6 Linear Two-class Classification

6.1 Introduction

In this Chapter we discuss *linear two-class classification*, another kind of *supervised learning* problem. At the outset the difference between this problem and linear regression (detailed in Chapter 5) is very subtle: two-class (or binary) classification is the name we give to a regression problem when the output of a dataset takes on only two *discrete* values, often referred to as two *classes*. Many popular machine learning problems fall into this category, including face detection[1] (and object detection in general), text-based sentiment analysis[2], automatic diagnosis of medical conditions[3], and more.

This subtle difference is important, and spurs the development of new cost functions that are better-suited to deal with such data. These new cost functions are formulated based on a wide array of motivating perspectives including *logistic regression*, *perceptron*, and *support vector machines*. While these perspectives widely differ on the surface they all (as we will see) reduce to virtually the same essential principle for two-class classification.

6.2 Logistic regression and the cross entropy cost

In this Section we describe a fundamental framework for linear two-class classification referred to as *logistic regression* employing the so-called *cross entropy* cost function.

6.2.1 Notation and modeling

Two-class classification is a particular instance of regression wherein the data still comes in the form of P input/output pairs $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$, and each input $\mathbf{x}_p$ is an N dimensional vector. However the corresponding output $\mathbf{x}_p$ is no longer continuous but takes on only two discrete numbers. While the actual value of

[1] The two classes here include face and non-face objects.
[2] The two classes here consist of written product reviews ascribing a positive or negative opinion.
[3] The two classes in this case refer to medical data corresponding to patients who either do or do not have a specific malady.

these numbers is in principle arbitrary, particular value pairs are more helpful than others for derivation purposes. In this Section we suppose that the output of our data takes on either the value 0 or +1, i.e., $y_p \in \{0, +1\}$. Often in the context of classification the output values y_p are called *labels*, and all points sharing the same label value are referred to as a *class* of data. Hence a dataset containing points with label values $y_p \in \{0, +1\}$ is said to be a dataset consisting of two classes.

The simplest way such a dataset can be distributed is on a set of adjacent *steps* as illustrated in the top two panels of Figure 6.1 for $N = 1$ (on the left) and $N = 2$ (on the right). Here the *bottom step* is the region of the space containing class 0, i.e., all of the points that have label value $y_p = 0$. Likewise the *top step* contains class 1, i.e., all of the points having label value $y_p = +1$. From this perspective, the problem of two-class classification can be naturally viewed as a case of *nonlinear* regression where our goal is to regress (or fit) a nonlinear step function to the data. We call this the *regression perspective* on classification.

Alternatively we can change perspective and view the dataset directly from 'above' where we can imagine looking down on the data from a point high up on the y axis. In other words, we look at the data as if it is projected onto the $y = 0$ plane. From this perspective, which is illustrated in the bottom panels of Figure 6.1, we remove the vertical y dimension of the data and visually represent the dataset using its input only, displaying the output values of each point by coloring the points one of two unique colors: we use blue for points with label $y_p = 0$, and red for those having label $y_p = +1$. From this second perspective which we call the *perceptron pespective* and illustrate in the bottom two panels of Figure 6.1, the edge separating the two steps (and consequently the datapoints on them) when projected onto the input space, takes the form of a single point when $N = 1$ (as illustrated in the bottom-left panel) and a *line* when $N = 2$ (as illustrated in the bottom-right panel). For general N what separates the two classes of data will be a *hyperplane*[4] which is also called a *decision boundary* in the context of classification.

In the current Section and the one that follows we focus solely on the regression perspective on two-class classification. We come back to the perceptron perspective again in Section 6.4.

6.2.2 Fitting a discontinuous step function

Adopting the regression perspective on two-class classification, we might be tempted at first to simply apply the linear regression framework described in Chapter 5 to fit such data. We do exactly this in Example 6.1.

Example 6.1 Fitting a linear regressor to classification data

[4] A point and a line are special low dimensional instances of a hyperplane.

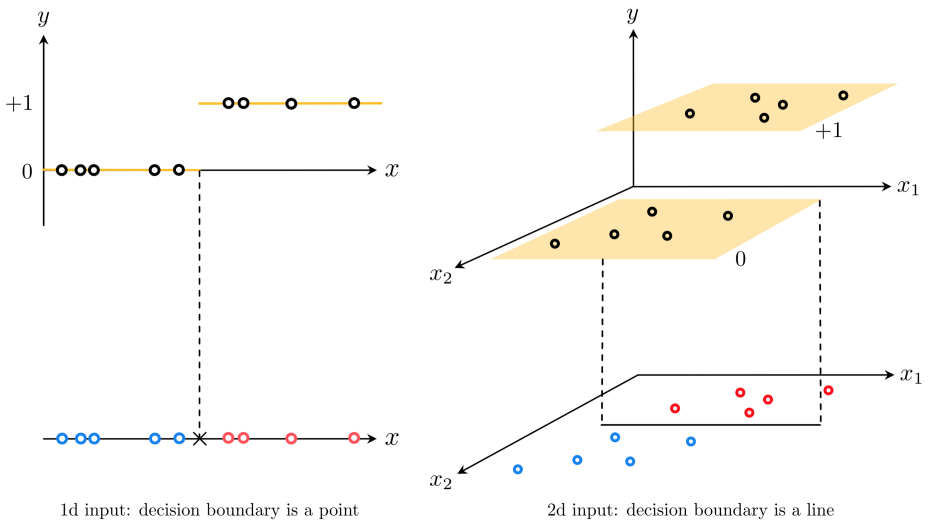

Figure 6.1 Two perspectives on classification illustrated using single-input (left column) and two-input (right column) toy datasets. The regression perspective shown in the top panels is equivalent to the perceptron perspective shown in the bottom panels, where we look at each respective dataset from 'above'. In the perceptron perspective we also mark the decision boundary. This is where the step function (colored in yellow in the top panels) transitions from its bottom to top step. See text for further details.

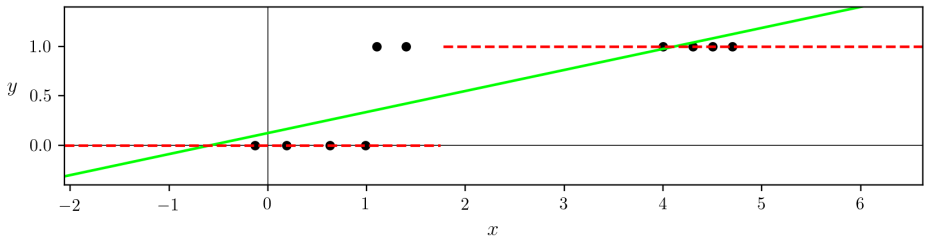

Figure 6.2 Figure associated with Example 6.1. See text for details.

In Figure 6.2 we show a simple two-class dataset where we have fit a line to this dataset via linear regression (shown in green). The line itself provides a poor representation of this data since its output takes on just two discrete values. Even when we pass this fully tuned linear regressor through a discrete step function by assigning the label +1 to all output values greater than 0.5 and the label 0 to all output values less than 0.5, the resulting step function (shown in dashed red in the Figure) still provides a less than ideal representation for the data. This is because the parameters of the (green) line were tuned first (before passing the resulting model through the step) causing the final step model to fail to properly identify two of the points on the top step. In the parlance of classification these types of points are referred to as *misclassified points* or *misclassifications* for short.

Example 6.1 hints at the fact that using linear regression outright to represent classification data is a poor option. Even after passing the resulting linear model through a step function the result still did not capture the true step function on which the data of that example lay. Instead of tuning the parameters of a linear model (performing linear regression) and then passing the result through a step function, we can in principle do a better job if we tune the parameters of the linear model *after* passing it through a step function.

To describe this sort of regression more formally, first recall our notation (introduced in Section 5.1) for denoting a linear model of N dimensional input

$$\mathring{\mathbf{x}}^T\mathbf{w} = w_0 + x_1w_1 + x_2w_2 + \cdots + x_Nw_N \tag{6.1}$$

where

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \quad \text{and} \quad \mathring{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}. \tag{6.2}$$

Next, let us denote a *step function* algebraically as[5]

$$\text{step}(t) = \begin{cases} 1 & \text{if } t > 0 \\ 0 & \text{if } t < 0 \end{cases} \tag{6.3}$$

Shoving our linear model in Equation (6.1) through this gives us a step function[6]

$$\text{step}\left(\mathring{\mathbf{x}}^T\mathbf{w}\right) \tag{6.4}$$

with a *linear decision boundary* between its lower and upper steps, defined by all points $\mathring{\mathbf{x}}$ where $\mathring{\mathbf{x}}^T\mathbf{w} = 0$. Any input lying *exactly* on the decision boundary can be assigned a label at random.

To tune the weight vector $\mathbf{w}$ properly we can (once again, as with linear regression in Chapter 5) setup a Least Squares cost function by reflecting on the sort of ideal relationship we want to find between the input and output of

[5] What happens with step (0) is, for our purposes, arbitrary. It can be set to any fixed value or left undefined as we have done here.

[6] Technically, $\text{step}\left(\mathring{\mathbf{x}}^T\mathbf{w} - 0.5\right)$ is the function that maps output values $\mathring{\mathbf{x}}^T\mathbf{w}$ greater (smaller) than 0.5 to 1 (0). However we can fuse the constant -0.5 into the bias weight w_0 by re-writing it as $w_0 \longleftarrow w_0 - 0.5$ (after all it is a parameter that must be learned) so that the step function can be expressed more compactly, as is done in Equation (6.4).

our dataset. Ideally, we want the point $\left(\mathbf{x}_p,\, y_p\right)$ to lie on the correct side of the optimal decision boundary, or in other words, the output y_p to lie on the proper step. Expressed algebraically, this ideal desire can be written as

$$\text{step}\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) = y_p \qquad p = 1, ..., P. \tag{6.5}$$

To find weights that satisfy this set of P equalities we form a Least Squares cost function by squaring the difference between both sides of each equality, and taking their average

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{step}\left(\mathring{\mathbf{x}}^T\mathbf{w}\right) - y_p\right)^2. \tag{6.6}$$

Our ideal weights then correspond to a minimizer of this cost function.

Unfortunately it is very difficult (if not impossible) to properly minimize this Least Squares cost using local optimization, as at virtually every point the function is *completely flat* locally (see Example 6.2). This problem, which is inherited from our use of the step function, renders both gradient descent and Newton's method ineffective, since both methods immediately halt at flat areas of a cost function.

Example 6.2 **Visual inspection of classification cost functions**

In the left panel of Figure 6.3 we plot the Least Squares cost function in Equation (6.6) for the dataset shown previously in Figure 6.2, over a wide range of values of its two parameters w_0 and w_1. This Least Squares surface consists of discrete steps at many different levels, and each step is *completely flat*. Because of this no local optimization method can be used to effectively minimize it. But all is not lost!

In the middle and right panels of Figure 6.3 we plot the surfaces of two related cost functions over the same dataset. We introduce the cost function shown in the middle panel in Figure 6.3, and the cost in the right panel in Figure 6.3. Both are far superior, in terms of our ability to properly minimize them, than the step-based Least Squares cost function shown on the left.

6.2.3 The logistic sigmoid function

To make the minimization of the Least Squares cost possible we can replace the step function in Equation (6.6) with a *continuous* approximation that matches it closely. The *logistic sigmoid function*

$$\sigma\left(x\right) = \frac{1}{1 + e^{-x}} \tag{6.7}$$

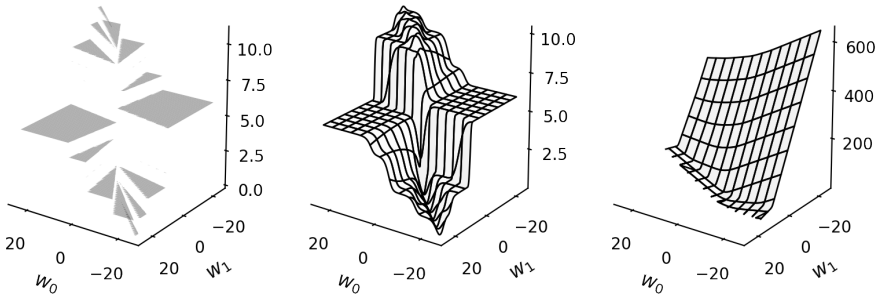

Figure 6.3 Figure associated with Example 6.2. See text for details.

