)

- 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 = ['o', 's', '^']
>>> 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= f'Class {l}', marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

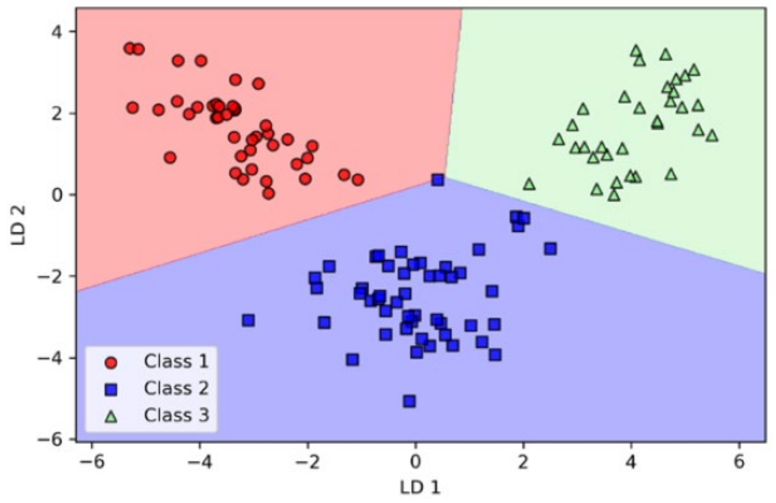


Linear Discriminant Analysis (LDA)

- Using scikit-learn:

```
>>> # the following import statement is one line
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

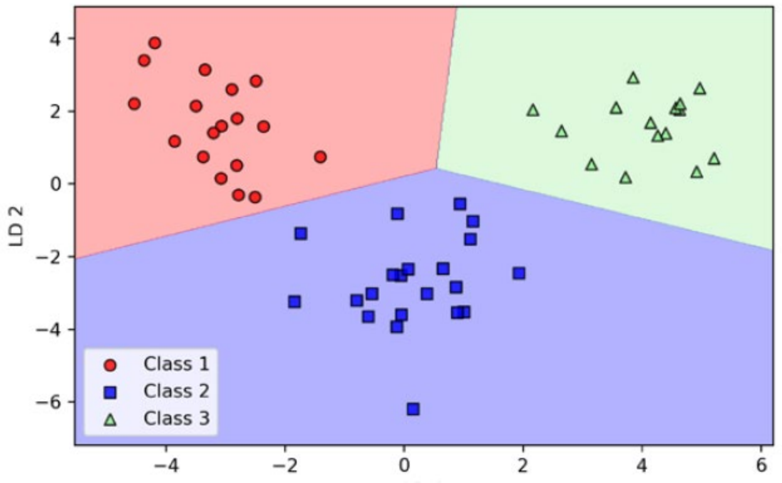
```
>>> lr = LogisticRegression(multi_class='ovr', random_state=1,
...                         solver='lbfgs')
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```



Linear Discriminant Analysis (LDA)

- Using scikit-learn:

```
>>> 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()
```



Linear Discriminant Analysis (LDA)

- **Ejercicio**

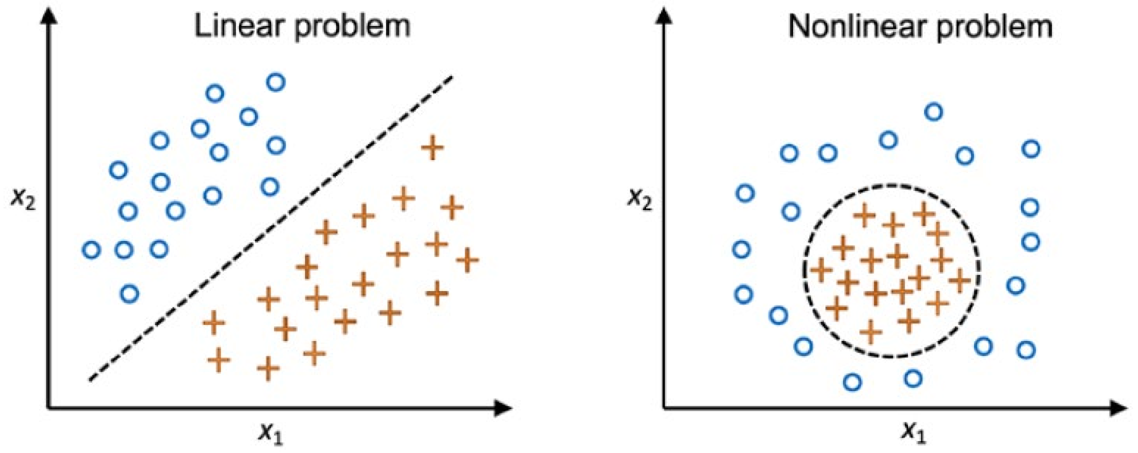
  - En este ejercicio, deberás realizar un análisis de discriminantes lineales (LDA) utilizando el conjunto de datos proporcionado en el archivo 'dataset1.csv'.
  - El análisis se realizará de dos formas diferentes: primero mediante una implementación manual (sin utilizar librerías específicas de LDA) y posteriormente usando *scikit-learn*.
  - Para ambos casos, deberás calcular las discriminantes lineales que se consideren y obtener: la matriz de pesos (*W*) y las proyecciones (*X'*).
  - Además, deberás crear visualizaciones que muestren las proyecciones (*X'*) etiquetadas según su categoría.
  - Finalmente, compararás los resultados obtenidos por ambos métodos.

