5

Compressing Data via Dimensionality Reduction

In *Chapter 4, Building Good Training Datasets – Data Preprocessing*, you learned about the different approaches for reducing the dimensionality of a dataset using different feature selection techniques. An alternative approach to feature selection for dimensionality reduction is **feature extraction**. In this chapter, you will learn about two fundamental techniques that will help you to summarize the information content of a dataset by transforming it onto a new feature subspace of lower dimensionality than the original one. Data compression is an important topic in machine learning, and it helps us to store and analyze the increasing amounts of data that are produced and collected in the modern age of technology.

In this chapter, we will cover the following topics:

- Principal component analysis for unsupervised data compression
- Linear discriminant analysis as a supervised dimensionality reduction technique for maximizing class separability
- A brief overview of nonlinear dimensionality reduction techniques and t-distributed stochastic neighbor embedding for data visualization

Unsupervised dimensionality reduction via principal component analysis

Similar to feature selection, we can use different feature extraction techniques to reduce the number of features in a dataset. The difference between feature selection and feature extraction is that while we maintain the original features when we use feature selection algorithms, such as **sequential backward selection**, we use feature extraction to transform or project the data onto a new feature space.

In the context of dimensionality reduction, 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 not only used to improve storage space or the computational efficiency of the learning algorithm but can also improve the predictive performance by reducing the **curse of dimensionality**—especially if we are working with non-regularized models.

The main steps in principal component analysis

In this section, we will discuss **principal component analysis** (**PCA**), an unsupervised linear transformation technique that is widely used across different fields, most prominently for feature extraction and dimensionality reduction. Other popular applications of PCA include exploratory data analysis and the denoising of signals in stock market trading, and the analysis of genome data and gene expression levels in the field of bioinformatics.

PCA helps us to identify patterns in data based on the correlation between features. In a nutshell, 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, as illustrated in *Figure 5.1*:

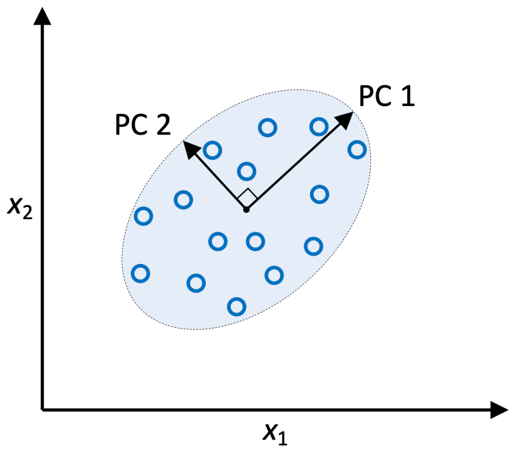

Figure 5.1: Using PCA to find the directions of maximum variance in a dataset

In *Figure 5.1*, x_1 and x_2 are the original feature axes, and **PC 1** and **PC 2** are the principal components.

If we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix, $\boldsymbol{W}$, that allows us to map a vector of the features of the training example, $\boldsymbol{x}$, onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space. For instance, the process is as follows. Suppose we have a feature vector, $\boldsymbol{x}$:

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

which is then transformed by a transformation matrix, $\boldsymbol{W} \in \mathbb{R}^{d \times k}$:

$$\boldsymbol{x}\boldsymbol{W} = \boldsymbol{z}$$

resulting in the output vector:

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

As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically $k << d$), 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). Note that the 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.

Before looking at the PCA algorithm for dimensionality reduction in more detail, 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 ($k \leq d$).
6. Construct a projection matrix, $\boldsymbol{W}$, from the "top" k eigenvectors.
7. Transform the d-dimensional input dataset, $\boldsymbol{X}$, using the projection matrix, $\boldsymbol{W}$, to obtain the new k-dimensional feature subspace.

In the following sections, we will perform a PCA step by step using Python as a learning exercise. Then, we will see how to perform a PCA more conveniently using scikit-learn.

Eigendecomposition: Decomposing a Matrix into Eigenvectors and Eigenvalues

Eigendecomposition, the factorization of a square matrix into so-called **eigenvalues** and **eigenvectors**, is at the core of the PCA procedure described in this section.

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, $A = A^T$.

When we decompose 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. Here, this "direction" is a linear transformation of the dataset's feature columns.

While a more detailed discussion of eigenvalues and eigenvectors is beyond the scope of this book, a relatively thorough treatment with pointers to additional resources can be found on Wikipedia at `https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors`.

Extracting the principal components step by step

In this subsection, we will tackle the first four steps of a PCA:

1. Standardizing the data
2. Constructing the covariance matrix
3. Obtaining the eigenvalues and eigenvectors of the covariance matrix
4. Sorting the eigenvalues by decreasing order to rank the eigenvectors

First, we will start by loading the Wine dataset that we worked with in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv(
...     'https://archive.ics.uci.edu/ml/'
...     'machine-learning-databases/wine/wine.data',
...     header=None
... )
```

Obtaining the Wine dataset

You can find a copy of the Wine dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data` is temporarily unavailable. For instance, to load the Wine dataset from a local directory, you can replace the following lines:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None
)
```

with these ones:

```
df = pd.read_csv(
    'your/local/path/to/wine.data',
    header=None
)
```

Next, we will process the Wine data into separate training and test datasets—using 70 percent and 30 percent of the data, respectively—and standardize it to unit variance:

```
>>> 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)
```

After completing the mandatory preprocessing by executing the preceding code, let's advance to the second step: constructing the covariance matrix. The symmetric d×d-dimensional covariance matrix, where d is the number of dimensions in the dataset, stores the pairwise covariances between the different features. For example, the covariance between two features, x_j and x_k, on the population level can be calculated via the following equation:

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

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. For example, the covariance matrix of three features can then be written as follows (note that Σ is the Greek uppercase letter sigma, which is not to be confused with the summation symbol):

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

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 13×13-dimensional covariance matrix.