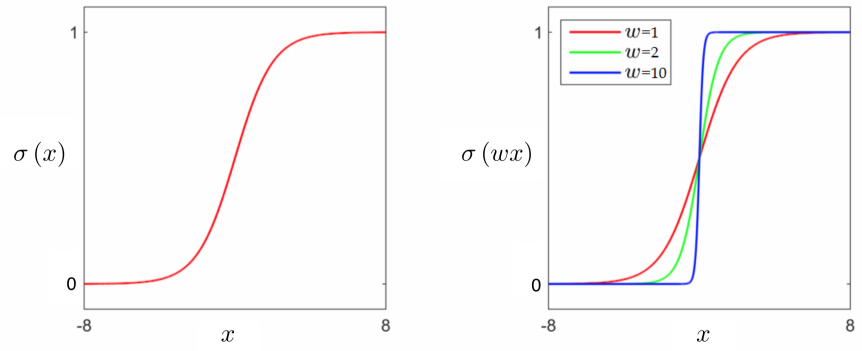

Figure 6.4 (left panel) Plot of the sigmoid function $\sigma(x)$. (right panel) By increasing the weight w in $\sigma(wx)$ from $w = 1$ (shown in red) to $w = 2$ (shown in green) and finally to $w = 10$ (shown in blue), the internally weighted version of the sigmoid function becomes an increasingly good approximator of the step function.

is such an approximation. In Figure 6.4 we plot this function (left panel) as well as several internally weighted versions of it (right panel). As we can see for the correct setting of internal weights the logistic sigmoid can be made to look arbitrarily similar to the step function.

6.2.4 Logistic regression using the Least Squares cost

Swapping out the step function with its sigmoid approximation in Equation (6.5) gives the related set of *approximate* equalities we desire to hold

$$\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) \approx y_p \qquad p = 1, ..., P \tag{6.8}$$

as well as the corresponding Least Squares cost function

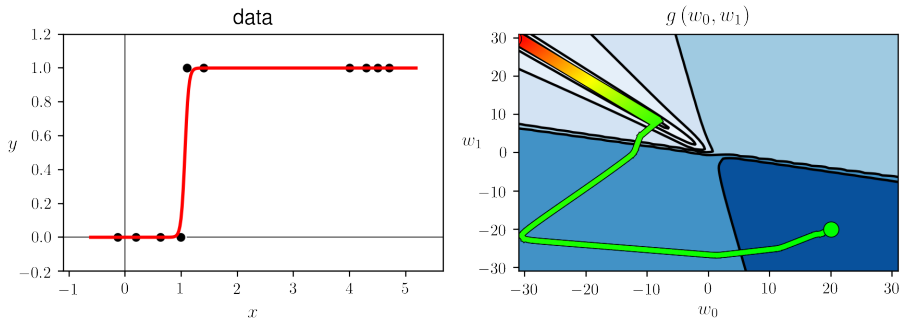

Figure 6.5 Figure associated with Example 6.3. See text for details.

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\sigma\left(\mathring{\mathbf{x}}^T\mathbf{w}\right) - y_p\right)^2 \tag{6.9}$$

Fitting a logistic sigmoid to classification data by minimizing this cost function is often referred to as performing *logistic regression*[7]. While the resulting cost function is generally non-convex, it can be properly minimized nonetheless using a host of local optimization techniques.

Example 6.3 Minimizing Least Squares logistic regression using normalized gradient descent

In Figure 6.5 we show how normalized gradient descent (see Section 3.9) can be used to minimize the Least Squares cost in Equation (6.9) over the dataset first shown in Figure 6.2. Here 6.5 run normalized gradient descent, initialized at the point $w_0 = -w_1 = 20$. Plotted in the left panel is the sigmoidal fit provided by properly minimizing the Least Squares cost. In the right panel a contour plot of the cost function is shown with the (normalized) gradient descent path colored green to red as the run progresses towards the cost function's minimizer. A surface plot of this cost function (in three dimensions) is shown in the middle panel of Figure 6.3. Although this cost function is very flat in many places, normalized gradient descent is designed specifically to deal with costs like this (see Section 3.9).

6.2.5 Logistic regression using the Cross Entropy cost

The *squared error* point-wise cost $g_p(\mathbf{w}) = \left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) - y_p\right)^2$ that we average over all P points in the data to form the Least Squares cost in Equation (6.9) is

[7] Because we are essentially performing *regression* using a *logistic* function.

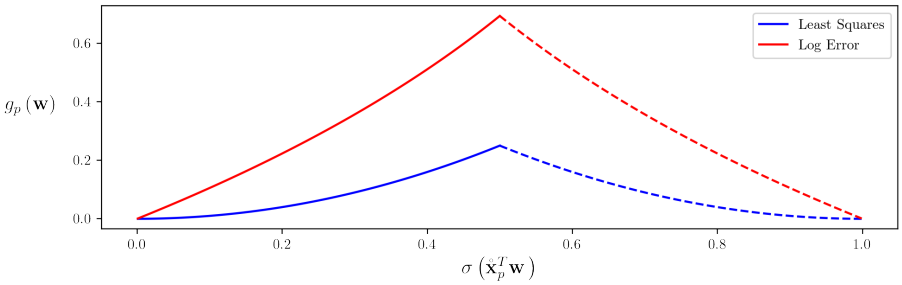

Figure 6.6 Visual comparison of the squared error (in blue) and the log error (in red) for two cases: $y_p = 0$ (solid curves) and $y_p = 1$ (dashed curves). In both cases the log error penalizes deviation from the true label value to a greater extent than the squared error.

universally defined, regardless of the values taken by the output by y_p. However because we *know* that the output we deal with in a two-class classification setting is limited to the *discrete* values $y_p \in \{0, 1\}$, it is reasonable to wonder if we can create a more appropriate cost customized to deal with just such values.

One such point-wise cost, which we refer to as the *log error*, is defined as follows

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = 1 \\ -\log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = 0. \end{cases} \tag{6.10}$$

First, notice that this point-wise cost is always non-negative (regardless of the input and weight values) with a minimum[8] value of 0.

Secondly, notice how this *log error* cost penalizes violations of our desired (approximate) equalities in Equation (6.8) much more harshly than a *squared error* does, as can be seen in Figure 6.6 where both are plotted for comparison.

Finally, notice that since our label values $_p \in \{0, 1\}$ we can write the *log error* in Equation (6.10) equivalently in a single line as

$$g_p(\mathbf{w}) = -y_p \log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) - \left(1 - y_p\right)\log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right). \tag{6.11}$$

This equivalent form allows us to write the overall cost function (formed by taking the average of the point-wise costs over all P datapoints) as

$$g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P} y_p \log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) + \left(1 - y_p\right)\log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) \tag{6.12}$$

This is referred to as the *cross entropy cost* for logistic regression.

[8] Technically, an infimum.

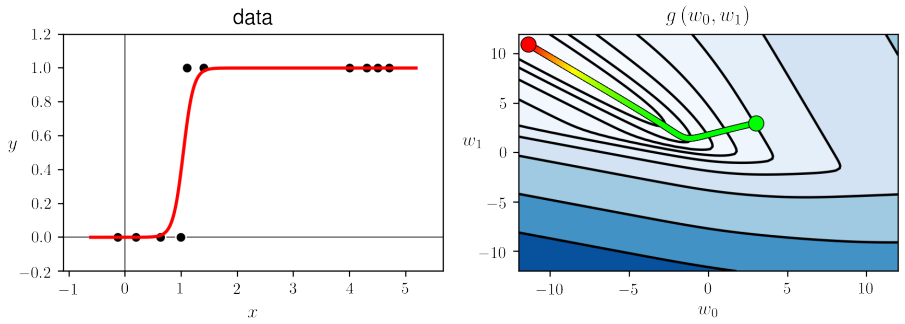

Figure 6.7 Figure associated with Example 6.4. See text for details.

6.2.6 Minimizing the cross entropy cost

The right panel of Figure 6.3 shows the surface of the cross entropy cost taken over the dataset shown in Figure 6.2. That the plotted surface looks convex is not accidental. Indeed (unlike Least Squares) the cross entropy cost is always convex regardless of the dataset used (See Chapter's exercises). This means that a wider variety of local optimization schemes can be used to properly minimize it (compared to the generally non-convex sigmoid-based Least Squares cost) including standard gradient descent schemes (see Section 3.6) and second order Newton's methods (see Section 4.3). For this reason the cross entropy cost is very often used in practice to perform logistic regression.

Example 6.4 **Minimizing cross entropy logistic regression using standard gradient descent**

In this Example we repeat the experiments of Example 6.3 using (instead) the cross entropy cost and standard gradient descent, initialized at the point $w_0 = 3$ and $w_1 = 3$. The left panel of Figure 6.7 shows the sigmoidal fit provided by properly minimizing the cross entropy cost. In the right panel a contour plot of the cost function is shown with the gradient descent path colored green to red as the run progresses towards the cost's minimizer. A surface plot of this cost function is shown in the right panel of Figure 6.3.

6.2.7 `Python` implementation

We can implement the Cross Entropy costs very similarly to the way we did the Least Squares cost for linear regression (see Section 5.1.4) breaking down our implementation into a linear `model` and the error function itself. Our linear `model` takes in both an appended input point $\mathring{\mathbf{x}}_p$ and a set of weights $\mathbf{w}$

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \mathring{\mathbf{x}}_p^T\mathbf{w} \tag{6.13}$$

which we can still implement as shown below.

```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

We can then implement the Cross Entropy cost using the log error in Equation 6.10 as shown below.

```
# define sigmoid function
def sigmoid(t):
    return 1/(1 + np.exp(-t))

# the convex cross-entropy cost function
def cross_entropy(w):
    # compute sigmoid of model
    a = sigmoid(model(x,w))

