3

A Tour of Machine Learning Classifiers Using scikit-learn

In this chapter, we will take a tour of a selection of popular and powerful machine learning algorithms that are commonly used in academia as well as in industry. While learning about the differences between several supervised learning algorithms for classification, we will also develop an appreciation of their individual strengths and weaknesses. In addition, we will take our first steps with the scikit-learn library, which offers a user-friendly and consistent interface for using those algorithms efficiently and productively.

The topics that will be covered throughout this chapter are as follows:

- An introduction to robust and popular algorithms for classification, such as logistic regression, support vector machines, and decision trees
- Examples and explanations using the scikit-learn machine learning library, which provides a wide variety of machine learning algorithms via a user-friendly Python API
- Discussions about the strengths and weaknesses of classifiers with linear and nonlinear decision boundaries

Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice and experience; each algorithm has its own quirks and is based on certain assumptions. To restate the **no free lunch theorem** by David H. Wolpert, no single classifier works best across all possible scenarios (*The Lack of A Priori Distinctions Between Learning Algorithms*, *Wolpert, David H*, *Neural Computation 8.7* (1996): 1341-1390). In practice, it is always recommended that you compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem; these may differ in the number of features or examples, the amount of noise in a dataset, and whether the classes are linearly separable or not.

Eventually, the performance of a classifier—computational performance as well as predictive power—depends heavily on the underlying data that is available for learning. The five main steps that are involved in training a supervised machine learning algorithm can be summarized as follows:

1. Selecting features and collecting labeled training examples.
2. Choosing a performance metric.
3. Choosing a classifier and optimization algorithm.
4. Evaluating the performance of the model.
5. Tuning the algorithm.

Since the approach of this book is to build machine learning knowledge step by step, we will mainly focus on the main concepts of the different algorithms in this chapter and revisit topics such as feature selection and preprocessing, performance metrics, and hyperparameter tuning for more detailed discussions later in the book.

First steps with scikit-learn – training a perceptron

In *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, you learned about two related learning algorithms for classification, the **perceptron** rule and **Adaline**, which we implemented in Python and NumPy by ourselves. Now we will take a look at the scikit-learn API, which, as mentioned, combines a user-friendly and consistent interface with a highly optimized implementation of several classification algorithms. The scikit-learn library offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models. We will discuss this in more detail, together with the underlying concepts, in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, and *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

To get started with the scikit-learn library, we will train a perceptron model similar to the one that we implemented in *Chapter 2*. For simplicity, we will use the already familiar **Iris dataset** throughout the following sections. Conveniently, the Iris dataset is already available via scikit-learn, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms. Similar to the previous chapter, we will only use two features from the Iris dataset for visualization purposes.

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

```
>>> from sklearn import datasets
>>> import numpy as np

>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> print('Class labels:', np.unique(y))
Class labels: [0 1 2]
```

The `np.unique(y)` function returned the three unique class labels stored in `iris.target`, and as we can see, the Iris flower class names, `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`, are already stored as integers (here: `0`, `1`, `2`). Although many scikit-learn functions and class methods also work with class labels in string format, using integer labels is a recommended approach to avoid technical glitches and improve computational performance due to a smaller memory footprint; furthermore, encoding class labels as integers is a common convention among most 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. In *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will discuss the best practices around model evaluation in more detail:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=1, stratify=y)
```

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

Note that the `train_test_split` function already shuffles the training datasets internally before splitting; otherwise, all examples from class `0` and class `1` would have ended up in the training datasets, and the test dataset would consist of 45 examples from class `2`. Via the `random_state` parameter, we provided a fixed random seed (`random_state=1`) for the internal pseudo-random number generator that is used for shuffling the datasets prior to splitting. Using such a fixed `random_state` ensures that our results are reproducible.

Lastly, we took advantage of the built-in support for stratification via `stratify=y`. In this context, stratification means that the `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset. We can use NumPy's `bincount` function, which counts the number of occurrences of each value in an array, to verify that this is indeed the case:

```
>>> print('Labels counts in y:', np.bincount(y))
```

```
Labels counts in y: [50 50 50]
>>> print('Labels counts in y_train:', np.bincount(y_train))
Labels counts in y_train: [35 35 35]
>>> print('Labels counts in y_test:', np.bincount(y_test))
Labels counts in y_test: [15 15 15]
```

Many machine learning and optimization algorithms also require feature scaling for optimal performance, as we saw in the **gradient descent** example in *Chapter 2, Training Simple 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)
```

Using the preceding code, we loaded the `StandardScaler` class from the `preprocessing` module and initialized a new `StandardScaler` object that we assigned to the `sc` variable. Using the `fit` method, `StandardScaler` estimated the parameters, μ (sample mean) and σ (standard deviation), for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters, μ and σ. Note that we used the same scaling parameters to standardize the test dataset so that both the values in the training and test dataset are comparable to each other.

Having standardized the training data, we can now train a perceptron model. Most algorithms in scikit-learn already support multiclass classification by default via the **one-vs.-rest** (**OvR**) method, which allows us to feed the three flower classes to the perceptron all at once. The code is as follows:

```
>>> from sklearn.linear_model import Perceptron

>>> ppn = Perceptron(eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

The scikit-learn interface will remind you of our perceptron implementation in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*. After loading the `Perceptron` class from the `linear_model` module, we initialized a new `Perceptron` object and trained the model via the `fit` method. Here, the model parameter, `eta0`, is equivalent to the learning rate, `eta`, that we used in our own perceptron implementation, and the `n_iter` parameter defines the number of epochs (passes over the training dataset).

As you will remember from *Chapter 2*, finding an appropriate learning rate requires some experimentation. If the learning rate is too large, the algorithm will overshoot the global cost minimum. If the learning rate is too small, the algorithm will require more epochs until convergence, which can make the learning slow—especially for large datasets. Also, we used the `random_state` parameter to ensure the reproducibility of the initial shuffling of the training dataset after each epoch.

Having trained a model in scikit-learn, we can make predictions via the `predict` method, just like in our own perceptron implementation in *Chapter 2*. The code is as follows:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified examples: %d' % (y_test != y_pred).sum())
Misclassified examples: 1
```

Executing the code, we can see that the perceptron misclassifies 1 out of the 45 flower examples. Thus, the misclassification error on the test dataset is approximately 0.022 or 2.2 percent ($1/45 \approx 0.022$).

Classification error versus accuracy

Instead of the misclassification error, many machine learning practitioners report the classification accuracy of a model, which is simply calculated as follows:

1–*error* = 0.978 or 97.8 percent

Whether we use the classification error or accuracy is merely a matter of preference.

Note that scikit-learn also implements a large variety of different performance metrics that are available via the `metrics` module. For example, we can calculate the classification accuracy of the perceptron on the test dataset as follows:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
Accuracy: 0.978
```

Here, `y_test` are the true class labels and `y_pred` are the class labels that we predicted previously. Alternatively, each classifier in scikit-learn has a `score` method, which computes a classifier's prediction accuracy by combining the `predict` call with `accuracy_score`, as shown here:

```
>>> print('Accuracy: %.3f' % ppn.score(X_test_std, y_test))
Accuracy: 0.978
```

Overfitting

Note that we will evaluate the performance of our models based on the test dataset in this chapter. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, you will learn about useful techniques, including graphical analysis, such as learning curves, to detect and prevent **overfitting**. Overfitting, which we will return to later in this chapter, means that the model captures the patterns in the training data well but fails to generalize well to unseen data.

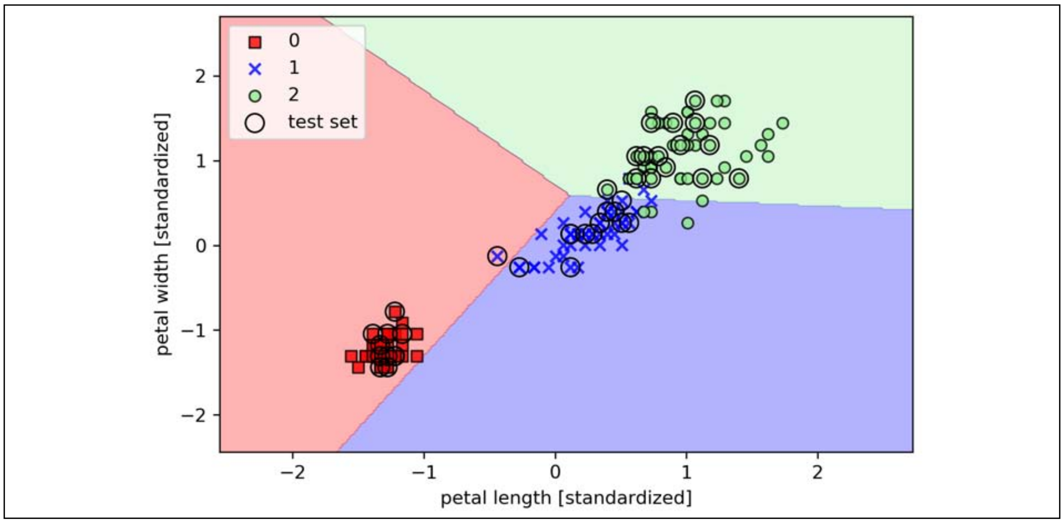
Finally, we can use our `plot_decision_regions` function from *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, to plot the **decision regions** of our newly trained perceptron model and visualize how well it separates the different flower examples. However, let's add a small modification to highlight the data instances from the test dataset via small circles:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


def plot_decision_regions(X, y, classifier, test_idx=None,
                          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
    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=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    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=cl,
                    edgecolor='black')
    # highlight test examples
    if test_idx:
        # plot all examples
        X_test, y_test = X[test_idx, :], y[test_idx]

        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='test set')
```

With the slight modification that we made to the `plot_decision_regions` function, we can now specify the indices of the examples that we want to mark on the resulting plots. The code is as follows:

```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                       y=y_combined,
...                       classifier=ppn,
...                       test_idx=range(105, 150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the resulting plot, the three flower classes cannot be perfectly separated by a linear decision boundary:



Remember from our discussion in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, that the perceptron algorithm never converges on datasets that aren't perfectly linearly separable, which is why the use of the perceptron algorithm is typically not recommended in practice. In the following sections, we will look at more powerful linear classifiers that converge to a cost minimum even if the classes are not perfectly linearly separable.

Additional perceptron settings

The `Perceptron`, as well as other scikit-learn functions and classes, often has additional parameters that we omit for clarity. You can read more about those parameters using the `help` function in Python (for instance, `help(Perceptron)`) or by going through the excellent scikit-learn online documentation at `http://scikit-learn.org/stable/`.

Modeling class probabilities via logistic regression

Although the perceptron rule offers a nice and easy-going introduction to machine learning algorithms for classification, its biggest disadvantage is that it never converges if the classes are not perfectly linearly separable. The classification task in the previous section would be an example of such a scenario. The reason for this is that the weights are continuously being updated, since there is always at least one misclassified training example present in each epoch. Of course, you can change the learning rate and increase the number of epochs, but be warned that the perceptron will never converge on this dataset.

To make better use of our time, we will now take a look at another simple, yet more powerful, algorithm for linear and binary classification problems: **logistic regression**. Note that, in spite of its name, logistic regression is a model for classification, not regression.

Logistic regression and conditional probabilities

Logistic regression is a classification model that is very easy to implement and performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry. Similar to the perceptron and Adaline, the logistic regression model in this chapter is also a linear model for binary classification.

Logistic regression for multiple classes

Note that logistic regression can be readily generalized to multiclass settings, which is known as multinomial logistic regression or softmax regression. A more detailed coverage of multinomial logistic regression is outside the scope of this book, but the interested reader can find more information in my lecture notes at `https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L05_gradient-descent_slides.pdf` or `http://rasbt.github.io/mlxtend/user_guide/classifier/SoftmaxRegression/`.

Another way to use logistic regression in multiclass settings is via the OvR technique, which we discussed previously.

To explain the idea behind logistic regression as a probabilistic model for binary classification, let's first introduce the **odds**: the odds in favor of a particular event. The odds can be written as $\frac{p}{(1-p)}$ where p stands for the probability of the positive event. The term "positive event" does not necessarily mean "good," but refers to the event that we want to predict, for example, the probability that a patient has a certain disease; we can think of the positive event as class label $y = 1$. We can then further define the **logit** function, which is simply the logarithm of the odds (log-odds):

$$logit(p) = \log \frac{p}{(1-p)}$$

Note that *log* refers to the natural logarithm, as it is the common convention in computer science. The *logit* function takes input values in the range 0 to 1 and transforms them to values over the entire real-number range, which we can use to express a linear relationship between feature values and the log-odds:

$$logit\big(p(y=1|\boldsymbol{x})\big) = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \sum_{i=0}^{m} w_ix_i = \boldsymbol{w}^T\boldsymbol{x}$$

Here, $p(y=1|\boldsymbol{x})$ is the conditional probability that a particular example belongs to class 1 given its features, $\boldsymbol{x}$.

Now, we are actually interested in predicting the probability that a certain example belongs to a particular class, which is the inverse form of the logit function.

It is also called the **logistic sigmoid function**, which is sometimes simply abbreviated to **sigmoid function** due to its characteristic S-shape:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Here, z is the net input, the linear combination of weights, and the inputs (that is, the features associated with the training examples):

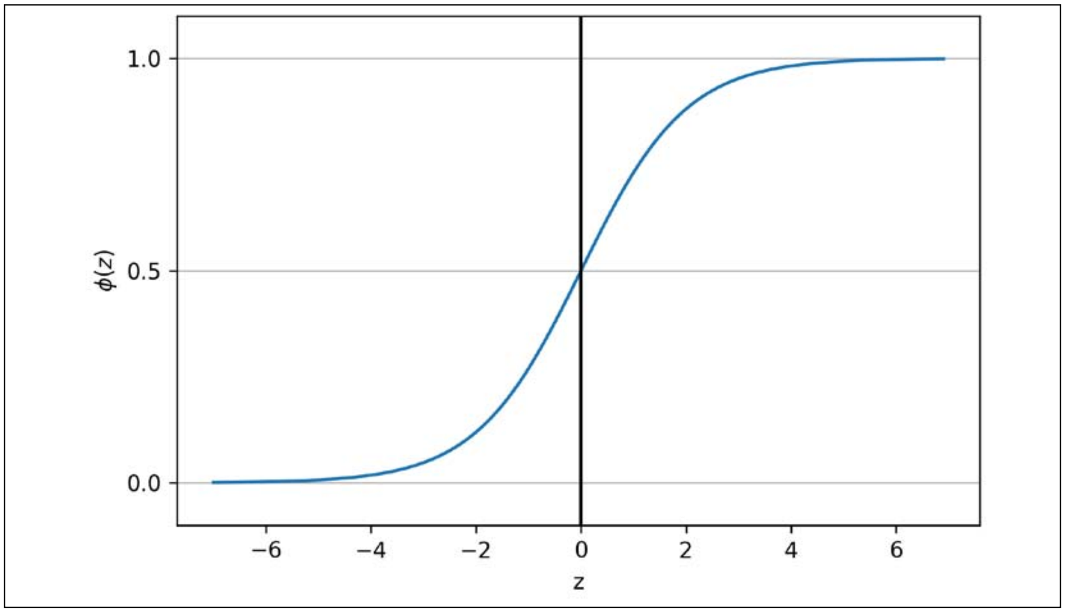
$$z = \boldsymbol{w}^T \boldsymbol{x} = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m$$

Note that similar to the convention we used in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*, w_0 refers to the bias unit and is an additional input value that we provide to x_0, which is set equal to 1.

Now, let's simply plot the sigmoid function for some values in the range –7 to 7 to see how it looks:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi (z)$')
>>> # y axis ticks and gridline
>>> plt.yticks([0.0, 0.5, 1.0])
>>> ax = plt.gca()
>>> ax.yaxis.grid(True)
>>> plt.tight_layout()
>>> plt.show()
```