Now, for our third step, let's obtain the eigenpairs of the covariance matrix. If you have taken a linear algebra class, you may have learned that an eigenvector, $\boldsymbol{v}$, satisfies the following condition:

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

Here, λ is a scalar: the eigenvalue. Since the manual computation of eigenvectors and eigenvalues is a somewhat tedious and elaborate task, we will use the `linalg.eig` function from NumPy to obtain the eigenpairs of the Wine 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 ]
```

Using the `numpy.cov` function, we computed the covariance matrix of the standardized training dataset. Using the `linalg.eig` function, we performed the eigendecomposition, which yielded a vector (`eigen_vals`) consisting of 13 eigenvalues and the corresponding eigenvectors stored as columns in a 13×13-dimensional matrix (`eigen_vecs`).

Eigendecomposition in NumPy

The `numpy.linalg.eig` function was designed to operate on both symmetric and non-symmetric square matrices. However, you may find that it returns complex eigenvalues in certain cases.

A related function, `numpy.linalg.eigh`, has been implemented to decompose Hermetian matrices, which is a numerically more stable approach to working with symmetric matrices such as the covariance matrix; `numpy.linalg.eigh` always returns real eigenvalues.

Total and explained variance

Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, 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. But before we collect those k most informative eigenvectors, let's plot the **variance explained ratios** of the eigenvalues. The variance explained ratio of an eigenvalue, λ_j, is simply the fraction of an eigenvalue, λ_j, and the total sum of the eigenvalues:

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

Using the NumPy `cumsum` function, we can then calculate the cumulative sum of explained variances, which we will then plot via Matplotlib's `step` function:

```
>>> 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()
```

The resulting plot indicates that the first principal component alone accounts for approximately 40 percent of the variance.

Also, we can see that the first two principal components combined explain almost 60 percent of the variance in the dataset:

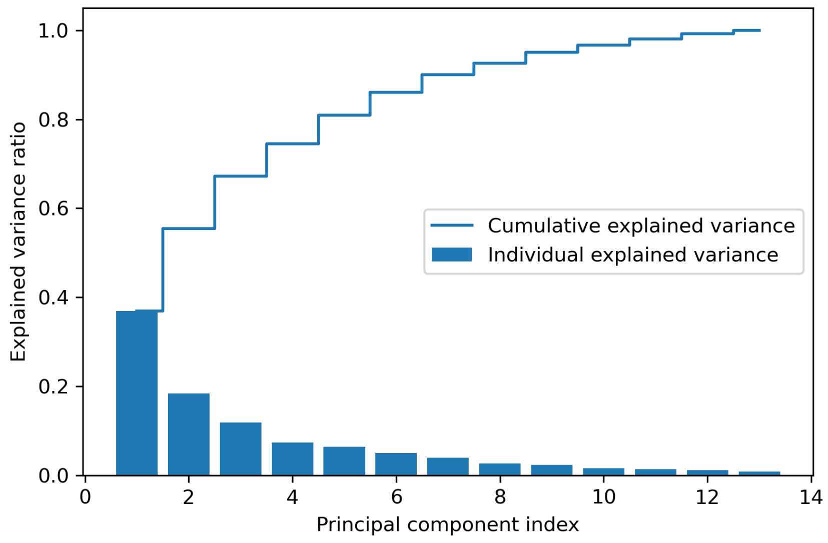

Figure 5.2: The proportion of the total variance captured by the principal components

Although the explained variance plot reminds us of the feature importance values that we computed in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, via random forests, we should remind ourselves that PCA is an unsupervised method, which means that information about the class labels is ignored. Whereas a random forest uses the class membership information to compute the node impurities, variance measures the spread of values along a feature axis.

Feature transformation

Now that we have successfully decomposed the covariance matrix into eigenpairs, let's proceed with the last three steps to transform the Wine dataset onto the new principal component axes. The remaining steps we are going to tackle in this section are the following:

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, $\boldsymbol{W}$, from the "top" k eigenvectors.
3. Transform the d-dimensional input dataset, $\boldsymbol{X}$, using the projection matrix, $\boldsymbol{W}$, to obtain the new k-dimensional feature subspace.

Or, in less technical terms, we will sort the eigenpairs by descending order of the eigenvalues, construct a projection matrix from the selected eigenvectors, and use the projection matrix to transform the data onto the lower-dimensional subspace.

We start by sorting the eigenpairs by decreasing order of the 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)
```

Next, we collect the two eigenvectors that correspond to the two largest eigenvalues, to capture about 60 percent of the variance in this dataset. Note that two eigenvectors have been chosen for the purpose of illustration, since we are going to plot the data via a two-dimensional scatterplot later in this subsection. In practice, the number of principal components has to be determined by a tradeoff between computational efficiency and the performance of the classifier:

```
>>> 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]]
```

By executing the preceding code, we have created a 13×2-dimensional projection matrix, ***W***, from the top two eigenvectors.

Mirrored projections

Depending on which versions of NumPy and LAPACK you are using, you may obtain the matrix, ***W***, with its signs flipped. Please note that this is not an issue; if ***v*** is an eigenvector of a matrix, Σ, we have:

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

Here, ***v*** is the eigenvector, and –***v*** is also an eigenvector, which we can show as follows. Using basic algebra, we can multiply both sides of the equation by a scalar, α:

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

Since matrix multiplication is associative for scalar multiplication, we can then rearrange this to the following:

$$\Sigma(\alpha \boldsymbol{v}) = \lambda(\alpha \boldsymbol{v})$$

Now, we can see that $\alpha \boldsymbol{v}$ is an eigenvector with the same eigenvalue, λ, for both $\alpha = 1$ and $\alpha = -1$. Hence, both ***v*** and –***v*** are eigenvectors.