    # compute cost of label 0 points
    ind = np.argwhere(y == 0)[:,1]
    cost = -np.sum(np.log(1 - a[:,ind]))

    # add cost on label 1 points
    ind = np.argwhere(y==1)[:,1]
    cost -= np.sum(np.log(a[:,ind]))

    # compute cross-entropy
    return cost/y.size
```

To minimize this cost we can use virtually any local optimization method detailed in Chapters 2 - 4. For first and second order methods (e.g., gradient descent and Newton's method schemes) an `autograd` (see Section 3.5) can be used automatically compute its gradient and Hessian.

Alternatively one can indeed compute the gradient and Hessian of the Cross Entropy cost in closed form, and implement them directly. Using the simple derivative rules outlined in Section 7.2.1 gradient can be computed as

$$\nabla g\left(\mathbf{w}\right) = -\frac{1}{P}\sum_{p=1}^{P}\left(y_p - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p \tag{6.14}$$

In addition to employ Newton's method 'by hand' one can hand compute the Hessian of the Cross Entropy function as

$$\nabla^2 g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T. \tag{6.15}$$

6.3 Logistic regression and the Softmax cost

In the previous Section we saw how to derive logistic regression when employing label values $y_p \in \{0, 1\}$. However as mentioned earlier these label values are arbitrary, and one can derive logistic regression using a different set of label values, e.g., $y_p \in \{-1, +1\}$. In this brief Section we do just this, tracing out entirely similar steps to what we have seen previously, resulting in new cost function called the *Softmax cost* for logistic regression. While the Softmax differs in form algebraically, it is in fact equivalent to the cross entropy cost. However conceptually speaking the Softmax cost is considerably valuable, since it allows us to sew together the many different perspectives on two-class classification into a single coherent idea, as we will see in the Sections that follow.

6.3.1 Different labels, same story

If we change the label values from $y_p \in \{0, 1\}$ to $y_p \in \{-1, +1\}$ much of the story we saw unfold previously unfolds here as well. That is, instead of our data ideally sitting on a step function with lower and upper steps taking on the values 0 and 1 respectively, they take on values -1 and $+1$ as shown in Figure 6.8 for prototypical cases where $N = 1$ (left panel) and $N = 2$ (right panel).

This particular step function is called a *sign function*, since it returns the numerical *sign* of its input

$$\text{sign}(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x < 0. \end{cases} \tag{6.16}$$

Shoving a *linear* model through the sign function gives us a step function

$$\text{sign}\left(\mathring{\mathbf{x}}^T\mathbf{w}\right) \tag{6.17}$$

with a *linear* decision boundary between its two steps defined by all points $\mathring{\mathbf{x}}$ where $\mathring{\mathbf{x}}^T\mathbf{w} = 0$. Any input lying *exactly* on the decision boundary can be assigned a label at random. A point is classified correctly when its true label is predicted correctly, that is when $\text{sign}\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) = y_p$. Otherwise, it is said to have been *mislcassified*.

As when using label values $y_p \in \{0, 1\}$, we can again attempt to form a Least Squares cost using the sign function. However, just as with the original step-based Least Squares in Equation (6.6) this too would be completely flat almost everywhere, and therefore extremely difficult to minimize properly.

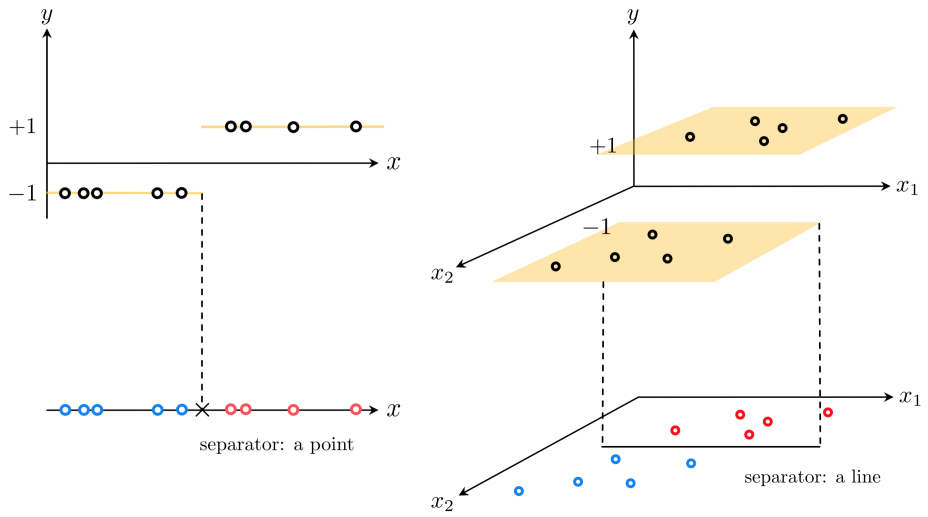

Figure 6.8 The analogous setup to Figure 6.1, only here we use label values $y_p \in \{-1, +1\}$.

Akin to what we did previously, we can look to replace the discontinuous $\text{sign}(\cdot)$ function with a *smooth approximation*: a slightly adjusted version of the *logistic sigmoid* function so that its values range between -1 and 1 (instead of 0 and 1). This scaled version of the sigmoid, called the *hyperbolic tangent function*, is written as

$$\tanh(x) = 2\,\sigma(x) - 1 = \frac{2}{1 + e^{-x}} - 1. \tag{6.18}$$

Given that the sigmoid function $\sigma(\cdot)$ ranges smoothly between 0 and 1, it is easy to see why $\tanh(\cdot)$ ranges smoothly between -1 and $+1$. In Figure 6.9 we plot the tanh function as well as several internally weighted versions of it, which can be made to look arbitrarily similar to the sign function.

Analogous to the Least Squares cost in Equation (6.9), we can form a Least Squares cost for recovering optimal model weights using the $\tanh(\cdot)$ function

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\tanh\left(\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) - y_p\right)^2. \tag{6.19}$$

which is likewise non-convex with undesirable flat regions, requiring specialized local methods for its proper minimization (see Example 6.3).

As with 0/1 labels here too we can employ the point-wise *log error* cost

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = +1 \\ -\log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = -1 \end{cases} \tag{6.20}$$

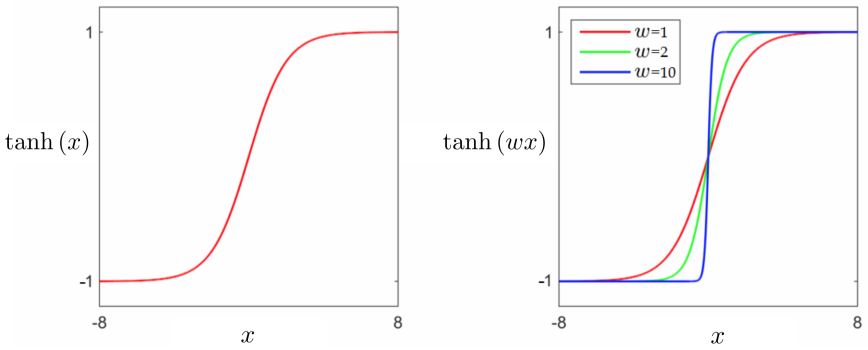

Figure 6.9 (left panel) Plot of the hyperbolic tangent function $\tanh(x)$. (right panel) By increasing the weight w in $\tanh(wx)$ from $w = 1$ (shown in red) to $w = 2$ (shown in green) and finally to $w = 10$ (shown in blue), the internally weighted version of the hyperbolic tangent function becomes an increasingly good approximator of the sign function.

which we can then use to form the so-called *Softmax cost* for logistic regression

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} g_p(\mathbf{w}). \tag{6.21}$$

As with the cross entropy cost it is far more common to express the Softmax cost differently by re-writing the log error in a equivalent way as follows. First, notice that because $1 - \sigma(x) = \sigma(-x)$ the point-wise cost in Equation(6.20) can be re-written equivalently as $-\log\left(\sigma\left(-\mathring{\mathbf{x}}^T\mathbf{w}\right)\right)$ and so the point-wise cost function can be written as

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = +1 \\ -\log\left(\sigma\left(-\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = -1. \end{cases} \tag{6.22}$$

Now notice, because of the particular choice of label values we are using here, i.e., $y_p \in \{-1, +1\}$, that we can move the label value in each case *inside* the inner most parentheses, and write both cases in a single line as

$$g_p(\mathbf{w}) = -\log\left(\sigma\left(y_p\,\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right). \tag{6.23}$$

Finally, since $-\log(x) = \log\left(\frac{1}{x}\right)$, we can write the point-wise cost in Equation (6.23) equivalently (using the definition of the sigmoid) as

$$g_p(\mathbf{w}) = \log\left(1 + e^{-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}}\right) \tag{6.24}$$

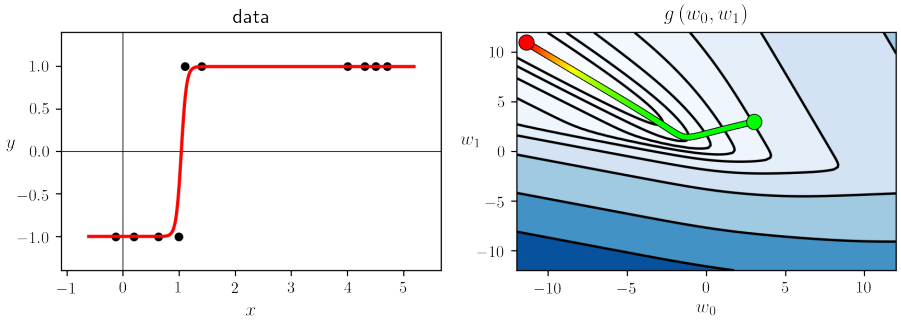

Figure 6.10 Figure associated with Example 6.5. See text for details.

Substituting this form of the point-wise log error function into Equation (6.21) we have a more common appearance of the *Softmax cost* for logistic regression

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) \tag{6.25}$$

This cost function, like the cross entropy cost detailed in the previous Section, is always convex regardless of the dataset used (see Chapter's exercises). Moreover, as we can see here by its derivation, the Softmax and cross entropy cost functions are completely equivalent (upon change of label value $y_p = -1$ to $y_p = 0$ and vice versa), having been built using the same point-wise log error cost function.

Example 6.5 **Minimizing Softmax logistic regression using standard gradient descent**

In this example we repeat the experiments of Example 6.4, swapping out labels $y_p = 0$ with $y_p = -1$, to form the Softmax cost and use gradient descent (with the same initial point, steplength parameter, and number of iterations) for its minimization. The results are shown in Figure 6.10.

6.3.2 Python implementation

If we express the Softmax cost using the Log Error as in Equation 6.21, then we can implement it almost entirely the same way we did the Cross Entropy cost as shown in Section 6.2.7.

To implement the Softmax cost as shown in Equation 6.3.1 we first implement the linear `model`, which takes in both an appended input point $\mathring{\mathbf{x}}_p$ and a set of weights $\mathbf{w}$ as

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \mathring{\mathbf{x}}_p^T \mathbf{w}\,. \tag{6.26}$$

With this notation for our model, the corresponding Softmax cost can be written as

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1 + e^{-y_p \text{model}\left(\mathbf{x}_p, \mathbf{w}\right)}\right).$$

We can then implement the cost in chunks - first the `model` function below precisely as we did with linear regression (see Section 5.1.4)

```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

We can then implement the Softmax cost as shown below.

```
# the convex softmax cost function
def softmax(w):
    cost = np.sum(np.log(1 + np.exp(-y*model(x,w))))
    return cost/float(np.size(y))
```

As alternative to using an automatic differentiator (which we use by default - employing `autograd` (see Section 3.5), one can perform gradient descent and Newton's method here by hand-computing the gradient and Hessian of the Softmax cost function. Using the simple derivative rules outlined in Section 7.2.1 gradient can be computed as

$$\nabla g\left(\mathbf{w}\right) = -\frac{1}{P}\sum_{p=1}^{P} \frac{e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}}{1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}} y_p \mathring{\mathbf{x}}_p \tag{6.27}$$

[9]

In addition to employ Newton's method 'by hand' one can hand compute the Hessian of the Softmax function as

[9] Writing the gradient in this way helps avoid numerical problems associated with using the exponential function on a modern computer. This is due to the exponential 'overflowing' with large exponents, e.g., e^{1000}, as these numbers are too large to store explicitly on the computer and so are represented symbolically as ∞. This becomes a problem when evaluating $\frac{e^{1000}}{1+e^{1000}}$ which, although basically equal to the value 1, is thought of by the computer to be a NaN (not a number) as it thinks $\frac{e^{1000}}{1+e^{1000}} = \frac{\infty}{\infty}$ which is undefined. By writing each summand of the gradient such that it has an exponential in its denominator only, we avoid the problem of dividing two overflowing exponentials.

$$\nabla^2 g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\frac{1}{1+e^{y_p\mathring{\mathbf{x}}_p^T\mathbf{w}}}\right)\left(1-\frac{1}{1+e^{y_p\mathring{\mathbf{x}}_p^T\mathbf{w}}}\right)\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T. \tag{6.28}$$

6.3.3 Noisy classification datasets

The discrete nature of the output in classification makes the concepts of *noise* and *noisy* data different in linear classification than what we saw previously with linear regression in Chapter 5. As described there, in the case of linear regression noise causes the data to *not* fall precisely on a single line (or hyperplane in higher dimensions). With linear two-class classification, noise manifests itself in our *inability* to find a single line (or hyperplane in higher dimensions) to separate the two classes of data. Figure 6.11 shows such a noisy classification dataset consisting of $P = 100$ points, whose two classes can be separated by a line but not perfectly. The left panel of this Figure shows the data in three dimensions (viewed from a regression perspective) along with the trained classifier: a three dimensional hyperbolic tangent function. The right panel shows the same dataset in two dimensions (viewed from a perceptron perspective) along with the learned linear decision boundary. Here the two half-spaces created by the decision boundary are also colored (light blue and light red) according to the class confined within each. As you can see, there are three points in this case (two blue points and one red points) that look like they are on the *wrong side* of our classifier. Such *noisy* points are often misclassified by a trained classifier, meaning that their true label value will not be correctly predicted. Two-class classification datasets typically have noise of this kind, and thus are not often perfectly separable by a linear decision boundary.

6.4 The Perceptron

As we have seen with logistic regression in the previous Section we treat classification as a particular form of nonlinear regression (employing - with the choice of label values $y_p \in \{-1, +1\}$ - a tanh non-linearity). This results in the learning of a proper nonlinear regressor, and a corresponding *linear decision boundary*

$$\mathring{\mathbf{x}}^T\mathbf{w} = 0. \tag{6.29}$$

Instead of learning this decision boundary as a result of a nonlinear regression, the *Perceptron* derivation described in this Section aims at determining this ideal linear decision boundary directly. While we will see how this direct approach leads back to the *Softmax cost function*, and that practically speaking the Perceptron and logistic regression often results in learning *the same linear decision boundary*, the Perceptron's focus on learning the decision boundary directly

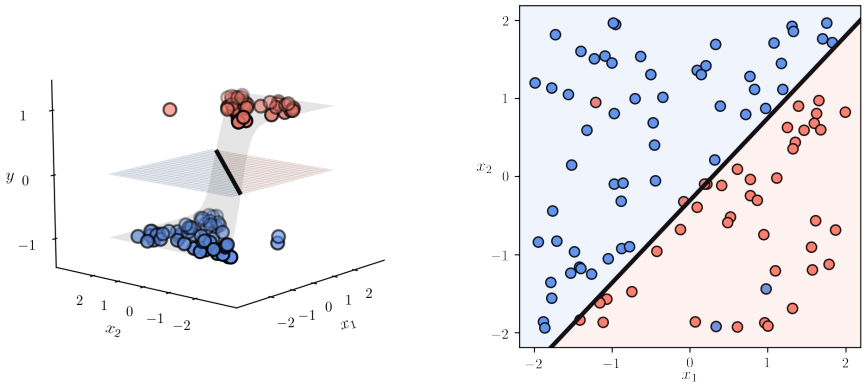

Figure 6.11 A two-class classification dataset viewed from the regression perspective on the left and from the perceptron perspective on the right, with three *noisy* datapoints pointed to by small arrows. See text for further details.

provides a valuable new perspective on the process of two-class classification. In particular - as we will see here - the Perceptron provides a simple geometric context for introducing the important concept of *regularization* (an idea we will see arise in various forms throughout the remainder of the text).