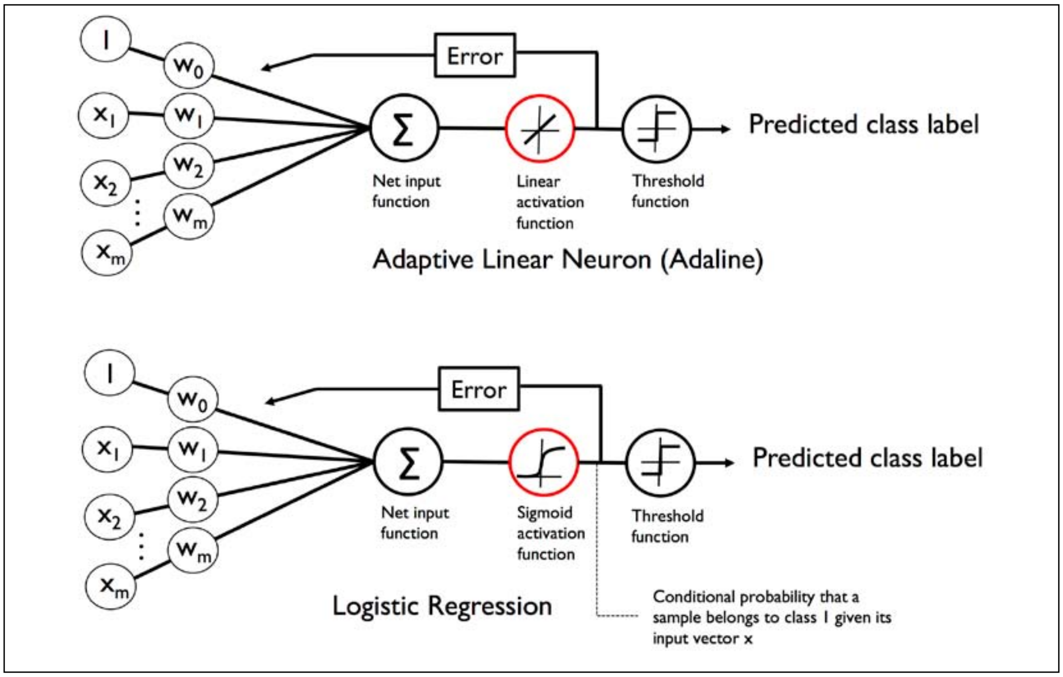
As a result of executing the previous code example, we should now see the S-shaped (sigmoidal) curve:



We can see that $\phi(z)$ approaches 1 if z goes toward infinity ($z \rightarrow \infty$) since e^{-z} becomes very small for large values of z. Similarly, $\phi(z)$ goes toward 0 for $z \rightarrow -\infty$ as a result of an increasingly large denominator. Thus, we can conclude that this sigmoid function takes real-number values as input and transforms them into values in the range [0, 1] with an intercept at $\phi(z) = 0.5$.

To build some understanding of the logistic regression model, we can relate it to *Chapter 2*. In Adaline, we used the identity function, $\phi(z) = z$, as the activation function. In logistic regression, this activation function simply becomes the sigmoid function that we defined earlier.

The difference between Adaline and logistic regression is illustrated in the following figure:


1
w_0
x_1
w_1
x_2
w_2
x_m
w_m
Error
Net input function
Linear activation function
Threshold function
Predicted class label
Adaptive Linear Neuron (Adaline)
Sigmoid activation function
Logistic Regression
Conditional probability that a sample belongs to class 1 given its input vector x

The output of the sigmoid function is then interpreted as the probability of a particular example belonging to class 1, $\phi(z) = P(y = 1|\boldsymbol{x}; \boldsymbol{w})$, given its features, $\boldsymbol{x}$, parameterized by the weights, $\boldsymbol{w}$. For example, if we compute $\phi(z) = 0.8$ for a particular flower example, it means that the chance that this example is an `Iris-versicolor` flower is 80 percent. Therefore, the probability that this flower is an `Iris-setosa` flower can be calculated as $P(y = 0|\boldsymbol{x}; \boldsymbol{w}) = 1 - P(y = 1|\boldsymbol{x}; \boldsymbol{w}) = 0.2$ or 20 percent. The predicted probability can then simply be converted into a binary outcome via a threshold function:

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

If we look at the preceding plot of the sigmoid function, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

In fact, there are many applications where we are not only interested in the predicted class labels, but where the estimation of the class-membership probability is particularly useful (the output of the sigmoid function prior to applying the threshold function). Logistic regression is used in weather forecasting, for example, not only to predict whether it will rain on a particular day but also to report the chance of rain. Similarly, logistic regression can be used to predict the chance that a patient has a particular disease given certain symptoms, which is why logistic regression enjoys great popularity in the field of medicine.

Learning the weights of the logistic cost function

You have learned how we can use the logistic regression model to predict probabilities and class labels; now, let's briefly talk about how we fit the parameters of the model, for instance the weights, ***w***. In the previous chapter, we defined the sum-squared-error cost function as follows:

$$J(\boldsymbol{w}) = \sum_i \frac{1}{2} \left(\phi\left(z^{(i)}\right) - y^{(i)}\right)^2$$

We minimized this function in order to learn the weights, ***w***, for our Adaline classification model. To explain how we can derive the cost function for logistic regression, let's first define the likelihood, *L*, that we want to maximize when we build a logistic regression model, assuming that the individual examples in our dataset are independent of one another. The formula is as follows:

$$L(\boldsymbol{w}) = P(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{w}) = \prod_{i=1}^{n} P\left(y^{(i)} \mid x^{(i)}; \boldsymbol{w}\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the **log-likelihood** function:

$$l(\boldsymbol{w}) = \log L(\boldsymbol{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)\right]$$

Firstly, applying the log function reduces the potential for numerical underflow, which can occur if the likelihoods are very small. Secondly, we can convert the product of factors into a summation of factors, which makes it easier to obtain the derivative of this function via the addition trick, as you may remember from calculus.

Now, we could use an optimization algorithm such as gradient ascent to maximize this log-likelihood function. Alternatively, let's rewrite the log-likelihood as a cost function, J, that can be minimized using gradient descent as in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*:

$$\mathrm{J}(\mathbf{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)\right]$$

To get a better grasp of this cost function, let's take a look at the cost that we calculate for one single training example:

$$J(\phi(z), y; \boldsymbol{w}) = -y \log(\phi(z)) - (1-y)\log(1-\phi(z))$$

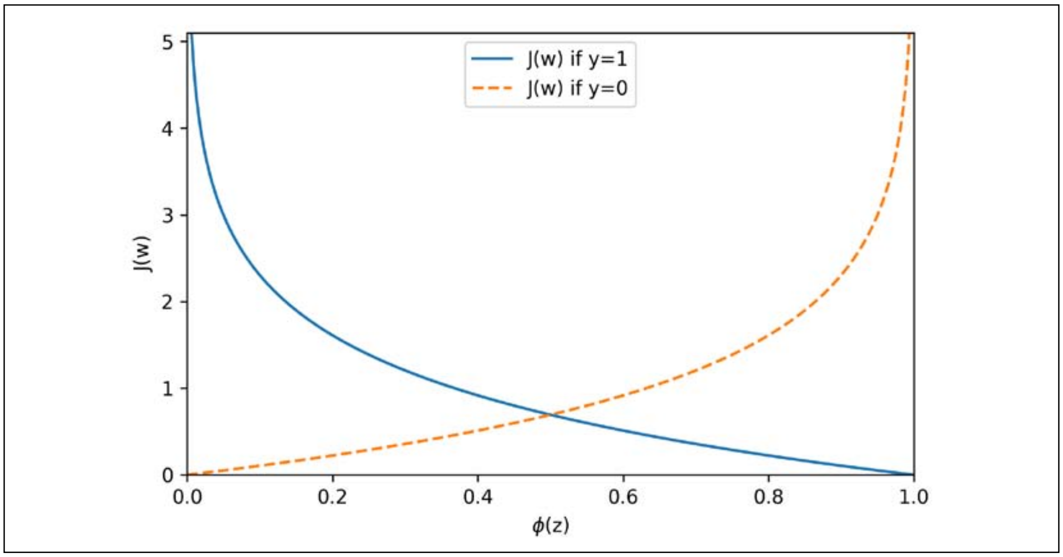
Looking at the equation, we can see that the first term becomes zero if $y = 0$, and the second term becomes zero if $y = 1$:

$$J(\phi(z), y; \boldsymbol{w}) = \begin{cases} -\log\big(\phi(z)\big) & \text{if } y = 1 \\ -\log\big(1-\phi(z)\big) & \text{if } y = 0 \end{cases}$$

