2

Training Simple Machine Learning Algorithms for Classification

In this chapter, we will make use of two of the first algorithmically described machine learning algorithms for classification: the perceptron and adaptive linear neurons. We will start by implementing a perceptron step by step in Python and training it to classify different flower species in the Iris dataset. This will help us to understand the concept of machine learning algorithms for classification and how they can be efficiently implemented in Python.

Discussing the basics of optimization using adaptive linear neurons will then lay the groundwork for using more sophisticated classifiers via the scikit-learn machine learning library in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*.

The topics that we will cover in this chapter are as follows:

- Building an understanding of machine learning algorithms
- Using pandas, NumPy, and Matplotlib to read in, process, and visualize data
- Implementing linear classifiers for 2-class problems in Python

Artificial neurons – a brief glimpse into the early history of machine learning

Before we discuss the perceptron and related algorithms in more detail, let's take a brief tour of the beginnings of machine learning. Trying to understand how the biological brain works in order to design an **artificial intelligence** (**AI**), Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the so-called **McCulloch-Pitts** (**MCP**) neuron, in 1943 (*A Logical Calculus of the Ideas Immanent in Nervous Activity* by *W. S. McCulloch* and *W. Pitts*, *Bulletin of Mathematical Biophysics*, 5(4): 115-133, 1943).

Biological neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in *Figure 2.1*:

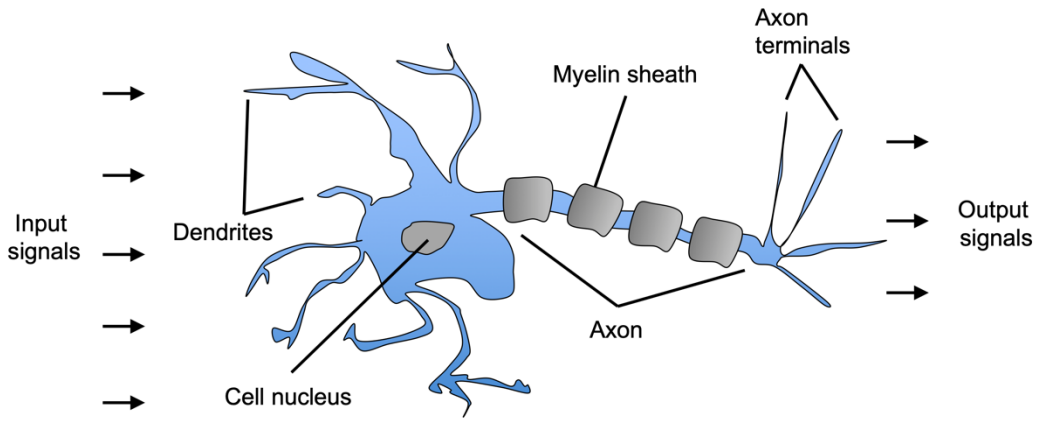

Figure 2.1: A neuron processing chemical and electrical signals

McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (*The Perceptron: A Perceiving and Recognizing Automaton* by *F. Rosenblatt*, *Cornell Aeronautical Laboratory*, 1957). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not. In the context of supervised learning and classification, such an algorithm could then be used to predict whether a new data point belongs to one class or the other.

The formal definition of an artificial neuron

More formally, we can put the idea behind **artificial neurons** into the context of a binary classification task with two classes: 0 and 1. We can then define a decision function, $\sigma(z)$, that takes a linear combination of certain input values, x, and a corresponding weight vector, w, where z is the so-called net input $z = w_1x_1 + w_2x_2 + \ldots + w_mx_m$:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Now, if the net input of a particular example, $\boldsymbol{x}^{(i)}$, is greater than a defined threshold, θ, we predict class 1, and class 0 otherwise. In the perceptron algorithm, the decision function, $\sigma(\cdot)$, is a variant of a **unit step function**:

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

To simplify the code implementation later, we can modify this setup via a couple of steps. First, we move the threshold, θ, to the left side of the equation:

$$z \geq \theta$$
$$z - \theta \geq 0$$

Second, we define a so-called *bias unit* as $b = -\theta$ and make it part of the net input:

$$z = w_1x_1 + \ldots + w_mx_m + b = \boldsymbol{w}^T\boldsymbol{x} + b$$

Third, given the introduction of the bias unit and the redefinition of the net input z above, we can redefine the decision function as follows:

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Linear algebra basics: dot product and matrix transpose

In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in $\boldsymbol{x}$ and $\boldsymbol{w}$ using a vector dot product, whereas the superscript T stands for transpose, which is an operation that transforms a column vector into a row vector and vice versa. For example, assume we have the following two column vectors:

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Then, we can write the transpose of vector $\boldsymbol{a}$ as $\boldsymbol{a}^T = [a_1\ a_2\ a_3]$ and write the dot product as

$$\boldsymbol{a}^T\boldsymbol{b} = \sum_i a_i b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

Furthermore, the transpose operation can also be applied to matrices to reflect it over its diagonal, for example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Please note that the transpose operation is strictly only defined for matrices; however, in the context of machine learning, we refer to $n \times 1$ or $1 \times m$ matrices when we use the term "vector."

In this book, we will only use very basic concepts from linear algebra; however, if you need a quick refresher, please take a look at Zico Kolter's excellent *Linear Algebra Review and Reference*, which is freely available at `http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf`.

Figure 2.2 illustrates how the net input $z = \boldsymbol{w}^T\boldsymbol{x} + b$ is squashed into a binary output (0 or 1) by the decision function of the perceptron (left subfigure) and how it can be used to discriminate between two classes separable by a linear decision boundary (right subfigure):

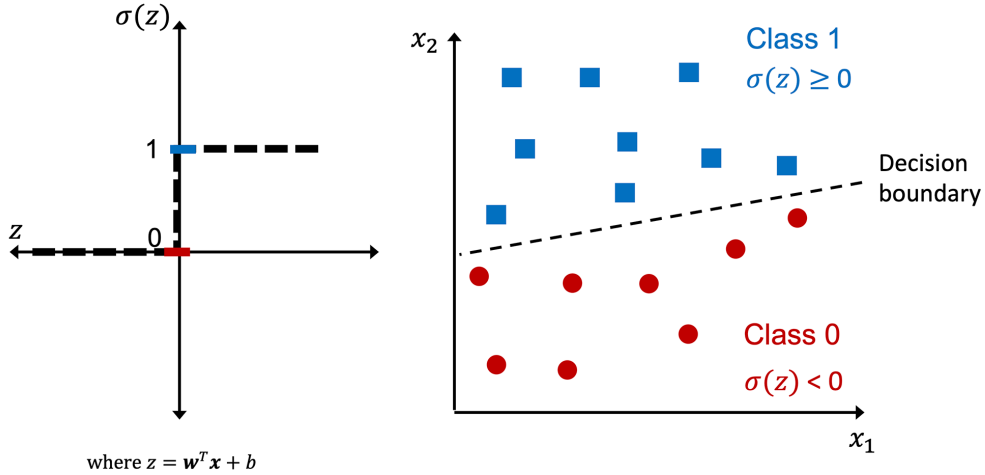

Figure 2.2: A threshold function producing a linear decision boundary for a binary classification problem