6.4.1 The Perceptron cost function

As we saw in the previous Section with our discussion of logistic regression (where our output / label values $y_p \in \{-1, +1\}$), in the simplest instance our two classes of data are largely separated by a *linear decision boundary* given by the collection of input $\mathbf{x}$ where $\mathring{\mathbf{x}}^T\mathbf{w} = 0$, with each class (mostly) lying on either side. This decision boundary is a *point* when the dimension of the input is $N = 1$, a *line* when $N = 2$, and is more generally for arbitrary N a *hyperplane* defined in the input space of a dataset.

This classification scenario can be best visualized in the case $N = 2$, where we view the problem of classification 'from above' - showing the input of a dataset colored to denote class membership. The default coloring scheme we use here - matching the scheme used in the previous Section - is to color points with label $y_p = -1$ blue and $y_p = +1$ red. The linear decision boundary is here a line that best separates points from the $y_p = -1$ class from those of the $y_p = +1$ class, as shown for a prototypical dataset in Figure 6.12.

The linear decision boundary cuts the input space into two *half-spaces*, one lying 'above' the hyperplane where $\mathring{\mathbf{x}}^T\mathbf{w} > 0$ and one lying 'below' it where $\mathring{\mathbf{x}}^T\mathbf{w} < 0$. Notice then, as depicted visually in Figure 6.12, that a proper set of weights $\mathbf{w}$ define a linear decision boundary that separates a two-class dataset as well as possible with as many members of one class as possible lying above it, and likewise as many members as possible of the other class lying below it.

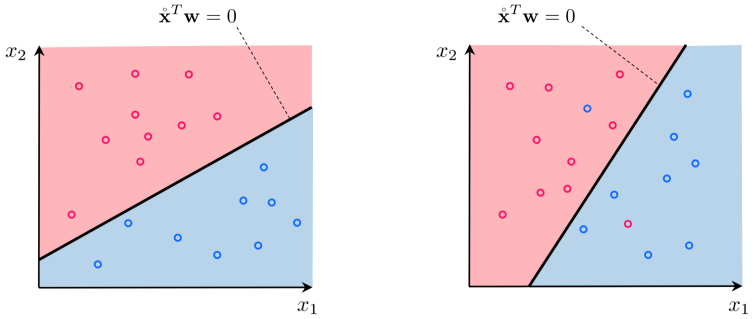

Figure 6.12 With the Perceptron we aim to directly learn the linear decision boundary $\mathring{\mathbf{x}}^T\mathbf{w} = 0$ (shown here in black) to separate two classes of data, colored red (class +1) and blue (class −1), by dividing the input space into a red half-space where $\mathring{\mathbf{x}}^T\mathbf{w} > 0$, and a blue half-space where $\mathring{\mathbf{x}}^T\mathbf{w} < 0$. (left panel) A linearly separable dataset where it is possible to learn a hyperplane to perfectly separate the two classes. (right panel) A dataset with two overlapping classes. Although the distribution of data does not allow for perfect linear separation, the Perceptron still aims to find a hyperplane that minimizes the number of misclassified points that end up in the wrong half-space.

Because we can always flip the orientation of an ideal hyperplane by multiplying it by −1 (or likewise because we can always swap our two label values) we can say in general that when the weights of a hyperplane are tuned properly members of the class $y_p = +1$ lie (mostly)'above' it, while members of the $y_p = -1$ class lie (mostly) 'below' it. In other words, our *desired* set of weights define a hyperplane where as often as possible we have that

$$\begin{aligned} \mathring{\mathbf{x}}_p^T\mathbf{w} &> 0 \quad \text{if} \;\; y_p = +1 \\ \mathring{\mathbf{x}}_p^T\mathbf{w} &< 0 \quad \text{if} \;\; y_p = -1. \end{aligned} \tag{6.30}$$

Because of our choice of label values we can consolidate the ideal conditions above into the single equation below

$$-y_p\mathring{\mathbf{x}}_p^T\mathbf{w} < 0. \tag{6.31}$$

Again we can do so specifically because we chose the label values $y_p \in \{-1, +1\}$. Likewise by taking the maximum of this quantity and zero we can then write this ideal condition, which states that a hyperplane correctly classifies the point $\mathbf{x}_p$, equivalently forming a *point-wise cost*

$$g_p(\mathbf{w}) = \max\left(0, -y_p\mathring{\mathbf{x}}_p^T\mathbf{w}\right) = 0. \tag{6.32}$$

Note that the expression $\max\left(0, -y_p\mathring{\mathbf{x}}_p^T\mathbf{w}\right)$ is always non-negative, since it returns zero if $\mathbf{x}_p$ is classified correctly, and returns a *positive value* if the point

is classified incorrectly. The functional form of this point-wise cost $\max(0, \cdot)$ is often called a *rectified linear unit* for historical reasons (see Section 6.11). Because these point-wise costs are non-negative and equal *zero* when our weights are tuned correctly, we can take their average over the entire dataset to form a proper cost function as

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} \max\left(0, -y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right). \tag{6.33}$$

When minimized appropriately this cost function can be used to recover the ideal weights satisfying the desired equations above as often as possible.

6.4.2 Minimizing the Perceptron cost

This cost function goes by many names such as the *Perceptron* cost, the *rectified linear unit* cost (or ReLU cost for short), and the *hinge cost* (since when plotted a ReLU function looks like a hinge - see Figure 6.13). This cost function is *always convex* but only has a single (discontinuous) derivative in each input dimension. This implies that we can only use zero and first order local optimization schemes (but not Newton's method). Note that the Perceptron cost *always* has a trivial solution at $\mathbf{w} = \mathbf{0}$, since indeed $g(\mathbf{0}) = 0$, thus one may need to take care in practice to avoid finding it (or a point too close to it) accidentally.

6.4.3 The softmax approximation to the Perceptron

Here we describe a common approach to ameliorating the optimization issues detailed above concerning the Perceptron cost. Somewhat akin to our replacement of the discrete step function with a smooth approximating sigmoid function in previous Sections, here we replace the max function portion of the Perceptron cost with a smooth (or at least twice differentiable) alternative that closely matches it everywhere. We do this via the *softmax* function defined as

$$\text{soft}(s_0, s_1, ..., s_{C-1}) = \log\left(e^{s_0} + e^{s_1} + \cdots + e^{s_{C-1}}\right) \tag{6.34}$$

where s_0, s_1, ..., s_{C-1} are any C scalar vaules - which is a generic smooth approximation to the *max* function, i.e.,

$$\text{soft}(s_0, s_1, ..., s_{C-1}) \approx \max(s_0, s_1, ..., s_{C-1}) \tag{6.35}$$

To see why the softmax approximates the max function let us look at the simple case when $C = 2$.

Suppose momentarily that $s_0 \leq s_1$, so that $\max(s_0, s_1) = s_1$. Therefore $\max(s_0, s_1)$ can be written as $\max(s_0, s_1) = s_0 + (s_1 - s_0)$, or equivalently as $\max(s_0, s_1) =$

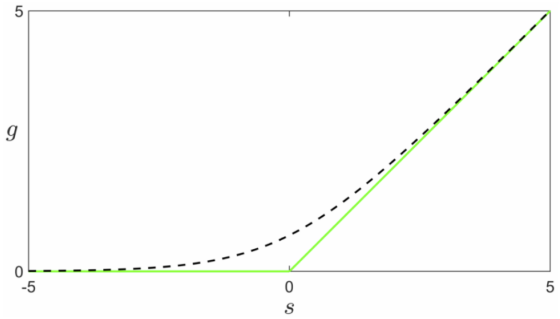

Figure 6.13 Plots of the Perceptron $g(s) = \max(0, s)$ (shown in green) as well as its smooth Softmax approximation $g(s) = \text{soft}(0, s) = \log(1 + e^s)$ (shown in dashed black).

$\log(e^{s_0}) + \log(e^{s_1 - s_0})$. Written in this way we can see that $\log(e^{s_0}) + \log(1 + e^{s_1 s_0}) = \log(e^{s_0} + e^{s_1}) = \text{soft}(s_0, s_1)$ is always larger than $\max(s_0, s_1)$ but not by much, especially when $e^{s_1 - s_0} \gg 1$. Since the same argument can be made if $s_0 \geq s_1$ we can say generally that $\text{soft}(s_0, s_1) \approx \max(s_0, s_1)$. The more general case follows similarly as well.

Returning to the Perceptron cost function in Equation 6.4.1, we replace the p^{th} summand with its Softmax approximation making our point-wise cost

$$g_p(\mathbf{w}) = \text{soft}\left(0, -y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right) = \log\left(e^0 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) = \log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) \tag{6.36}$$

giving the overall cost function as

$$g(\mathbf{w}) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) \tag{6.37}$$

which is the *Softmax cost* we saw previously derived from the logistic regression perspective on two-class classification. This is why the cost function is called *Softmax*, since it derives from the general softmax approximation to the max function.

Note that *like* the Perceptron cost - as we already know - the Softmax cost is convex. However *unlike* the Perceptron cost, the Softmax cost has infinitely many derivatives and Newton's method can therefore be used to minimize it. Moreover, it does not have a trivial solution at zero like the Perceptron cost does. Nonetheless, the fact that the Softmax cost so closely approximates the Perceptron shows just how closely aligned - in the end - both logistic regression and the Perceptron perspectives on classification truly are. Practically speaking their differences lie in how well - for a particular dataset - one can optimize either cost function, along with (what is very often slight) differences in the quality of each cost function's learned decision boundary. Of course when the Softmax

is employed from the Perceptron perspective there is no qualitative difference between the Perceptron and logistic regression at all.

6.4.4 The Softmax cost and linearly separable datasets

Imagine that we have a dataset whose two classes can be perfectly separated by a hyperplane, and that we have chosen an appropriate cost function to minimize it in order to determine proper weights for our model. Imagine further that we are *extremely lucky* and our *initialization* $\mathbf{w}^0$ produces a linear decision boundary $\mathring{\mathbf{x}}^T\mathbf{w}^0 = 0$ with *perfect separation*. This means, according to Equation (6.4.1), that for each of our P points we have that $-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}^0 < 0$ and likewise that the point-wise Perceptron cost in Equation is zero for every point i.e., $g_p\left(\mathbf{w}^0\right) = \max\left(0, -y_p\mathring{\mathbf{x}}_p^T\mathbf{w}^0\right) = 0$ and so the Perceptron cost in Eequation 6.4.1 is *exactly equal to zero*.

Since the Perceptron cost value is already zero, its lowest value, any local optimization algorithm will halt immediately (that is, we would never take a single optimization step). However this will *not* be the case if we used the same initialization but employed the Softmax cost instead of the Perceptron.

Since we always have that $e^{-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}^0} > 0$, the Softmax point-wise cost is always non-negative $g_p\left(\mathbf{w}^0\right) = \log\left(1 + e^{-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}^0}\right) > 0$ and hence too the Softmax cost. This means that we in applying any local optimization scheme like e.g., gradient descent we will indeed take steps away from the initialization $\mathbf{w}^0$ in order to drive the value of the Softmax cost lower and lower towards its minimum at zero. In fact - with data that is indeed linearly separable - the Softmax cost achieves this lower bound *only when the magnitude of the weights grows to infinity*. This is clear from the fact each individual term $\log\left(1 + e^{-C}\right) = 0$ only as $C \longrightarrow \infty$. Indeed if we multiply our initialization $\mathbf{w}^0$ by any constant $C > 1$ we can *decrease* the value of any negative exponential involving one of our data points since $e^{-C} < 1$ and so $e^{-y_p\mathring{\mathbf{x}}_p^T C\mathbf{w}^0} = e^{-C}e^{-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}^0} < e^{-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}^0}$.

This likewise decreases the Softmax cost as well with the minimum achieved only as $C \longrightarrow \infty$. However, importantly, regardless of the scalar $C > 1$ value involved the decision boundary defined by the initial weights $\mathring{\mathbf{x}}^T\mathbf{w}^0 = 0$ *does not change location*, since we still have that $C\,\mathring{\mathbf{x}}^T\mathbf{w}^0 = 0$ (indeed this is true for any non-zero scalar C). So even though the location of the separating hyperplane need not change, with the Softmax cost we still take more and more steps in minimization since (in the case of perfectly linearly separable data) its minimum lies off at infinity. This fact can cause severe numerical instability issues with local optimization schemes that make *large progress* at each step - particularly Newton's method (see Section 4.3) - since they will tend to rapidly diverge to infinity[10].

[10] Notice: because the Softmax and Cross Entropy costs are equivalent (as discussed in the previous Section), this issue equally presents itself when using the Cross Entropy cost as well.

