Chapter 1

Training Simple Machine Learning Algorithms for Classification

1.1 Implementing a perceptron learning algorithm in Python

1.1.1 An object-oriented perceptron API

If all the weights are initialized to zero, the learning rate parameter, η, affects only the scale of the weight vector, not the direction. Consider a vector,$v_1 = [1\ 2\ 3]$, where the angle between v_1 and a vector, $v_2 = 0.5 \times v_1$, would be exactly zero.

1.2 Adaptive linear neurons and the convergence of learning

We will take a look at another type of single-layer **neural network (NN): ADAptive LInear NEuron (Adaline)**. The Adaline algorithm is particularly interesting because it illustrates the key concepts of defining and minimizing continuous loss functions.

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 $\sigma(z) = z$.

As Figure 1.1 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.

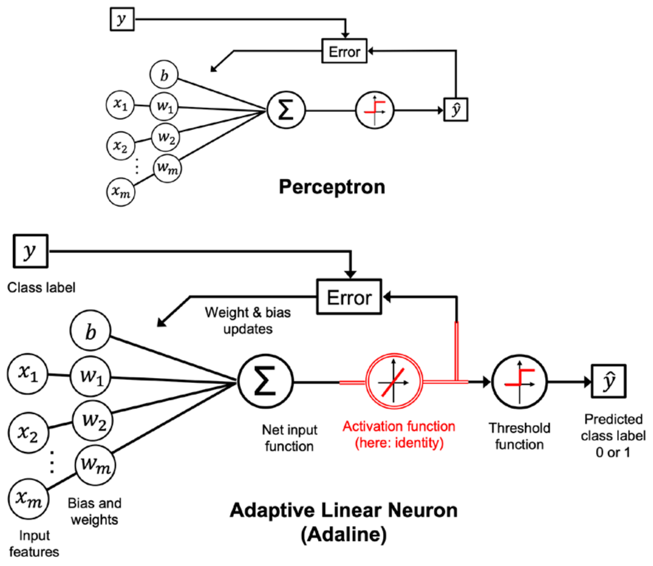

Figure 1.1: A comparison between a perceptron and the Adaline algorithm

Although the Adaline learning rule looks identical to the perceptron rule, we should note that $\sigma(z^{(i)})$ with $\sigma(z^{(i)}) = \mathbf{w}^T\mathbf{x}^{(i)} + b$ 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, we will refer to this process as **full batch gradient descent**.

1.2.1 Improving gradient descent through feature scaling

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.

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).

1.2.2 Large-scale machine learning and stochastic gradient descent

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, $\mathbf{x}^{(i)}$:

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

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

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

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:

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

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.

Chapter 2

A Tour of Machine Learning Classifiers Using Scikit-Learn

2.1 Modeling class probabilities via logistic regression

2.1.1 Logistic regression and conditional probabilities

Under the logistic model, we assume that there is a linear relationship between the weighted inputs and the log-odds:

$$logit(p) = w_1x_1 + w_2x_2 + \cdots + w_mx_m + b = \sum_{i=j} w_jx_j = \mathbf{w}^T\mathbf{x} + b \tag{2.1}$$

2.1.2 Learning the model weights via the logistic loss function

To explain how we can derive the loss function for logistic regression, let's first define the likelihood, $\mathscr{L}$, 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:

$$\begin{aligned}\mathscr{L}(\mathbf{w}, b|\mathbf{x}) = p(y|\mathbf{x}; \mathbf{w}, b) &= \prod_{i=1}^{m} p\left(y^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}, b\right) \\ &= \prod_{i=1}^{m} \left(\sigma(z^{(i)})\right)^{y^{(i)}} \left(1 - \sigma(z^{(i)})\right)^{1-y^{(i)}}\end{aligned} \tag{2.2}$$

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

$$\begin{aligned}l(\mathbf{w}, b|\mathbf{x}) &= \log \mathscr{L}(\mathbf{w}, b|\mathbf{x}) \\ &= \sum_{i=1} \left[y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))\right]\end{aligned} \tag{2.3}$$

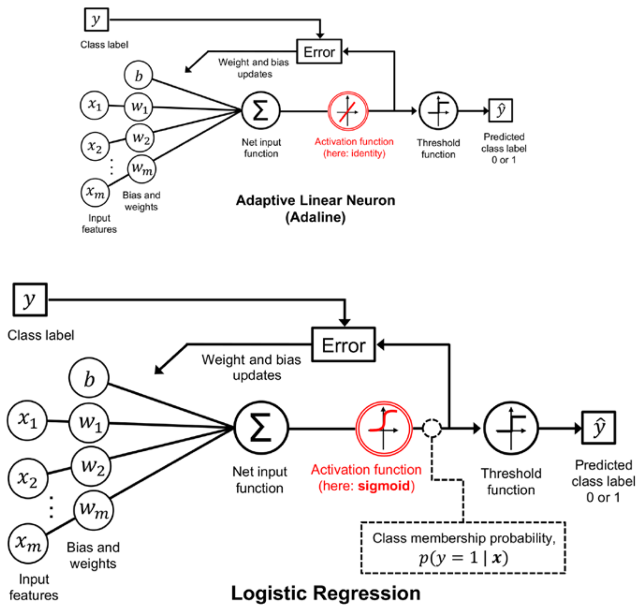

Figure 2.1: Logistic regression compared to Adaline