Community Experience Distilled

Python Machine Learning

Unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics

Foreword by Dr. Randal S. Olson
Artificial Intelligence and Machine Learning Researcher, University of Pennsylvania

Sebastian Raschka

[PACKT] PUBLISHING open source* community experience distilled

www.it-ebooks.info

Python Machine Learning

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2015

Production reference: 1160915

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78355-513-0

www.packtpub.com

www.it-ebooks.info

5
Compressing Data via Dimensionality Reduction

In *Chapter 4*, *Building Good Training Sets – 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 three fundamental techniques that will help us 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:

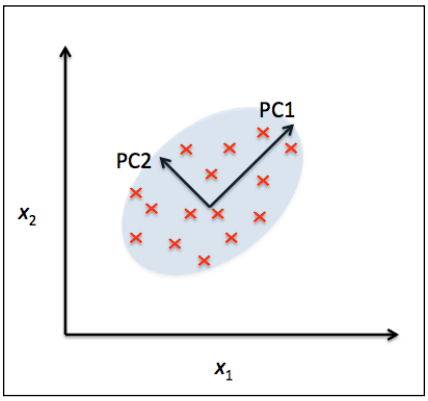
- **Principal component analysis** (**PCA**) for unsupervised data compression
- **Linear Discriminant Analysis** (**LDA**) as a supervised dimensionality reduction technique for maximizing class separability
- Nonlinear dimensionality reduction via **kernel principal component analysis**

www.it-ebooks.info

Unsupervised dimensionality reduction via principal component analysis

Similar to feature selection, we can use feature extraction to reduce the number of features in a dataset. However, while we maintained the original features when we used 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. Feature extraction is typically used to improve computational efficiency but can also help to reduce the *curse of dimensionality*—especially if we are working with nonregularized models.

Principal component analysis (**PCA**) is an unsupervised linear transformation technique that is widely used across different fields, most prominently for dimensionality reduction. Other popular applications of PCA include exploratory data analyses and de-noising of signals in stock market trading, and the analysis 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 it onto a new subspace with equal or fewer dimensions that 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 the following figure. Here, x_1 and x_2 are the original feature axes, and **PC1** and **PC2** are the principal components:



www.it-ebooks.info

If we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix $\boldsymbol{W}$ that allows us to map a sample vector $\boldsymbol{x}$ onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space:

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

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

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

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, and all consequent principal components will have the largest possible variance given that they are uncorrelated (orthogonal) to the other principal components. 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. Select k eigenvectors that correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
5. Construct a projection matrix $\boldsymbol{W}$ from the "top" k eigenvectors.
6. Transform the d-dimensional input dataset $\boldsymbol{X}$ using the projection matrix $\boldsymbol{W}$ to obtain the new k-dimensional feature subspace.

www.it-ebooks.info

Total and explained variance

In this subsection, we will tackle the first four steps of a principal component analysis: standardizing the data, constructing the covariance matrix, obtaining the eigenvalues and eigenvectors of the covariance matrix, and sorting the eigenvalues by decreasing order to rank the eigenvectors.

First, we will start by loading the *Wine* dataset that we have been working with in *Chapter 4*, *Building Good Training Sets – 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)
```

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

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> 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, random_state=0)
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.fit_transform(X_test)
```

After completing the mandatory preprocessing steps by executing the preceding code, let's advance to the second step: constructing the covariance matrix. The symmetric $d \times d$-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 $\boldsymbol{x}_j$ and x_k on the population level can be calculated via the following equation:

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

Here, μ_j and μ_k are the sample means of feature j and k, respectively. Note that the sample means are zero if we standardize 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, a covariance matrix of three features can then be written as (note that Σ stands for the Greek letter *sigma*, which is not to be confused with the *sum* symbol):

www.it-ebooks.info

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

The 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, let's obtain the eigenpairs of the covariance matrix. As we surely remember from our introductory linear algebra or calculus classes, an eigenvalue $\boldsymbol{v}$ 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%s' % eigen_vals)
Eigenvalues
[ 4.8923083   2.46635032  1.42809973  1.01233462  0.84906459
0.60181514
0.52251546  0.08414846  0.33051429  0.29595018  0.16831254  0.21432212
0.2399553 ]
```

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

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). Since the eigenvalues define the magnitude of the eigenvectors, 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.

www.it-ebooks.info

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:

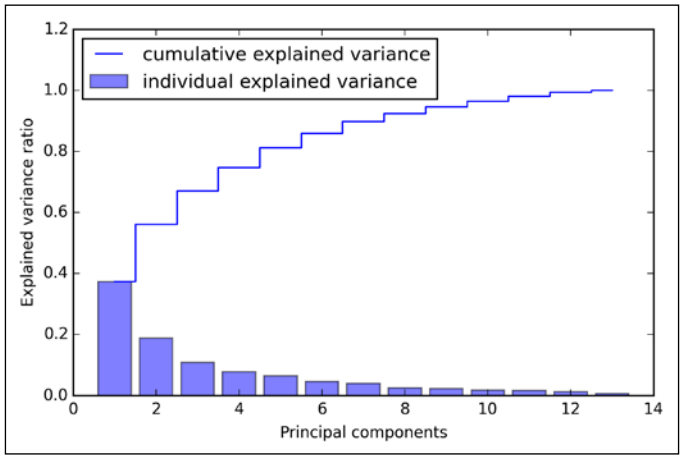
$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

Using the NumPy `cumsum` function, we can then calculate the cumulative sum of explained variances, which we will 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, alpha=0.5, align='center',
...         label='individual explained variance')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...         label='cumulative explained variance')
>>> plt.ylabel('Explained variance ratio')
>>> plt.xlabel('Principal components')
>>> plt.legend(loc='best')
>>> plt.show()
```

The resulting plot indicates that the first principal component alone accounts for 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 data:



www.it-ebooks.info

Although the explained variance plot reminds us of the feature importance that we computed in *Chapter 4*, *Building Good Training Sets – Data Preprocessing*, via random forests, we shall 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

After we have successfully decomposed the covariance matrix into eigenpairs, let's now proceed with the last three steps to transform the *Wine* dataset onto the new principal component axes. In this section, 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:

```
>>> eigen_pairs =[(np.abs(eigen_vals[i]),eigen_vecs[:,i])
...               for i inrange(len(eigen_vals))]
>>> eigen_pairs.sort(reverse=True)
```

Next, we collect the two eigenvectors that correspond to the two largest values to capture about 60 percent of the variance in this dataset. Note that we only chose two eigenvectors for the purpose of illustration, since we are going to plot the data via a two-dimensional scatter plot later in this subsection. In practice, the number of principal components has to be determined from a trade-off 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:
[[ 0.14669811  0.50417079]
[-0.24224554  0.24216889]
[-0.02993442  0.28698484]
[-0.25519002 -0.06468718]
[ 0.12079772  0.22995385]
[ 0.38934455  0.09363991]
[ 0.42326486  0.01088622]
[-0.30634956  0.01870216]
[ 0.30572219  0.03040352]
[-0.09869191  0.54527081]
```


www.it-ebooks.info