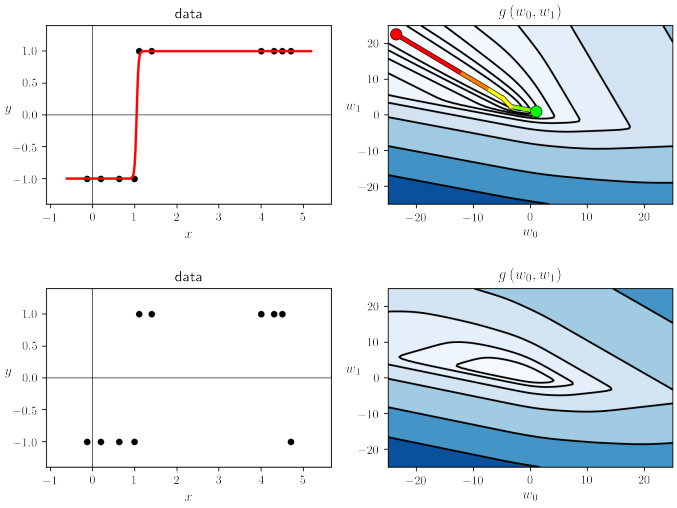

Figure 6.14 (top row) Figure associated with Example 6.6. See text for details.

Example 6.6 **Perfectly separable data and the Softmax cost**

In applying Newton's method to minimize the Softmax cost over perfectly linearly separable data it is easy to run into numerical instability issues as the global minimum of the cost technically lies at infinity. Here we examine a simple instance of this behavior using the single input dataset shown Figure 6.10. Here in the top row of Figure 6.14 we illustrate the progress of 5 Newton steps in beginning at the point $\mathbf{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Within 5 steps we have reached a point providing a very good fit to the data (in the top-left panel of the Figure we plot the $\tanh(\cdot)$ fit using the logistic regressoion perspective on the Softmax cost), and one that is already quite large in magnitude (as can be seen in the top-right panel of the Figure, where the contour plot of the cost function is shown). We can see here by the trajectory of the steps in the right panel, which are traveling linearly towards the minimum out at $\begin{bmatrix} -\infty \\ \infty \end{bmatrix}$, that the location of the linear decision boundary (here a point) is not changing after the first step or two. In other words, after the first few steps we each subsequent step is simply multiplying its predecessor by a scalar value $C > 1$.

Notice that if we simply flip one of the labels - making this dataset not perfectly linearly separable - the corresponding cost function does not have a global minimum out at infinity, as illustrated in the contour plot shown in the bottom row of Figure 6.14.

6.4.5 Normalizing feature-touching weights

How can we prevent this potential problem when employing the Softmax / Cross Entropy cost? One approach can be to employ our local optimization schemes more carefully by e.g., taking fewer steps and / or halting optimization if the magnitude of the weights grows larger than a large pre-defined constant (this is called *early-stopping*). Another approach is to control the magnitude of the weights during the optimization procedure itself. Both approaches are generally referred to in the jargon of machine learning as *regularization strategies*. The former strategy is straightforward, requiring slight adjustments to the way we have typically employed local optimization, but the latter approach requires some further explanation which we now provide.

To control the magnitude of $\mathbf{w}$ means that we want to control the size of the $N + 1$ individual weights it contains

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_N \end{bmatrix}. \tag{6.38}$$

We can do this by *directly* controlling the size of just N of these weights, and it is particularly convenient to do so using the final N feature-touching weights $w_1, w_2, ..., w_N$ because these define the *normal vector* to the linear decision boundary $\mathring{\mathbf{x}}^T\mathbf{w} = 0$. To more easily introduce the geometric concepts that follow we will use our bias / feature weight notation for $\mathbf{w}$ first introduced in Section 5.1.4. This provides us with individual notation for the bias and feature-touching weights as

$$\text{(bias): } b = w_0 \quad \text{(feature-touching weights): } \quad \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{6.39}$$

With this notation we can express a linear decision boundary as

$$\mathring{\mathbf{x}}^T\mathbf{w} = b + \mathbf{x}^T\boldsymbol{\omega} = 0. \tag{6.40}$$

To see why this notation is useful first note how - geometrically speaking - the feature-touching weights $\boldsymbol{\omega}$ define the *normal vector of the linear decision boundary*. The *normal vector* to a hyperplane (like our decision boundary) is always *perpendicular* to it - as illustrated in Figure 6.15. We can always compute the error - also called the signed distance - of a point $\mathbf{x}_p$ to a linear decision boundary in terms of the normal vector $\boldsymbol{\omega}$.

To see how this is possible, imagine we have a point $\mathbf{x}_p$ lying 'above' the linear

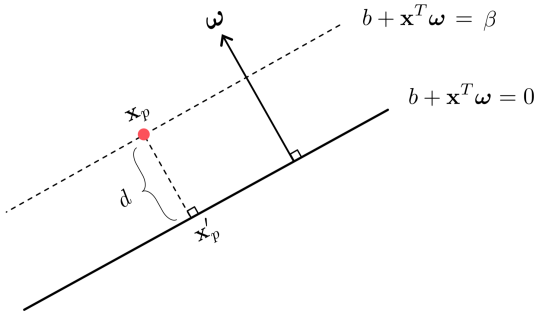

Figure 6.15 A linear decision boundary written as $b + \mathbf{x}^T\boldsymbol{\omega} = 0$ has a normal vector $\boldsymbol{\omega}$ defined by its feature-touching weights. To compute the signed distance of a point $\mathbf{x}_p$ to the boundary we mark the translation of the decision boundary passing through this point as $b + \mathbf{x}^T\boldsymbol{\omega} = \beta$, and the projection of the point onto the decision boundary as $\mathbf{x}'_p$.

decision boundary on a translate of the decision boundary where $b+\mathbf{x}^T\boldsymbol{\omega} = \beta > 0$, as illustrated in the Figure 6.15 (the same simple argument that follows can be made if $\mathbf{x}_p$ lies 'below' it as well). To compute the distance of $\mathbf{x}_p$ to the decision boundary imagine we know the location of its *vertical projection* onto the decision boundary which will call $\mathbf{x}'_p$. To compute our desired error we want to compute the signed distance between $\mathbf{x}_p$ and its vertical projection, i.e., the length of the vector $\mathbf{x}'_p - \mathbf{x}_p$ times the sign of β which here is +1 since we assume the point lies above the decision boundary, i.e., $d = \left\|\mathbf{x}'_p - \mathbf{x}_p\right\|_2 \text{sign}(\beta) = \left\|\mathbf{x}'_p - \mathbf{x}_p\right\|_2$. Now, because *this vector is also perpendicular* to the decision boundary (and so is *parallel* to the normal vector $\boldsymbol{\omega}$) the *inner-product rule* (see Section 7.9) gives

$$\left(\mathbf{x}'_p - \mathbf{x}_p\right)^T \boldsymbol{\omega} = \left\|\mathbf{x}'_p - \mathbf{x}_p\right\|_2 \left\|\boldsymbol{\omega}\right\|_2 = d \left\|\boldsymbol{\omega}\right\|_2. \tag{6.41}$$

Now if we take the difference between our decision boundary and its translation evaluated at $\mathbf{x}'_p$ and $\mathbf{x}_p$ respectively, we have simplifying

$$\beta - 0 = \left(b + \left(\mathbf{x}'_p\right)^T \boldsymbol{\omega}\right) - \left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right) = \left(\mathbf{x}'_p - \mathbf{x}_p\right)^T \boldsymbol{\omega}. \tag{6.42}$$

Since both formulae are equal to $\left(\mathbf{x}'_p - \mathbf{x}_p\right)^T \boldsymbol{\omega}$ we can set them equal to each other, which gives

$$d \left\|\boldsymbol{\omega}\right\|_2 = \beta \tag{6.43}$$

or in other words that the signed distance d of $\mathbf{x}_p$ to the decision boundary is

$$d = \frac{\beta}{\left\|\boldsymbol{\omega}\right\|_2} = \frac{b + \mathbf{x}_p^T\boldsymbol{\omega}}{\left\|\boldsymbol{\omega}\right\|_2}. \tag{6.44}$$

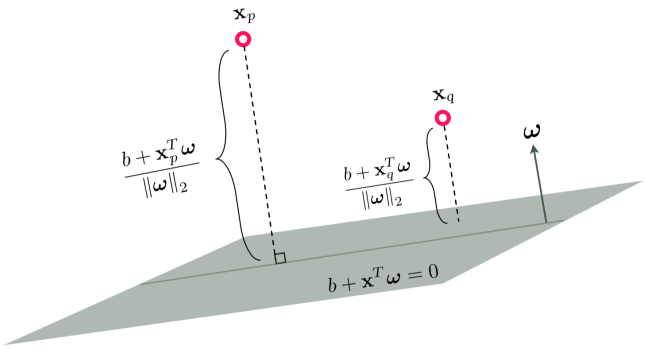

Figure 6.16 Visual representation of the distance to the hyperplane $b + \mathbf{x}^T\boldsymbol{\omega}$, of two points $\mathbf{x}_p$ and $\mathbf{x}_q$ lying above it.

Note that we need not worry dividing by zero here since if the feature-touching weights $\boldsymbol{\omega}$ were all zero, this would imply that the bias $b = 0$ as well and we have no decision boundary at all. Also notice, this analysis implies that if the feature-touching weights have *unit length* as $\|\boldsymbol{\omega}\|_2 = 1$ then the signed distance d of a point $\mathbf{x}_p$ to the decision boundary is given *simply by its evaluation* $b + \mathbf{x}_p^T\boldsymbol{\omega}$. Finally note that if $\mathbf{x}_p$ were to lie below the decision boundary and $\beta < 0$ nothing about the final formulae derived above will change.

We mark this point-to-decision-boundary distance on points in the Figure 6.16, here the input dimension $N = 3$ and the decision boundary is a true hyperplane.

Remember, as detailed above, we can scale any linear decision boundary by a non-zero scalar C and it still defines the same hyperplane. So if - in particular - we multiply by $C = \frac{1}{\|\boldsymbol{\omega}\|_2}$ we have

$$\frac{b + \mathbf{x}^T\boldsymbol{\omega}}{\|\boldsymbol{\omega}\|_2} = \frac{b}{\|\boldsymbol{\omega}\|_2} + \mathbf{x}^T\frac{\boldsymbol{\omega}}{\|\boldsymbol{\omega}\|_2} = 0 \tag{6.45}$$

we do not change the nature of our decision boundary and now our feature-touching weights have unit length as $\left\|\frac{\boldsymbol{\omega}}{\|\boldsymbol{\omega}\|_2}\right\|_2 = 1$. In other words, regardless of how large our weights $\mathbf{w}$ were to begin with we can always normalize them in a consistent way by dividing off the magnitude of $\boldsymbol{\omega}$.

6.4.6 Regularizing two-class classification

The normalization scheme described above is particularly useful in the context of the technical issue with the Softmax / Cross entropy highlighted above because clearly a decision boundary that perfectly separates two classes of data *can be feature-weight normalized* to prevent its weights from growing too large (and diverging too infinity). Of course we do not want to wait to perform this normalization until *after* we run our local optimization, since this will not prevent the

magnitude of the weights from potentially diverging, but *during* optimization. We can achieve this by *constraining* the Softmax / Cross-Entropy cost so that feature-touching weights always have length one i.e., $\|\boldsymbol{\omega}\|_2 = 1$. Formally this minimization problem (employing the Softmax cost) can be phrased as follows

$$\begin{aligned} &\underset{b,\boldsymbol{\omega}}{\text{minimize}} \quad \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b+\mathbf{x}_p^T\boldsymbol{\omega}\right)}\right) \\ &\text{subject to} \quad \|\boldsymbol{\omega}\|_2^2 = 1. \end{aligned} \tag{6.46}$$

By solving this *constrained* version of the Softmax cost we can still learn a decision boundary that perfectly separates two classes of data, but we avoid divergence in the magnitude of the weights by keeping their magnitude *feature-weight normalized*. This formulation can indeed be solved by simple extensions of the local optimization methods detailed in Chapters 2 - 4 (see this Chapter's exercises for further details). However a more popular approach in the machine learning community is to 'relax' this constrained formulation and instead solve the highly related unconstrained *regularized* version of the original Softmax cost. This relaxed form of the problem consists in minimizing a cost function that is a linear combination of our original Softmax cost and the magnitude of the feature weights

$$g\left(b,\boldsymbol{\omega}\right) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b+\mathbf{x}_p^T\boldsymbol{\omega}\right)}\right) + \lambda\ \|\boldsymbol{\omega}\|_2^2 \tag{6.47}$$

which we can minimize using any of our familiar local optimization schemes. Here the term $\|\boldsymbol{\omega}\|_2^2$ is referred to as a *regularizer*, with the parameter $\lambda \geq 0$ being called a *regularization parameter*. The parameter λ is used to balance how strongly we pressure one term or the other in minimizing their sum. In minimizing the first term, our Softmax cost, we are still looking to learn an excellent linear decision boundary. In also minimizing the second term, the magnitude of the feature-touching weights, we incentivize the learning of *small* weights. This prevents the divergence of their magnitude since if their size does start to grow our entire cost function 'suffers' because of it, and becomes large. Because of this the value of λ is typically chosen to be small (and positive) in practice, although some fine-tuning can be useful.