Let's write a short code snippet to create a plot that illustrates the cost of classifying a single training example for different values of $\phi(z)$:

```
>>> def cost_1(z):
...     return - np.log(sigmoid(z))
>>> def cost_0(z):
...     return - np.log(1 - sigmoid(z))
>>> z = np.arange(-10, 10, 0.1)
>>> phi_z = sigmoid(z)
>>> c1 = [cost_1(x) for x in z]
>>> plt.plot(phi_z, c1, label='J(w) if y=1')
>>> c0 = [cost_0(x) for x in z]
>>> plt.plot(phi_z, c0, linestyle='--', label='J(w) if y=0')
>>> plt.ylim(0.0, 5.1)
>>> plt.xlim([0, 1])
>>> plt.xlabel('$\phi$(z)')
>>> plt.ylabel('J(w)')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

The resulting plot shows the sigmoid activation on the x-axis in the range 0 to 1 (the inputs to the sigmoid function were z values in the range –10 to 10) and the associated logistic cost on the y-axis:



We can see that the cost approaches 0 (continuous line) if we correctly predict that an example belongs to class 1. Similarly, we can see on the y-axis that the cost also approaches 0 if we correctly predict $y = 0$ (dashed line). However, if the prediction is wrong, the cost goes toward infinity. The main point is that we penalize wrong predictions with an increasingly larger cost.

Converting an Adaline implementation into an algorithm for logistic regression

If we were to implement logistic regression ourselves, we could simply substitute the cost function, J, in our Adaline implementation from *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, with the new cost function:

$$J(\boldsymbol{w}) = -\sum_i y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)$$

We use this to compute the cost of classifying all training examples per epoch. Also, we need to swap the linear activation function with the sigmoid activation and change the threshold function to return class labels 0 and 1, instead of –1 and 1. If we make those changes to the Adaline code, we will end up with a working logistic regression implementation, as shown here:

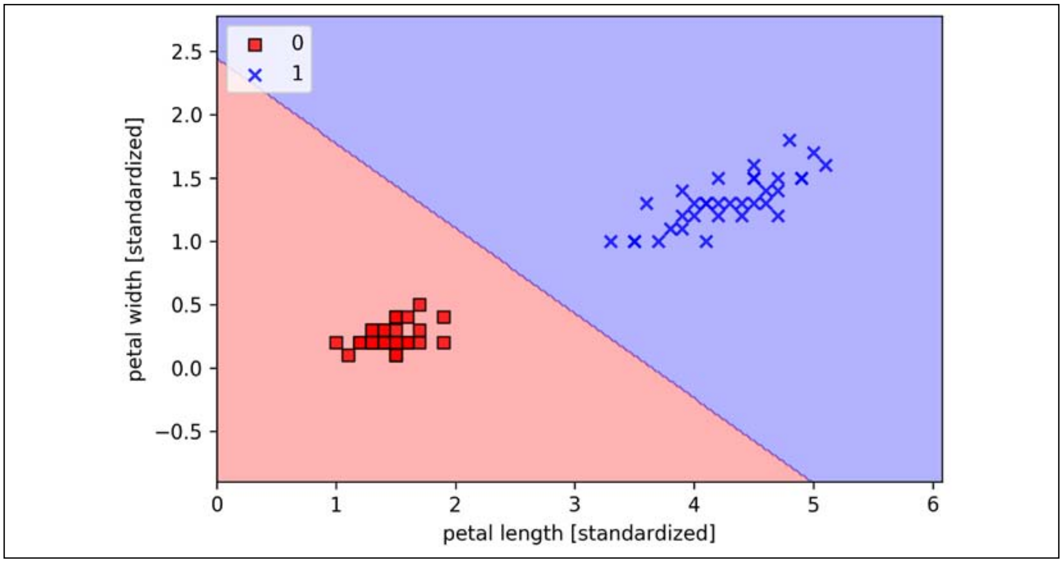
```
class LogisticRegressionGD(object):
    """Logistic Regression Classifier using gradient descent.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    random_state : int
        Random number generator seed for random weight
        initialization.


    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Logistic cost function value in each epoch.

    """
    def __init__(self, eta=0.05, n_iter=100, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
            Training vectors, where n_examples is the number of
            examples and n_features is the number of features.
        y : array-like, shape = [n_examples]
            Target values.
        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()

            # note that we compute the logistic 'cost' now
            # instead of the sum of squared errors cost
            cost = (-y.dot(np.log(output)) -
                        ((1 - y).dot(np.log(1 - output))))
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, z):
        """Compute logistic sigmoid activation"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, 0)
        # equivalent to:
        # return np.where(self.activation(self.net_input(X))
        #                 >= 0.5, 1, 0)
```

When we fit a logistic regression model, we have to keep in mind that it only works for binary classification tasks.

So, let's consider only `Iris-setosa` and `Iris-versicolor` flowers (classes `0` and `1`) and check that our implementation of logistic regression works:

```
>>> X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
>>> lrgd = LogisticRegressionGD(eta=0.05,
...                              n_iter=1000,
...                              random_state=1)
>>> lrgd.fit(X_train_01_subset,
...          y_train_01_subset)
>>> plot_decision_regions(X=X_train_01_subset,
...                       y=y_train_01_subset,
...                       classifier=lrgd)
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

The resulting decision region plot looks as follows:



The gradient descent learning algorithm for logistic regression

Using calculus, we can show that the weight update in logistic regression via gradient descent is equal to the equation that we used in Adaline in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. However, please note that the following derivation of the gradient descent learning rule is intended for readers who are interested in the mathematical concepts behind the gradient descent learning rule for logistic regression. It is not essential for following the rest of this chapter.

Let's start by calculating the partial derivative of the log-likelihood function with respect to the *j*th weight:

$$\frac{\partial}{\partial w_j} l(\boldsymbol{w}) = \left(y \frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial}{\partial w_j}\phi(z)$$

Before we continue, let's also calculate the partial derivative of the sigmoid function:

$$\frac{\partial}{\partial z}\phi(z) = \frac{\partial}{\partial z}\frac{1}{1+e^{-z}} = \frac{1}{(1+e^{-z})^2}\, e^{-z} = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) = \phi(z)\big(1-\phi(z)\big)$$

Now, we can resubstitute $\frac{\partial}{\partial z}\phi(z) = \phi(z)\big(1-\phi(z)\big)$ in our first equation to obtain the following:

$$\begin{aligned}
\left(y \frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial}{\partial w_j}\phi(z) &= \left(y \frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\phi(z)\big(1-\phi(z)\big)\frac{\partial}{\partial w_j} z \\
&= \Big(y\big(1-\phi(z)\big) - (1-y)\phi(z)\Big)x_j \\
&= \big(y - \phi(z)\big)x_j
\end{aligned}$$

Remember that the goal is to find the weights that maximize the log-likelihood so that we perform the update for each weight as follows:

$$w_j := w_j + \eta \sum_{i=1}^{n}\left(y^{(i)} - \phi\big(z^{(i)}\big)\right) x_j^{(i)}$$

Since we update all weights simultaneously, we can write the general update rule as follows:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}$$

We define $\Delta\boldsymbol{w}$ as follows:

$$\Delta\boldsymbol{w} = \eta\nabla l(\boldsymbol{w})$$

Since maximizing the log-likelihood is equal to minimizing the cost function, *J*, that we defined earlier, we can write the gradient descent update rule as follows:

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_{i=1}^{n}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j{}^{(i)}$$

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \quad \Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w})$$

This is equal to the gradient descent rule for Adaline in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*.

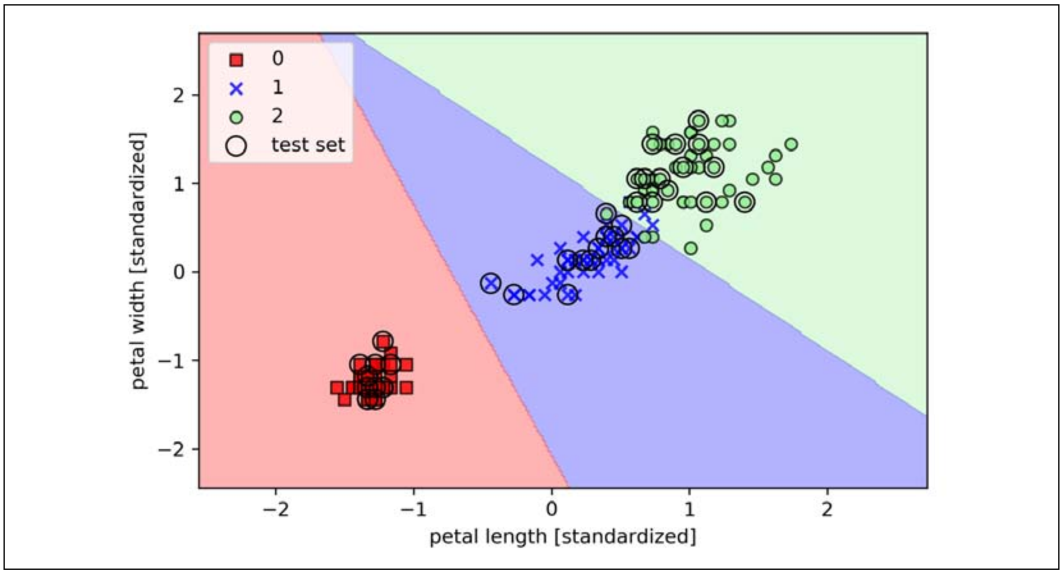
Training a logistic regression model with scikit-learn

We just went through useful coding and math exercises in the previous subsection, which helped to illustrate the conceptual differences between Adaline and logistic regression. Now, let's learn how to use scikit-learn's more optimized implementation of logistic regression, which also supports multiclass settings off the shelf. Note that in recent versions of scikit-learn, the technique used for multiclass classification, multinomial, or OvR, is chosen automatically. In the following code example, we will use the `sklearn.linear_model.LogisticRegression` class as well as the familiar `fit` method to train the model on all three classes in the standardized flower training dataset. Also, we set `multi_class='ovr'` for illustration purposes. As an exercise for the reader, you may want to compare the results with `multi_class='multinomial'`. Note that the `multinomial` setting is usually recommended in practice for mutually exclusive classes, such as those found in the Iris dataset. Here, "mutually exclusive" means that each training example can only belong to a single class (in contrast to multilabel classification, where a training example can be a member of multiple classes).

Now, let's have a look at the code example:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, random_state=1,
...                          solver='lbfgs', multi_class='ovr')
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=lr,
...                       test_idx=range(105, 150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After fitting the model on the training data, we plotted the decision regions, training examples, and test examples, as shown in the following plot:



Note that there exist many different optimization algorithms for solving optimization problems. For minimizing convex loss functions, such as the logistic regression loss, it is recommended to use more advanced approaches than regular stochastic gradient descent (SGD). In fact, scikit-learn implements a whole range of such optimization algorithms, which can be specified via the solver parameter, namely, `'newton-cg'`, `'lbfgs'`, `'liblinear'`, `'sag'`, and `'saga'`.

While the logistic regression loss is convex, most optimization algorithms should converge to the global loss minimum with ease. However, there are certain advantages of using one algorithm over the other. For instance, in the current version (v 0.21), scikit-learn uses `'liblinear'` as a default, which cannot handle the multinomial loss and is limited to the OvR scheme for multi-class classification. However, in future versions of scikit-learn (that is, v 0.22), the default solver will be changed to `'lbfgs'`, which stands for the **limited-memory Broyden–Fletcher–Goldfarb–Shanno** (**BFGS**) algorithm (`https://en.wikipedia.org/wiki/Limited-memory_BFGS`) and is more flexible in this regard. To adopt this new default choice, we will specify `solver='lbfgs'` explicitly when using logistic regression throughout this book.

Looking at the preceding code that we used to train the `LogisticRegression` model, you might now be wondering, "What is this mysterious parameter C?" We will discuss this parameter in the next subsection, where we will introduce the concepts of overfitting and regularization. However, before we move on to those topics, let's finish our discussion of class-membership probabilities.

The probability that training examples belong to a certain class can be computed using the `predict_proba` method. For example, we can predict the probabilities of the first three examples in the test dataset as follows:

```
>>> lr.predict_proba(X_test_std[:3, :])
```

This code snippet returns the following array:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

The first row corresponds to the class-membership probabilities of the first flower, the second row corresponds to the class-membership probabilities of the second flower, and so forth. Notice that the columns all sum up to one, as expected. (You can confirm this by executing `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`.)

The highest value in the first row is approximately 0.85, which means that the first example belongs to class three (`Iris-virginica`) with a predicted probability of 85 percent. So, as you may have already noticed, we can get the predicted class labels by identifying the largest column in each row, for example, using NumPy's `argmax` function:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

The returned class indices are shown here (they correspond to `Iris-virginica`, `Iris-setosa`, and `Iris-setosa`):

```
array([2, 0, 0])
```

In the preceding code example, we computed the conditional probabilities and converted these into class labels manually by using NumPy's `argmax` function. In practice, the more convenient way of obtaining class labels when using scikit-learn is to call the `predict` method directly:

```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

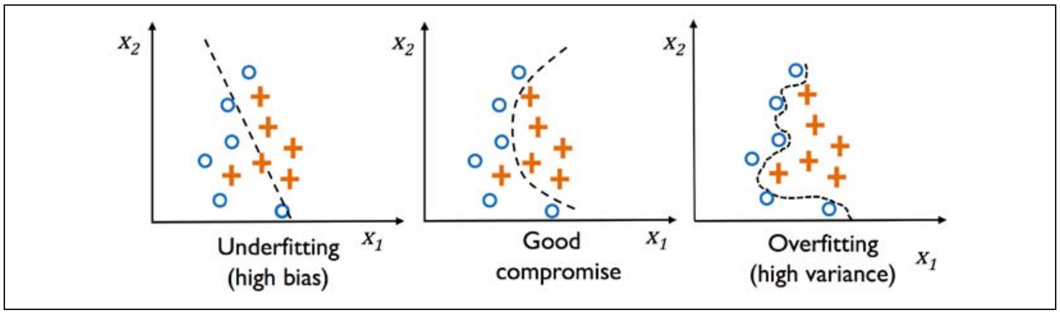
Lastly, a word of caution if you want to predict the class label of a single flower example: scikit-learn expects a two-dimensional array as data input; thus, we have to convert a single row slice into such a format first. One way to convert a single row entry into a two-dimensional data array is to use NumPy's `reshape` method to add a new dimension, as demonstrated here:

```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

Tackling overfitting via regularization

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, we also say that the model has a high variance, which can be caused by having too many parameters, leading to a model that is too complex given the underlying data. Similarly, our model can also suffer from **underfitting** (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.