```
[ 0.30032535 -0.27924322]
[ 0.36821154 -0.174365  ]
[ 0.29259713  0.36315461]]
```

By executing the preceding code, we have created a 13×2-dimensional projection matrix W from the top two eigenvectors. Using the projection matrix, we can now transform a sample x (represented as 1×13-dimensional row vector) onto the PCA subspace obtaining x', a now two-dimensional sample vector consisting of two new features:

$$x' = xW$$

```
>>> X_train_std[0].dot(w)
array([ 2.59891628,  0.00484089])
```

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

$$X' = XW$$

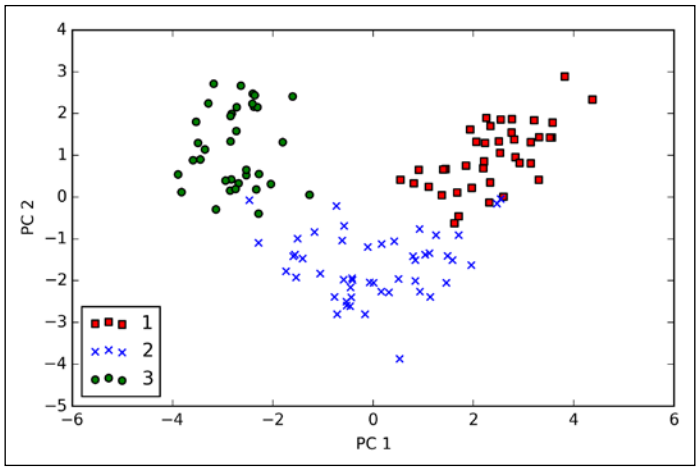
```
>>> X_train_pca = X_train_std.dot(w)
```

Lastly, let's visualize the transformed *Wine* training set, now stored as an 124×2-dimensional matrix, in a two-dimensional scatterplot:

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

As we can see in the resulting plot (shown in the next figure), the data is more spread along the *x*-axis—the first principal component—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 intuitively see that a linear classifier will likely be able to separate the classes well:

www.it-ebooks.info



Although we encoded the class labels information for the purpose of illustration in the preceding scatter plot, we have to keep in mind that PCA is an unsupervised technique that doesn't use 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. `PCA` is another one of scikit-learn's transformer classes, where we first fit the model using the training data before we transform both the training data and the test data using the same model parameters. Now, let's use the `PCA` from scikit-learn on the *Wine* training dataset, classify the transformed samples via logistic regression, and visualize the decision regions via the `plot_decision_region` function that we defined in *Chapter 2, Training Machine Learning Algorithms for Classification*:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

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

    # plot the decision surface
```

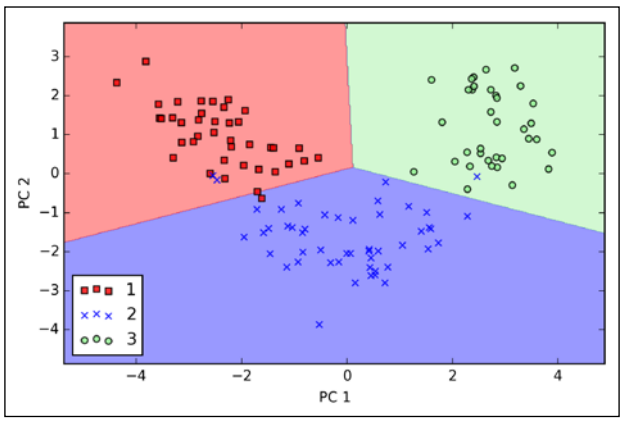

www.it-ebooks.info

```
    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))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
```

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression()
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

By executing the preceding code, we should now see the decision regions for the training model reduced to the two principal component axes.

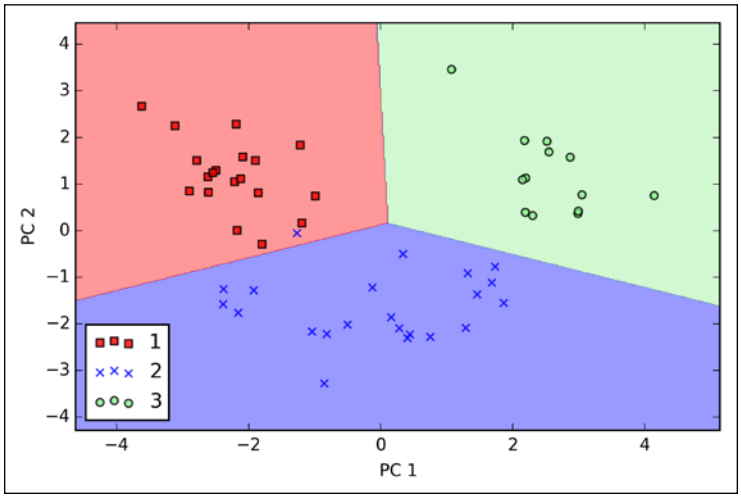


www.it-ebooks.info

If we compare the PCA projection via scikit-learn with our own PCA implementation, we notice that the plot above is a mirror image of the previous PCA via our step-by-step approach. Note that this is not due to an error in any of those two implementations, but 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 with `-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('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