t-distributed Stochastic Neighbor Embedding (t-SNE)

- **PCA and LDA** are **linear transformation techniques** for feature extraction.
- However, there are **nonlinear dimensionality reduction techniques**.
  - One is t-distributed stochastic neighbor embedding (**t-SNE**), very **used to visualize high-dimensional datasets** in two or three dimensions.

t-distributed Stochastic Neighbor Embedding (t-SNE)

- **Why** consider nonlinear dimensionality reduction?
  - Many machine learning algorithms make assumptions about the linear separability of the input data.
  - However, if **we are dealing with nonlinear problems**, which we may encounter rather frequently **in real-world applications**, linear transformation techniques for dimensionality reduction, such as PCA and LDA, may not be the best choice.



t-distributed Stochastic Neighbor Embedding (t-SNE)

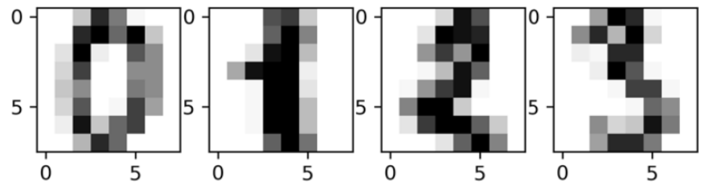
- **t-SNE** learns to **embed data points into a lower-dimensional space** such that the **pairwise distances in the original space are preserved**.
- t-SNE is **a technique intended for visualization purposes** as it **requires the whole dataset** for the projection.
- Since it projects the points directly (unlike PCA, it does **not involve a projection matrix**), **we cannot apply t-SNE to new data points**.

t-distributed Stochastic Neighbor Embedding (t-SNE)

- We show a quick demonstration of how t-SNE can be applied to the 64-dimensional Digits dataset

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
```

```
>>> fig, ax = plt.subplots(1, 4)
>>> for i in range(4):
>>>     ax[i].imshow(digits.images[i], cmap='Greys')
>>> plt.show()
```



Digits are 8×8 grayscale images

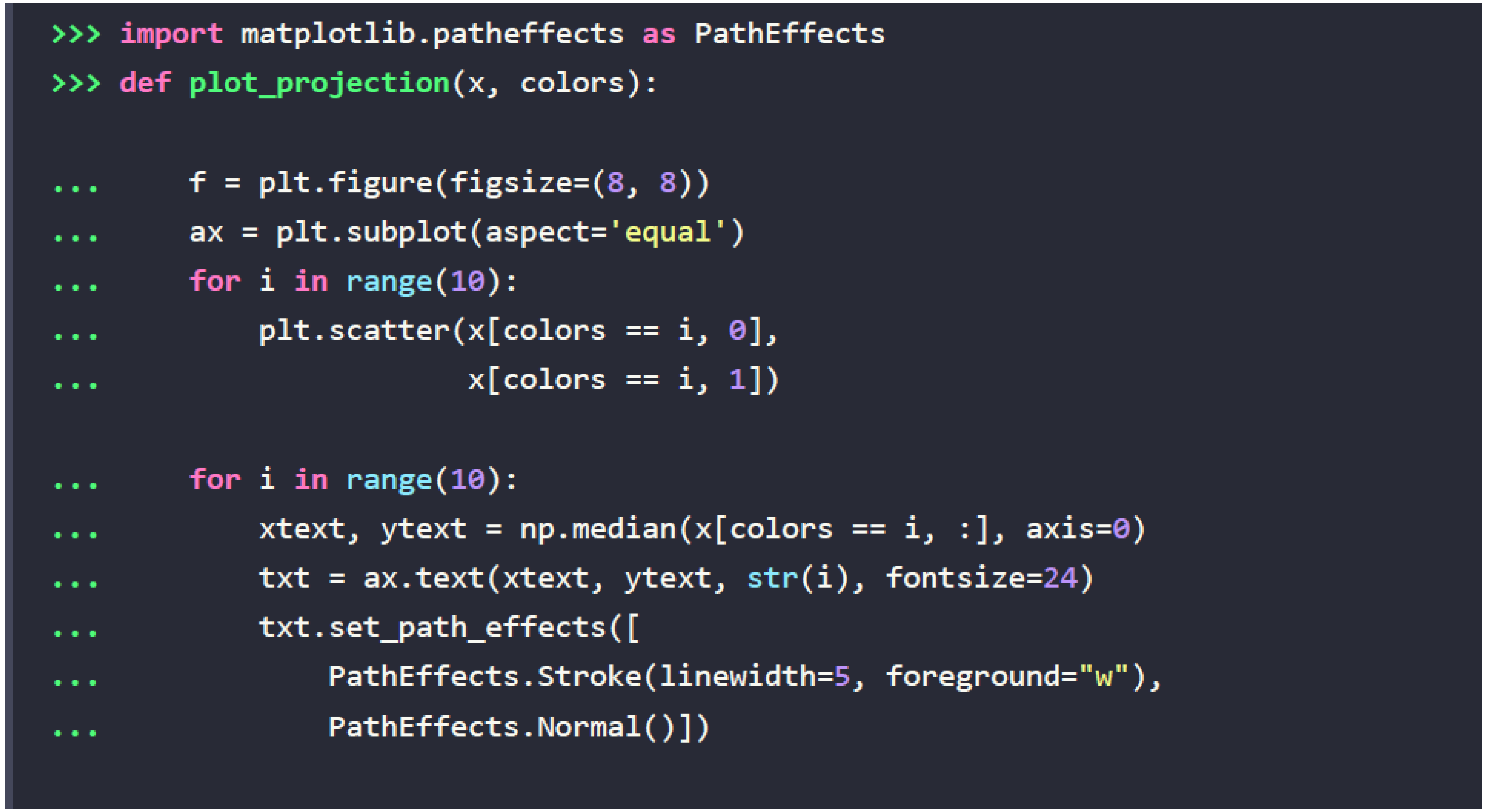
```
>>> digits.data.shape
(1797, 64)
```

t-distributed Stochastic Neighbor Embedding (t-SNE)