Example 6.7 **The regularized Softmax cost**

Here we repeat the experiment of the previous Example, but add a regularizer with $\lambda = 10^{-3}$ to the Softmax cost as in Equation 6.4.6. In the right panel of Figure 6.17 we show the contour plot of the regularized cost, and we can see (in the contour plot shown in the right panel of the Figure) its global minimum no

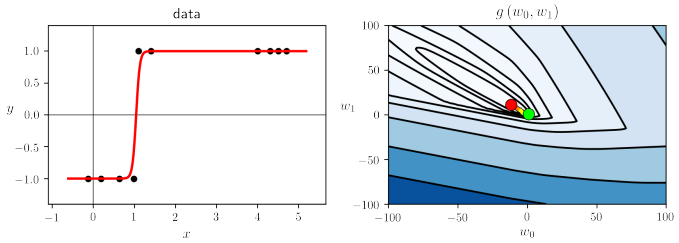

Figure 6.17 Figure associated with Example 6.7. See text for details.

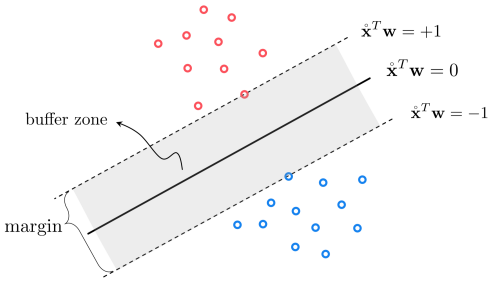

Figure 6.18 For linearly separable data the width of the buffer zone (in gray) confined between two evenly spaced translates of a separating hyperplane that just touch each respective class, defines the margin of that separating hyperplane.

longer lies at infinity. However we still learn a perfect decision boundary as illustrated in the left panel by a tightly fitting $\tanh(\cdot)$ function.

6.5 Support vector machines

In this Section we describe *Support Vector Machines*, or SVMs for short. This approach provides interesting theoretical insight into the two-class classification process - particularly under the assumption that the data is perfectly linearly separable. However we will see that in the more realistic scenario when data is not perfectly separable the Support Vector Machines approach does not provide a learned decision boundary that substantially differs from those provided by logistic regression or the Perceptron.

6.5.1 The Margin-Perceptron

Suppose once again that we have a two-class classification training dataset of P points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ with the labels $y_p \in \{-1, +1\}$. Also suppose for the time being that we are dealing with a two-class dataset that is perfectly linearly separable with a known linear decision boundary $\mathring{\mathbf{x}}^T\mathbf{w} = 0$ like the one illustrated in Figure 6.18.

This separating hyperplane creates a buffer between the two classes confined

between two evenly shifted versions of itself: one version that lies on the positive side of the separator and just touches the class having labels $y_p = +1$ (colored red) taking the form $\mathring{\mathbf{x}}^T\mathbf{w} = +1$, and one lying on the negative side of it just touching the class with labels $y_p = -1$ (colored blue) taking the form $\mathring{\mathbf{x}}^T\mathbf{w} = -1$. The translations above and below the separating hyperplane are more generally defined as $\mathring{\mathbf{x}}^T\mathbf{w} = +\beta$ and $\mathring{\mathbf{x}}^T\mathbf{w} = -\beta$ respectively, where $\beta > 0$. However by dividing off β in both equations and reassigning the variables as $\mathbf{w} \longleftarrow \frac{\mathbf{w}}{\beta}$ we can leave out the redundant parameter β and have the two translations as stated $\mathring{\mathbf{x}}^T\mathbf{w} = \pm 1$.

The fact that all points in the +1 class lie exactly on or on the positive side of $\mathring{\mathbf{x}}^T\mathbf{w} = +1$, and all points in the -1 class lie exactly on or on the negative side of $\mathring{\mathbf{x}}^T\mathbf{w} = -1$ can be written formally as the following conditions

$$\begin{array}{ll} \mathring{\mathbf{x}}^T\mathbf{w} \geq 1 & \text{if } \ y_p = +1 \\ \mathring{\mathbf{x}}^T\mathbf{w} \leq -1 & \text{if } \ y_p = -1 \end{array} \tag{6.48}$$

which is a generalization of the conditions which led to the Perceptron cost in Equation 6.4.1.

We can combine these conditions into a single statement by multiplying each by their respective label values, giving the single inequality $y_p\,\mathring{\mathbf{x}}^T\mathbf{w} \geq 1$ which can be equivalently written as a point-wise cost

$$g_p(\mathbf{w}) = \max\left(0,\, 1 - y_p\,\mathring{\mathbf{x}}^T\mathbf{w}\right) = 0 \tag{6.49}$$

Again, this value is always non-negative. Summing up all P equations of the form above gives the *Margin-Perceptron* cost

$$g(\mathbf{w}) = \sum_{p=1}^{P} \max\left(0,\, 1 - y_p\,\mathring{\mathbf{x}}^T\mathbf{w}\right) \tag{6.50}$$

Notice the striking similarity between the original Perceptron cost from the previous Section and the margin Perceptron cost above: naively we have just 'added a 1' to the non-zero input of the max function in each summand. However this additional 1 prevents the issue of a trivial zero solution with the original Perceptron discussed previously, which simply does not arise here.

If the data is indeed perfectly linearly separable any hyperplane passing between the two classes will have parameters $\mathbf{w}$ where $g(\mathbf{w}) = 0$. However the Margin-Perceptron is still a valid cost function even if the data is not linearly separable. The only difference is that with such a dataset we can not make the criteria above hold for all points in the dataset. Thus a violation for the p^{th} point adds the positive value of $1 - y_p\,\mathring{\mathbf{x}}^T\mathbf{w}$ to the cost function.

6.5.2 Relation to the Softmax cost

As with the Perceptron, we can smooth out the Margin-Perceptron by replacing the max operator with softmax (see Section 6.4.3). Doing so in one summand of the Margin-Perceptron gives the related summand

$$\text{soft}\left(0, 1 - y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right) = \log\left(1 + e^{1 - y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) \tag{6.51}$$

Right away if we were to sum over all P we could form a Softmax-like cost function that closely matches the Margin-Perceptron. But note how in the derivation of the margin Perceptron above the '1' used in the $1 - y_p\left(\mathring{\mathbf{x}}_p^T \mathbf{w}\right)$ component of the cost could have been chosen to be any number we desire. Indeed we chose the value '1' out of convenience. Instead we could have chosen any value $\epsilon > 0$ in which case the set of P conditions stated in Equation 6.5.1 would be equivalently stated as

$$\max\left(0, \epsilon - y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right) = 0 \tag{6.52}$$

for all p and the Margin-Perceptron equivalently stated as

$$g(\mathbf{w}) = \sum_{p=1}^{P} \max\left(0, \epsilon - y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right) \tag{6.53}$$

and, finally, the softmax version of one summand here being

$$\text{soft}\left(0, \epsilon - y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right) = \log\left(1 + e^{\epsilon - y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) \tag{6.54}$$

When ϵ is quite small we of course have that $\log\left(1 + e^{\epsilon - y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) \approx \log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right)$, the same summand used in the Softmax cost. Thus we can, roughly speaking, interpret Softmax cost function as a smoothed version of our Margin-Perceptron cost as well.

6.5.3 Maximum margin decision boundaries

When two classes of data are perfectly linearly separable infinitely many hyperplanes perfectly divide up the data. In Figure 6.19 we display two such hyperplanes for a prototypical perfectly separable dataset. Given that both classifiers (as well as any other decision boundary perfectly separating the data) would perfectly classify this dataset, is there one that we can say is the 'best' of all possible separating hyperplanes?

One reasonable standard for judging the quality of these hyperplanes is via their margin lengths, that is the distance between the evenly spaced translates that just touch each class. The larger this distance is the intuitively better the associated hyperplane separates the entire space given the particular distribution of the data. This idea is illustrated pictorially in the Figure. In this illustration

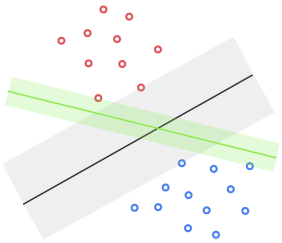

Figure 6.19 Infinitely many linear decision boundaries can perfectly separate a dataset like the one shown here, where two linear decision boundaries are shown in green and black. The decision boundary with the maximum margin - here the one shown in black - is intuitively the best choice. See text for further details.

two separators are shown along with their respective margins. While both perfectly distinguish between the two classes the green separator (with smaller margin) divides up the space in a rather awkward fashion given how the data is distributed, and will therefore tend to more easily misclassify future datapoints. On the other hand, the black separator (having a larger margin) divides up the space more evenly with respect to the given data, and will tend to classify future points more accurately.

In our venture to recover the maximum margin separating decision boundary, it will be convenient to use our individual notation for the bias and feature-touching weights (used in e.g., Section 6.4.5)

$$\text{(bias): } b = w_0 \quad \text{(feature-touching weights): } \quad \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{6.55}$$

With this notation we can express a linear decision boundary as

$$\mathring{\mathbf{x}}^T\mathbf{w} = b + \mathbf{x}^T\boldsymbol{\omega} = 0. \tag{6.56}$$

To find the separating hyperplane with maximum margin we aim to find a set of parameters so that the region defined by $b + \mathbf{x}^T\boldsymbol{\omega} = \pm 1$, with each translate just touching one of the two classes, has the largest possible margin. As depicted in Figure 6.20 the margin can be determined by calculating the distance between any two points (one from each translated hyperplane) both lying on the normal vector $\boldsymbol{\omega}$. Denoting by $\mathbf{x}_1$ and $\mathbf{x}_2$ the points on vector $\boldsymbol{\omega}$ belonging to the *positive* and *negative* translated hyperplanes, respectively, the margin is computed simply as the length of the line segment connecting $\mathbf{x}_1$ and $\mathbf{x}_2$, i.e., $\|\mathbf{x}_1 - \mathbf{x}_2\|_2$.

The margin can be written much more conveniently by taking the difference of the two translates evaluated at $\mathbf{x}_1$ and $\mathbf{x}_2$ respectively, as

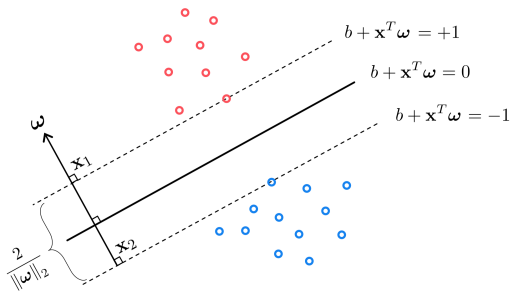

Figure 6.20 The margin of a separating hyperplane can be calculated by measuring the distance between the two points of intersection of the normal vector $\boldsymbol{\omega}$ and the two equidistant translations of the hyperplane. This distance can be shown to have the value of $\frac{2}{\|\boldsymbol{\omega}\|_2}$ (see text for further details).

$$\left(w_0 + \mathbf{x}_1^T\mathbf{w}\right) - \left(w_0 + \mathbf{x}_2^T\mathbf{w}\right) = \left(\mathbf{x}_1 - \mathbf{x}_2\right)^T\boldsymbol{\omega} = 2 \tag{6.57}$$

Using the inner product rule (see Section 7.9), and the fact that the two vectors $\mathbf{x}_1 - \mathbf{x}_2$ and $\boldsymbol{\omega}$ are parallel to each other, we can solve for the margin directly in terms of $\boldsymbol{\omega}$, as

$$\|\mathbf{x}_1 - \mathbf{x}_2\|_2 = \frac{2}{\|\boldsymbol{\omega}\|_2} \tag{6.58}$$

Therefore finding the separating hyperplane with maximum margin is equivalent to finding the one with the *smallest* possible normal vector $\boldsymbol{\omega}$.