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



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_
```


www.it-ebooks.info

```
array([ 0.37329648,  0.18818926,  0.10896791,  0.07724389,
0.06478595,
0.04592014,  0.03986936,  0.02521914,  0.02258181,  0.01830924,
0.01635336,  0.01284271,  0.00642076])
```

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

Supervised data compression via linear discriminant analysis

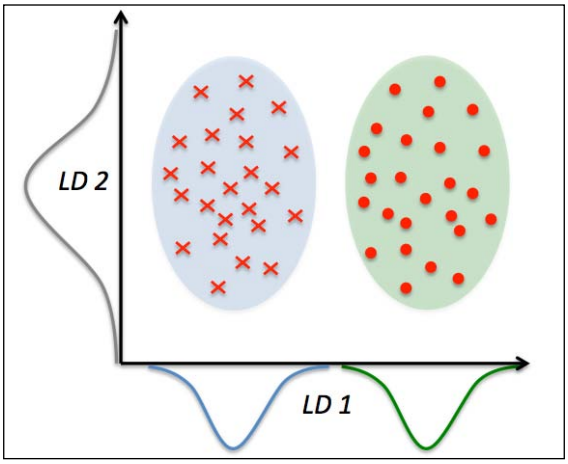
Linear Discriminant Analysis (**LDA**) can be used as a technique for feature extraction to increase the computational efficiency and reduce the degree of over-fitting due to the curse of dimensionality in nonregularized models.

The general concept behind LDA is very similar to PCA, 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. Both LDA and PCA 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 intuitively 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 samples (A. M. Martinez and A. C. Kak. *PCA Versus LDA*. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 23(2):228–233, 2001).

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

www.it-ebooks.info

The following figure summarizes the concept of LDA for a two-class problem. Samples from class 1 are shown as crosses and samples from class 2 are shown as circles, respectively:



A linear discriminant, as shown on the *x*-axis (LD 1), would separate the two normally 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 features are statistically independent of each other. However, even if one or more of those assumptions are slightly violated, LDA for dimensionality reduction can still work reasonably well (R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. 2nd. Edition. New York, 2001).

Before we take a look into the inner workings of LDA in the following subsections, let's summarize the key steps of the LDA approach:

1. Standardize the d-dimensional dataset (d is the number of features).
2. For each class, compute the d-dimensional mean vector.
3. Construct the between-class scatter matrix $\boldsymbol{S}_B$ and the within-class scatter matrix $\boldsymbol{S}_w$.

www.it-ebooks.info

4. Compute the eigenvectors and corresponding eigenvalues of the matrix $\boldsymbol{S}_w^{-1}\boldsymbol{S}_B$.
5. Choose the k eigenvectors that correspond to the k largest eigenvalues to construct a $d \times k$-dimensional transformation matrix $\boldsymbol{W}$; the eigenvectors are the columns of this matrix.
6. Project the samples onto the new feature subspace using the transformation matrix $\boldsymbol{W}$.

The assumptions that we make when we are using LDA are that the features are normally distributed and independent of each other. Also, the LDA algorithm assumes that the covariance matrices for the individual classes are identical. However, even if we violate those assumptions to a certain extent, LDA may still work reasonably well in dimensionality reduction and classification tasks (R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. 2nd. Edition. New York, 2001).

Computing the scatter matrices

Since we have 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 samples of class i:

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

This results in three mean vectors:

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

www.it-ebooks.info

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...                X_train_std[y_train==label], axis=0))
...     print('MV %s: %s\n' %(label, mean_vecs[label-1]))
MV 1: [ 0.9259 -0.3091  0.2592 -0.7989  0.3039  0.9608  1.0515 -0.6306
0.5354
  0.2209  0.4855  0.798   1.2017]

MV 2: [-0.8727 -0.3854 -0.4437  0.2481 -0.2409 -0.1059  0.0187 -0.0164
0.1095
 -0.8796  0.4392  0.2776 -0.7016]

MV 3: [ 0.1637  0.8929  0.3249  0.5658 -0.01   -0.9499 -1.228   0.7436
-0.7652
  0.979  -1.1698 -1.3007 -0.3912]
```

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:

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

```
>>> 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[y == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row-mv).dot((row-mv).T)
...     S_W += class_scatter
>>> print('Within-class scatter matrix: %sx%s'
...        % (S_W.shape[0], S_W.shape[1]))
Within-class scatter matrix: 13x13
```


www.it-ebooks.info

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

```
>>> print('Class label distribution: %s'
...       % np.bincount(y_train)[1:])
Class label distribution: [40 49 35]
```

Thus, we want to scale the individual scatter matrices $\boldsymbol{S}_i$ before we sum them up as scatter matrix $\boldsymbol{S}_w$. When we divide the scatter matrices by the number of class samples 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:

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

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

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

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

Here, $\boldsymbol{m}$ is the overall mean that is computed, including samples from all classes.

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # number of features
>>> S_B = np.zeros((d, d))
>>> for i,mean_vec in enumerate(mean_vecs):
...     n = X[y==i+1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1)
...     mean_overall = mean_overall.reshape(d, 1)
    S_B += n * (mean_vec - mean_overall).dot(
```


www.it-ebooks.info

```
...                (mean_vec - mean_overall).T)
print('Between-class scatter matrix: %sx%s'
...    % (S_B.shape[0], S_B.shape[1]))
Between-class scatter matrix: 13x13
```

Selecting linear discriminants for the new feature 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 $S_w^{-1}S_B$:

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

After we computed the eigenpairs, we can now 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 decreasing order:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])


Eigenvalues in decreasing order:

643.015384346
225.086981854
1.37146633984e-13
5.68434188608e-14
4.16877714935e-14
4.16877714935e-14
3.76733516161e-14
3.7544790902e-14
3.7544790902e-14
2.30295239559e-14
2.30295239559e-14
1.9101018959e-14
3.86601693797e-16
```

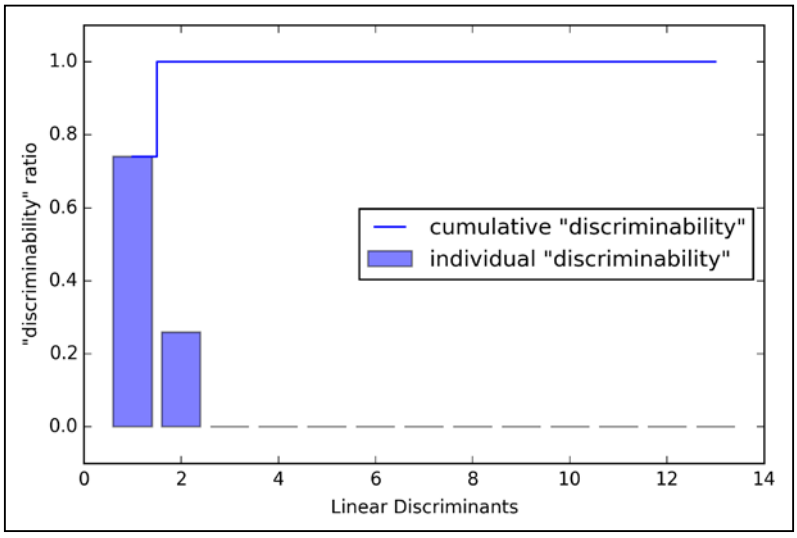

www.it-ebooks.info

Those who are a little more familiar with linear algebra may know that the rank of the $d \times d$-dimensional covariance matrix can be at most $d-1$, and 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). Note that in the rare case of perfect collinearity (all aligned sample 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 the 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, alpha=0.5, align='center',
...         label='individual "discriminability"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...          label='cumulative "discriminability"')
>>> plt.ylabel('"discriminability" ratio')
>>> plt.xlabel('Linear Discriminants')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.show()
```

As we can see in the resulting figure, the first two linear discriminants capture about 100 percent of the useful information in the *Wine* training dataset:



www.it-ebooks.info

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:
 [[-0.0707 -0.3778]
 [ 0.0359 -0.2223]
 [-0.0263 -0.3813]
 [ 0.1875  0.2955]
 [-0.0033  0.0143]
 [ 0.2328  0.0151]
 [-0.7719  0.2149]
 [-0.0803  0.0726]
 [ 0.0896  0.1767]
 [ 0.1815 -0.2909]
 [-0.0631  0.2376]
 [-0.3794  0.0867]
 [-0.3355 -0.586 ]]
```