Although we have only encountered linear models for classification so far, the problems of overfitting and underfitting can be best illustrated by comparing a linear decision boundary to more complex, nonlinear decision boundaries, as shown in the following figure:



The bias-variance tradeoff

Often, researchers use the terms "bias" and "variance" or "bias-variance tradeoff" to describe the performance of a model—that is, you may stumble upon talks, books, or articles where people say that a model has a "high variance" or "high bias." So, what does that mean? In general, we might say that "high variance" is proportional to overfitting and "high bias" is proportional to underfitting.

In the context of machine learning models, **variance** measures the consistency (or variability) of the model prediction for classifying a particular example if we retrain the model multiple times, for example, on different subsets of the training dataset. We can say that the model is sensitive to the randomness in the training data. In contrast, **bias** measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.

If you are interested in the technical specification and derivation of the "bias" and "variance" terms, I've written about it in my lecture notes here: `https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/08_eval-intro_notes.pdf`.

One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization. Regularization is a very useful method for handling collinearity (high correlation among features), filtering out noise from data, and eventually preventing overfitting.

The concept behind regularization is to introduce additional information (bias) to penalize extreme parameter (weight) values. The most common form of regularization is so-called **L2 regularization** (sometimes also called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

Here, λ is the so-called **regularization parameter**.

Regularization and feature normalization

Regularization is another reason why feature scaling such as standardization is important. For regularization to work properly, we need to ensure that all our features are on comparable scales.

The cost function for logistic regression can be regularized by adding a simple regularization term, which will shrink the weights during model training:

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right] + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

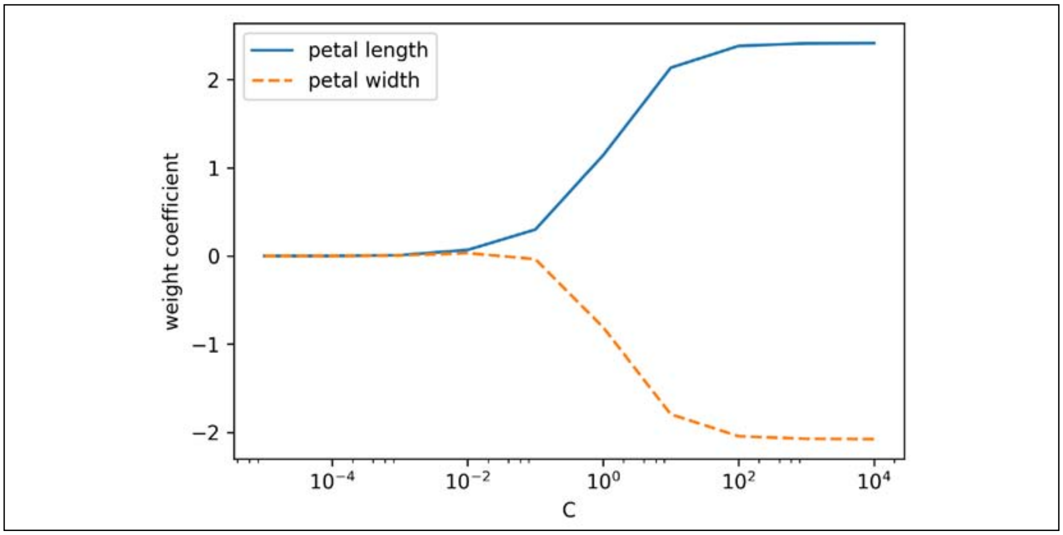
Via the regularization parameter, λ, we can then control how well we fit the training data, while keeping the weights small. By increasing the value of λ, we increase the regularization strength.

The parameter, `C`, that is implemented for the `LogisticRegression` class in scikit-learn comes from a convention in support vector machines, which will be the topic of the next section. The term `C` is directly related to the regularization parameter, λ, which is its inverse. Consequently, decreasing the value of the inverse regularization parameter, `C`, means that we are increasing the regularization strength, which we can visualize by plotting the L2 regularization path for the two weight coefficients:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10.**c, random_state=1,
...                             solver='lbfgs', multi_class='ovr')
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10.**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...          label='petal length')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...          label='petal width')
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

By executing the preceding code, we fitted 10 logistic regression models with different values for the inverse-regularization parameter, `C`. For the purposes of illustration, we only collected the weight coefficients of class `1` (here, the second class in the dataset: `Iris-versicolor`) versus all classifiers—remember that we are using the OvR technique for multiclass classification.

As we can see in the resulting plot, the weight coefficients shrink if we decrease parameter `C`, that is, if we increase the regularization strength:

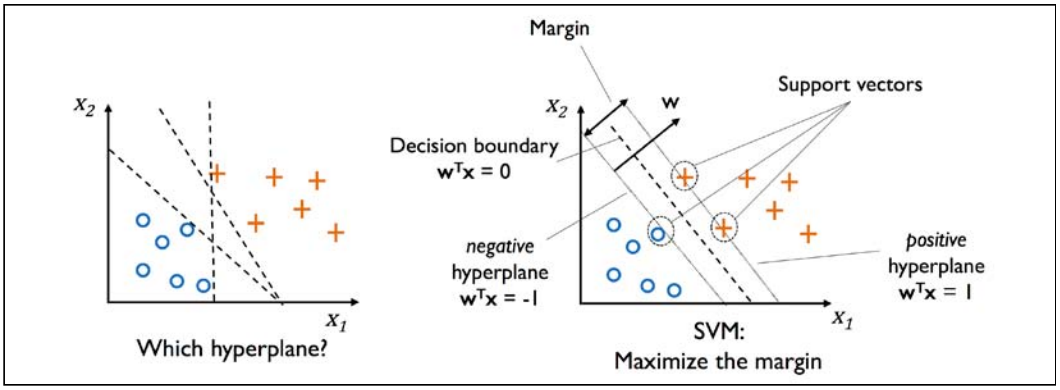


An additional resource on logistic regression

Since an in-depth coverage of the individual classification algorithms exceeds the scope of this book, *Logistic Regression: From Introductory to Advanced Concepts and Applications*, *Dr. Scott Menard*, *Sage Publications*, *2009*, is recommended to readers who want to learn more about logistic regression.

Maximum margin classification with support vector machines

Another powerful and widely used learning algorithm is the **support vector machine** (**SVM**), which can be considered an extension of the perceptron. Using the perceptron algorithm, we minimized misclassification errors. However, in SVMs our optimization objective is to maximize the margin. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called **support vectors**. This is illustrated in the following figure:



Maximum margin intuition

The rationale behind having decision boundaries with large margins is that they tend to have a lower generalization error, whereas models with small margins are more prone to overfitting. To get an idea of the margin maximization, let's take a closer look at those positive and negative hyperplanes that are parallel to the decision boundary, which can be expressed as follows:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{pos} = 1 \quad (1)$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1 \quad (2)$$

If we subtract those two linear equations (1) and (2) from each other, we get:

$$\Rightarrow \boldsymbol{w}^T\left(\boldsymbol{x}_{\boldsymbol{pos}} - \boldsymbol{x}_{\boldsymbol{neg}}\right) = 2$$

We can normalize this equation by the length of the vector ***w***, which is defined as follows:

$$\|\boldsymbol{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

So, we arrive at the following equation:

$$\frac{\boldsymbol{w}^{\boldsymbol{T}}\left(\boldsymbol{x}_{\boldsymbol{pos}} - \boldsymbol{x}_{\boldsymbol{neg}}\right)}{\|\boldsymbol{w}\|} = \frac{2}{\|\boldsymbol{w}\|}$$

The left side of the preceding equation can then be interpreted as the distance between the positive and negative hyperplane, which is the so-called **margin** that we want to maximize.

Now, the objective function of the SVM becomes the maximization of this margin by maximizing $\frac{2}{\|\boldsymbol{w}\|}$ under the constraint that the examples are classified correctly, which can be written as:

$$w_{\mathbf{0}} + \boldsymbol{w}^T\boldsymbol{x}^{(i)} \geq 1 \quad if\ y^{(i)} = 1$$

$$w_{\mathbf{0}} + \boldsymbol{w}^T\boldsymbol{x}^{(i)} \leq -1 \quad if\ y^{(i)} = -1$$

$$\text{for } i = 1 \ldots N$$

Here, *N* is the number of examples in our dataset.

These two equations basically say that all negative-class examples should fall on one side of the negative hyperplane, whereas all the positive-class examples should fall behind the positive hyperplane, which can also be written more compactly as follows:

$$y^{(i)}\left(w_{\mathbf{0}} + \boldsymbol{w}^T \boldsymbol{x}^{(i)}\right) \geq 1 \qquad \forall_i$$

In practice, though, it is easier to minimize the reciprocal term, $\frac{1}{2}\|\boldsymbol{w}\|^2$, which can be solved by quadratic programming. However, a detailed discussion about quadratic programming is beyond the scope of this book. You can learn more about SVMs in *The Nature of Statistical Learning Theory*, *Springer Science+Business Media*, *Vladimir Vapnik*, 2000, or read Chris J.C. Burges' excellent explanation in *A Tutorial on Support Vector Machines for Pattern Recognition* (*Data Mining and Knowledge Discovery*, *2(2)*: 121-167, *1998*).

Dealing with a nonlinearly separable case using slack variables

Although we don't want to dive much deeper into the more involved mathematical concepts behind the maximum-margin classification, let's briefly mention the slack variable, ξ, which was introduced by Vladimir Vapnik in 1995 and led to the so-called **soft-margin classification**. The motivation for introducing the slack variable was that the linear constraints need to be relaxed for nonlinearly separable data to allow the convergence of the optimization in the presence of misclassifications, under appropriate cost penalization.

The positive-valued slack variable is simply added to the linear constraints:

$$w_{\mathbf{0}} + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 - \xi^{(i)} \quad if\ y^{(i)} = 1$$

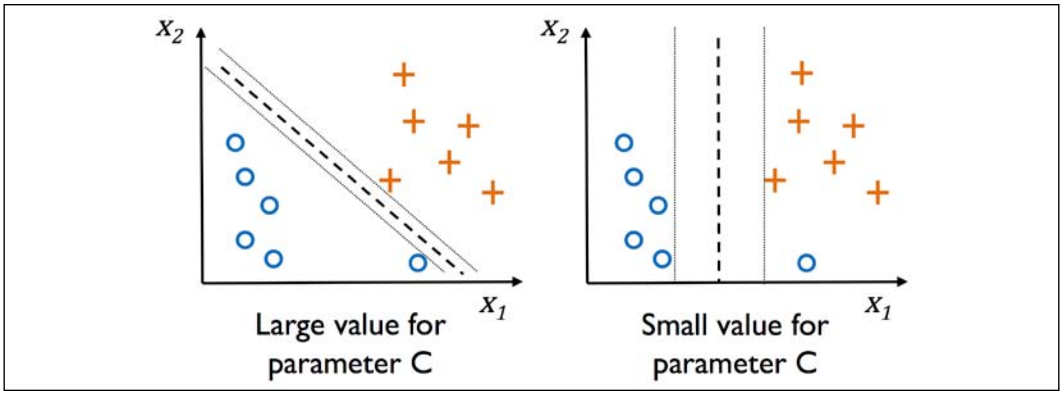
$$w_{\mathbf{0}} + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 + \xi^{(i)} \quad if\ y^{(i)} = -1$$

for $i \; = 1 \ldots N$

Here, *N* is the number of examples in our dataset. So, the new objective to be minimized (subject to the constraints) becomes:

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

Via the variable, `C`, we can then control the penalty for misclassification. Large values of `C` correspond to large error penalties, whereas we are less strict about misclassification errors if we choose smaller values for `C`. We can then use the `C` parameter to control the width of the margin and therefore tune the bias-variance tradeoff, as illustrated in the following figure:

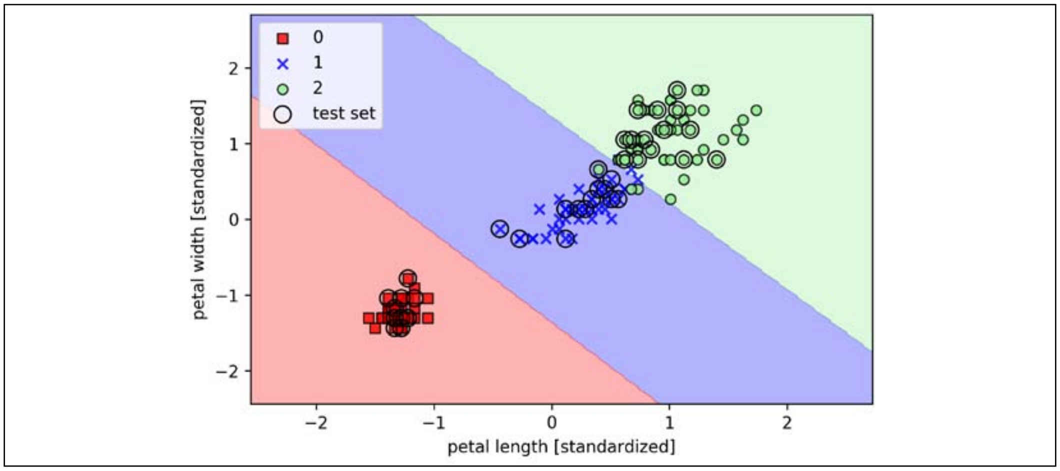


This concept is related to regularization, which we discussed in the previous section in the context of regularized regression, where decreasing the value of `C` increases the bias and lowers the variance of the model.