```
>>> y_digits = digits.target
>>> X_digits = digits.data
```

```
>>> from sklearn.manifold import TSNE
>>> tsne = TSNE(n_components=2, init='pca',
...             random_state=123)
>>> X_digits_tsne = tsne.fit_transform(X_digits)
```

t-distributed Stochastic Neighbor Embedding (t-SNE)



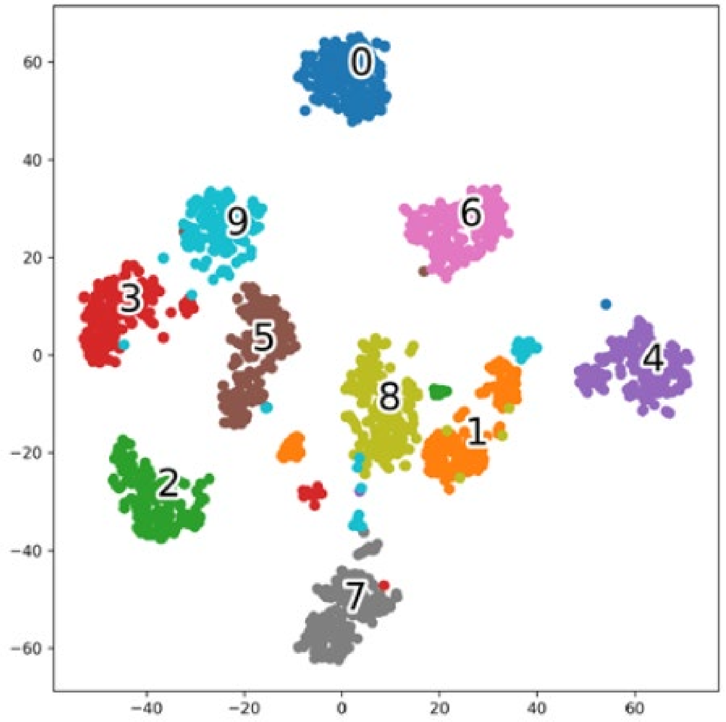
```
>>> import matplotlib.patheffects as PathEffects
>>> def plot_projection(x, colors):

...     f = plt.figure(figsize=(8, 8))
...     ax = plt.subplot(aspect='equal')
...     for i in range(10):
...         plt.scatter(x[colors == i, 0],
...                     x[colors == i, 1])

...     for i in range(10):
...         xtext, ytext = np.median(x[colors == i, :], axis=0)
...         txt = ax.text(xtext, ytext, str(i), fontsize=24)
...         txt.set_path_effects([
...             PathEffects.Stroke(linewidth=5, foreground="w"),
...             PathEffects.Normal()])
```

t-distributed Stochastic Neighbor Embedding (t-SNE)

```
>>> plot_projection(X_digits_tsne, y_digits)
>>> plt.show()
```


0
1
2
3
4
5
6
7
8
9
60
40
20
0
−20
−40
−60
−40
−20
0
20
40
60