Projecting samples onto the new feature space

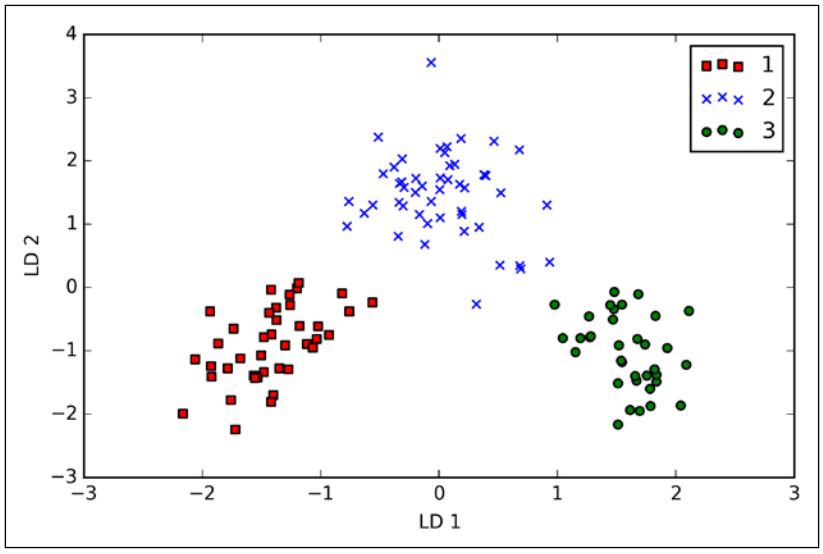
Using the transformation matrix W that we created in the previous subsection, we can now transform the training data set by multiplying the matrices:

$$X' = XW$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0],
...                 X_train_lda[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='upper right')
>>> plt.show()
```


www.it-ebooks.info

As we can see in the resulting plot, the three wine classes are now linearly separable in the new feature subspace:



LDA via scikit-learn

The step-by-step implementation was a good exercise for understanding the inner workings of LDA and understanding the differences between LDA and PCA. Now, let's take a look at the `LDA` class implemented in scikit-learn:

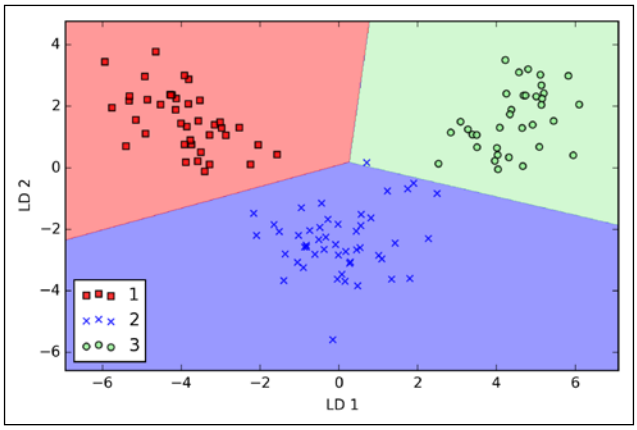
```
>>> from sklearn.lda import LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

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

```
>>> lr = LogisticRegression()
>>> 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.show()
```


www.it-ebooks.info

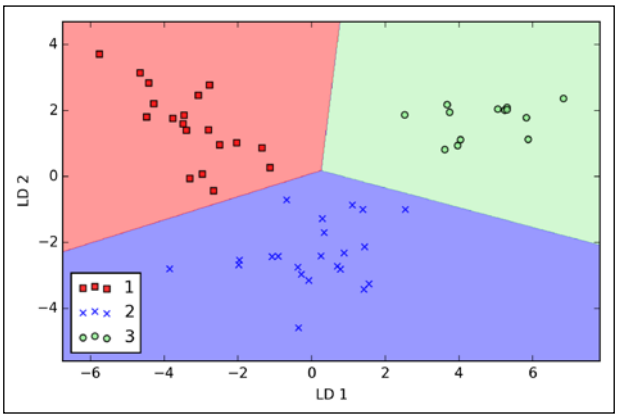
Looking at the resulting plot, we see that the logistic regression model misclassifies one of the samples from class 2:



By lowering the regularization strength, we could probably shift the decision boundaries so that the logistic regression models classify all samples in the training dataset correctly. However, let's take a look at the results on the test set:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

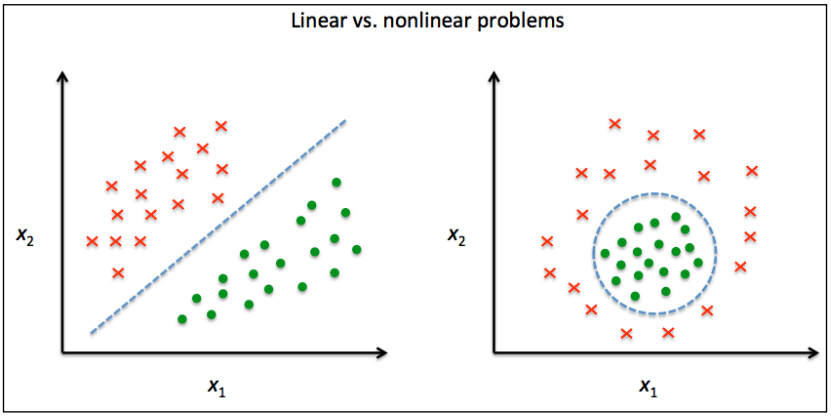
As we can see in the resulting plot, the logistic regression classifier is able to get a perfect accuracy score for classifying the samples in the test dataset by only using a two-dimensional feature subspace instead of the original 13 *Wine* features:



www.it-ebooks.info

Using kernel principal component analysis for nonlinear mappings

Many machine learning algorithms make assumptions about the linear separability of the input data. You 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) **support vector machine** (**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. In this section, we will take a look at a kernelized version of PCA, or *kernel PCA*, which relates to the concepts of kernel SVM that we remember from *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*. Using kernel PCA, we will learn how to transform data that is not linearly separable onto a new, lower-dimensional subspace that is suitable for linear classifiers.