Using the projection matrix, we can now transform an example, ***x*** (represented as a 13-dimensional row vector), onto the PCA subspace (the principal components one and two) obtaining ***x***′, now a two-dimensional example vector consisting of two new features:

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

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

Similarly, we can transform the entire 124×13-dimensional training dataset onto the two principal components by calculating the matrix dot product:

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

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

Lastly, let's visualize the transformed Wine training dataset, now stored as an 124×2-dimensional matrix, in a two-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()
```

As we can see in *Figure 5.3*, the data is more spread along the first principal component (*x* axis) than the second principal component (*y* axis), which is consistent with the explained variance ratio plot that we created in the previous subsection. However, we can tell that a linear classifier will likely be able to separate the classes well:

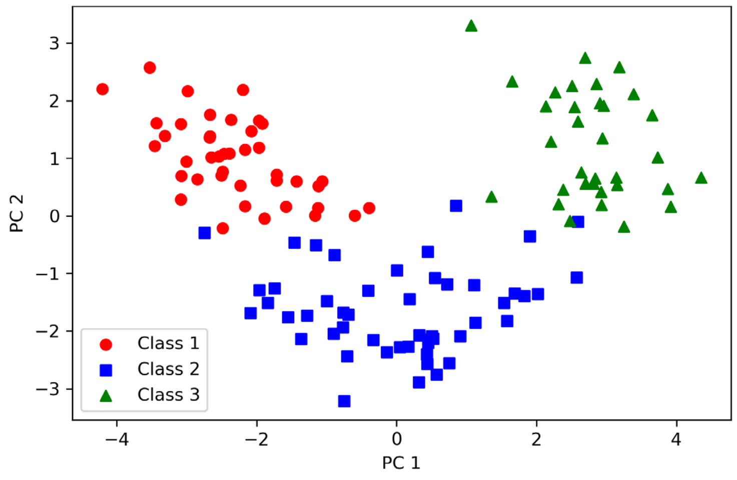

Figure 5.3: Data records from the Wine dataset projected onto a 2D feature space via PCA

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 in scikit-learn

Although the verbose approach in the previous subsection helped us to follow the inner workings of PCA, we will now discuss how to use the `PCA` class implemented in scikit-learn.

The `PCA` class is another one of scikit-learn's transformer classes, with which we first fit the model using the training data before we transform both the training data and the test dataset using the same model parameters. Now, let's use the `PCA` class from scikit-learn on the Wine training dataset, classify the transformed examples via logistic regression, and visualize the decision regions via the `plot_decision_regions` function that we defined in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*:

```python
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')
```

For your convenience, you can place the preceding `plot_decision_regions` code into a separate code file in your current working directory, for example, `plot_decision_regions_script.py`, and import it into your current Python session:

```
>>> 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()
```

By executing this code, we should now see the decision regions for the training data reduced to two principal component axes:

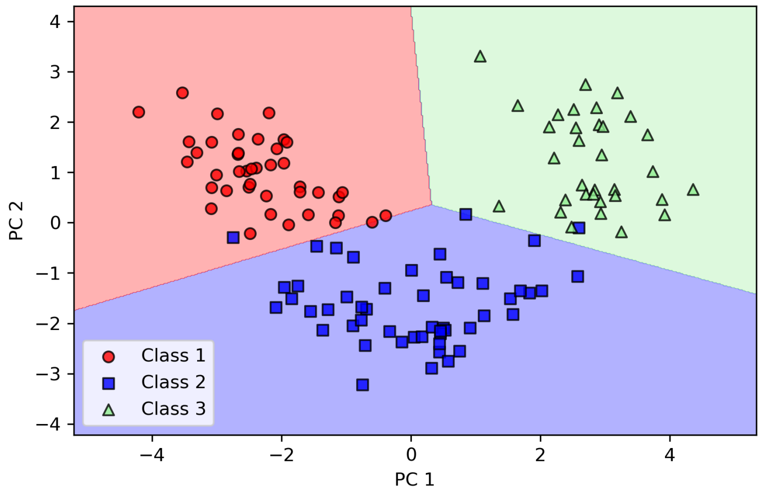

Figure 5.4: Training examples and logistic regression decision regions after using scikit-learn's PCA for dimensionality reduction

When we compare the PCA projections via scikit-learn with our own PCA implementation, we might see that the resulting plots are mirror images of each other. Note that this is not due to an error in either of those two implementations; the reason for this difference is that, depending on the eigensolver, eigenvectors can have either negative or positive signs.

Not that it matters, but we could simply revert the mirror image by multiplying the data by –1 if we wanted to; note that eigenvectors are typically scaled to unit length 1. For the sake of completeness, let's plot the decision regions of the logistic regression on the transformed test dataset to see if it can separate the classes well:

```
>>> 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()
```

After we plot the decision regions for the test dataset by executing the preceding code, we can see that logistic regression performs quite well on this small two-dimensional feature subspace and only misclassifies a few examples in the test dataset:

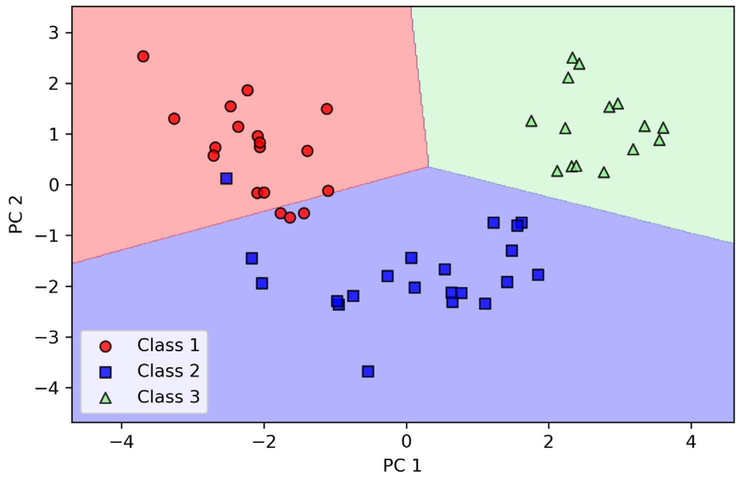

Figure 5.5: Test datapoints with logistic regression decision regions in the PCA-based feature space

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,
          0.06422108, 0.05051724, 0.03954654, 0.02643918,
          0.02389319, 0.01629614, 0.01380021, 0.01172226,
          0.00820609])
```

Note that we set `n_components=None` when we initialized the `PCA` class so that it will return all principal components in a sorted order, instead of performing a dimensionality reduction.