The perceptron learning rule

The whole idea behind the MCP neuron and Rosenblatt's *thresholded* perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either *fires* or it doesn't. Thus, Rosenblatt's classic perceptron rule is fairly simple, and the perceptron algorithm can be summarized by the following steps:

1. Initialize the weights and bias unit to 0 or small random numbers
2. For each training example, $\boldsymbol{x}^{(i)}$:
    a. Compute the output value, $\hat{y}^{(i)}$
    b. Update the weights and bias unit

Here, the output value is the class label predicted by the unit step function that we defined earlier, and the simultaneous update of the bias unit and each weight, w_j, in the weight vector, $\boldsymbol{w}$, can be more formally written as:

$$w_j := w_j + \Delta w_j$$
$$\text{and} \quad b := b + \Delta b$$

The update values ("deltas") are computed as follows:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$
$$\text{and} \quad \Delta b = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

Note that unlike the bias unit, each weight, w_j, corresponds to a feature, x_j, in the dataset, which is involved in determining the update value, Δw_j, defined above. Furthermore, η is the **learning rate** (typically a constant between 0.0 and 1.0), $y^{(i)}$ is the **true class label** of the ith training example, and $\hat{y}^{(i)}$ is the **predicted class label**. It is important to note that the bias unit and all weights in the weight vector are being updated simultaneously, which means that we don't recompute the predicted label, $\hat{y}^{(i)}$, before the bias unit and all of the weights are updated via the respective update values, Δw_j and Δb. Concretely, for a two-dimensional dataset, we would write the update as:

$$\begin{aligned}\Delta w_1 &= \eta(y^{(i)} - \text{output}^{(i)})x_1^{(i)};\\ \Delta w_2 &= \eta(y^{(i)} - \text{output}^{(i)})x_2^{(i)};\\ \Delta b &= \eta(y^{(i)} - \text{output}^{(i)})\end{aligned}$$

Before we implement the perceptron rule in Python, let's go through a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the bias unit and weights remain unchanged, since the update values are 0:

(1) $y^{(i)} = 0, \quad \hat{y}^{(i)} = 0, \quad \Delta w_j = \eta(0-0)x_j^{(i)} = 0, \quad \Delta b = \eta(0-0) = 0$

(2) $y^{(i)} = 1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(1-1)x_j^{(i)} = 0, \quad \Delta b = \eta(1-1) = 0$

However, in the case of a wrong prediction, the weights are being pushed toward the direction of the positive or negative target class:

(3) $y^{(i)} = 1, \quad \hat{y}^{(i)} = 0, \quad \Delta w_j = \eta(1-0)x_j^{(i)} = \eta x_j^{(i)}, \quad \Delta b = \eta(1-0) = \eta$

(4) $y^{(i)} = 0, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(0-1)x_j^{(i)} = -\eta x_j^{(i)}, \quad \Delta b = \eta(0-1) = -\eta$

To get a better understanding of the feature value as a multiplicative factor, $x_j^{(i)}$, let's go through another simple example, where:

$$y^{(i)} = 1, \quad \hat{y}^{(i)} = 0, \quad \eta = 1$$

Let's assume that $x_j^{(i)} = 1.5$ and we misclassify this example as *class 0*. In this case, we would increase the corresponding weight by 2.5 in total so that the net input, $z = x_j^{(i)} \times w_j + b$, would be more positive the next time we encounter this example, and thus be more likely to be above the threshold of the unit step function to classify the example as *class 1*:

$$\Delta w_j = (1-0)1.5 = 1.5, \quad \Delta b = (1-0) = 1$$

The weight update, Δw_j, is proportional to the value of $x_j^{(i)}$. For instance, if we have another example, $x_j^{(i)} = 2$, that is incorrectly classified as *class 0*, we will push the decision boundary by an even larger extent to classify this example correctly the next time:

$$\Delta w_j = (1-0)2 = 2, \quad \Delta b = (1-0) = 1$$

It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable, which means that the two classes can be perfectly separated by a linear decision boundary. (Interested readers can find the convergence proof in my lecture notes: `https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf`). *Figure 2.3* shows visual examples of linearly separable and linearly inseparable scenarios:

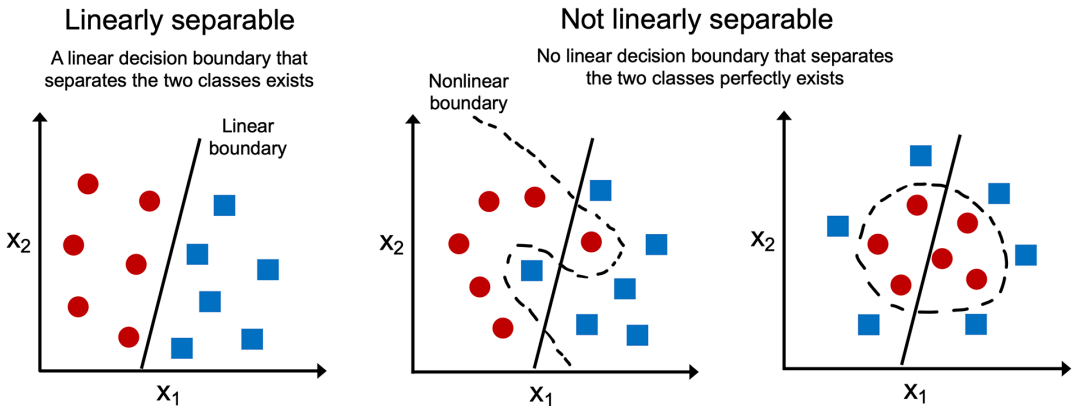

Figure 2.3: Examples of linearly and nonlinearly separable classes

If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (**epochs**) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise. Later in this chapter, we will cover the Adaline algorithm that produces linear decision boundaries and converges even if the classes are not perfectly linearly separable. In *Chapter 3*, we will learn about algorithms that can produce nonlinear decision boundaries.

Downloading the example code

If you bought this book directly from Packt, you can download the example code files from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can download all code examples and datasets directly from `https://github.com/rasbt/machine-learning-book`.

Now, before we jump into the implementation in the next section, what you just learned can be summarized in a simple diagram that illustrates the general concept of the perceptron:

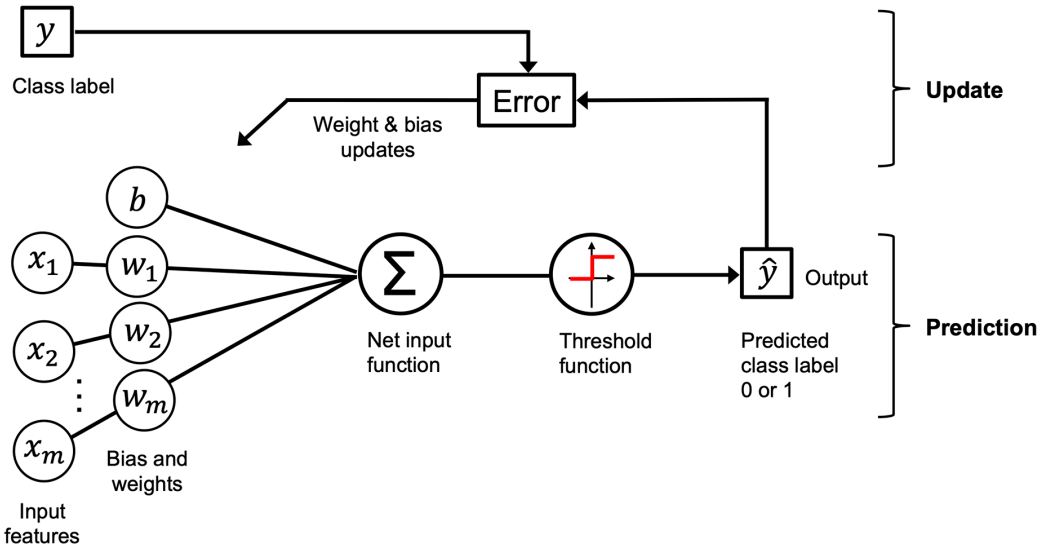

Figure 2.4: Weights and bias of the model are updated based on the error function

The preceding diagram illustrates how the perceptron receives the inputs of an example (x) and combines them with the bias unit (b) and weights (w) to compute the net input. The net input is then passed on to the threshold function, which generates a binary output of 0 or 1—the predicted class label of the example. During the learning phase, this output is used to calculate the error of the prediction and update the weights and bias unit.