Kernel functions and the kernel trick

As we remember from our discussion about kernel SVMs in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we can tackle nonlinear problems by projecting them onto a new feature space of higher dimensionality where the classes become linearly separable. To transform the samples $x \in \mathbb{R}^d$ onto this higher k-dimensional subspace, we defined a nonlinear mapping function ϕ:

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

www.it-ebooks.info

We can think of ϕ as a function that creates nonlinear combinations of the original features to map the original d-dimensional dataset onto a larger, k-dimensional feature space. For example, if we had feature vector $\boldsymbol{x} \in \mathbb{R}^d$ ($\boldsymbol{x}$ is a column vector consisting of d features) with two dimensions $(d = 2)$, a potential mapping onto a 3D space could be as follows:

$$\boldsymbol{x} = [x_1,\ x_2]^T$$

$$\downarrow \phi$$

$$\mathbf{z} = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2 \right]^T$$

In other words, via kernel PCA we perform a nonlinear mapping that transforms the data onto a higher-dimensional space and use standard PCA in this higher-dimensional space to project the data back onto a lower-dimensional space where the samples can be separated by a linear classifier (under the condition that the samples can be separated by density in the input space). However, one downside of this approach is that it is computationally very expensive, and this is where we use the *kernel trick*. Using the kernel trick, we can compute the similarity between two high-dimension feature vectors in the original feature space.

Before we proceed with more details about using the kernel trick to tackle this computationally expensive problem, let's look back at the *standard* PCA approach that we implemented at the beginning of this chapter. We computed the covariance between two features k and j as follows:

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

Since the standardizing of features centers them at mean zero, for instance, $\frac{1}{n} \sum_i x_j^{(i)} = 0$, we can simplify this equation as follows:

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

www.it-ebooks.info

Note that the preceding equation refers to the covariance between two features; now, let's write the general equation to calculate the covariance *matrix* Σ:

$$\sum = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}^{(i)}\, \boldsymbol{x}^{(i)T}$$

Bernhard Scholkopf generalized this approach (B. Scholkopf, A. Smola, and K.-R. Muller. *Kernel Principal Component Analysis*. pages 583–588, 1997) so that we can replace the dot products between samples in the original feature space by the nonlinear feature combinations via ϕ:

$$\sum = \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi(\boldsymbol{x}^{(i)})^{T}$$

To obtain the eigenvectors—the principal components—from this covariance matrix, we have to solve the following equation:

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

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow v = \frac{1}{n\lambda}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} \boldsymbol{v} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{a}^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right)$$

Here, λ and $\boldsymbol{v}$ are the eigenvalues and eigenvectors of the covariance matrix Σ, and $\boldsymbol{a}$ can be obtained by extracting the eigenvectors of the kernel (similarity) matrix $\boldsymbol{K}$ as we will see in the following paragraphs.

The derivation of the kernel matrix is as follows:

First, let's write the covariance matrix as in matrix notation, where $\phi(X)$ is an $n \times k$-dimensional matrix:

$$\sum = \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} = \frac{1}{n}\phi(\boldsymbol{X})^{T}\,\phi(\boldsymbol{X})$$

www.it-ebooks.info

Now, we can write the eigenvector equation as follows:

$$v = \frac{1}{n}\sum_{i=1}^{n} a^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right) = \lambda\phi(\boldsymbol{X})^{T}\,\boldsymbol{a}$$

Since $\Sigma\boldsymbol{v} = \lambda\boldsymbol{v}$, we get:

$$\frac{1}{n}\phi(\boldsymbol{X})^{T}\,\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\boldsymbol{a} = \lambda\phi(\boldsymbol{X})^{T}\,\boldsymbol{a}$$

Multiplying it by $\phi(\boldsymbol{X})$ on both sides yields the following result:

$$\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\boldsymbol{a} = \lambda\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\,\boldsymbol{a} = \lambda\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\boldsymbol{K}\boldsymbol{a} = \lambda\boldsymbol{a}$$

Here, $\boldsymbol{K}$ is the similarity (kernel) matrix:

$$\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}$$

As we recall from the SVM section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-learn*, we use the kernel trick to avoid calculating the pairwise dot products of the samples $\boldsymbol{x}$ under ϕ explicitly by using a kernel function $\boldsymbol{K}$ so that we don't need to calculate the eigenvectors explicitly:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^{T}\,\phi\left(\boldsymbol{x}^{(j)}\right)$$

In other words, what we obtain after kernel PCA are the samples already projected onto the respective components rather than constructing a transformation matrix as in the standard PCA approach. Basically, the kernel function (or simply *kernel*) can be understood as a function that calculates a dot product between two vectors—a measure of similarity.

www.it-ebooks.info

The most commonly used kernels are as follows:

- The polynomial kernel:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\left(\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}+\theta\right)^{p}$$

  Here, θ is the threshold and p is the power that has to be specified by the user.

- The hyperbolic tangent (sigmoid) kernel:

$$k\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\tanh\left(\eta\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}+\theta\right)$$

- The **Radial Basis Function** (**RBF**) or Gaussian kernel that we will use in the following examples in the next subsection:

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

  It is also written as follows:

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

To summarize what we have discussed so far, we can define the following three steps to implement an RBF kernel PCA:

1. We compute the kernel (similarity) matrix k, where we need to calculate the following:

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

www.it-ebooks.info

We do this for each pair of samples:

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

For example, if our dataset contains 100 training samples, the symmetric kernel matrix of the pair-wise similarities would be 100×100 dimensional.

2. We center the kernel matrix k using the following equation:

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

Here, $1_{\boldsymbol{n}}$ is an $n \times n$ - dimensional matrix (the same dimensions as the kernel matrix) where all values are equal to $\frac{1}{n}$.

3. We collect the top k eigenvectors of the centered kernel matrix based on their corresponding eigenvalues, which are ranked by decreasing magnitude. In contrast to standard PCA, the eigenvectors are not the principal component axes but the samples projected onto those axes.

At this point, you may be wondering why we need to center the kernel matrix in the second step. We previously assumed that we are working with standardized data, where all features have mean zero when we formulated the covariance matrix and replaced the dot products by the nonlinear feature combinations via ϕ.Thus, the centering of the kernel matrix in the second step becomes necessary, since we do not compute the new feature space explicitly and we cannot guarantee that the new feature space is also centered at zero.

In the next section, we will put those three steps into action by implementing a kernel PCA in Python.

www.it-ebooks.info

Implementing a kernel principal component analysis in Python

In the previous subsection, we discussed the core concepts behind kernel PCA. Now, we are going to implement an RBF kernel PCA in Python following the three steps that summarized the kernel PCA approach. Using the SciPy and NumPy helper functions, we will see that implementing a kernel PCA is actually really simple:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
      Tuning parameter of the RBF kernel

    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
```


www.it-ebooks.info

```
    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # numpy.eigh returns them in sorted order
    eigvals, eigvecs = eigh(K)