6.5.4 The hard-margin and soft-margin SVM problems

In order to find a separating hyperplane for the data with minimum length normal vector we can simply combine our desire to minimize $\|\boldsymbol{\omega}\|_2^2$ subject to the constraint (defined by the Margin-Perceptron) that the hyperplane perfectly separates the data (given by the margin criterion described above). This gives the so-called *hard-margin* Support Vector Machine problem

$$\begin{aligned} &\underset{b,\,\boldsymbol{\omega}}{\text{minimize}} \;\; \|\boldsymbol{\omega}\|_2^2 \\ &\text{subject to} \;\; \max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)\right) = 0, \quad p = 1, ..., P. \end{aligned} \tag{6.59}$$

The set of constraints here are precisely the margin-perceptron guarantee that the hyperplane we recover separates the data perfectly. While there are *constrained optimization* algorithms that are designed to solve problems like this as stated, we can also solve the hard-margin problem by *relaxing* the constraints and forming an unconstrained formulation of the problem (to which we can apply familiar algorithmic methods to minimize). This is precisely the *regularization* approach detailed previously in Section 6.4.6. To do this we merely bring the constraints up, forming a single cost function

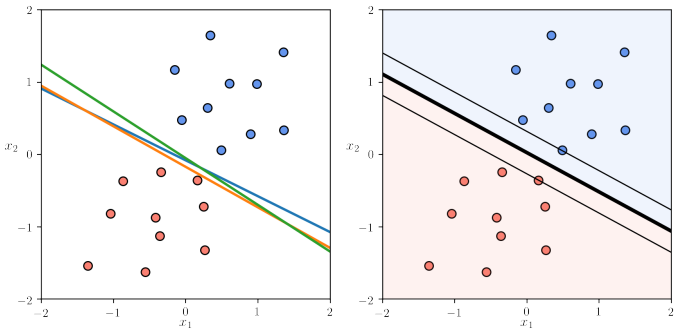

Figure 6.21 Figure associated with Example 6.8. See text for details.

$$g(b, \boldsymbol{\omega}) = \sum_{p=1}^{P} \max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T \boldsymbol{\omega}\right)\right) + \lambda \left\|\boldsymbol{\omega}\right\|_2^2 \tag{6.60}$$

to be minimized. Here the parameter $\lambda \geq 0$ is called a penalty or regularization parameter (as we saw previously in Section 6.4.6). When λ is set to a small positive value we put more 'pressure' on the cost function to make sure the constraints indeed hold, and (in theory) when λ is made very small the formulation above matches the original constrained form. Because of this λ is often set to be quite small in practice.

This *regularized* form of the Margin-Perceptron cost function is referred to as the *soft-margin Support Vector Machine cost*.

Example 6.8 **The SVM decision boundary**

In the left panel of Figure 6.21 we show three boundaries learned via minimizing the Margin-Perceptron cost (shown in Equation 6.5.1) three separate times with different random initializations. In the right panel of this Figure we show the decision boundary provided by properly minimizing the SVM soft-margin SVM cost in Equation 6.5.4 with regularization parameter $\lambda = 10^{-3}$.

Each of the boundaries shown in the left panel perfectly separate the two classes, but the SVM decision boundary in the right panel provides the maximum margin. Note how in the right panel the margins pass through points from both classes - equidistant from the SVM linear decision boundary. These points are called *Support Vectors*, hence the name Support Vector Machines.

6.5.5 SVMs and noisy data

A very big practical benefit of the softmargin SVM problem in equation 6.5.4 is that it allows us it to deal with noisy imperfectly (linearly) separable data - which arise far more commonly in practice than datasets that are perfectly

linearly separable. 'Noise' is permissible with the soft-margin problem since regardless of the weights b and $\boldsymbol{\omega}$, we always have some data point which is misclassified, i.e., for some p that $\max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)\right) > 0$. Such a case makes the hard-margin formulation in Equation 6.5.4 -, as no such b and $\boldsymbol{\omega}$ even exist that satisfy the constraints. Because we commonly deal with non-separable datasets in practice the softm-margin form of SVM is the version that is more frequently used.

6.6 Which approach produces the best results?

Once we forgo the assumption of perfectly (linear) separability the added value of a 'maximum margin hyperplane' proided by the SVM solution disappears since we *no longer have a margin to begin with*. Thus with many datasets in practice the soft-margin problem does *not* provide a solution remarkably different than the Perceptron or logistic regression. Indeed - with datasets that are not linearly separable - it often returns *exactly* the same solution provided by the perceptron or logistic regression.

Furthermore if we compare the soft-margin SVM problem in Equation 6.5.4, and *smooth* the Margin-Perceptron portion of the cost using the *softmax* function as detailed in Section 6.5.2. This gives a smoothed out soft-margin SVM cost function of the form

$$g\left(b, \boldsymbol{\omega}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)}\right) + \lambda \left\|\boldsymbol{\omega}\right\|_2^2 . \tag{6.61}$$

While this is interpreted through the lens of SVMs, it can also be immediately identified as a regularized Softmax cost (i.e., as a regularized Perceptron or logistic regression). Therefore we can see that all three methods of linear two-class classification we have seen - logistic regression, the Perceptron, and SVMs - are very deeply connected, and why they tend to provide similar results on realistic (not linearly separable) datasets.

6.7 The Categorical Cross Entropy cost

In Sections 6.2 and 6.3 we saw how two different choices for label values, $y_p \in \{0, 1\}$ or $y_p \in \{-1, +1\}$, result in precisely the same two class classification via Cross Entropy / Softmax cost function minimization. In each instance we formed a *Log Error* cost per datapoint, and averaging these over all P datapoints provided a proper and convex cost function. In other words, the numerical values of the label pairs themselves were largely used just to *simplify* the expression of these cost functions. Given the pattern for deriving convex cost functions for logistic regression, given any two *numerical* label values $y_p \in \{a, b\}$ it would be a

straightforward affair to derive an appropriate convex cost function based on a Log Error like pointwise cost.

However the true range of label value choices is even broader than this - than two *numerical* values. We can indeed use *any two distinct objects as labels* as well, i.e., two un-ordered values. However regardless of how we define our labels we still end up building the same sort two class classifier we have seen previously - tuning its weights by minimization of a familiar cost function like e.g., the Cross Entropy / Softmax cost.

To drive home this point, in this brief Section we show how to derive the same Cross Entropy cost function seen in Section 6.2 employing *categorical labels* instead of numerical ones. This leads to the derivation of the so-called *Categorical Cross Entropy cost* function which - as we will see - is equivalent to the Cross Entropy cost.

6.7.1 One-hot encoded categorical labels

Suppose we begin with a two-class classification dataset $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ with N dimenisional input and transform our original numerical label values $y_p \in \{0, 1\}$ with *one-hot encoded vectors* of the form

$$y_p = 0 \longleftarrow \mathbf{y}_p = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad y_p = 1 \longleftarrow \mathbf{y}_p = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{6.62}$$

Each vector representation uniquely identifies its corresponding label value, but now our label values are no longer *ordered numerical values*, and our dataset now takes the form $\left\{\left(\mathbf{x}_p, \mathbf{y}_p\right)\right\}_{p=1}^{P}$ where $\mathbf{y}_p$ is defined as above. However our goal here will remain the same: to properly tune a set of $N + 1$ weights $\mathbf{w}$ regress the input to the output of our dataset.

6.7.2 Choosing a nonlinearity

With these new *categorical labels* our classification task - when viewed as a *regression problem* - is a special case of *multi-output regression* (as detailed in Section 5.5) where we aim to regress N dimensional input against two dimensional categorical labels using a nonlinear function of the linear combination $\mathring{\mathbf{x}}_p^T\mathbf{w}$. Because our categorical labels have length two we need to use a nonlinear function of this linear combination that produces *two* outputs as well. Since the labels are one-hot encoded and we are familiar with the sigmoid function (see Section 6.2.3), it then makes sense to sense use the following nonlinear function of each input point $\mathbf{x}_p$

$$\boldsymbol{\sigma}_p = \begin{bmatrix} \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) \\ 1 - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) \end{bmatrix}. \tag{6.63}$$

Why? Because suppose for a particular point that $\mathbf{y}_p = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\mathbf{w}$ is tuned so that $\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) \approx 1$. By the definition of the sigmoid this implies that $1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) \approx 0$ and so that - for this point - $\boldsymbol{\sigma}_p \approx \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \mathbf{y}_p$ which is indeed our desire. And - of course - this same idea holds if $\mathbf{y}_p = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ as well.

Thus with this nonlinear transformation an ideal setting of our weights $\mathbf{w}$ will force

$$\boldsymbol{\sigma}_p \approx \mathbf{y}_p \tag{6.64}$$

to hold for as many points as possible.

6.7.3 Choosing a cost function

As was the case with numerical labels, here we could also very well propose a standard pointwise cost taken from our experience with regression like e.g., the Least Squares

$$g_p\left(\mathbf{w}\right) = \left\|\boldsymbol{\sigma}_p - \mathbf{y}_p\right\|_2^2 \tag{6.65}$$

and minimize the average of these over all P points to tune $\mathbf{w}$. However as was the case with numerical labels, here because our *categorical labels* take a very precise binary form we are better off employing a *Log Error* (see Section 6.2) to better incentivize learning (producing a convex cost function). Denoting $\log \boldsymbol{\sigma}_p$ the vector formed by taking the $\log(\cdot)$ of each entry of $\boldsymbol{\sigma}_p$, here the Log Error takes the form

$$g_p\left(\mathbf{w}\right) = -\mathbf{y}_p^T \log \boldsymbol{\sigma}_p = -y_{p,1} \log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) - y_{p,2} \log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right). \tag{6.66}$$

Taking the average of these P pointwise costs gives the so-called *Categorical Cross Entropy* cost function for two-class classification. Here the 'categorical' part of this name refers to the fact that our labels are one-hot encoded categorical (i.e., un-ordered) vectors.

However it is easy to see that cost function is precisely the Log Error we found in Section 6.2. In other words, written in terms of our original *numerical* labels, the pointwise cost above is precisely

$$g_p\left(\mathbf{w}\right) = -y_p \log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) - \left(1 - y_p\right) \log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right). \tag{6.67}$$

Therefore - even though we employed categorical versions of our original numerical label values - the cost function we minimize in the end to properly tune $\mathbf{w}$ is precisely the same we have seen previously - the Cross Entropy / Softmax cost where numerical labels were employed.

6.8 Making predictions and measuring the quality of a trained model

In this Section we describe simple metrics for judging the quality of a trained two-class classification model, as well as how to make predictions using one.

6.8.1 Making predictions using a trained model

If we denote the optimal set of weights found by minimizing a classification cost function, employing by default label values $y_p \in \{-1, +1\}$, by $\mathbf{w}^\star$ then note we can write our fully tuned linear model as

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = \mathring{\mathbf{x}}^T\mathbf{w}^\star = w_0^\star + x_1 w_1^\star + x_2 w_2^\star + \cdots + x_N w_N^\star. \tag{6.68}$$

This fully trained model defines an optimal decision boundary for the training dataset which takes the form

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = 0. \tag{6.69}$$

To predict the label y of an input $\mathbf{x}$ we then process this model through an appropriate step. Since we by default use label values $y_p \in \{-1, +1\}$ this step function is conveniently defined by the sign $(\cdot)$ function (as detailed in Section 6.3), and the predicted label for $\mathbf{x}$ is given as

$$\text{sign}\left(\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right)\right) = y\,. \tag{6.70}$$

This evaluation - which will always take on values ± 1 if $\mathbf{x}$ does not lie *exactly* on the decision boundary (in which case we assign a random value from ± 1) - simply computes which side of the decision boundary the input $\mathbf{x}$ lies on. If it lies 'above' the decision boundary then $y = +1$, and if 'below' then $y = -1$. This is illustrated for a prototypical dataset in Figure 6.22.

6.8.2 Confidence scoring

Once a proper decision boundary is learned, we can judge its *confidence* in any point based on *the point's distance to the decision boundary*. We say that our classifier has *zero* confidence in points lying along the decision boundary itself, because the boundary cannot tell us accurately which class such points belong too (which is why they are randomly assigned a label value if we ever need to make a prediction there). Likewise we say that *near* the decision boundary we have *little confidence* in the classifier's predictions. Why is this? Imagine we apply a small perturbation to the decision boundary, changing its location ever so slightly. Some points *very close to the original boundary* will end up on the *opposite* side of the new boundary, and will consequently have *different predicted*

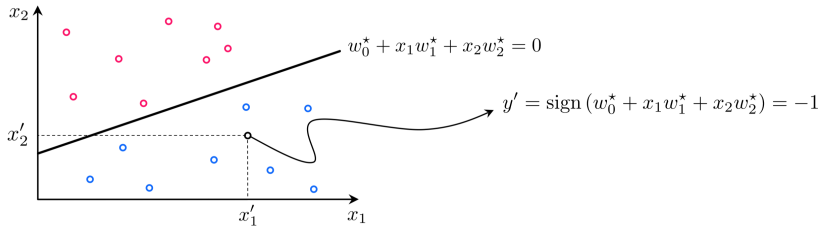

Figure 6.22 Once a decision boundary has been learned for the training dataset with optimal parameters $w_0^\star$ and $\mathbf{w}^\star$, the label y of a new point $\mathbf{x}$ can be predicted by simply checking which side of the boundary it lies on. In the illustration shown here $\mathbf{x}$ lies below the learned hyperplane, and as a result is given the label sign $(\mathring{\mathbf{x}}^T\mathbf{w}^\star) = -1$.

labels. Conversely, this is why we have *high confidence* in the predicted labels of points *far* from the decision boundary. These predicted labels will not change if we make a small change to the location of the decision boundary.