Now that we have learned the basic concepts behind a linear SVM, let's train an SVM model to classify the different flowers in our Iris dataset:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=1)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=svm,
...                       test_idx=range(105, 150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

The three decision regions of the SVM, visualized after training the classifier on the Iris dataset by executing the preceding code example, are shown in the following plot:



Logistic regression versus SVMs

In practical classification tasks, linear logistic regression and linear SVMs often yield very similar results. Logistic regression tries to maximize the conditional likelihoods of the training data, which makes it more prone to outliers than SVMs, which mostly care about the points that are closest to the decision boundary (support vectors). On the other hand, logistic regression has the advantage that it is a simpler model and can be implemented more easily. Furthermore, logistic regression models can be easily updated, which is attractive when working with streaming data.

Alternative implementations in scikit-learn

The scikit-learn library's `LogisticRegression` class, which we used in the previous sections, makes use of the LIBLINEAR library, which is a highly optimized C/C++ library developed at the National Taiwan University (`http://www.csie.ntu.edu.tw/~cjlin/liblinear/`).

Similarly, the `SVC` class that we used to train an SVM makes use of LIBSVM, which is an equivalent C/C++ library specialized for SVMs (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`).

The advantage of using LIBLINEAR and LIBSVM over native Python implementations is that they allow the extremely quick training of large amounts of linear classifiers. However, sometimes our datasets are too large to fit into computer memory. Thus, scikit-learn also offers alternative implementations via the `SGDClassifier` class, which also supports online learning via the `partial_fit` method. The concept behind the `SGDClassifier` class is similar to the stochastic gradient algorithm that we implemented in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, for Adaline. We could initialize the SGD version of the perceptron, logistic regression, and an SVM with default parameters, as follows:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

Solving nonlinear problems using a kernel SVM

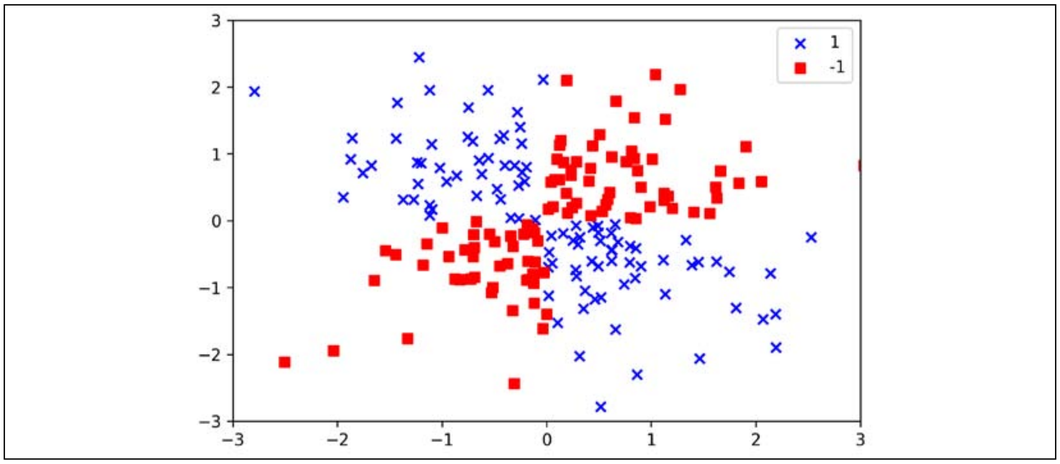
Another reason why SVMs enjoy high popularity among machine learning practitioners is that they can be easily **kernelized** to solve nonlinear classification problems. Before we discuss the main concept behind the so-called **kernel SVM**, the most common variant of SVMs, let's first create a synthetic dataset to see what such a nonlinear classification problem may look like.

Kernel methods for linearly inseparable data

Using the following code, we will create a simple dataset that has the form of an XOR gate using the `logical_or` function from NumPy, where 100 examples will be assigned the class label `1`, and 100 examples will be assigned the class label `-1`:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> np.random.seed(1)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0,
...                        X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)
>>> plt.scatter(X_xor[y_xor == 1, 0],
...                X_xor[y_xor == 1, 1],
...                c='b', marker='x',
...                label='1')
>>> plt.scatter(X_xor[y_xor == -1, 0],
...                X_xor[y_xor == -1, 1],
...                c='r',
...                marker='s',
...                label='-1')
>>> plt.xlim([-3, 3])
>>> plt.ylim([-3, 3])
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we will have an XOR dataset with random noise, as shown in the following plot:

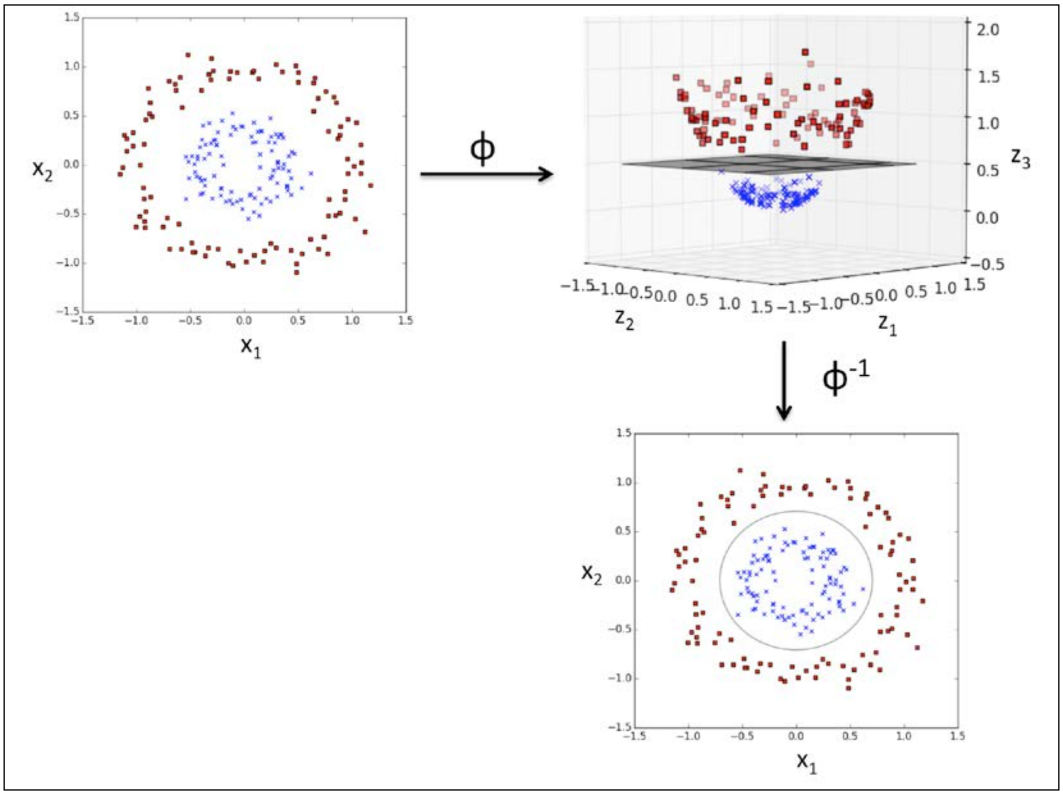


Obviously, we would not be able to separate the examples from the positive and negative class very well using a linear hyperplane as a decision boundary via the linear logistic regression or linear SVM model that we discussed in earlier sections.

The basic idea behind **kernel methods** to deal with such linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher-dimensional space via a mapping function, ϕ, where the data becomes linearly separable. As shown in the following plot, we can transform a two-dimensional dataset into a new three-dimensional feature space, where the classes become separable via the following projection:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

This allows us to separate the two classes shown in the plot via a linear hyperplane that becomes a nonlinear decision boundary if we project it back onto the original feature space:



Using the kernel trick to find separating hyperplanes in a high-dimensional space

To solve a nonlinear problem using an SVM, we would transform the training data into a higher-dimensional feature space via a mapping function, ϕ, and train a linear SVM model to classify the data in this new feature space. Then, we could use the same mapping function, ϕ, to transform new, unseen data to classify it using the linear SVM model.

However, one problem with this mapping approach is that the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data. This is where the so-called **kernel trick** comes into play.

Although we did not go into much detail about how to solve the quadratic programming task to train an SVM, in practice, we just need to replace the dot product $\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$ by $\phi(\boldsymbol{x}^{(i)})^T\phi(\boldsymbol{x}^{(j)})$. In order to save the expensive step of calculating this dot product between two points explicitly, we define a so-called **kernel function**:

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

One of the most widely used kernels is the **radial basis function** (**RBF**) kernel, which can simply be called the **Gaussian kernel**:

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

This is often simplified to:

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

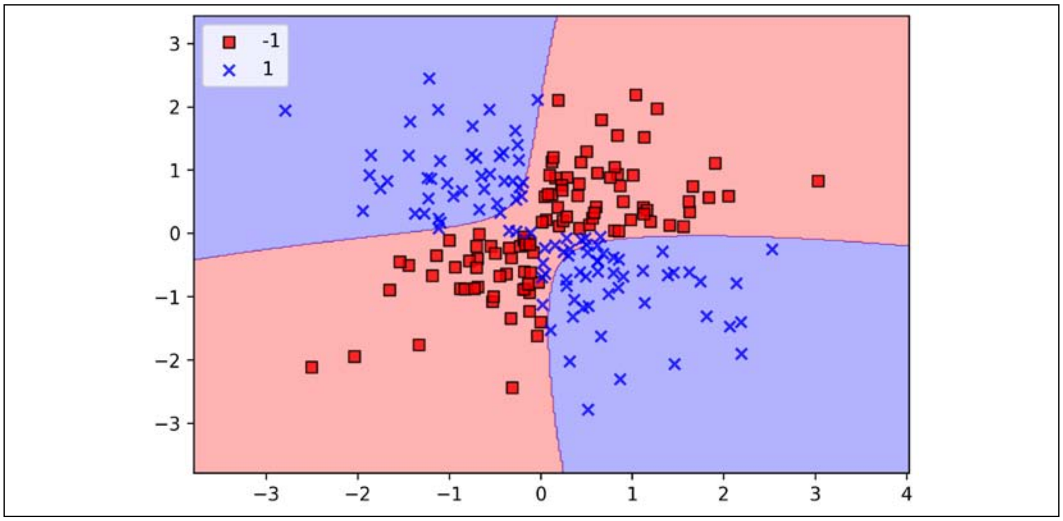
Here, $\gamma = \frac{1}{2\sigma^2}$ is a free parameter to be optimized.

Roughly speaking, the term "kernel" can be interpreted as a **similarity function** between a pair of examples. The minus sign inverts the distance measure into a similarity score, and, due to the exponential term, the resulting similarity score will fall into a range between 1 (for exactly similar examples) and 0 (for very dissimilar examples).