    # Collect the top k eigenvectors (projected samples)
    X_pc = np.column_stack((eigvecs[:, -i]
                            for i in range(1, n_components + 1)))

    return X_pc
```

One downside of using an RBF kernel PCA for dimensionality reduction is that we have to specify the parameter γ a priori. Finding an appropriate value for γ requires experimentation and is best done using algorithms for parameter tuning, for example, grid search, which we will discuss in more detail in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

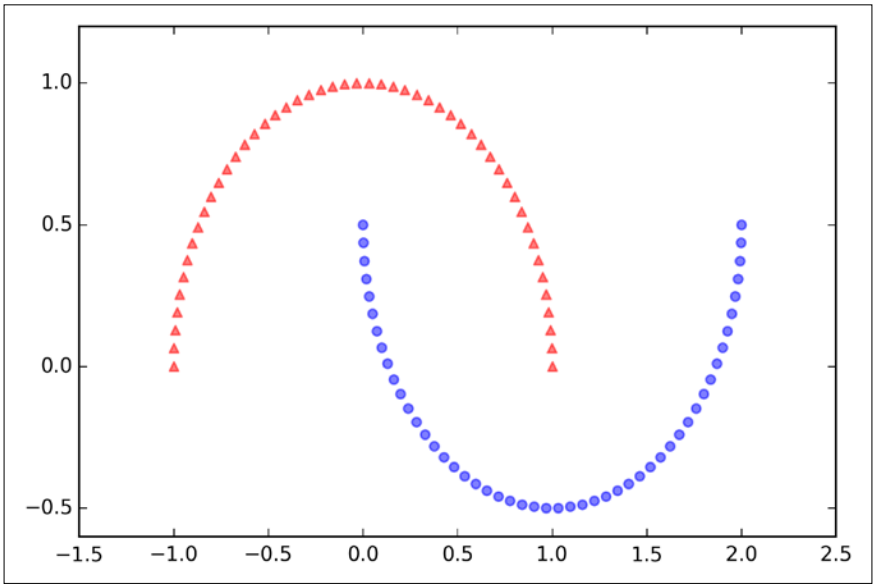
Example 1 – separating half-moon shapes

Now, let's apply our `rbf_kernel_pca` on some nonlinear example datasets. We will start by creating a two-dimensional dataset of 100 sample points representing two half-moon shapes:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```


www.it-ebooks.info

For the purposes of illustration, the half-moon of triangular symbols shall represent one class and the half-moon depicted by the circular symbols represent the samples from another class:



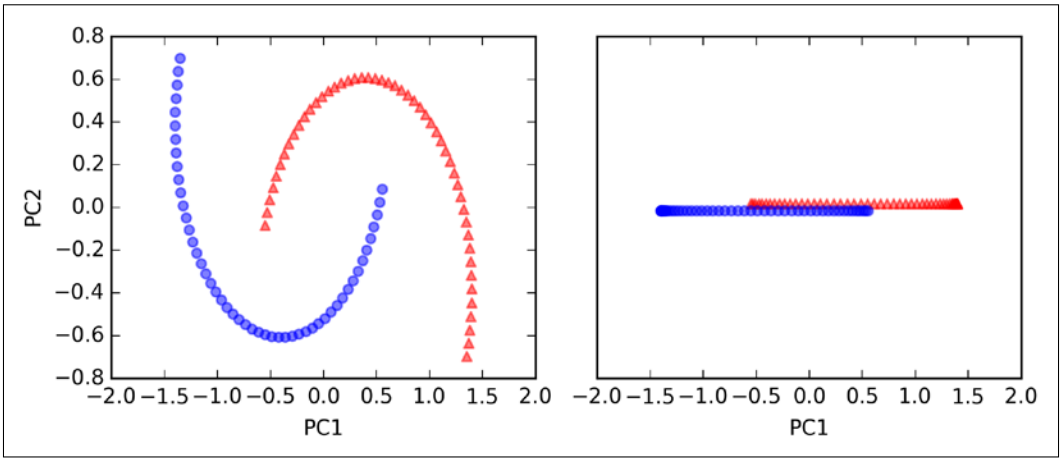
Clearly, these two half-moon shapes are not linearly separable and our goal is to *unfold* the half-moons via kernel PCA so that the dataset can serve as a suitable input for a linear classifier. But first, let's see what the dataset looks like if we project it onto the principal components via standard PCA:

```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
```


www.it-ebooks.info

```
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Clearly, we can see in the resulting figure that a linear classifier would be unable to perform well on the dataset transformed via standard PCA:



Note that when we plotted the first principal component only (right subplot), we shifted the triangular samples slightly upwards and the circular samples slightly downwards to better visualize the class overlap.

Please remember that PCA is an unsupervised method and does not use class label information in order to maximize the variance in contrast to LDA. Here, the triangular and circular symbols were just added for visualization purposes to indicate the degree of separation.

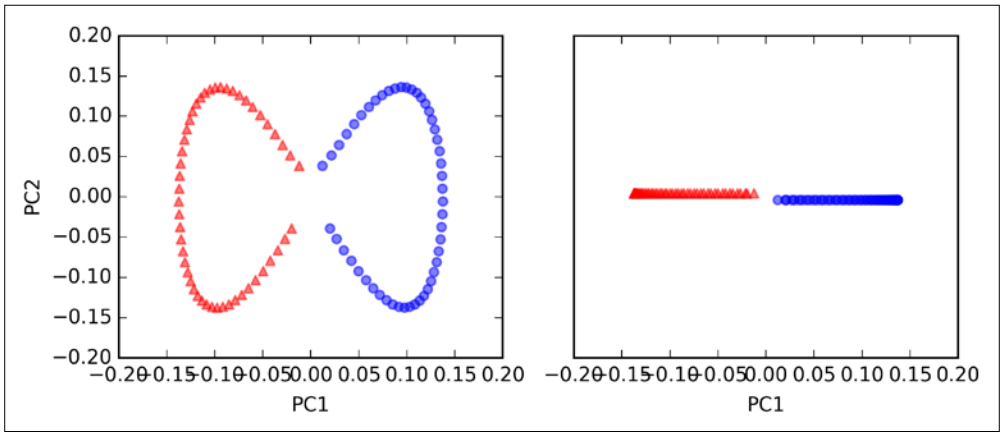
Now, let's try out our kernel PCA function `rbf_kernel_pca`, which we implemented in the previous subsection:

```
>>> from matplotlib.ticker import FormatStrFormatter
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
```


www.it-ebooks.info

```
...                color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...                color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...                color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> plt.show()
```

We can now see that the two classes (circles and triangles) are linearly well separated so that it becomes a suitable training dataset for linear classifiers:



Unfortunately, there is no universal value for the tuning parameter γ that works well for different datasets. To find a γ value that is appropriate for a given problem requires experimentation. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will discuss techniques that can help us to automate the task of optimizing such tuning parameters. Here, I will use values for γ that I found produce *good* results.