Assessing feature contributions

In this section, we will take a brief look at how we can assess the contributions of the original features to the principal components. As we learned, via PCA, we create principal components that represent linear combinations of the features. Sometimes, we are interested to know about how much each original feature contributes to a given principal component. These contributions are often called **loadings**.

The factor loadings can be computed by scaling the eigenvectors by the square root of the eigenvalues. The resulting values can then be interpreted as the correlation between the original features and the principal component. To illustrate this, let us plot the loadings for the first principal component.

First, we compute the 13×13-dimensional loadings matrix by multiplying the eigenvectors by the square root of the eigenvalues:

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

Then, we plot the loadings for the first principal component, `loadings[:, 0]`, which is the first column in this matrix:

```
>>> 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()
```

In *Figure 5.6*, we can see that, for example, **Alcohol** has a negative correlation with the first principal component (approximately –0.3), whereas **Malic acid** has a positive correlation (approximately 0.54). Note that a value of 1 describes a perfect positive correlation whereas a value of –1 corresponds to a perfect negative correlation:

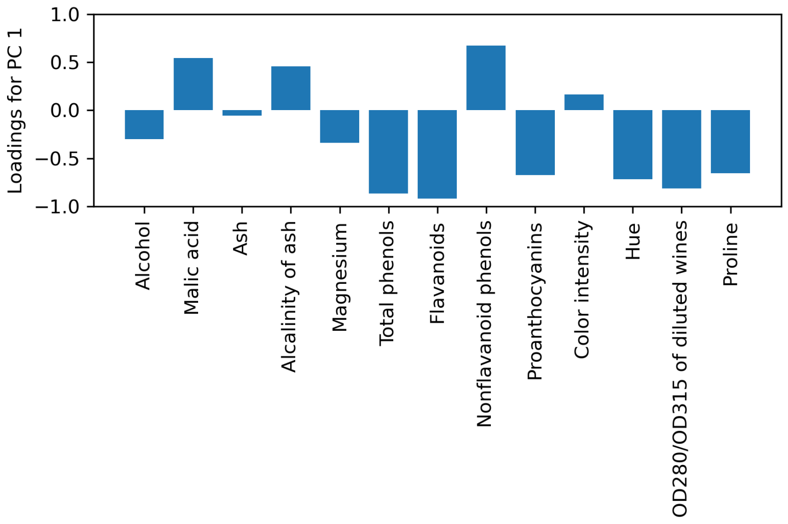

Figure 5.6: Feature correlations with the first principal component

In the preceding code example, we compute the factor loadings for our own PCA implementation. 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_)
```

To compare the scikit-learn PCA loadings with those we created previously, let us create a similar bar plot:

```
>>> 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()
```

As we can see, the bar plots look the same:

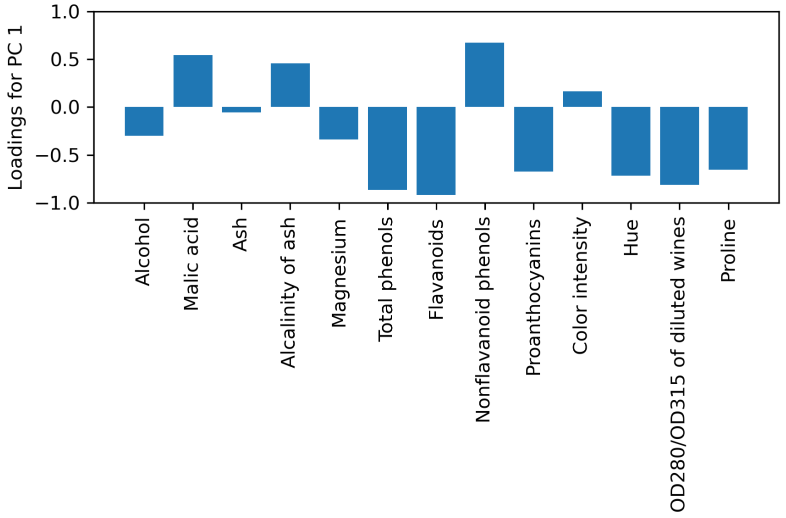

Figure 5.7: Feature correlations to the first principal component using scikit-learn

After exploring PCA as an unsupervised feature extraction technique, the next section will introduce **linear discriminant analysis** (**LDA**), which is a linear transformation technique that takes class label information into account.

Supervised data compression via linear discriminant analysis

LDA can be used as a technique for feature extraction to increase computational efficiency and reduce the degree of overfitting due to the curse of dimensionality in non-regularized models. The general concept behind LDA is very similar to PCA, but whereas PCA attempts to find the orthogonal component axes of maximum variance in a dataset, the goal in LDA is to find the feature subspace that optimizes class separability. In the following sections, we will discuss the similarities between LDA and PCA in more detail and walk through the LDA approach step by step.

Principal component analysis versus linear discriminant analysis

Both PCA and LDA are *linear transformation techniques* that can be used to reduce the number of dimensions in a dataset; the former is an unsupervised algorithm, whereas the latter is supervised. Thus, we might think that LDA is a superior feature extraction technique for classification tasks compared to PCA. However, A.M. Martinez 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 examples (*PCA Versus LDA* by *A. M. Martinez* and *A. C. Kak*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(2): 228-233, 2001).

Fisher LDA

LDA is sometimes also called **Fisher's LDA**. Ronald A. Fisher initially formulated *Fisher's Linear Discriminant* for two-class classification problems in 1936 (*The Use of Multiple Measurements in Taxonomic Problems*, *R. A. Fisher*, *Annals of Eugenics*, 7(2): 179-188, 1936). In 1948, Fisher's linear discriminant was generalized for multiclass problems by C. Radhakrishna Rao under the assumption of equal class covariances and normally distributed classes, which we now call LDA (*The Utilization of Multiple Measurements in Problems of Biological Classification* by *C. R. Rao*, *Journal of the Royal Statistical Society*. Series B (Methodological), 10(2): 159-203, 1948).

Figure 5.8 summarizes the concept of LDA for a two-class problem. Examples from class 1 are shown as circles, and examples from class 2 are shown as crosses:

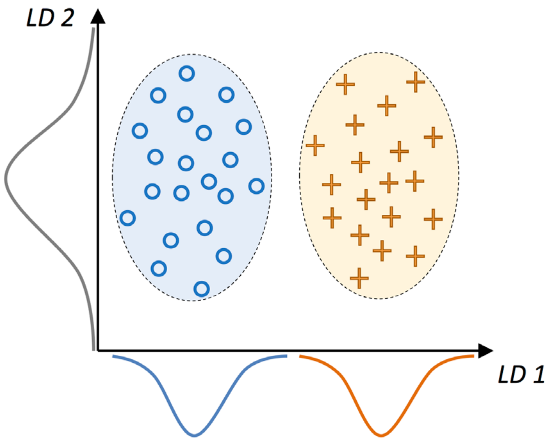

Figure 5.8: The concept of LDA for a two-class problem

A linear discriminant, as shown on the *x* axis (*LD 1*), would separate the two normal distributed classes well. Although the exemplary linear discriminant shown on the *y* axis (*LD 2*) captures a lot of the variance in the dataset, it would fail as a good linear discriminant since it does not capture any of the class-discriminatory information.

One assumption in LDA is that the data is normally distributed. Also, we assume that the classes have identical covariance matrices and that the training examples are statistically independent of each other. However, even if one, or more, of those assumptions is (slightly) violated, LDA for dimensionality reduction can still work reasonably well (*Pattern Classification 2nd Edition* by *R. O. Duda*, *P. E. Hart*, and *D. G. Stork*, *New York*, 2001).

The inner workings of linear discriminant analysis

Before we dive into the code implementation, 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. Construct the between-class scatter matrix, $\boldsymbol{S}_B$, and the within-class scatter matrix, $\boldsymbol{S}_W$.
4. Compute the eigenvectors and corresponding eigenvalues of the matrix, $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$.
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, $\boldsymbol{W}$; the eigenvectors are the columns of this matrix.
7. Project the examples onto the new feature subspace using the transformation matrix, $\boldsymbol{W}$.

As we can see, LDA is quite similar to PCA in the sense that we are decomposing matrices into eigenvalues and eigenvectors, which will form the new lower-dimensional feature space. However, as mentioned before, LDA takes class label information into account, which is represented in the form of the mean vectors computed in *step 2*. In the following sections, we will discuss these seven steps in more detail, accompanied by illustrative code implementations.

Computing the scatter matrices

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 calculation of the mean vectors, which we will use to construct the within-class scatter matrix and between-class scatter matrix, respectively. Each mean vector, $\boldsymbol{m}_i$, stores the mean feature value, μ_m, with respect to the examples of class *i*:

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

This results in three mean vectors:

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

These mean vectors can be computed by the following code, where we compute one mean vector for each of the three labels:

```
>>> 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')
```

```
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]
```

Using the mean vectors, we can now compute the within-class scatter matrix, $\boldsymbol{S}_W$:

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

This is calculated by summing up the individual scatter matrices, $\boldsymbol{S}_i$, of each individual class i:

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

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     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
```

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]
```

Thus, we want to scale the individual scatter matrices, $\boldsymbol{S}_i$, before we sum them up as the scatter matrix, $\boldsymbol{S}_W$. 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:

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

The code for computing the scaled within-class scatter matrix is as follows:

```
>>> 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
```

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 $\boldsymbol{S}_B$:

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

Here, $\boldsymbol{m}$ is the overall mean that is computed, including examples from all c classes:

```
>>> 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
```

Selecting linear discriminants for the new feature subspace

The remaining steps of the LDA are similar to the steps of the PCA. However, instead of performing the eigendecomposition on the covariance matrix, we solve the generalized eigenvalue problem of the matrix, $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$:

```
>>> 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))]
```

```
>>> 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
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
```

In LDA, the number of linear discriminants is at most $c - 1$, where c is the number of class labels, since the in-between scatter matrix, S_B, is the sum of c matrices with rank one or less. We can indeed see that we only have two nonzero eigenvalues (the eigenvalues 3-13 are not exactly zero, but this is due to the floating-point arithmetic in NumPy.)

Collinearity

Note that in the rare case of perfect collinearity (all aligned example points fall on a straight line), the covariance matrix would have rank one, which would result in only one eigenvector with a nonzero eigenvalue.

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. For simplicity, we will call the content of class-discriminatory information **discriminability**:

```
>>> 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, 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()
```

As we can see in *Figure 5.9*, the first two linear discriminants alone capture 100 percent of the useful information in the Wine training dataset:

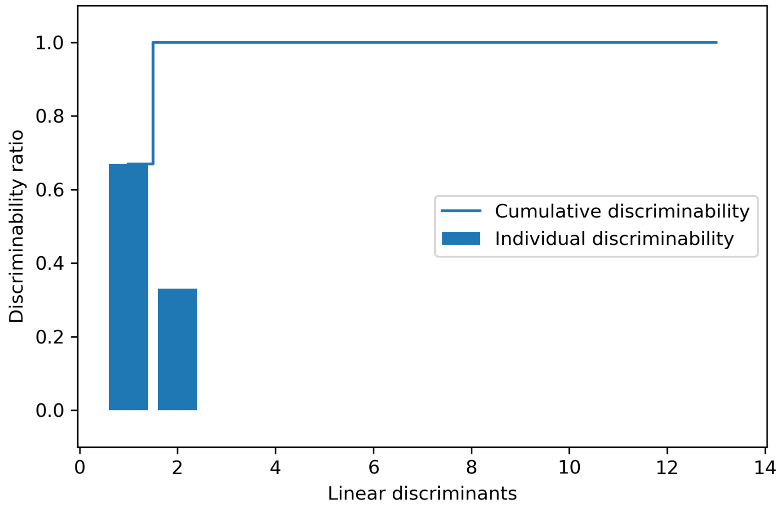

Figure 5.9: The top two discriminants capture 100 percent of the useful information

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]]
```

Projecting examples onto the new feature space

Using the transformation matrix *W* that we created in the previous subsection, 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()
```

As we can see in *Figure 5.10*, the three Wine classes are now perfectly linearly separable in the new feature subspace:

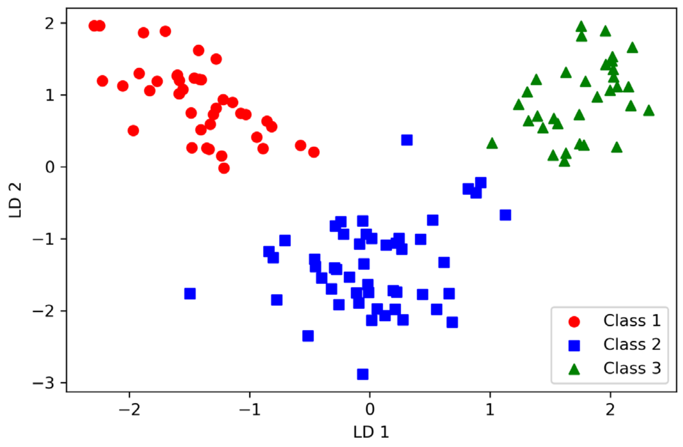

Figure 5.10: Wine classes perfectly separable after projecting the data onto the first two discriminants

LDA via scikit-learn

That step-by-step implementation was a good exercise to understand the inner workings of LDA and understand the differences between LDA and PCA. Now, let's look at the `LDA` class implemented in 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)
```

Next, let's see how the logistic regression classifier handles the lower-dimensional training dataset after the LDA transformation:

```
>>> 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()
```

Looking at *Figure 5.11*, we can see that the logistic regression model misclassifies one of the examples from class 2:

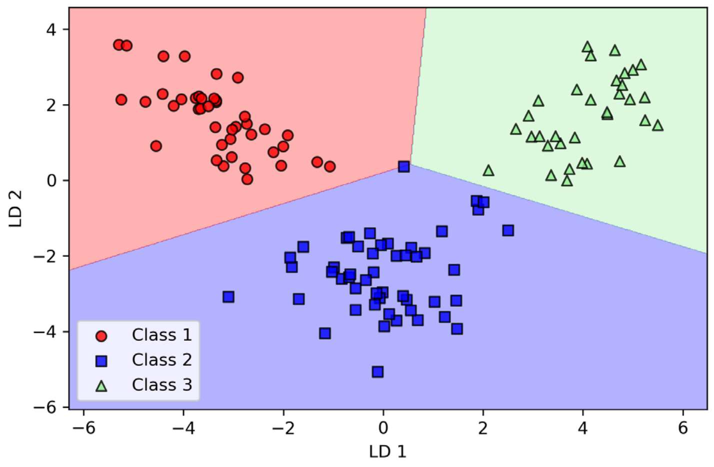

Figure 5.11: The logistic regression model misclassifies one of the classes

By lowering the regularization strength, we could probably shift the decision boundaries so that the logistic regression model classifies all examples in the training dataset correctly. However, and more importantly, let's take a look at the results on the 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.show()
```

As we can see in *Figure 5.12*, the logistic regression classifier is able to get a perfect accuracy score for classifying the examples in the test dataset by only using a two-dimensional feature subspace, instead of the original 13 Wine features:

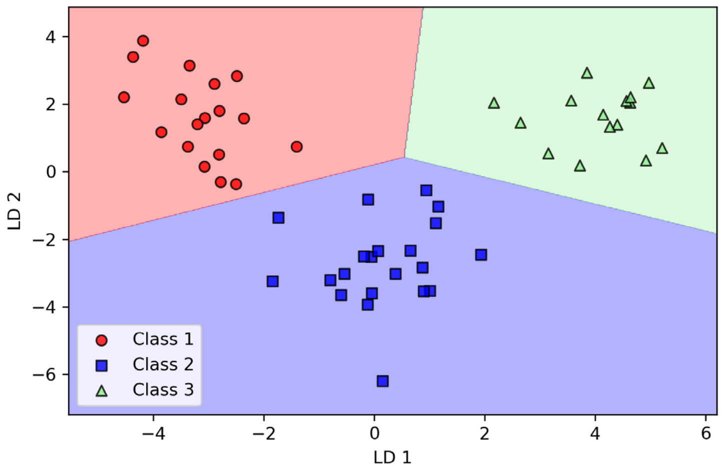

Figure 5.12: The logistic regression model works perfectly on the test data

Nonlinear dimensionality reduction and visualization

In the previous section, we covered linear transformation techniques, such as PCA and LDA, for feature extraction. In this section, we will discuss why considering nonlinear dimensionality reduction techniques might be worthwhile.

One nonlinear dimensionality reduction technique that is particularly worth highlighting is **t-distributed stochastic neighbor embedding** (**t-SNE**) since it is frequently used in literature to visualize high-dimensional datasets in two or three dimensions. We will see how we can apply t-SNE to plot images of handwritten images in a 2-dimensional feature space.

Why consider nonlinear dimensionality reduction?

Many machine learning algorithms make assumptions about the linear separability of the input data.

You have learned that the perceptron even requires perfectly linearly separable training data to converge. Other algorithms that we have covered so far assume that the lack of perfect linear separability is due to noise: Adaline, logistic regression, and the (standard) SVM to just name a few.

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:

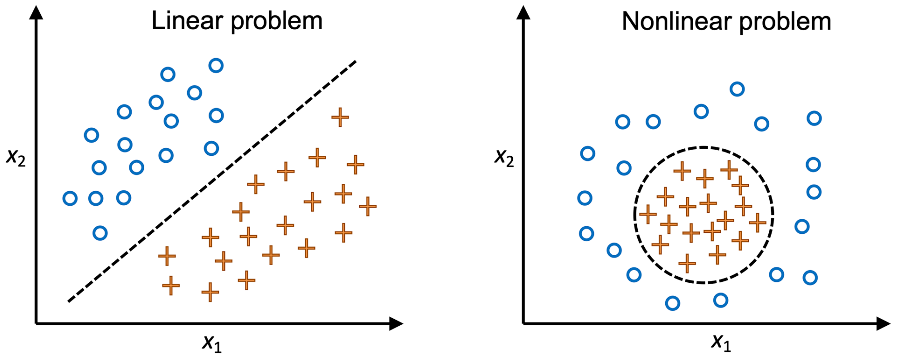

Figure 5.13: The difference between linear and nonlinear problems

The scikit-learn library implements a selection of advanced techniques for nonlinear dimensionality reduction that are beyond the scope of this book. The interested reader can find a nice overview of the current implementations in scikit-learn, complemented by illustrative examples, at `http://scikit-learn.org/stable/modules/manifold.html`.

The development and application of nonlinear dimensionality reduction techniques is also often referred to as manifold learning, where a manifold refers to a lower dimensional topological space embedded in a high-dimensional space. Algorithms for manifold learning have to capture the complicated structure of the data in order to project it onto a lower-dimensional space where the relationship between data points is preserved.

A classic example of manifold learning is the 3-dimensional Swiss roll illustrated in *Figure 5.14*:

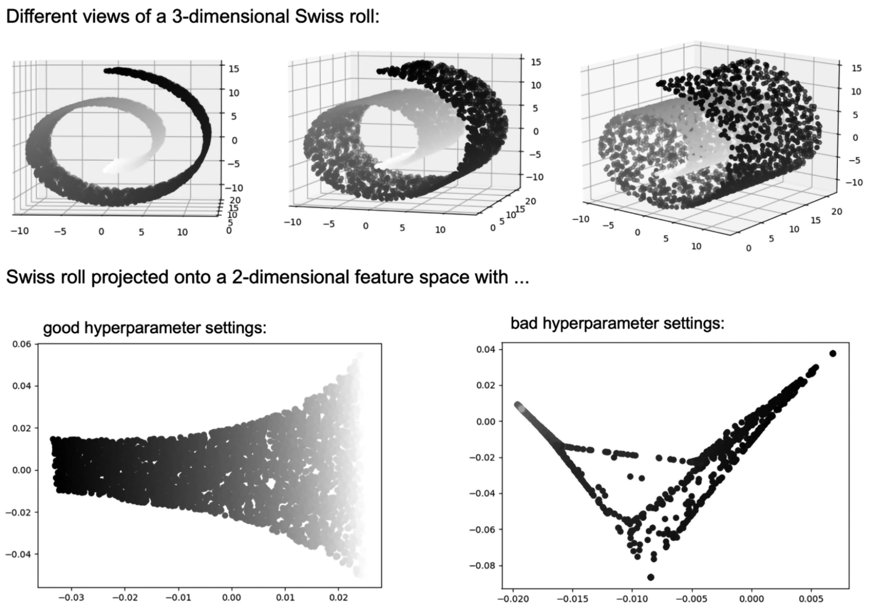

Figure 5.14: Three-dimensional Swiss roll projected into a lower, two-dimensional space

While nonlinear dimensionality reduction and manifold learning algorithms are very powerful, we should note that these techniques are notoriously hard to use, and with non-ideal hyperparameter choices, they may cause more harm than good. The reason behind this difficulty is that we are often working with high-dimensional datasets that we cannot readily visualize and where the structure is not obvious (unlike the Swiss roll example in *Figure 5.14*). Moreover, unless we project the dataset into two or three dimensions (which is often not sufficient for capturing more complicated relationships), it is hard or even impossible to assess the quality of the results. Hence, many people still rely on simpler techniques such as PCA and LDA for dimensionality reduction.

Visualizing data via t-distributed stochastic neighbor embedding

After introducing nonlinear dimensionality reduction and discussing some of its challenges, let's take a look at a hands-on example involving t-SNE, which is often used for visualizing complex datasets in two or three dimensions.

In a nutshell, t-SNE is modeling data points based on their pair-wise distances in the high-dimensional (original) feature space. Then, it finds a probability distribution of pair-wise distances in the new, lower-dimensional space that is close to the probability distribution of pair-wise distances in the original space. Or, in other words, t-SNE learns to embed data points into a lower-dimensional space such that the pairwise distances in the original space are preserved. You can find more details about this method in the original research paper *Visualizing data using t-SNE* by *Maaten and Hinton, Journal of Machine Learning Research*, 2018 (`https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf`). However, as the research paper title suggests, 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.

The following code shows a quick demonstration of how t-SNE can be applied to a 64-dimensional dataset. First, we load the Digits dataset from scikit-learn, which consists of low-resolution handwritten digits (the numbers 0-9):

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

The digits are 8×8 grayscale images. The following code plots the first four images in the dataset, which consists of 1,797 images in total:

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

As we can see in *Figure 5.15*, the images are relatively low resolution, 8×8 pixels (that is, 64 pixels per image):

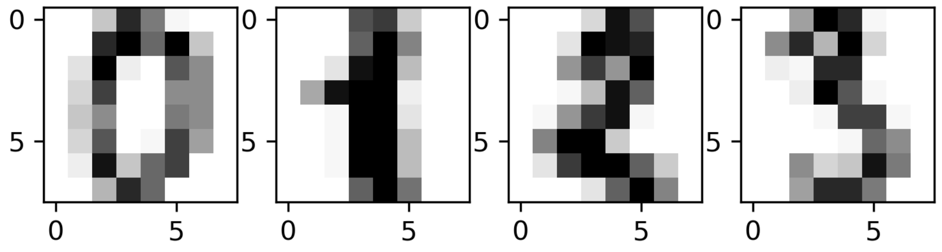

Figure 5.15: Low resolution images of handwritten digits

Note that the `digits.data` attribute lets us access a tabular version of this dataset where the examples are represented by the rows, and the columns correspond to the pixels:

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

Next, let us assign the features (pixels) to a new variable `X_digits` and the labels to another new variable `y_digits`:

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

Then, we import the t-SNE class from scikit-learn and fit a new `tsne` object. Using `fit_transform`, we perform the t-SNE fitting and data transformation in one step:

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

Using this code, we projected the 64-dimensional dataset onto a 2-dimensional space. We specified `init='pca'`, which initializes the t-SNE embedding using PCA as it is recommended in the research article *Initialization is critical for preserving global data structure in both t-SNE and UMAP* by *Kobak* and *Linderman*, *Nature Biotechnology Volume 39*, pages 156–157, 2021 (`https://www.nature.com/articles/s41587-020-00809-z`).

Note that t-SNE includes additional hyperparameters such as the perplexity and learning rate (often called **epsilon**), which we omitted in the example (we used the scikit-learn default values). In practice, we recommend you explore these parameters as well. More information about these parameters and their effects on the results can be found in the excellent article *How to Use t-SNE Effectively* by *Wattenberg*, *Viegas*, and *Johnson*, *Distill*, 2016 (`https://distill.pub/2016/misread-tsne/`).

Finally, let us visualize the 2D t-SNE embeddings using the following code:

```
>>> 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()])
```

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

Like PCA, t-SNE is an unsupervised method, and in the preceding code, we use the class labels `y_digits` (0-9) only for visualization purposes via the functions color argument. Matplotlib's `PathEffects` are used for visual purposes, such that the class label is displayed in the center (via `np.median`) of data points belonging to each respective digit. The resulting plot is as follows:

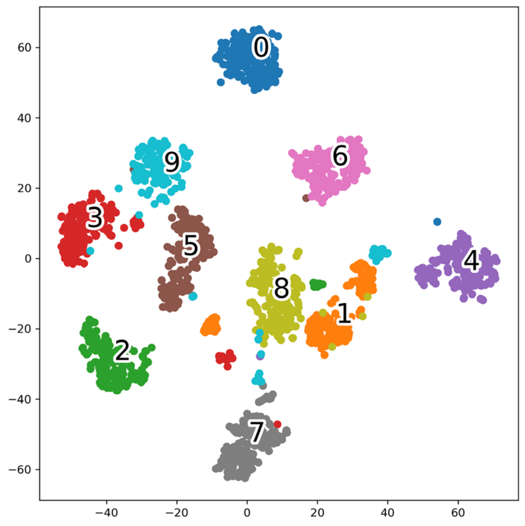

Figure 5.16: A visualization of how t-SNE embeds the handwritten digits in a 2D feature space

As we can see, t-SNE is able to separate the different digits (classes) nicely, although not perfectly. It might be possible to achieve better separation by tuning the hyperparameters. However, a certain degree of class mixing might be unavoidable due to illegible handwriting. For instance, by inspecting individual images, we might find that certain instances of the number 3 indeed look like the number 9, and so forth.

Uniform manifold approximation and projection

Another popular visualization technique is **uniform manifold approximation and projection (UMAP)**. While UMAP can produce similarly good results as t-SNE (for example, see the Kobak and Linderman paper referenced previously), it is typically faster, and it can also be used to project new data, which makes it more attractive as a dimensionality reduction technique in a machine learning context, similar to PCA. Interested readers can find more information about UMAP in the original paper: *UMAP: Uniform manifold approximation and projection for dimension reduction* by *McInnes*, *Healy*, and *Melville*, 2018 (`https://arxiv.org/abs/1802.03426`). A scikit-learn compatible implementation of UMAP can be found at `https://umap-learn.readthedocs.io`.

Summary

In this chapter, you learned about two fundamental dimensionality reduction techniques for feature extraction: PCA and LDA. Using PCA, we projected data onto a lower-dimensional subspace to maximize the variance along the orthogonal feature axes, while ignoring the class labels. LDA, in contrast to PCA, is a technique for supervised dimensionality reduction, which means that it considers class information in the training dataset to attempt to maximize the class separability in a linear feature space. Lastly, you also learned about t-SNE, which is a nonlinear feature extraction technique that can be used for visualizing data in two or three dimensions.

Equipped with PCA and LDA as fundamental data preprocessing techniques, you are now well prepared to learn about the best practices for efficiently incorporating different preprocessing techniques and evaluating the performance of different models in the next chapter.