Implementing a perceptron learning algorithm in Python

In the previous section, we learned how Rosenblatt's perceptron rule works; let's now implement it in Python and apply it to the Iris dataset that we introduced in *Chapter 1*, *Giving Computers the Ability to Learn from Data*.

An object-oriented perceptron API

We will take an object-oriented approach to defining the perceptron interface as a Python class, which will allow us to initialize new `Perceptron` objects that can learn from data via a `fit` method and make predictions via a separate `predict` method. As a convention, we append an underscore (_) to attributes that are not created upon the initialization of the object, but we do this by calling the object's other methods, for example, `self.w_`.

Additional resources for Python's scientific computing stack

If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:

- **NumPy**: `https://sebastianraschka.com/blog/2020/numpy-intro.html`
- **pandas**: `https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html`
- **Matplotlib**: `https://matplotlib.org/stable/tutorials/introductory/usage.html`

The following is the implementation of a perceptron in Python:

```
import numpy as np
class Perceptron:
    """Perceptron classifier.

    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.
    b_ : Scalar
      Bias unit after fitting.

    errors_ : list
      Number of misclassifications (updates) in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, 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=X.shape[1])
        self.b_ = np.float_(0.)
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_ += update * xi
                self.b_ += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

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

Using this perceptron implementation, we can now initialize new `Perceptron` objects with a given learning rate, `eta` (η), and the number of epochs, `n_iter` (passes over the training dataset).

Via the `fit` method, we initialize the bias `self.b_` to an initial value 0 and the weights in `self.w_` to a vector, $\mathbb{R}^m$, where m stands for the number of dimensions (features) in the dataset.

Notice that the initial weight vector contains small random numbers drawn from a normal distribution with a standard deviation of 0.01 via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, where `rgen` is a NumPy random number generator that we seeded with a user-specified random seed so that we can reproduce previous results if desired.

Technically, we could initialize the weights to zero (in fact, this is done in the original perceptron algorithm). However, if we did that, then the learning rate η (`eta`) would have no effect on the decision boundary. If all the weights are initialized to zero, the learning rate parameter, `eta`, affects only the scale of the weight vector, not the direction. If you are familiar with trigonometry, consider a vector, $v1$ =[1 2 3], where the angle between $v1$ and a vector, $v2 = 0.5 \times v1$, would be exactly zero, as demonstrated by the following code snippet:

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...           np.linalg.norm(v2)))
0.0
```

Here, `np.arccos` is the trigonometric inverse cosine, and `np.linalg.norm` is a function that computes the length of a vector. (Our decision to draw the random numbers from a random normal distribution—for example, instead of from a uniform distribution—and to use a standard deviation of `0.01` was arbitrary; remember, we are just interested in small random values to avoid the properties of all-zero vectors, as discussed earlier.)

As an optional exercise after reading this chapter, you can change `self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])` to `self.w_ = np.zeros(X.shape[1])` and run the perceptron training code presented in the next section with different values for `eta`. You will observe that the decision boundary does not change.

NumPy array indexing

NumPy indexing for one-dimensional arrays works similarly to Python lists using the square-bracket (`[]`) notation. For two-dimensional arrays, the first indexer refers to the row number and the second indexer to the column number. For example, we would use `X[2, 3]` to select the third row and fourth column of a two-dimensional array, `X`.

After the weights have been initialized, the `fit` method loops over all individual examples in the training dataset and updates the weights according to the perceptron learning rule that we discussed in the previous section.

The class labels are predicted by the `predict` method, which is called in the `fit` method during training to get the class label for the weight update; but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the `self.errors_` list so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product, $\boldsymbol{w}^T\boldsymbol{x} + b$.

Vectorization: Replacing for loops with vectorized code

Instead of using NumPy to calculate the vector dot product between two arrays, `a` and `b`, via `a.dot(b)` or `np.dot(a, b)`, we could also perform the calculation in pure Python via `sum([i * j for i, j in zip(a, b)])`. However, the advantage of using NumPy over classic Python `for` loop structures is that its arithmetic operations are vectorized. Vectorization means that an elemental arithmetic operation is automatically applied to all elements in an array. By formulating our arithmetic operations as a sequence of instructions on an array, rather than performing a set of operations for each element at a time, we can make better use of our modern **central processing unit** (**CPU**) architectures with **single instruction, multiple data** (**SIMD**) support. Furthermore, NumPy uses highly optimized linear algebra libraries, such as **Basic Linear Algebra Subprograms** (**BLAS**) and **Linear Algebra Package** (**LAPACK**), that have been written in C or Fortran. Lastly, NumPy also allows us to write our code in a more compact and intuitive way using the basics of linear algebra, such as vector and matrix dot products.

Training a perceptron model on the Iris dataset

To test our perceptron implementation, we will restrict the following analyses and examples in the remainder of this chapter to two feature variables (dimensions). Although the perceptron rule is not restricted to two dimensions, considering only two features, sepal length and petal length, will allow us to visualize the decision regions of the trained model in a scatterplot for learning purposes.

Note that we will also only consider two flower classes, setosa and versicolor, from the Iris dataset for practical reasons—remember, the perceptron is a binary classifier. However, the perceptron algorithm can be extended to multi-class classification—for example, the **one-versus-all** (**OvA**) technique.

The OvA method for multi-class classification

OvA, which is sometimes also called **one-versus-rest** (**OvR**), is a technique that allows us to extend any binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the examples from all other classes are considered negative classes. If we were to classify a new, unlabeled data instance, we would use our *n* classifiers, where *n* is the number of class labels, and assign the class label with the highest confidence to the particular instance we want to classify. In the case of the perceptron, we would use OvA to choose the class label that is associated with the largest absolute net input value.