www.it-ebooks.info

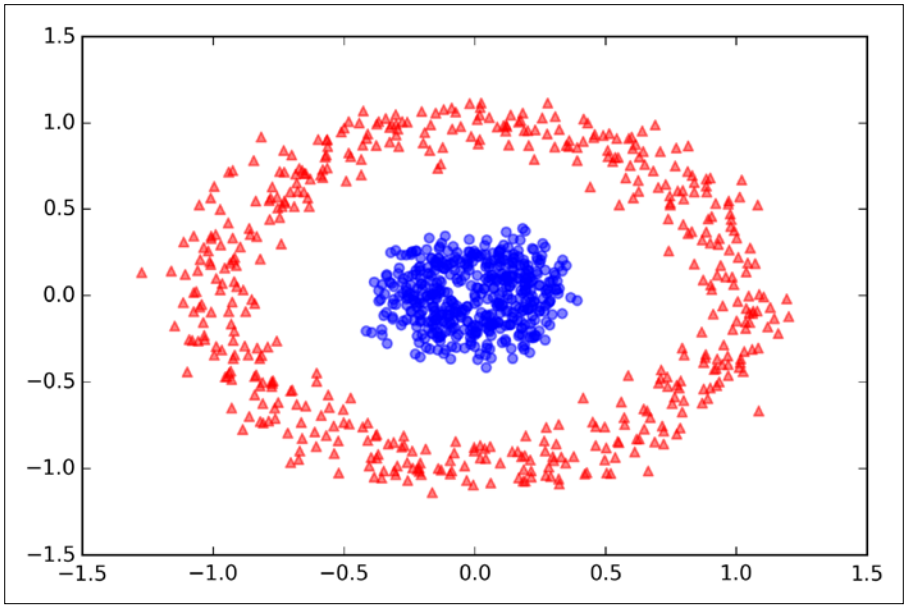
Example 2 – separating concentric circles

In the previous subsection, we showed you how to separate half-moon shapes via kernel PCA. Since we put so much effort into understanding the concepts of kernel PCA, let's take a look at another interesting example of a nonlinear problem: concentric circles.

The code is as follows:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...                     random_state=123, noise=0.1, factor=0.2)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

Again, we assume a two-class problem where the triangle shapes represent one class and the circle shapes represent another class, respectively:

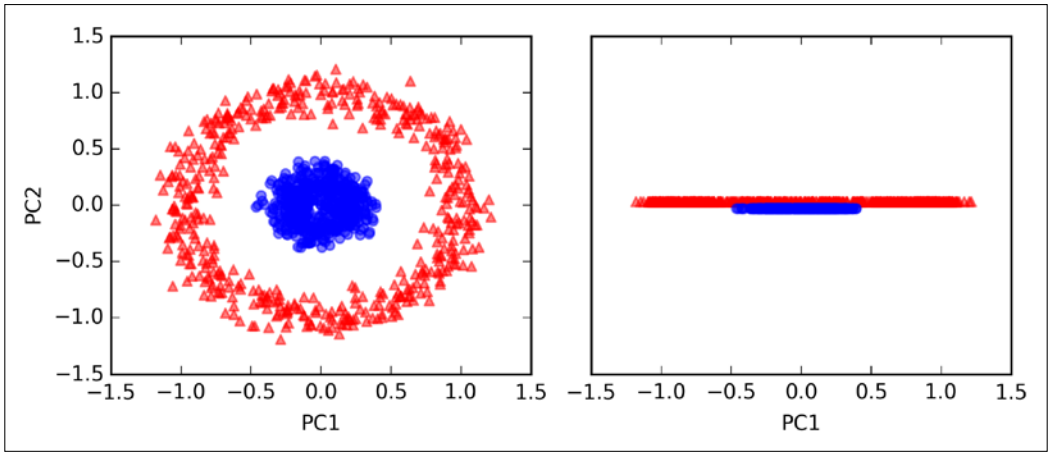


www.it-ebooks.info

Let's start with the standard PCA approach to compare it with the results of the RBF kernel PCA:

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...              color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Again, we can see that standard PCA is not able to produce results suitable for training a linear classifier:

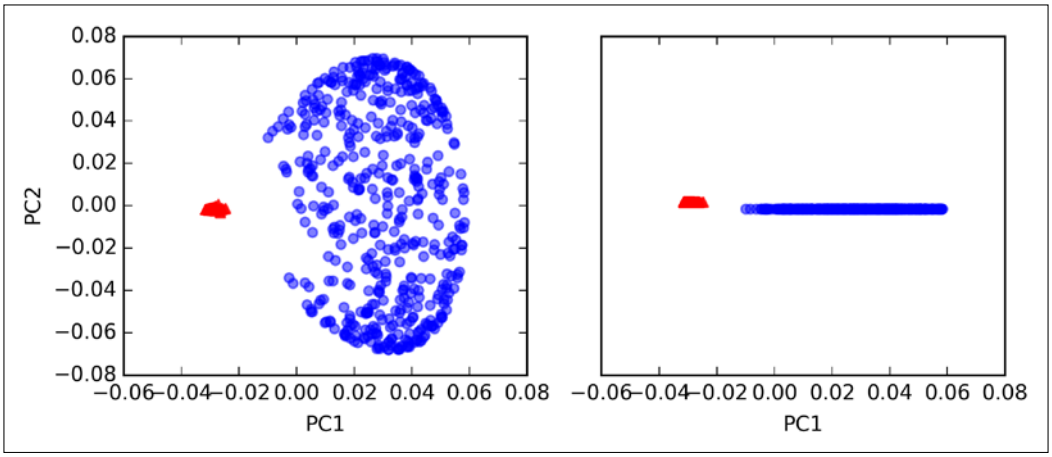


www.it-ebooks.info

Given an appropriate value for γ, let's see if we are luckier using the RBF kernel PCA implementation:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

Again, the RBF kernel PCA projected the data onto a new subspace where the two classes become linearly separable:



www.it-ebooks.info

Projecting new data points

In the two previous example applications of kernel PCA, the half-moon shapes and the concentric circles, we projected a single dataset onto a new feature. In real applications, however, we may have more than one dataset that we want to transform, for example, training and test data, and typically also new samples we will collect after the model building and evaluation. In this section, you will learn how to project data points that were not part of the training dataset.

