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.

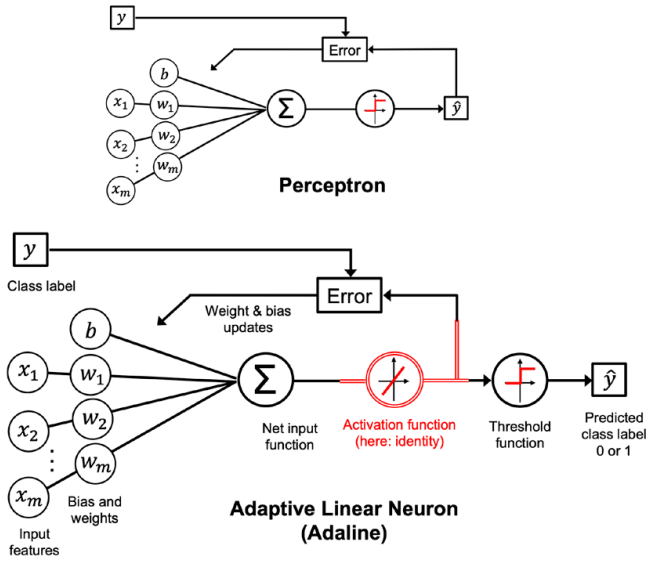

Figure 1.1: A comparison between a perceptron and the Adaline algorithm

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:

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

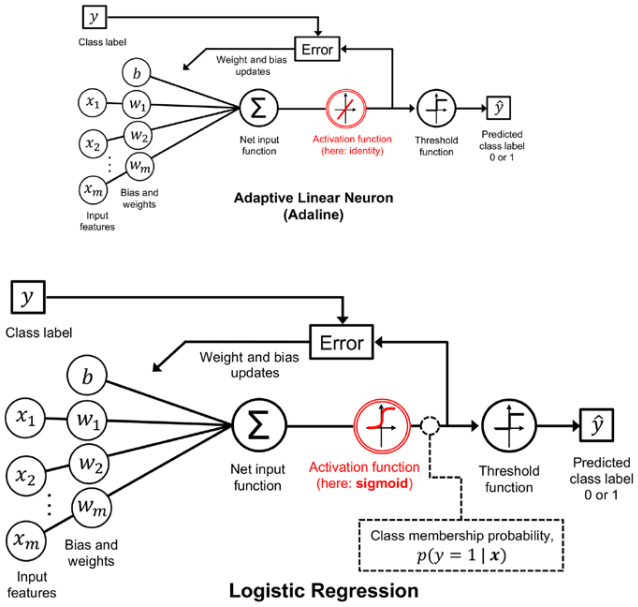

Figure 2.1: Logistic regression compared to Adaline

2.1.3 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.

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.

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 loss function for logistic regression can be regularized by adding a simple

regularization term, which will shrink the weights during model training:

$$L(\mathbf{w}, b|\mathbf{x}) = \sum_{i=1} \left[y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right] + \frac{\lambda}{2n} ||\mathbf{w}||^2 \qquad (2.4)$$

The parameter, C, that is implemented for the LogisticRegression class in scikit-learn comes from a convention in support vector machines. The term C is inversely proportional to the regularization parameter, λ . Consequently, decreasing the value of the inverse regularization parameter, C, means that we are increasing the regularization strength.

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 speople 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.

2.2 Maximum margin classification with support vector machines

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.

2.2.1 Dealing with a nonlinearly separable case using slack variables

The motivation for introducing the slack variable was that the linear constraints in the SVM optimization objective need to be relaxed for nonlinearly separable data to allow the convergence of the optimization in the presence of misclassifications, under appropriate loss penalization.

The use of the slack variable, in turn, introduces the variable, which is commonly referred to as C in SVM contexts. We can consider C as a hyperparameter for controlling the penalty for misclassification. Large values of C correspond to large error

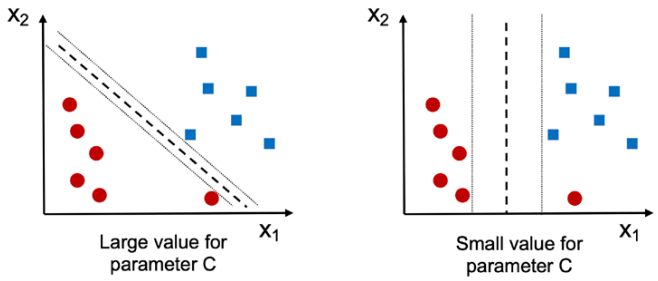

Figure 2.2: The impact of large and small values of the inverse regularization strength C on classification

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 Figure 2.2:

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 of being a simpler model and can be implemented more easily, and is mathematically easier to explain. Furthermore, **logistic regression models can be easily updated, which is attractive when working with streaming data**.

2.2.2 Alternative implementations in scikit-learn

The advantage of using LIBLINEAR and LIBSVM over, for example, 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.

2.3 Solving nonlinear problems using a kernel SVM

2.3.1 Kernel methods for linearly inseparable data

The basic idea behind kernel methods for dealing 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 Figure 3.14, we can transform a two-dimensional dataset into a new

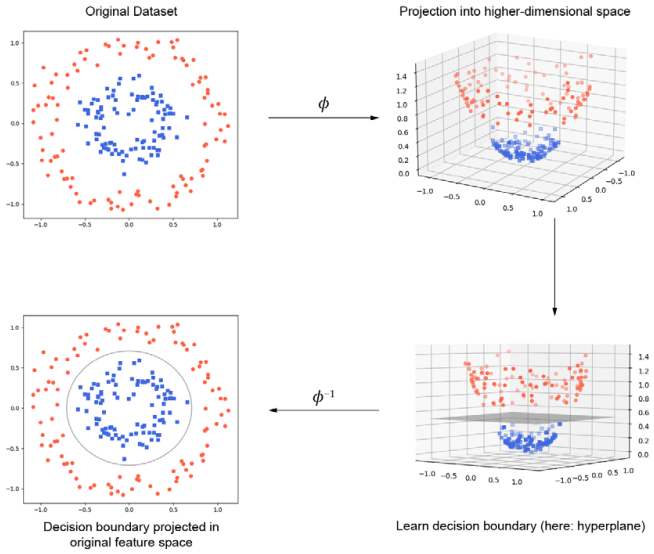

Figure 2.3: The process of classifying nonlinear data using kernel methods

three-dimensional feature space, where the classes become separable via the following projection:

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

2.3.2 Using the kernel trick to find separating hyperplanes in a highdimensional 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.

In practice, we just need to replace the dot product $\mathbf{x}^{(i)T}\mathbf{x}^{(j)}$ by $\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$. To save the expensive step of calculating this dot product between two points explicitly, we define a so-called kernel function:

$$k(\mathbf{x}^{(i)}\mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$$

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).