First, we will use the `pandas` library to load the Iris dataset directly from the *UCI Machine Learning Repository* into a `DataFrame` object and print the last five lines via the `tail` method to check that the data was loaded correctly:

```
>>> import os
>>> import pandas as pd
>>> s = 'https://archive.ics.uci.edu/ml/'\
...     'machine-learning-databases/iris/iris.data'
>>> print('From URL:', s)
From URL: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
data
>>> df = pd.read_csv(s,
...                  header=None,
...                  encoding='utf-8')
>>> df.tail()
```

After executing the previous code, we should see the following output, which shows the last five lines of the Iris dataset:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Figure 2.5: The last five lines of the Iris dataset

Loading the Iris dataset

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

```
df = pd.read_csv(
  'https://archive.ics.uci.edu/ml/'
  'machine-learning-databases/iris/iris.data',
  header=None, encoding='utf-8')
```

with the following one:

```
df = pd.read_csv(
  'your/local/path/to/iris.data',
  header=None, encoding='utf-8')
```

Next, we extract the first 100 class labels that correspond to the 50 Iris-setosa and 50 Iris-versicolor flowers and convert the class labels into the two integer class labels, `1` (versicolor) and `0` (setosa), that we assign to a vector, `y`, where the `values` method of a pandas `DataFrame` yields the corresponding NumPy representation.

Similarly, we extract the first feature column (sepal length) and the third feature column (petal length) of those 100 training examples and assign them to a feature matrix, `X`, which we can visualize via a two-dimensional scatterplot:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> # select setosa and versicolor
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', 0, 1)
>>> # extract sepal length and petal length
>>> X = df.iloc[0:100, [0, 2]].values
>>> # plot data
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='o', label='Setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='s', label='Versicolor')
>>> plt.xlabel('Sepal length [cm]')
>>> plt.ylabel('Petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should see the following scatterplot:

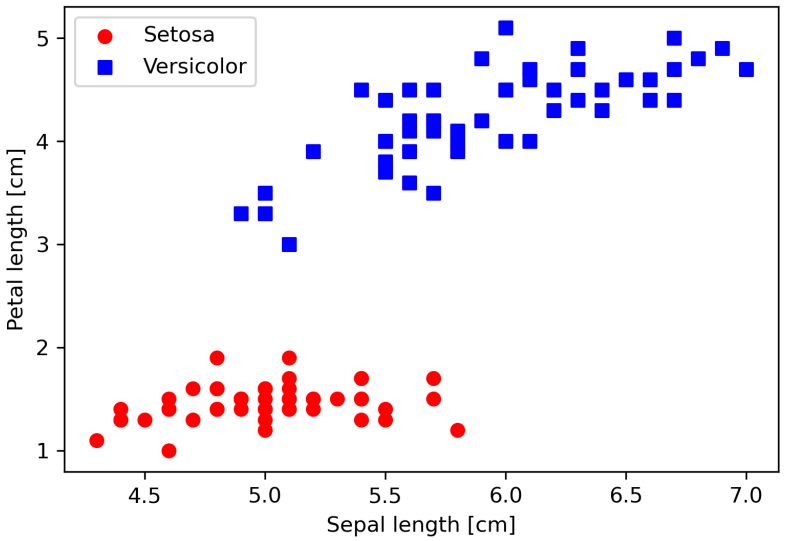

Figure 2.6: Scatterplot of setosa and versicolor flowers by sepal and petal length

Figure 2.6 shows the distribution of flower examples in the Iris dataset along the two feature axes: petal length and sepal length (measured in centimeters). In this two-dimensional feature subspace, we can see that a linear decision boundary should be sufficient to separate setosa from versicolor flowers. Thus, a linear classifier such as the perceptron should be able to classify the flowers in this dataset perfectly.

