Although stochastic gradient descent can be considered as an approximation of gradient descent, it typically reaches convergence much faster because of the more frequent weight updates. Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient descent, which can also have the advantage that stochastic gradient descent can escape shallow local minima more readily. To obtain accurate results via stochastic gradient descent, it is important to present it with data in a random order, which is why we want to shuffle the training set for every epoch to prevent cycles.

In stochastic gradient descent implementations, the fixed learning rate η is often replaced by an adaptive learning rate that decreases over time, for example, $\frac{c_1}{[number\ of\ iterations]+c_2}$ where c_1 and c_2 are constants. Note that stochastic gradient descent does not reach the global minimum but an area very close to it. By using an adaptive learning rate, we can achieve further annealing to a better global minimum

Another advantage of stochastic gradient descent is that we can use it for *online learning*. In online learning, our model is trained on-the-fly as new training data arrives. This is especially useful if we are accumulating large amounts of data—for example, customer data in typical web applications. Using online learning, the system can immediately adapt to changes and the training data can be discarded after updating the model if storage space ~~in~~ is an issue.

A compromise between batch gradient descent and stochastic gradient descent is the so-called *mini-batch learning*. Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data—for example, 50 samples at a time. The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates. Furthermore, mini-batch learning allows us to replace the for-loop over the training samples in **Stochastic Gradient Descent** (**SGD**) by vectorized operations, which can further improve the computational efficiency of our learning algorithm.

We will assign the *petal length* and *petal width* of the 150 flower samples to the feature matrix `X` and the corresponding class labels of the flower species to the vector `y`:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
```

If we executed `np.unique(y)` to return the different class labels stored in `iris.target`, we would see that the Iris flower class names, *Iris-Setosa*, *Iris-Versicolor*, and *Iris-Virginica*, are already stored as integers (`0`, `1`, `2`), which is recommended for the optimal performance of many machine learning libraries.

To evaluate how well a trained model performs on unseen data, we will further split the dataset into separate training and test datasets. Later in *Chapter 5*, *Compressing Data via Dimensionality Reduction*, we will discuss the best practices around model evaluation in more detail:

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...           X, y, test_size=0.3, random_state=0)
```

Using the `train_test_split` function from scikit-learn's `cross_validation` module, we randomly split the `X` and `y` arrays into 30 percent test data (45 samples) and 70 percent training data (105 samples).

Many machine learning and optimization algorithms also require feature scaling for optimal performance, as we remember from the **gradient descent** example in *Chapter 2*, *Training Machine Learning Algorithms for Classification*. Here, we will standardize the features using the `StandardScaler` class from scikit-learn's `preprocessing` module:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Applied to the standardized Wine data, the L1 regularized logistic regression would yield the following sparse solution:

```
>>> lr = LogisticRegression(penalty='l1', C=0.1)
>>> lr.fit(X_train_std, y_train)
>>> print('Training accuracy:', lr.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', lr.score(X_test_std, y_test))
Test accuracy: 0.981481481481
```

Both training and test accuracies (both 98 percent) do not indicate any overfitting of our model. When we access the intercept terms via the `lr.intercept_` attribute, we can see that the array returns three values:

```
>>> lr.intercept_
array([-0.38379237, -0.1580855 , -0.70047966])
```

Since we the fit the `LogisticRegression` object on a multiclass dataset, it uses the **One-vs-Rest** (**OvR**) approach by default where the first intercept belongs to the model that fits class 1 versus class 2 and 3; the second value is the intercept of the model that fits class 2 versus class 1 and 3; and the third value is the intercept of the model that fits class 3 versus class 1 and 2, respectively:

```
>>> lr.coef_
array([[ 0.280, 0.000, 0.000, -0.0282, 0.000,
         0.000, 0.710, 0.000, 0.000, 0.000,
         0.000, 0.000, 1.236],
       [-0.644, -0.0688 , -0.0572, 0.000, 0.000,
         0.000, 0.000, 0.000, 0.000, -0.927,
         0.060, 0.000, -0.371],
       [ 0.000, 0.061, 0.000, 0.000, 0.000,
         0.000, -0.637, 0.000, 0.000, 0.499,
        -0.358, -0.570, 0.000
       ]])
```

The weight array that we accessed via the `lr.coef_` attribute contains three rows of weight coefficients, one weight vector for each class. Each row consists of 13 weights where each weight is multiplied by the respective feature in the 13-dimensional Wine dataset to calculate the net input:

$$z = w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

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.

Due to the brevity of this statement, the sentence may potentially be confusing. Maybe something along the lines of

“Lastly, you learned about a nonlinear feature extractor, kernel PCA. Using the kernel trick, and a temporary projection into a higher-dimensional feature space, you were ultimately able to compress datasets consisting of nonlinear features onto a smaller dimensional subspace where the classes became linearly separable.“

would be better?

On a side note, it is also worth mentioning that we technically don't have to update the weights of the intercept if we are working with standardized variables since the *y* axis intercept is always 0 in those cases. We can quickly confirm this by printing the weights:

```
>>> print('Slope: %.3f' % lr.w_[1])
Slope: 0.695
>>> print('Intercept: %.3f' % lr.w_[0])
Intercept: -0.000
```

Estimating the coefficient of a regression model via scikit-learn

In the previous section, we implemented a working model for regression analysis. However, in a real-world application, we may be interested in more efficient implementations, for example, scikit-learn's `LinearRegression` object that makes use of the **LIBLINEAR** library and advanced optimization algorithms that work better with unstandardized variables. This is sometimes desirable for certain applications:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> print('Slope: %.3f' % slr.coef_[0])
Slope: 9.102
>>> print('Intercept: %.3f' % slr.intercept_)
Intercept: -34.671
```

As we can see by executing the preceding code, scikit-learn's `LinearRegression` model fitted with the unstandardized **RM** and **MEDV** variables yielded different model coefficients. Let's compare it to our own GD implementation by plotting MEDV against RM:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()w
```