Now that we have covered the big picture behind the kernel trick, let's see if we can train a kernel SVM that is able to draw a nonlinear decision boundary that separates the XOR data well. Here, we simply use the `SVC` class from scikit-learn that we imported earlier and replace the `kernel='linear'` parameter with `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

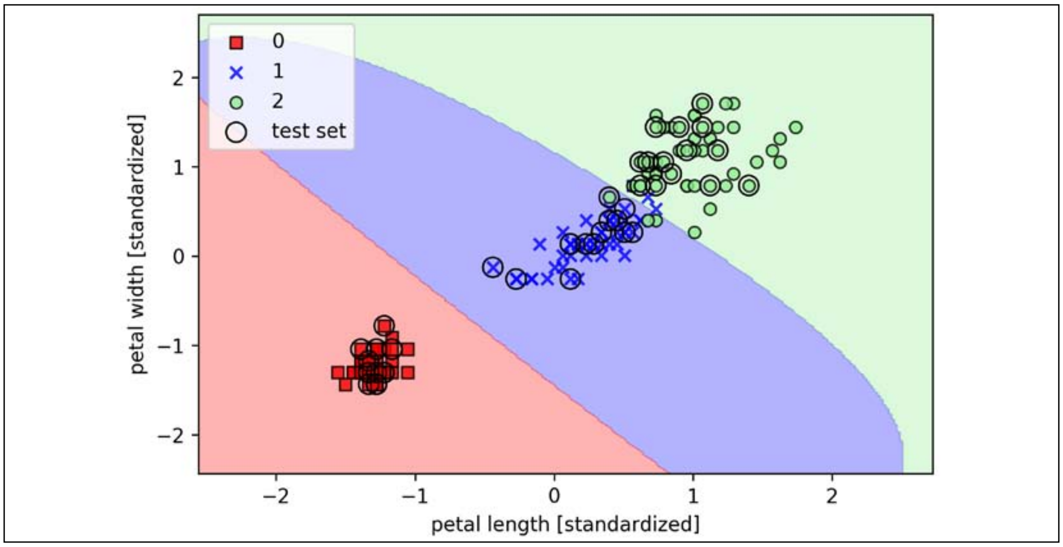
As we can see in the resulting plot, the kernel SVM separates the XOR data relatively well:



The γ parameter, which we set to `gamma=0.1`, can be understood as a cut-off parameter for the Gaussian sphere. If we increase the value for γ, we increase the influence or reach of the training examples, which leads to a tighter and bumpier decision boundary. To get a better understanding of γ, let's apply an RBF kernel SVM to our Iris flower dataset:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

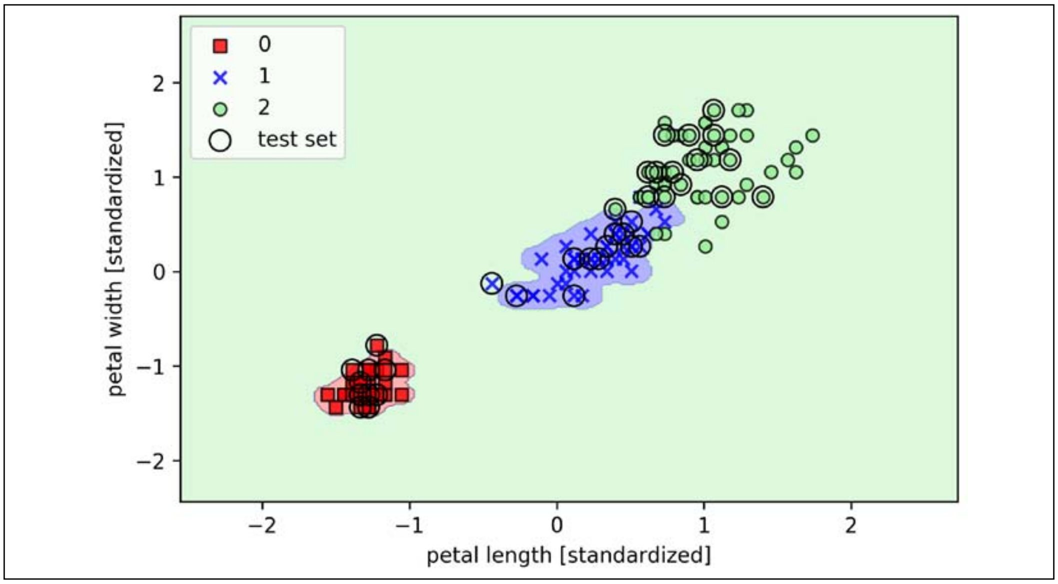
Since we chose a relatively small value for γ, the resulting decision boundary of the RBF kernel SVM model will be relatively soft, as shown in the following plot:



Now, let's increase the value of γ and observe the effect on the decision boundary:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

In the resulting plot, we can now see that the decision boundary around the classes `0` and `1` is much tighter using a relatively large value of γ:

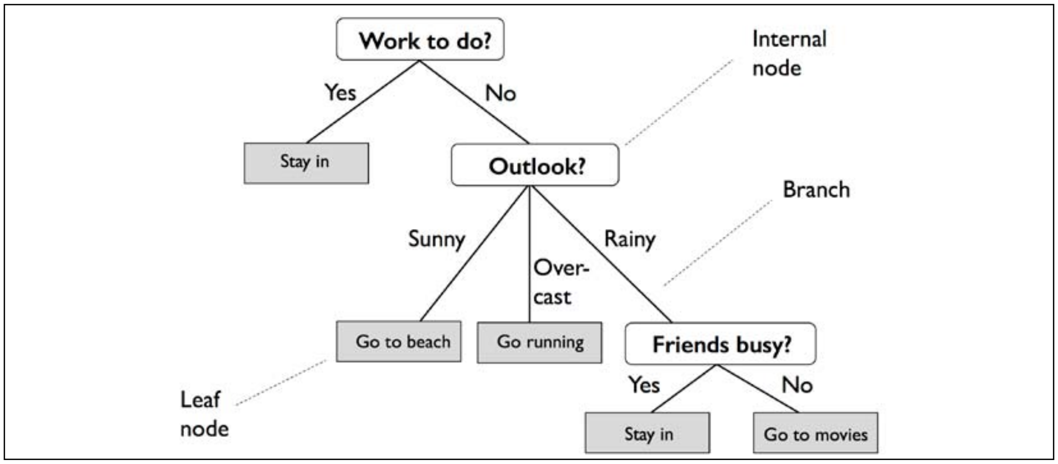


Although the model fits the training dataset very well, such a classifier will likely have a high generalization error on unseen data. This illustrates that the γ parameter also plays an important role in controlling overfitting or variance when the algorithm is too sensitive to fluctuations in the training dataset.

Decision tree learning

Decision tree classifiers are attractive models if we care about interpretability. As the name "decision tree" suggests, we can think of this model as breaking down our data by making a decision based on asking a series of questions.

Let's consider the following example in which we use a decision tree to decide upon an activity on a particular day:



Based on the features in our training dataset, the decision tree model learns a series of questions to infer the class labels of the examples. Although the preceding figure illustrates the concept of a decision tree based on categorical variables, the same concept applies if our features are real numbers, like in the Iris dataset. For example, we could simply define a cut-off value along the **sepal width** feature axis and ask a binary question: "Is the sepal width ≥ 2.8 cm?"

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain** (**IG**), which will be explained in more detail in the following section. In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the training examples at each node all belong to the same class. In practice, this can result in a very deep tree with many nodes, which can easily lead to overfitting. Thus, we typically want to **prune** the tree by setting a limit for the maximal depth of the tree.

Maximizing IG – getting the most bang for your buck

In order to split the nodes at the most informative features, we need to define an objective function that we want to optimize via the tree learning algorithm. Here, our objective function is to maximize the IG at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Here, *f* is the feature to perform the split; D_p and D_j are the dataset of the parent and *j*th child node; *I* is our **impurity** measure; N_p is the total number of training examples at the parent node; and N_j is the number of examples in the *j*th child node. As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurities of the child nodes, the larger the information gain. However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that each parent node is split into two child nodes, D_{left} and D_{right}:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

The three impurity measures or splitting criteria that are commonly used in binary decision trees are **Gini impurity** (I_G), **entropy** (I_H), and the **classification error** (I_E). Let's start with the definition of entropy for all **non-empty** classes ($p(i|t) \neq 0$):

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

Here, $p(i|t)$ is the proportion of the examples that belong to class *i* for a particular node, *t*. The entropy is therefore 0 if all examples at a node belong to the same class, and the entropy is maximal if we have a uniform class distribution. For example, in a binary class setting, the entropy is 0 if $p(i = 1|t) = 1$ or $p(i = 0|t) = 0$. If the classes are distributed uniformly with $p(i = 1|t) = 0.5$ and $p(i = 0|t) = 0.5$, the entropy is 1. Therefore, we can say that the entropy criterion attempts to maximize the mutual information in the tree.

The Gini impurity can be understood as a criterion to minimize the probability of misclassification:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)\big(1 - p(i|t)\big) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Similar to entropy, the Gini impurity is maximal if the classes are perfectly mixed, for example, in a binary class setting ($c = 2$):

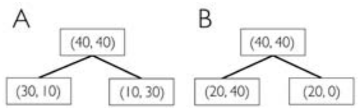
$$I_G(t) = 1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

However, in practice, both Gini impurity and entropy typically yield very similar results, and it is often not worth spending much time on evaluating trees using different impurity criteria rather than experimenting with different pruning cut-offs.

Another impurity measure is the classification error:

$$I_E(t) = 1 - \max\{\, p(i|t)\}$$

This is a useful criterion for pruning but not recommended for growing a decision tree, since it is less sensitive to changes in the class probabilities of the nodes. We can illustrate this by looking at the two possible splitting scenarios shown in the following figure:



We start with a dataset, D_p, at the parent node, which consists of 40 examples from class 1 and 40 examples from class 2 that we split into two datasets, D_{left} and D_{right}. The information gain using the classification error as a splitting criterion would be the same ($IG_E = 0.25$) in both scenarios, A and B:

$$I_E(D_p) = 1 - 0.5 = 0.5$$

$$A:\quad I_E(D_{left}) = 1 - \frac{3}{4} = 0.25$$

$$A:\quad I_E(D_{Right}) = 1 - \frac{3}{4} = 0.25$$

$$A:\quad IG_E = 0.5 - \frac{4}{8}0.25 - \frac{4}{8}0.25 = 0.25$$

$$B:\quad I_E(D_{left}) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B:\quad I_E(D_{Right}) = 1 - 1 = 0$$

$$B:\quad IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25$$

However, the Gini impurity would favor the split in scenario B ($IG_G = 0.1\bar{\bar{6}}$) over scenario A ($IG_G = 0.125$), which is indeed purer:

$$I_G(D_p) = 1 - (0.5^2 + 0.5^2) = 0.5$$

$$A: \quad I_G(D_{left}) = 1 - \left(\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A: \quad I_G(D_{right}) = 1 - \left(\left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A: \quad IG_G = 0.5 - \frac{4}{8}0.375 - \frac{4}{8}0.375 = 0.125$$

$$B: \quad I_G(D_{left}) = 1 - \left(\left(\frac{2}{6}\right)^2 + \left(\frac{4}{6}\right)^2\right) = \frac{4}{9} = 0.\bar{4}$$

$$B: \quad I_G(D_{right}) = 1 - (1^2 + 0^2) = 0$$

$$B: \quad IG_G = 0.5 - \frac{6}{8}0.\bar{4} - 0 = 0.\overline{16}$$

Similarly, the entropy criterion would also favor scenario B ($IG_H = 0.31$) over scenario A ($IG_H = 0.19$):

$$I_H(D_p) = -\left(0.5\, log_2(0.5) + 0.5\, log_2(0.5)\right) = 1$$

$$A: I_H(D_{left}) = -\left(\frac{3}{4}\, log_2\left(\frac{3}{4}\right) + \frac{1}{4}\, log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A: I_H(D_{right}) = -\left(\frac{1}{4}\, log_2\left(\frac{1}{4}\right) + \frac{3}{4}\, log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A: IG_H = 1 - \frac{4}{8}0.81 - \frac{4}{8}0.81 = 0.19$$

$$B: I_H(D_{left}) = -\left(\frac{2}{6}\, log_2\left(\frac{2}{6}\right) + \frac{4}{6}\, log_2\left(\frac{4}{6}\right)\right) = 0.92$$