Now, it's time to train our perceptron algorithm on the Iris data subset that we just extracted. Also, we will plot the misclassification error for each epoch to check whether the algorithm converged and found a decision boundary that separates the two Iris flower classes:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1),
...          ppn.errors_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of updates')
>>> plt.show()
```

Note that the number of misclassification errors and the number of updates is the same, since the perceptron weights and bias are updated each time it misclassifies an example. After executing the preceding code, we should see the plot of the misclassification errors versus the number of epochs, as shown in *Figure 2.7*:

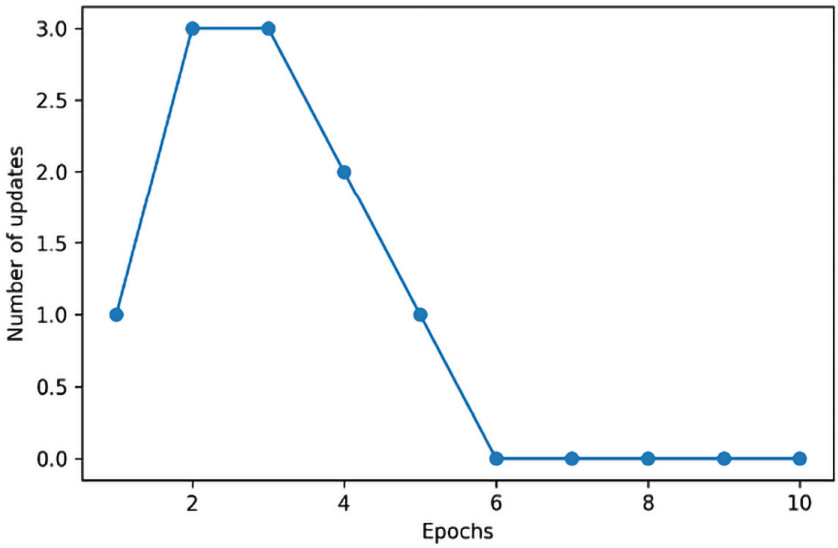

Figure 2.7: A plot of the misclassification errors against the number of epochs

As we can see in *Figure 2.7*, our perceptron converged after the sixth epoch and should now be able to classify the training examples perfectly. Let's implement a small convenience function to visualize the decision boundaries for two-dimensional datasets:

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    # setup marker generator and color map
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

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

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

First, we define a number of `colors` and `markers` and create a colormap from the list of colors via `ListedColormap`. Then, we determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays, `xx1` and `xx2`, via the NumPy `meshgrid` function. Since we trained our perceptron classifier on two feature dimensions, we need to flatten the grid arrays and create a matrix that has the same number of columns as the Iris training subset so that we can use the `predict` method to predict the class labels, `lab`, of the corresponding grid points.

After reshaping the predicted class labels, `lab`, into a grid with the same dimensions as `xx1` and `xx2`, we can now draw a contour plot via Matplotlib's `contourf` function, which maps the different decision regions to different colors for each predicted class in the grid array:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('Sepal length [cm]')
>>> plt.ylabel('Petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should now see a plot of the decision regions, as shown in *Figure 2.8*:

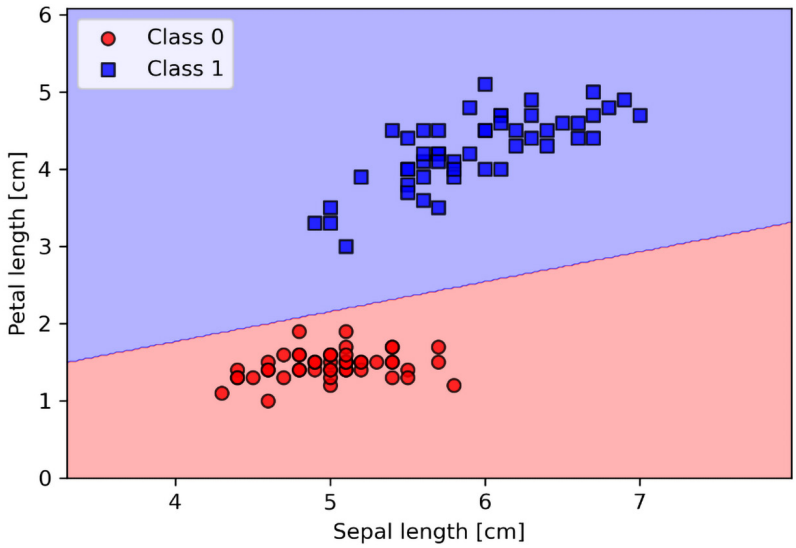

Figure 2.8: A plot of the perceptron's decision regions

As we can see in the plot, the perceptron learned a decision boundary that can classify all flower examples in the Iris training subset perfectly.

Perceptron convergence

Although the perceptron classified the two Iris flower classes perfectly, convergence is one of the biggest problems of the perceptron. Rosenblatt proved mathematically that the perceptron learning rule converges if the two classes can be separated by a linear hyperplane. However, if the classes cannot be separated perfectly by such a linear decision boundary, the weights will never stop updating unless we set a maximum number of epochs. Interested readers can find a summary of the proof in my lecture notes at `https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf`.

Adaptive linear neurons and the convergence of learning

In this section, we will take a look at another type of single-layer **neural network** (**NN**): **ADAptive LInear NEuron** (**Adaline**). Adaline was published by Bernard Widrow and his doctoral student Tedd Hoff only a few years after Rosenblatt's perceptron algorithm, and it can be considered an improvement on the latter (*An Adaptive "Adaline" Neuron Using Chemical "Memistors"*, *Technical Report Number 1553-2* by *B. Widrow and colleagues*, *Stanford Electron Labs*, Stanford, CA, *October* 1960).

The Adaline algorithm is particularly interesting because it illustrates the key concepts of defining and minimizing continuous loss functions. This lays the groundwork for understanding other machine learning algorithms for classification, such as logistic regression, support vector machines, and multilayer neural networks, as well as linear regression models, which we will discuss in future chapters.

The key difference between the Adaline rule (also known as the **Widrow-Hoff rule**) and Rosenblatt's perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function, $\sigma(z)$, is simply the identity function of the net input, so that $\sigma(z) = z$.

While the linear activation function is used for learning the weights, we still use a threshold function to make the final prediction, which is similar to the unit step function that we covered earlier.

The main differences between the perceptron and Adaline algorithm are highlighted in *Figure 2.9*:

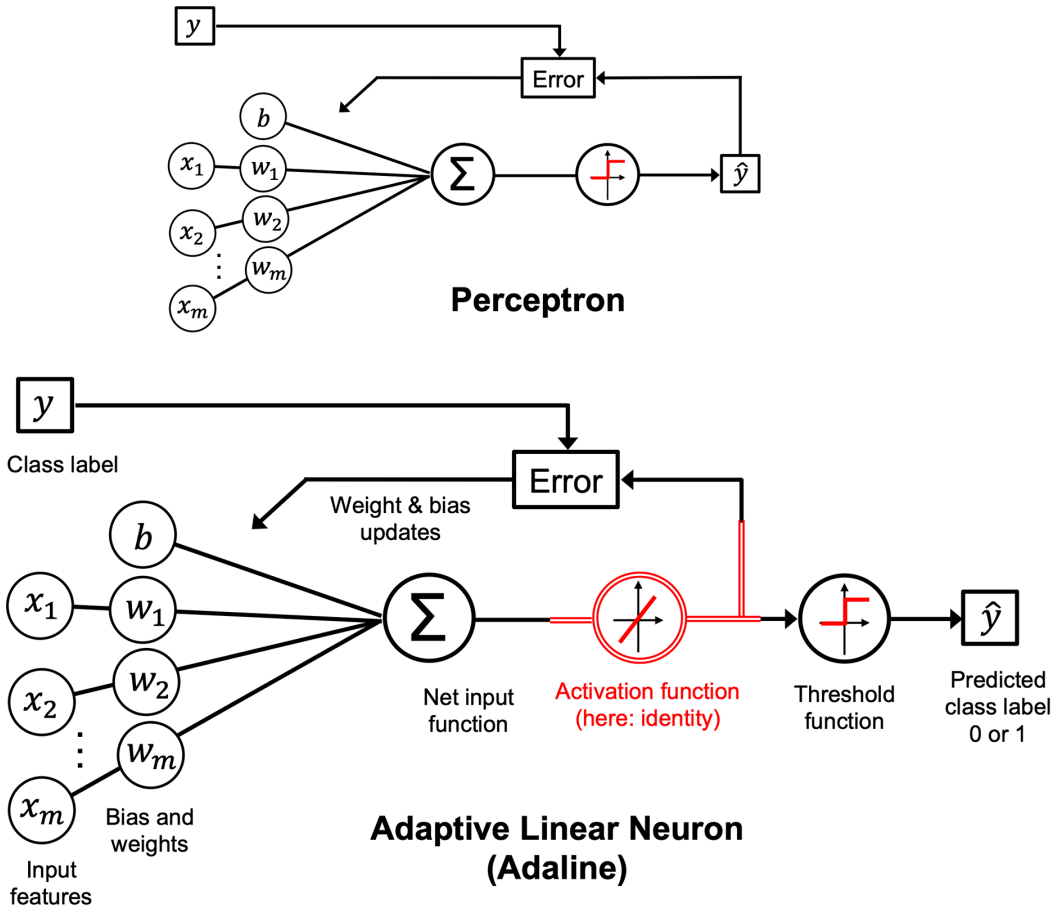

Figure 2.9: A comparison between a perceptron and the Adaline algorithm

As *Figure 2.9* indicates, the Adaline algorithm compares the true class labels with the linear activation function's continuous valued output to compute the model error and update the weights. In contrast, the perceptron compares the true class labels to the predicted class labels.