As we remember from the standard PCA approach at the beginning of this chapter, we project data by calculating the dot product between a transformation matrix and the input samples; the columns of the projection matrix are the top k eigenvectors ($\boldsymbol{v}$) that we obtained from the covariance matrix. Now, the question is how can we transfer this concept to kernel PCA? If we think back to the idea behind kernel PCA, we remember that we obtained an eigenvector ($\boldsymbol{a}$) of the centered kernel matrix (not the covariance matrix), which means that those are the samples that are already projected onto the principal component axis $\boldsymbol{v}$. Thus, if we want to project a new sample $\boldsymbol{x}'$ onto this principal component axis, we'd need to compute the following:

$$\phi(\boldsymbol{x}')^T \boldsymbol{v}$$

Fortunately, we can use the kernel trick so that we don't have to calculate the projection $\phi(\boldsymbol{x}')^T \boldsymbol{v}$ explicitly. However, it is worth noting that kernel PCA, in contrast to standard PCA, is a memory-based method, which means that we have to reuse the original training set each time to project new samples. We have to calculate the pairwise RBF kernel (similarity) between each i th sample in the training dataset and the new sample $\boldsymbol{x}'$:

$$\phi(\boldsymbol{x}')^T \boldsymbol{v} = \sum_i a^{(i)} \phi(\boldsymbol{x}')^T \phi\left(\boldsymbol{x}^{(i)}\right)$$

$$= \sum_i a^{(i)} k\left(\boldsymbol{x}', \boldsymbol{x}^{(i)}\right)^T$$

Here, eigenvectors $\boldsymbol{a}$ and eigenvalues λ of the Kernel matrix $\boldsymbol{K}$ satisfy the following condition in the equation:

$$\boldsymbol{K}\boldsymbol{a} = \lambda \boldsymbol{a}$$

www.it-ebooks.info

After calculating the similarity between the new samples and the samples in the training set, we have to normalize the eigenvector $\boldsymbol{a}$ by its eigenvalue. Thus, let's modify the `rbf_kernel_pca` function that we implemented earlier so that it also returns the eigenvalues of the kernel matrix:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
      Tuning parameter of the RBF kernel

    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset

     lambdas: list
       Eigenvalues

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
```


www.it-ebooks.info

```
    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # numpy.eigh returns them in sorted order
    eigvals, eigvecs = eigh(K)

    # Collect the top k eigenvectors (projected samples)
    alphas = np.column_stack((eigvecs[:,-i]
                    for i in range(1,n_components+1)))

    # Collect the corresponding eigenvalues
    lambdas = [eigvals[-i] for i in range(1,n_components+1)]

    return alphas, lambdas
```

Now, let's create a new half-moon dataset and project it onto a one-dimensional subspace using the updated RBF kernel PCA implementation:

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

To make sure that we implement the code for projecting new samples, let's assume that the 26th point from the half-moon dataset is a new data point $\boldsymbol{x}'$, and our task is to project it onto this new subspace:

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25] # original projection
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...                     (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
... return k.dot(alphas / lambdas)
```


www.it-ebooks.info

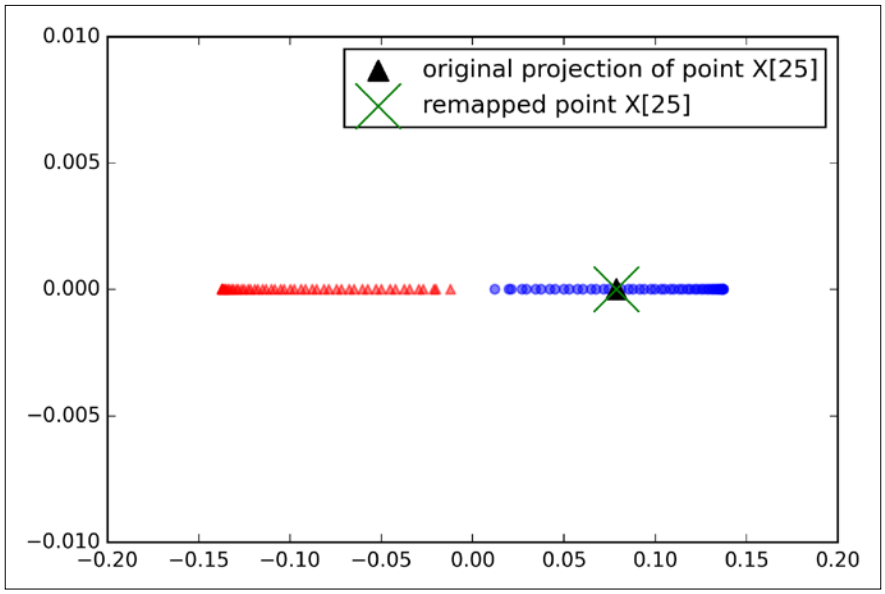
By executing the following code, we are able to reproduce the original projection. Using the `project_x` function, we will be able to project any new data samples as well. The code is as follows:

```
>>> x_reproj = project_x(x_new, X,
...       gamma=15, alphas=alphas, lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

Lastly, let's visualize the projection on the first principal component:

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

As we can see in the following scatterplot, we mapped the sample $\boldsymbol{x}'$ onto the first principal component correctly:



www.it-ebooks.info

Kernel principal component analysis in scikit-learn

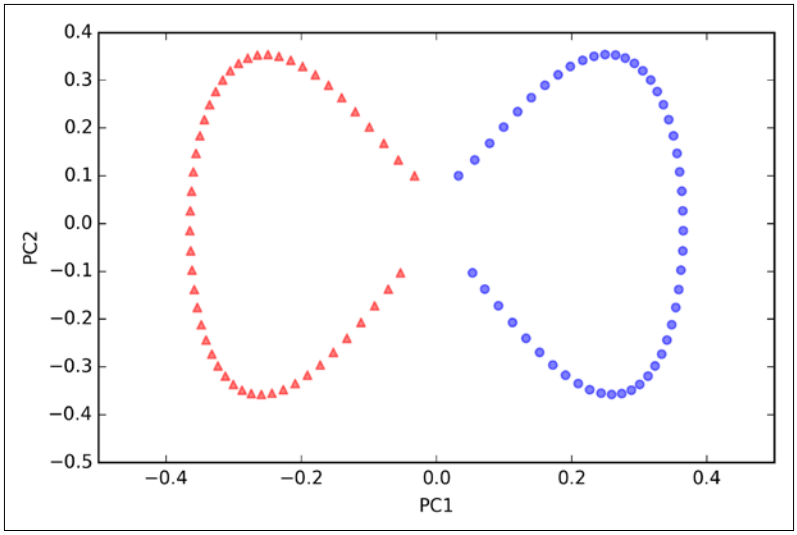
For our convenience, scikit-learn implements a kernel PCA class in the `sklearn.decomposition` submodule. The usage is similar to the standard PCA class, and we can specify the kernel via the `kernel` parameter:

```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...                         kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

To see if we get results that are consistent with our own kernel PCA implementation, let's plot the transformed half-moon shape data onto the first two principal components:

```
>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.show()
```

As we can see, the results of the scikit-learn `KernelPCA` are consistent with our own implementation:



www.it-ebooks.info

Scikit-learn also implements advanced techniques for nonlinear dimensionality reduction that are beyond the scope of this book. You can find a nice overview of the current implementations in scikit-learn complemented with illustrative examples at `http://scikit-learn.org/stable/modules/manifold.html`.

Summary

In this chapter, you learned about three different, fundamental dimensionality reduction techniques for feature extraction: standard PCA, LDA, and kernel PCA. 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 learned about a kernelized version of PCA, which allows you to map nonlinear datasets onto a lower-dimensional feature space where the classes become linearly separable.

Equipped with these essential 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.

www.it-ebooks.info