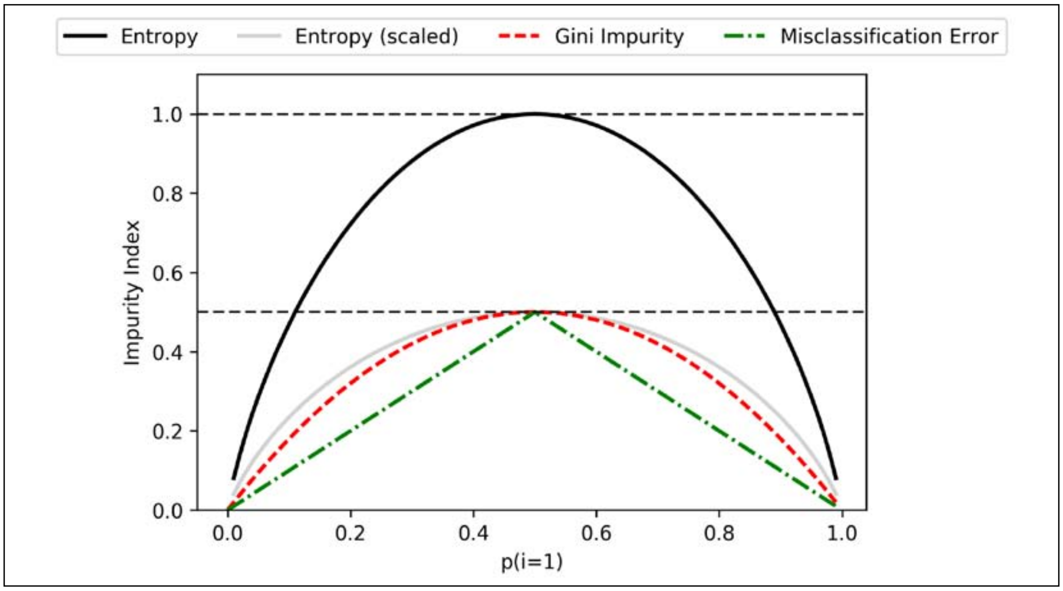
$$B: I_H(D_{right}) = 0$$

$$B: IG_H = 1 - \frac{6}{8}0.92 - 0 = 0.31$$

For a more visual comparison of the three different impurity criteria that we discussed previously, let's plot the impurity indices for the probability range [0, 1] for class 1. Note that we will also add a scaled version of the entropy (entropy / 2) to observe that the Gini impurity is an intermediate measure between entropy and the classification error. The code is as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                           ['Entropy', 'Entropy (scaled)',
...                            'Gini impurity',
...                            'Misclassification error'],
...                           ['-', '-', '--', '-.'],
...                           ['black', 'lightgray',
...                            'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                    linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...           ncol=5, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('impurity index')
```

The plot produced by the preceding code example is as follows:

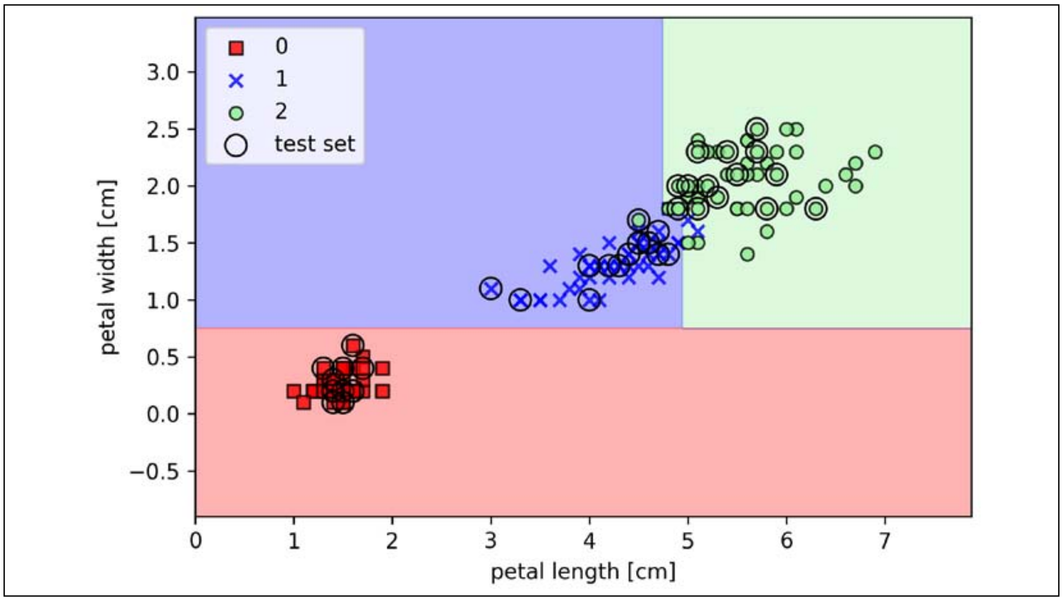


Building a decision tree

Decision trees can build complex decision boundaries by dividing the feature space into rectangles. However, we have to be careful since the deeper the decision tree, the more complex the decision boundary becomes, which can easily result in overfitting. Using scikit-learn, we will now train a decision tree with a maximum depth of 4, using Gini impurity as a criterion for impurity. Although feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms. The code is as follows:

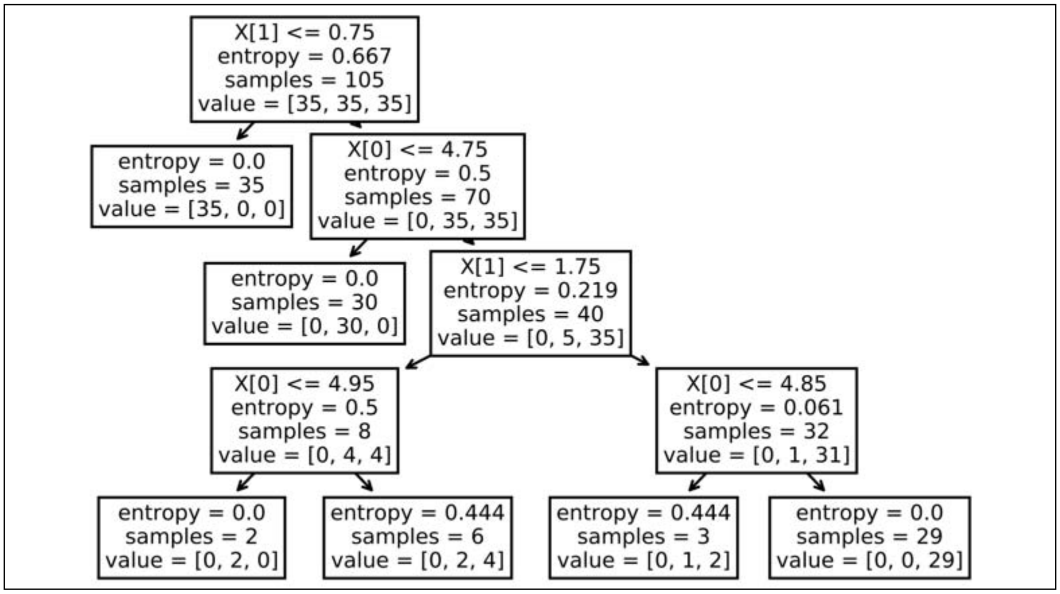
```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree_model = DecisionTreeClassifier(criterion='gini',
...                                     max_depth=4,
...                                     random_state=1)
>>> tree_model.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined,
...                       y_combined,
...                       classifier=tree_model,
...                       test_idx=range(105, 150))
>>> plt.xlabel('petal length [cm]')
>>> plt.ylabel('petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code example, we get the typical axis-parallel decision boundaries of the decision tree:



A nice feature in scikit-learn is that it allows us to readily visualize the decision tree model after training via the following code:

```
>>> from sklearn import tree
>>> tree.plot_tree(tree_model)
>>> plt.show()
```



However, nicer visualizations can be obtained by using the Graphviz program as a backend for plotting scikit-learn decision trees. This program is freely available from `http://www.graphviz.org` and is supported by Linux, Windows, and macOS. In addition to Graphviz, we will use a Python library called PyDotPlus, which has capabilities similar to Graphviz and allows us to convert `.dot` data files into a decision tree image file. After you have installed Graphviz (by following the instructions on `http://www.graphviz.org/download`), you can install PyDotPlus directly via the pip installer, for example, by executing the following command in your command-line terminal:

```
> pip3 install pydotplus
```

Installing PyDotPlus prerequisites

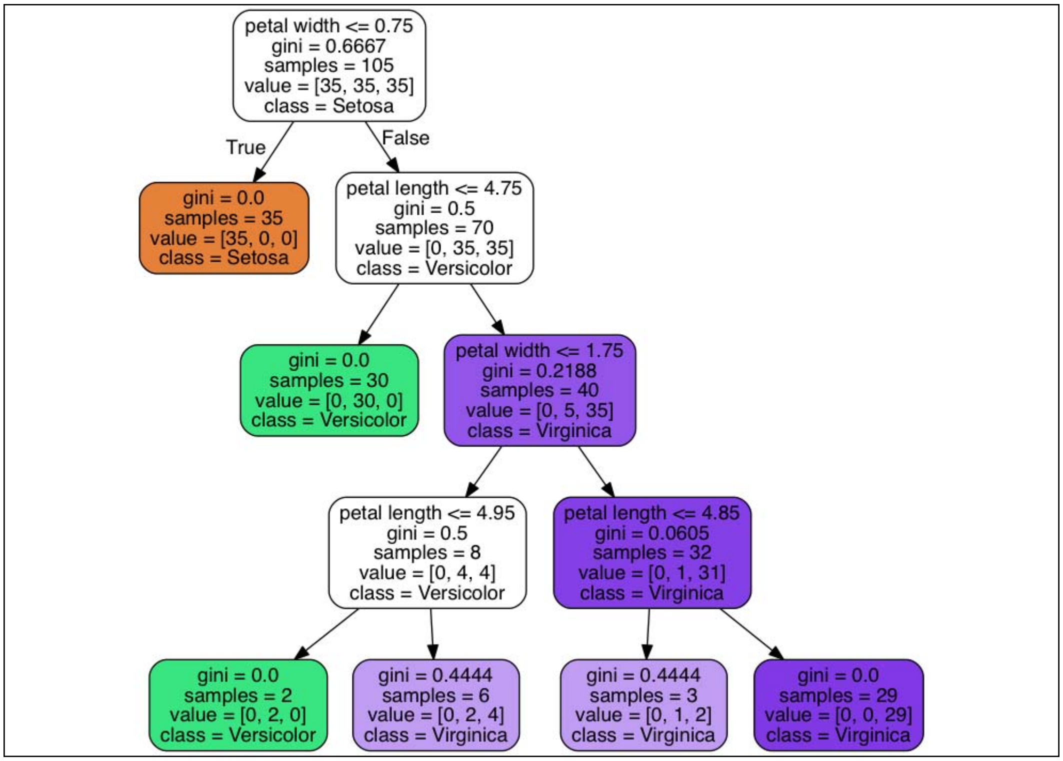
Note that on some systems, you may have to install the PyDotPlus prerequisites manually by executing the following commands:

```
pip3 install graphviz
pip3 install pyparsing
```

The following code will create an image of our decision tree in PNG format in our local directory:

```
>>> from pydotplus import graph_from_dot_data
>>> from sklearn.tree import export_graphviz
>>> dot_data = export_graphviz(tree_model,
...                            filled=True,
...                            rounded=True,
...                            class_names=['Setosa',
...                                         'Versicolor',
...                                         'Virginica'],
...                            feature_names=['petal length',
...                                           'petal width'],
...                            out_file=None)
>>> graph = graph_from_dot_data(dot_data)
>>> graph.write_png('tree.png')
```

By using the `out_file=None` setting, we directly assigned the DOT data to a `dot_data` variable, instead of writing an intermediate `tree.dot` file to disk. The arguments for `filled`, `rounded`, `class_names`, and `feature_names` are optional, but make the resulting image file visually more appealing by adding color, rounding the box edges, showing the name of the majority class label at each node, and displaying the feature name in each splitting criterion. These settings resulted in the following decision tree image:



Looking at the decision tree figure, we can now nicely trace back the splits that the decision tree determined from our training dataset. We started with 105 examples at the root and split them into two child nodes with 35 and 70 examples, using the petal width cut-off ≤ 0.75 cm. After the first split, we can see that the left child node is already pure and only contains examples from the `Iris-setosa` class (Gini impurity = 0). The further splits on the right are then used to separate the examples from the `Iris-versicolor` and `Iris-virginica` class.

Looking at this tree, and the decision region plot of the tree, we can see that the decision tree does a very good job of separating the flower classes. Unfortunately, scikit-learn currently does not implement functionality to manually post-prune a decision tree. However, we could go back to our previous code example, change the `max_depth` of our decision tree to `3`, and compare it to our current model, but we leave this as an exercise for the interested reader.

Combining multiple decision trees via random forests

Ensemble methods have gained huge popularity in applications of machine learning during the last decade due to their good classification performance and robustness toward overfitting. While we are going to cover different ensemble methods, including **bagging** and **boosting**, later in *Chapter 7*, *Combining Different Models for Ensemble Learning*, let's discuss the decision tree-based **random forest** algorithm, which is known for its good scalability and ease of use. A random forest can be considered as an **ensemble** of decision trees. The idea behind a random forest is to average multiple (deep) decision trees that individually suffer from high variance to build a more robust model that has a better generalization performance and is less susceptible to overfitting. The random forest algorithm can be summarized in four simple steps:

1. Draw a random **bootstrap** sample of size *n* (randomly choose *n* examples from the training dataset with replacement).
2. Grow a decision tree from the bootstrap sample. At each node:
    a. Randomly select *d* features without replacement.
    b. Split the node using the feature that provides the best split according to the objective function, for instance, maximizing the information gain.
3. Repeat the steps 1-2 *k* times.
4. Aggregate the prediction by each tree to assign the class label by **majority vote**. Majority voting will be discussed in more detail in *Chapter 7*, *Combining Different Models for Ensemble Learning*.

We should note one slight modification in step 2 when we are training the individual decision trees: instead of evaluating all features to determine the best split at each node, we only consider a random subset of those.

Sampling with and without replacement

In case you are not familiar with the terms sampling "with" and "without" replacement, let's walk through a simple thought experiment. let's assume that we are playing a lottery game where we randomly draw numbers from an urn. We start with an urn that holds five unique numbers, 0, 1, 2, 3, and 4, and we draw exactly one number each turn. In the first round, the chance of drawing a particular number from the urn would be 1/5. Now, in sampling without replacement, we do not put the number back into the urn after each turn. Consequently, the probability of drawing a particular number from the set of remaining numbers in the next round depends on the previous round. For example, if we have a remaining set of numbers 0, 1, 2, and 4, the chance of drawing number 0 would become 1/4 in the next turn.