Minimizing loss functions with gradient descent

One of the key ingredients of supervised machine learning algorithms is a defined **objective function** that is to be optimized during the learning process. This objective function is often a loss or cost function that we want to minimize. In the case of Adaline, we can define the loss function, L, to learn the model parameters as the **mean squared error** (**MSE**) between the calculated outcome and the true class label:

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)^2$$

The main advantage of this continuous linear activation function, in contrast to the unit step function, is that the loss function becomes differentiable. Another nice property of this loss function is that it is convex; thus, we can use a very simple yet powerful optimization algorithm called **gradient descent** to find the weights that minimize our loss function to classify the examples in the Iris dataset.

As illustrated in *Figure 2.10*, we can describe the main idea behind gradient descent as *climbing down a hill* until a local or global loss minimum is reached. In each iteration, we take a step in the opposite direction of the gradient, where the step size is determined by the value of the learning rate, as well as the slope of the gradient (for simplicity, the following figure visualizes this only for a single weight, w):

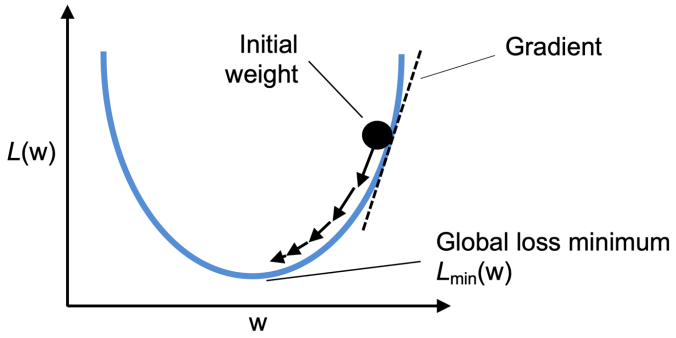

Figure 2.10: How gradient descent works

Using gradient descent, we can now update the model parameters by taking a step in the opposite direction of the gradient, $\nabla L(\boldsymbol{w}, b)$, of our loss function, $L(\boldsymbol{w}, b)$:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \quad b := b + \Delta b$$

The parameter changes, $\Delta \boldsymbol{w}$ and Δb, are defined as the negative gradient multiplied by the learning rate, η:

$$\Delta \boldsymbol{w} = -\eta \nabla_w L(\boldsymbol{w}, b), \quad \Delta b = -\eta \nabla_b L(\boldsymbol{w}, b)$$

To compute the gradient of the loss function, we need to compute the partial derivative of the loss function with respect to each weight, w_j:

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) x_j^{(i)}$$

Similarly, we compute the partial derivative of the loss with respect to the bias as:

$$\frac{\partial L}{\partial b} = -\frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right)$$

Please note that the 2 in the numerator above is merely a constant scaling factor, and we could omit it without affecting the algorithm. Removing the scaling factor has the same effect as changing the learning rate by a factor of 2. The following information box explains where this scaling factor originates.

So we can write the weight update as:

$$\Delta w_j = -\eta \frac{\partial L}{\partial w_j} \text{ and } \Delta b = -\eta \frac{\partial L}{\partial b}$$

Since we update all parameters simultaneously, our Adaline learning rule becomes:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}, \quad b := b + \Delta b$$

The mean squared error derivative

If you are familiar with calculus, the partial derivative of the MSE loss function with respect to the *j*th weight can be obtained as follows:

$$\begin{aligned}
\frac{\partial L}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right)^2 = \frac{1}{n} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right)^2 \\
&= \frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) \\
&= \frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_j \left(w_j x_j^{(i)} + b \right) \right) \\
&= \frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) \left(-x_j^{(i)} \right) = -\frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) x_j^{(i)}
\end{aligned}$$

The same approach can be used to find partial derivative $\frac{\partial L}{\partial b}$ except that $\frac{\partial}{\partial b}\left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)} + b \right) \right)$ is equal to –1 and thus the last step simplifies to $-\frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right)$.

Although the Adaline learning rule looks identical to the perceptron rule, we should note that $\sigma(z^{(i)})$ with $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)} + b$ is a real number and not an integer class label. Furthermore, the weight update is calculated based on all examples in the training dataset (instead of updating the parameters incrementally after each training example), which is why this approach is also referred to as **batch gradient descent**. To be more explicit and avoid confusion when talking about related concepts later in this chapter and this book, we will refer to this process as **full batch gradient descent**.

Implementing Adaline in Python

Since the perceptron rule and Adaline are very similar, we will take the perceptron implementation that we defined earlier and change the `fit` method so that the weight and bias parameters are now updated by minimizing the loss function via gradient descent:

```
class AdalineGD:
    """ADAptive LInear NEuron classifier.

    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.
    b_ : Scalar
        Bias unit after fitting.
    losses_ : list
      Mean squared error loss function values in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, 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=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors**2).mean()
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, X):
        """Compute linear activation"""
        return X

    def predict(self, X):
        """Return class label after unit step"""
```

```
        return np.where(self.activation(self.net_input(X))
                        >= 0.5, 1, 0)
```

Instead of updating the weights after evaluating each individual training example, as in the perceptron, we calculate the gradient based on the whole training dataset. For the bias unit, this is done via `self.eta * 2.0 * errors.mean()`, where `errors` is an array containing the partial derivative values $\frac{\partial}{\partial b}$. Similarly, we update the weights. However note that the weight updates via the partial derivatives $\frac{\partial L}{\partial w_j}$ involve the feature values x_j, which we can compute by multiplying `errors` with each feature value for each weight:

```
            for w_j in range(self.w_.shape[0]):
                self.w_[w_j] += self.eta *
                    (2.0 * (X[:, w_j]*errors)).mean()
```

To implement the weight update more efficiently without using a `for` loop, we can use a matrix-vector multiplication between our feature matrix and the error vector instead:

```
self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
```

Please note that the `activation` method has no effect on the code since it is simply an identity function. Here, we added the activation function (computed via the `activation` method) to illustrate the general concept with regard to how information flows through a single-layer NN: features from the input data, net input, activation, and output.

In the next chapter, we will learn about a logistic regression classifier that uses a non-identity, nonlinear activation function. We will see that a logistic regression model is closely related to Adaline, with the only difference being its activation and loss function.

Now, similar to the previous perceptron implementation, we collect the loss values in a `self.losses_` list to check whether the algorithm converged after training.

Matrix multiplication

Performing a matrix multiplication is similar to calculating a vector dot-product where each row in the matrix is treated as a single row vector. This vectorized approach represents a more compact notation and results in a more efficient computation using NumPy. For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

Please note that in the preceding equation, we are multiplying a matrix with a vector, which is mathematically not defined. However, remember that we use the convention that this preceding vector is regarded as a 3×1 matrix.

In practice, it often requires some experimentation to find a good learning rate, η, for optimal convergence. So, let's choose two different learning rates, $\eta = 0.1$ and $\eta = 0.0001$, to start with and plot the loss functions versus the number of epochs to see how well the Adaline implementation learns from the training data.

Hyperparameters

The learning rate, η (`eta`), as well as the number of epochs (`n_iter`), are the so-called hyperparameters (or tuning parameters) of the perceptron and Adaline learning algorithms. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will take a look at different techniques to automatically find the values of different hyperparameters that yield optimal performance of the classification model.

Let's now plot the loss against the number of epochs for the two different learning rates:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> ada1 = AdalineGD(n_iter=15, eta=0.1).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.losses_) + 1),
...            np.log10(ada1.losses_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Mean squared error)')
>>> ax[0].set_title('Adaline - Learning rate 0.1')
>>> ada2 = AdalineGD(n_iter=15, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.losses_) + 1),
...            ada2.losses_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Mean squared error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```

As we can see in the resulting loss function plots, we encountered two different types of problems. The left chart shows what could happen if we choose a learning rate that is too large. Instead of minimizing the loss function, the MSE becomes larger in every epoch, because we *overshoot* the global minimum. On the other hand, we can see that the loss decreases on the right plot, but the chosen learning rate, $\eta = 0.0001$, is so small that the algorithm would require a very large number of epochs to converge to the global loss minimum:

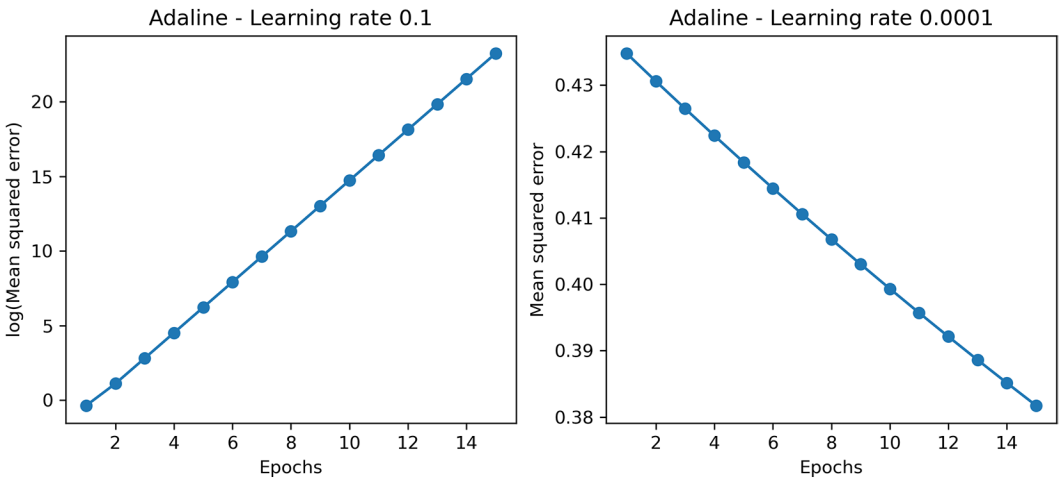

Figure 2.11: Error plots for suboptimal learning rates

Figure 2.12 illustrates what might happen if we change the value of a particular weight parameter to minimize the loss function, *L*. The left subfigure illustrates the case of a well-chosen learning rate, where the loss decreases gradually, moving in the direction of the global minimum.

The subfigure on the right, however, illustrates what happens if we choose a learning rate that is too large—we overshoot the global minimum:

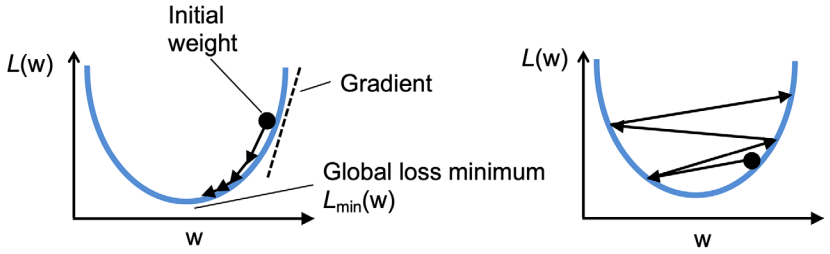

Figure 2.12: A comparison of a well-chosen learning rate and a learning rate that is too large

Improving gradient descent through feature scaling

Many machine learning algorithms that we will encounter throughout this book require some sort of feature scaling for optimal performance, which we will discuss in more detail in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, and *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*.

Gradient descent is one of the many algorithms that benefit from feature scaling. In this section, we will use a feature scaling method called **standardization**. This normalization procedure helps gradient descent learning to converge more quickly; however, it does not make the original dataset normally distributed. Standardization shifts the mean of each feature so that it is centered at zero and each feature has a standard deviation of 1 (unit variance). For instance, to standardize the *j*th feature, we can simply subtract the sample mean, μ_j, from every training example and divide it by its standard deviation, σ_j:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Here, x_j is a vector consisting of the *j*th feature values of all training examples, n, and this standardization technique is applied to each feature, j, in our dataset.

One of the reasons why standardization helps with gradient descent learning is that it is easier to find a learning rate that works well for all weights (and the bias). If the features are on vastly different scales, a learning rate that works well for updating one weight might be too large or too small to update the other weight equally well. Overall, using standardized features can stabilize the training such that the optimizer has to go through fewer steps to find a good or optimal solution (the global loss minimum). *Figure 2.13* illustrates possible gradient updates with unscaled features (left) and standardized features (right), where the concentric circles represent the loss surface as a function of two model weights in a two-dimensional classification problem:

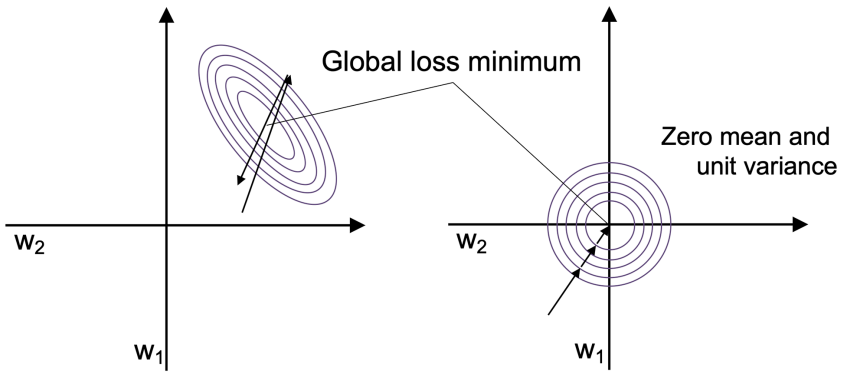

Figure 2.13: A comparison of unscaled and standardized features on gradient updates

Standardization can easily be achieved by using the built-in NumPy methods `mean` and `std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

After standardization, we will train Adaline again and see that it now converges after a small number of epochs using a learning rate of $\eta = 0.5$:

```
>>> ada_gd = AdalineGD(n_iter=20, eta=0.5)
>>> ada_gd.fit(X_std, y)
```

```
>>> plot_decision_regions(X_std, y, classifier=ada_gd)
>>> plt.title('Adaline - Gradient descent')
>>> plt.xlabel('Sepal length [standardized]')
>>> plt.ylabel('Petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_gd.losses_) + 1),
...          ada_gd.losses_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Mean squared error')
>>> plt.tight_layout()
>>> plt.show()
```

After executing this code, we should see a figure of the decision regions, as well as a plot of the declining loss, as shown in *Figure 2.14*:

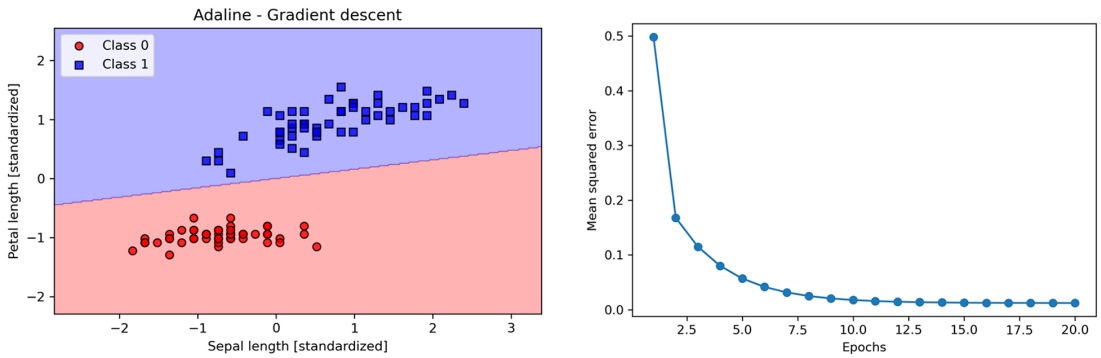

Figure 2.14: Plots of Adaline's decision regions and MSE by number of epochs

As we can see in the plots, Adaline has now converged after training on the standardized features. However, note that the MSE remains non-zero even though all flower examples were classified correctly.

Large-scale machine learning and stochastic gradient descent

In the previous section, we learned how to minimize a loss function by taking a step in the opposite direction of the loss gradient that is calculated from the whole training dataset; this is why this approach is sometimes also referred to as full batch gradient descent. Now imagine that we have a very large dataset with millions of data points, which is not uncommon in many machine learning applications. Running full batch gradient descent can be computationally quite costly in such scenarios, since we need to reevaluate the whole training dataset each time we take one step toward the global minimum.

A popular alternative to the batch gradient descent algorithm is **stochastic gradient descent** (**SGD**), which is sometimes also called iterative or online gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all training examples, $\boldsymbol{x}^{(i)}$:

$$\Delta w_j = \frac{2\eta}{n}\sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)x_j^{(i)}$$

we update the parameters incrementally for each training example, for instance:

$$\Delta w_j = \eta\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)x_j^{(i)}, \quad \Delta b = \eta\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)$$

Although SGD can be considered as an approximation of gradient descent, it typically reaches convergence much faster because of the more frequent weight updates. Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient descent, which can also have the advantage that SGD can escape shallow local minima more readily if we are working with nonlinear loss functions, as we will see later in *Chapter 11*, *Implementing a Multilayer Artificial Neural Network from Scratch*. To obtain satisfying results via SGD, it is important to present training data in a random order; also, we want to shuffle the training dataset for every epoch to prevent cycles.

Adjusting the learning rate during training

In SGD implementations, the fixed learning rate, η, is often replaced by an adaptive learning rate that decreases over time, for example:

$$\frac{c_1}{[\text{number of iterations}] + c_2}$$

where c_1 and c_2 are constants. Note that SGD does not reach the global loss minimum but an area very close to it. And using an adaptive learning rate, we can achieve further annealing to the loss minimum.

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

Mini-batch gradient descent

A compromise between full batch gradient descent and SGD is so-called **mini-batch gradient descent**. Mini-batch gradient descent can be understood as applying full batch gradient descent to smaller subsets of the training data, for example, 32 training examples at a time. The advantage over full batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates. Furthermore, mini-batch learning allows us to replace the `for` loop over the training examples in SGD with vectorized operations leveraging concepts from linear algebra (for example, implementing a weighted sum via a dot product), which can further improve the computational efficiency of our learning algorithm.

Since we already implemented the Adaline learning rule using gradient descent, we only need to make a few adjustments to modify the learning algorithm to update the weights via SGD. Inside the `fit` method, we will now update the weights after each training example. Furthermore, we will implement an additional `partial_fit` method, which does not reinitialize the weights, for online learning. In order to check whether our algorithm converged after training, we will calculate the loss as the average loss of the training examples in each epoch. Furthermore, we will add an option to shuffle the training data before each epoch to avoid repetitive cycles when we are optimizing the loss function; via the `random_state` parameter, we allow the specification of a random seed for reproducibility:

```
class AdalineSGD:
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    shuffle : bool (default: True)
        Shuffles training data every epoch if True to prevent
        cycles.
    random_state : int
        Random number generator seed for random weight
        initialization.
```

```
    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    b_ : Scalar
        Bias unit after fitting.
    losses_ : list
        Mean squared error loss function value averaged over all
        training examples in each epoch.


    """
    def __init__(self, eta=0.01, n_iter=10,
                 shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        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

        """
        self._initialize_weights(X.shape[1])
        self.losses_ = []
        for i in range(self.n_iter):
            if self.shuffle:
```

```
                X, y = self._shuffle(X, y)
            losses = []
            for xi, target in zip(X, y):
                losses.append(self._update_weights(xi, target))
            avg_loss = np.mean(losses)
            self.losses_.append(avg_loss)
        return self

    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Shuffle training data"""
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to small random numbers"""
        self.rgen = np.random.RandomState(self.random_state)
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01,
                                   size=m)
        self.b_ = np.float_(0.)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.activation(self.net_input(xi))
        error = (target - output)
        self.w_ += self.eta * 2.0 * xi * (error)
        self.b_ += self.eta * 2.0 * error
        loss = error**2
        return loss
```

```
    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, X):
        """Compute linear activation"""
        return X

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

The `_shuffle` method that we are now using in the `AdalineSGD` classifier works as follows: via the `permutation` function in `np.random`, we generate a random sequence of unique numbers in the range 0 to 100. Those numbers can then be used as indices to shuffle our feature matrix and class label vector.

We can then use the `fit` method to train the `AdalineSGD` classifier and use our `plot_decision_regions` to plot our training results:

```
>>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada_sgd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada_sgd)
>>> plt.title('Adaline - Stochastic gradient descent')
>>> plt.xlabel('Sepal length [standardized]')
>>> plt.ylabel('Petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_sgd.losses_) + 1), ada_sgd.losses_,
...          marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average loss')
>>> plt.tight_layout()
>>> plt.show()
```

The two plots that we obtain from executing the preceding code example are shown in *Figure 2.15*:

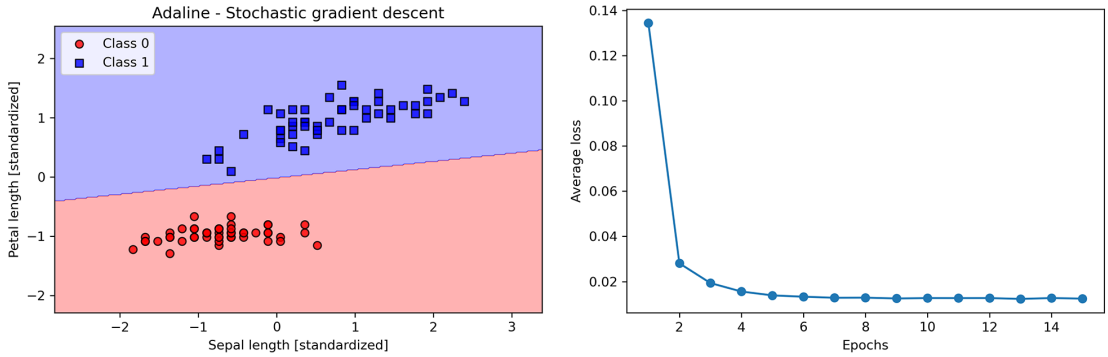

Figure 2.15: Decision regions and average loss plots after training an Adaline model using SGD

As you can see, the average loss goes down pretty quickly, and the final decision boundary after 15 epochs looks similar to the batch gradient descent Adaline. If we want to update our model, for example, in an online learning scenario with streaming data, we could simply call the `partial_fit` method on individual training examples—for instance, `ada_sgd.partial_fit(X_std[0, :], y[0])`.

Summary

In this chapter, we gained a good understanding of the basic concepts of linear classifiers for supervised learning. After we implemented a perceptron, we saw how we can train adaptive linear neurons efficiently via a vectorized implementation of gradient descent and online learning via SGD.

Now that we have seen how to implement simple classifiers in Python, we are ready to move on to the next chapter, where we will use the Python scikit-learn machine learning library to get access to more advanced and powerful machine learning classifiers, which are commonly used in academia as well as in industry.

The object-oriented approach that we used to implement the perceptron and Adaline algorithms will help with understanding the scikit-learn API, which is implemented based on the same core concepts that we used in this chapter: the `fit` and `predict` methods. Based on these core concepts, we will learn about logistic regression for modeling class probabilities and support vector machines for working with nonlinear decision boundaries. In addition, we will introduce a different class of supervised learning algorithms, tree-based algorithms, which are commonly combined into robust ensemble classifiers.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`