However, in random sampling with replacement, we always return the drawn number to the urn so that the probability of drawing a particular number at each turn does not change; we can draw the same number more than once. In other words, in sampling *with* replacement, the samples (numbers) are independent and have a covariance of zero. For example, the results from five rounds of drawing random numbers could look like this:

- Random sampling without replacement: 2, 1, 3, 4, 0
- Random sampling with replacement: 1, 3, 3, 4, 1

Although random forests don't offer the same level of interpretability as decision trees, a big advantage of random forests is that we don't have to worry so much about choosing good hyperparameter values. We typically don't need to prune the random forest since the ensemble model is quite robust to noise from the individual decision trees. The only parameter that we really need to care about in practice is the number of trees, *k*, (step 3) that we choose for the random forest. Typically, the larger the number of trees, the better the performance of the random forest classifier at the expense of an increased computational cost.

Although it is less common in practice, other hyperparameters of the random forest classifier that can be optimized—using techniques that we will discuss in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*—are the size, n, of the bootstrap sample (step 1), and the number of features, d, that are randomly chosen for each split (step 2.a), respectively. Via the sample size, n, of the bootstrap sample, we control the bias-variance tradeoff of the random forest.

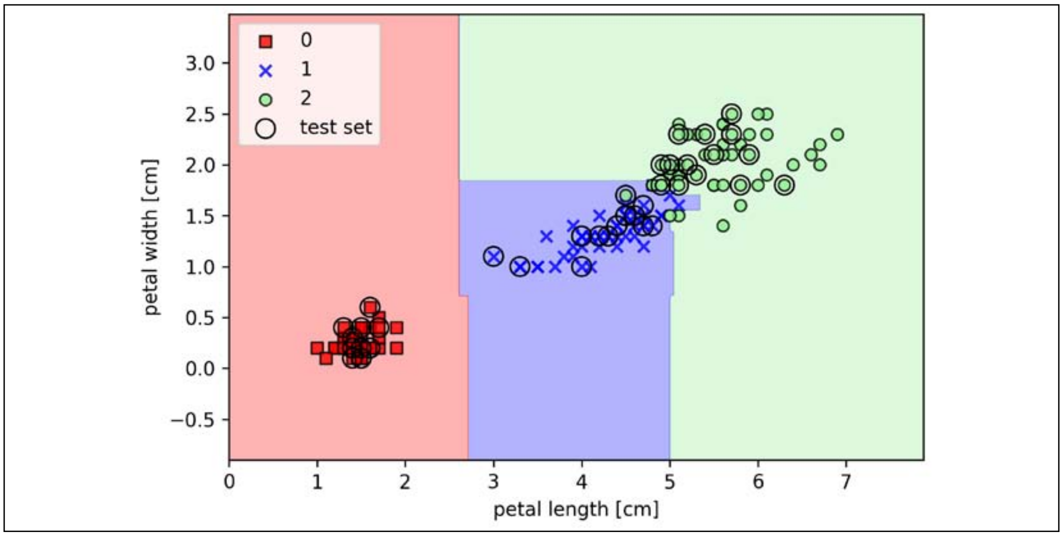
Decreasing the size of the bootstrap sample increases the diversity among the individual trees, since the probability that a particular training example is included in the bootstrap sample is lower. Thus, shrinking the size of the bootstrap samples may increase the *randomness* of the random forest, and it can help to reduce the effect of overfitting. However, smaller bootstrap samples typically result in a lower overall performance of the random forest, and a small gap between training and test performance, but a low test performance overall. Conversely, increasing the size of the bootstrap sample may increase the degree of overfitting. Because the bootstrap samples, and consequently the individual decision trees, become more similar to each other, they learn to fit the original training dataset more closely.

In most implementations, including the `RandomForestClassifier` implementation in scikit-learn, the size of the bootstrap sample is chosen to be equal to the number of training examples in the original training dataset, which usually provides a good bias-variance tradeoff. For the number of features, d, at each split, we want to choose a value that is smaller than the total number of features in the training dataset. A reasonable default that is used in scikit-learn and other implementations is $d = \sqrt{m}$, where m is the number of features in the training dataset.

Conveniently, we don't have to construct the random forest classifier from individual decision trees by ourselves because there is already an implementation in scikit-learn that we can use:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='gini',
...                                 n_estimators=25,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('petal length [cm]')
>>> plt.ylabel('petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the preceding code, we should see the decision regions formed by the ensemble of trees in the random forest, as shown in the following plot:



Using the preceding code, we trained a random forest from 25 decision trees via the `n_estimators` parameter and used the Gini impurity measure as a criterion to split the nodes. Although we are growing a very small random forest from a very small training dataset, we used the `n_jobs` parameter for demonstration purposes, which allows us to parallelize the model training using multiple cores of our computer (here, two cores).

K-nearest neighbors – a lazy learning algorithm

The last supervised learning algorithm that we want to discuss in this chapter is the **k-nearest neighbor** (**KNN**) classifier, which is particularly interesting because it is fundamentally different from the learning algorithms that we have discussed so far.

KNN is a typical example of a **lazy learner**. It is called "lazy" not because of its apparent simplicity, but because it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

Parametric versus nonparametric models

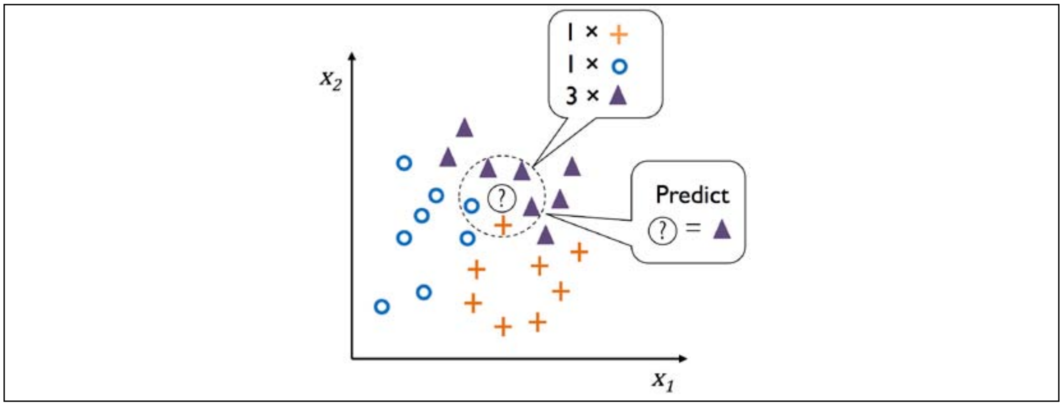
Machine learning algorithms can be grouped into **parametric** and **nonparametric** models. Using parametric models, we estimate parameters from the training dataset to learn a function that can classify new data points without requiring the original training dataset anymore. Typical examples of parametric models are the perceptron, logistic regression, and the linear SVM. In contrast, nonparametric models can't be characterized by a fixed set of parameters, and the number of parameters grows with the training data. Two examples of nonparametric models that we have seen so far are the decision tree classifier/random forest and the kernel SVM.

KNN belongs to a subcategory of nonparametric models that is described as **instance-based learning**. Models based on instance-based learning are characterized by memorizing the training dataset, and lazy learning is a special case of instance-based learning that is associated with no (zero) cost during the learning process.

The KNN algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number of *k* and a distance metric.
2. Find the *k*-nearest neighbors of the data record that we want to classify.
3. Assign the class label by majority vote.

The following figure illustrates how a new data point (**?**) is assigned the triangle class label based on majority voting among its five nearest neighbors.



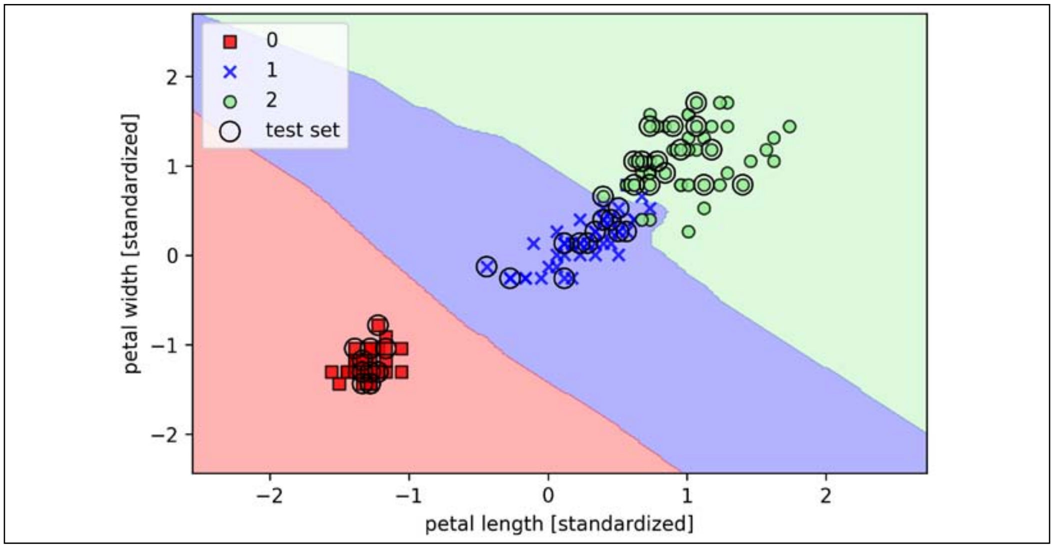
Based on the chosen distance metric, the KNN algorithm finds the *k* examples in the training dataset that are closest (most similar) to the point that we want to classify. The class label of the data point is then determined by a majority vote among its *k* nearest neighbors.

The main advantage of such a memory-based approach is that the classifier immediately adapts as we collect new training data. However, the downside is that the computational complexity for classifying new examples grows linearly with the number of examples in the training dataset in the worst-case scenario—unless the dataset has very few dimensions (features) and the algorithm has been implemented using efficient data structures such as k-d trees (*An Algorithm for Finding Best Matches in Logarithmic Expected Time*, *J. H. Friedman*, *J. L. Bentley*, and *R.A. Finkel*, *ACM transactions on mathematical software* (*TOMS*), *3(3): 209–226, 1977*). Furthermore, we can't discard training examples since no training step is involved. Thus, storage space can become a challenge if we are working with large datasets.

By executing the following code, we will now implement a KNN model in scikit-learn using a Euclidean distance metric:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

By specifying five neighbors in the KNN model for this dataset, we obtain a relatively smooth decision boundary, as shown in the following plot:



Resolving ties

In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the data record to be classified. If the neighbors have similar distances, the algorithm will choose the class label that comes first in the training dataset.

The *right* choice of *k* is crucial to finding a good balance between overfitting and underfitting. We also have to make sure that we choose a distance metric that is appropriate for the features in the dataset. Often, a simple Euclidean distance measure is used for real-value examples, for example, the flowers in our Iris dataset, which have features measured in centimeters. However, if we are using a Euclidean distance measure, it is also important to standardize the data so that each feature contributes equally to the distance. The `minkowski` distance that we used in the previous code is just a generalization of the Euclidean and Manhattan distance, which can be written as follows:

$$d\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} \quad x_k^{(j)}\right|^p},$$

It becomes the Euclidean distance if we set the parameter `p=2` or the Manhattan distance at `p=1`. Many other distance metrics are available in scikit-learn and can be provided to the `metric` parameter. They are listed at `http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html`.

The curse of dimensionality

It is important to mention that KNN is very susceptible to overfitting due to the **curse of dimensionality**. The curse of dimensionality describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. We can think of even the closest neighbors as being too far away in a high-dimensional space to give a good estimate.

We discussed the concept of regularization in the section about logistic regression as one way to avoid overfitting. However, in models where regularization is not applicable, such as decision trees and KNN, we can use feature selection and dimensionality reduction techniques to help us to avoid the curse of dimensionality. This will be discussed in more detail in the next chapter.

Summary

In this chapter, you learned about many different machine learning algorithms that are used to tackle linear and nonlinear problems. You have seen that decision trees are particularly attractive if we care about interpretability. Logistic regression is not only a useful model for online learning via SGD, but also allows us to predict the probability of a particular event.

Although SVMs are powerful linear models that can be extended to nonlinear problems via the kernel trick, they have many parameters that have to be tuned in order to make good predictions. In contrast, ensemble methods, such as random forests, don't require much parameter tuning and don't overfit as easily as decision trees, which makes them attractive models for many practical problem domains. The KNN classifier offers an alternative approach to classification via lazy learning that allows us to make predictions without any model training, but with a more computationally expensive prediction step.

However, even more important than the choice of an appropriate learning algorithm is the available data in our training dataset. No algorithm will be able to make good predictions without informative and discriminatory features.

In the next chapter, we will discuss important topics regarding the preprocessing of data, feature selection, and dimensionality reduction, which means that we will need to build powerful machine learning models. Later, in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will see how we can evaluate and compare the performance of our models and learn useful tricks to fine-tune the different algorithms.