The notion of 'confidence' can be made precise and normalized to be universally applicable by running the point's distance to the boundary through the *sigmoid function* (see Section 6.2.3). This gives the confidence that a point belongs to class +1.

The signed distance d from a point to the decision boundary provided by our trained model can be computed (see Section 6.4.5) as

$$d = \frac{b^\star + \mathbf{x}_p^T\boldsymbol{\omega}^\star}{\left\|\boldsymbol{\omega}^\star\right\|_2}. \tag{6.71}$$

where we denote

$$\text{(bias): } b^\star = w_0^\star \quad \text{(feature-touching weights): } \boldsymbol{\omega}^\star = \begin{bmatrix} w_1^\star \\ w_2^\star \\ \vdots \\ w_N^\star \end{bmatrix}. \tag{6.72}$$

By evaluating d using the *sigmoid function* we squash it smoothly onto the interval $[0, 1]$, giving a 'confidence' score

$$\text{confidence in the predicted label of a point } \mathbf{x} \;\; = \sigma\left(d\right). \tag{6.73}$$

When this value equals 0.5 the point lies on the boundary itself. If the value is greater than 0.5 the point lies on the positive side of the decision boundary and so we have larger confidence in its predicted label being +1. When the value is less than 0.5 the point lies on the *negative* side of the classifier, and so we have less confidence that it truly has label value +1. Because normalization employing

the sigmoid squashes $(-\infty, +\infty)$ down to the interval $[0, 1]$ this confidence value is often interpreted as a *probability*.

6.8.3 Judging the quality of a trained model using *accuracy*

Once we have successfully minimized the a cost function for linear two-class classification it can be a delicate matter to determine our trained model's quality. The simplest metric for judging the quality of a fully trained model is to simply *count the number of misclassifications* it forms over our training dataset. This is a raw count of the number of training datapoints $\mathbf{x}_p$ whose true label y_p is predicted *incorrectly* by our trained model.

To compare the point $\mathbf{x}_p$'s predicted label $\hat{y}_p = \text{sign}\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}^\star\right)\right)$ and true true label y_p we can use an identity function $\mathcal{I}(\cdot)$ and compute

$$\mathcal{I}\left(\hat{y}_p,\, y_p\right) = \begin{cases} 0 & \text{if } \hat{y}_p, = y_p \\ 1 & \text{if } \hat{y}_p, \neq y_p. \end{cases} \tag{6.74}$$

Summing all P points gives the total number of misclassifications of our trained model

$$\text{number of misclassifications} = \sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p,\, y_p\right). \tag{6.75}$$

Using this we can also compute the *accuracy* - denoted $\mathcal{A}$ - of a trained model. This is simply the percentage of training dataset whose labels are *correctly* predicted by the model.

$$\mathcal{A} = 1 - \frac{1}{P}\sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p,\, y_p\right). \tag{6.76}$$

The accuracy ranges from 0 (no points are classified correctly) to 1 (all points are classified correctly).

Example 6.9 **Comparing cost function and misclassification histories**

Our classification cost functions are - in the end - based on smooth approximations to a discrete step function (as detailed in Sections 6.2 and 6.3). This is the function we truly wish to use, i.e., the function through which we truly want to tune the parameters of our model. However since we cannot optimize this parameterized step function directly we settle for a smooth approximation. The consequences of this practical choice are seen when we compare the cost function history from a run of gradient descent to the corresponding misclassification count measured at each step of the run. In Figure 6.23 show such a comparison using the dataset shown in Figure 6.11. In fact we show such results

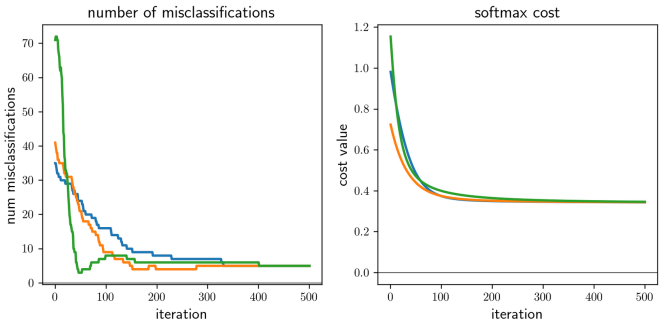

Figure 6.23 Figure associated with Example 6.9. See text for details.

of three independent runs of gradient descent, with the history of misclassifications shown in the left panel and corresponding Softmax cost histories show in the right panel. Each run is color-coded to distinguish it from the other runs.

Comparing the left and right panels of the Figure we can see that the number of misclassifications and Softmax evaluations at each step of a gradient descent run do not perfectly track one another. That is, it is not the case that just because the cost function value is decreasing that so too is the number of misclassifications. Again, this occurs because our Softmax cost is only an approximation of the true quantity we would like to minimize.

This simple example has an extremely practical implication: after a running a local optimization to minimize a two-class classification cost function the best step, and corresponding weights, are associated with the lowest *number of misclassifications* (or likewise the *highest accuracy*) **not** the lowest cost function value.

6.8.4 Judging the quality of a trained model using *balanced accuracy*

Classification accuracy is an excellent basic measurement of a trained classifier's performance. However in certain scenarios using the accuracy metric can paint an incomplete picture of how well we have really solved a classification problem. For example when a dataset consists of *highly imbalanced classes* - that is when a dataset has far more examples of one class than the other - the 'accuracy' of a trained model loses its value as a quality metric. This is because when one class greatly outnumbers the other in a dataset an accuracy value close to 1 can be misleading. For example, if one class makes up 95% percent of all data points, a naive classifier that blindly assigns the label of the majority class to *every training point* achieves an accuracy of 95%. But here misclassifying 5% amounts to *completely misclassifying an entire class of data*.

This idea of 'sacrificing' members of the smaller class by misclassifying them (instead of members from the majority class) is - depending on the application - very undesirable. For example if the classification determines whether or not

someone has a particularly rare but deadly disease that requires further examination one would likely rather misclassify a healthy individual (and give them further testing) then miss someone with the disease. As another example, if the classification determines whether or not a particular financial transaction was *fraudulent*, one would likely rather misclassify a standard transaction (to review further or alert a customer) than miss an actually fraudulent transaction

These sorts of scenarios point to a problem with the use of *accuracy* as a proper metric for diagnosing classifier performance on datasets with highly imbalanced classes: because it weights *misclassifications from both classes equally* it fails to convey how well a trained model performs on each class of the data individually. This results in the potential for strong performance on a very large class of data masking poor performance on a very small one. The simplest way to improve the accuracy metric is to take this potential problem into account, and instead of computing accuracy over *both classes together* to compute accuracy on *each class individually and average the results*.

If we denote the *indices* of those points with labels $y_p = +1$ and $y_p = -1$ as Ω_{+1} and Ω_{-1} respectively, then we can compute the number of misclassifications on each class individually (employing the notation and indicator function introduced above) as

$$\begin{array}{c} \text{number of misclassifications on } +1 \text{ class} = \sum_{p\in\Omega_{+1}} \mathcal{I}\left(\hat{y}_p,\, y_p\right) \\ \text{number of misclassifications on } -1 \text{ class} = \sum_{p\in\Omega_{-1}} \mathcal{I}\left(\hat{y}_p,\, y_p\right). \end{array} \tag{6.77}$$

The accuracy on each class individually can then be likewise computed as (denoting the accuracy on class +1 and −1 individually as $\mathcal{A}_{+1}$ and $\mathcal{A}_{-1}$ respectively)

$$\begin{array}{c} \mathcal{A}_{+1} = 1 - \frac{1}{|\Omega_{+1}|}\sum_{p\in\Omega_{+1}} \mathcal{I}\left(\hat{y}_p,\, y_p\right) \\ \mathcal{A}_{-1} = 1 - \frac{1}{|\Omega_{-1}|}\sum_{p\in\Omega_{-1}} \mathcal{I}\left(\hat{y}_p,\, y_p\right) \end{array} \tag{6.78}$$

Note here the $|\Omega_{+1}|$ and $|\Omega_{-1}|$ denote the number of points belonging to the +1 and −1 class respectively. We can then combine these two metrics into a single value by *taking their average*. This combined metric is called *balanced accuracy* (which we denote as $\mathcal{A}_{\text{balanced}}$)

$$\mathcal{A}_{\text{balanced}} = \frac{\mathcal{A}_{+1} + \mathcal{A}_{-1}}{2}. \tag{6.79}$$

Notice if both classes have equal representation then balanced accuracy reduces to the overall accuracy value $\mathcal{A}$.

The balanced accuracy metric ranges from 0 to 1. When equal to 0 no point is classified correctly, and when both classes are classified perfectly $\mathcal{A}_{\text{balanced}} = 1$. Values of the metric in between 0 and 1 measure how well - on average - each class is classified individually. If, for example, one class of data is classified

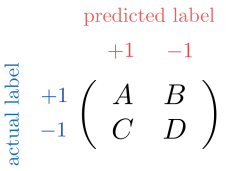

Figure 6.24 A generic *confusion matrix* as a metric for classification quality.

completely correct and the other completely incorrect (as in our imaginary scenario where we have an imbalanced dataset with 95% membership in one class and 5% in the other, and where have simply classified the entire space as the majority class) then $\mathcal{A}_{\text{balanced}} = 0.5$.

Thus balanced accuracy is a simple metric for helping us understand whether our learned model has 'behaved poorly' on highly imbalanced datasets. In order to *improve the behavior* of our learned model in such instances we have to adjust the way we perform two class classification. One popular way of doing this - called weighted classification - is discussed in the next Section.

6.8.5 The confusion matrix and additional quality metrics

Additional metrics for judging the quality of a trained model for two class classification can be formed using the *confusion matrix*, shown in the Figure 6.24. A confusion matrix is a simple look-up table where classification results are broken down by actual (across rows) and predicted (across columns) class membership. Here we denote A is the number of data points whose actual label, +1, is identical to the label assigned to them by the trained classifier. The other diagonal entry D is similarly defined as the number of data points whose predicted class, -1, is equal to their actual class. The off-diagonal entries denoted by B and C represent the two types of classification errors wherein the actual and predicted labels do not match one another. In practice we want these two values to be as small as possible.

Our *accuracy* metric can be expressed in terms of the confusion matrix quantities shown in the Figure as

$$\mathcal{A} = \frac{A + D}{A + B + C + D}, \tag{6.80}$$

and our accuracy on each individual class likewise as

$$\begin{aligned} \mathcal{A}_{+1} &= \tfrac{A}{A+C} \\ \mathcal{A}_{-1} &= \tfrac{D}{B+D}. \end{aligned} \tag{6.81}$$

In the jargon of machine learning these individual accuracy metrics are often called *precision* and *specificity* respectively. The *balanced accuracy* metric can likewise be expressed as

$$\mathcal{A}_{\text{balanced}} = \frac{1}{2}\frac{A}{A+C} + \frac{1}{2}\frac{D}{B+D}. \tag{6.82}$$

6.9 Weighted two-class classification

Because our two-class classification cost functions are *summable over individual points* we can - as we did with regression in Section 5.4 - weight individual points in order to emphasize or de-emphasize their importance to a classification model. This is called *weighted classification*. This idea is often implemented when dealing with *highly imbalanced* two class datasets (see Section 6.8.4).

6.9.1 Weighted two-class classification

Just as we saw with regression in Section 5.4, weighted classification cost functions naturally arise due to repeated points in a dataset. For example, with metadata (e.g., census data) datasets it is not uncommon to receive duplicate data points due to multiple individuals reporting the same answers to a survey.

In Figure 6.25 we take a standard census dataset and plot a subset of it along a single input feature. With only one feature taken into account we end up with multiple entries of the same datapoint, which we show visually via the radius of each point (the more times a given datapoint appears in the dataset the larger we make the radius). These datapoints should not be thrown away - they did not arise due to some error in data collecting - they represent the true dataset.

Just as with a regression cost, if we examine any two-class classification cost it will 'collapse', with summands containing identical points naturally combining into weighted terms. One can see this by performing the same kind of simple exercise used in Section 6.25 to illustrate this fact for regression. This leads to the notion of *weighting* two-class cost functions, like e.g., the weighted Softmax cost which we write below using the generic `model` notation used used in e.g., Section 6.8 to represent our linear model

$$g(\mathbf{w}) = \sum_{p=1}^{P} \beta_p \log\left(1 + e^{-y_p \text{model}\left(x_p, \mathbf{w}\right)}\right). \tag{6.83}$$

Here the values β_1, β_2, ..., β_P are fixed *point-wise* weights. That is, a unique point $\left(x_p,\, y_p\right)$ in the dataset has weight $\beta_p = 1$ whereas if this point is repeated R times in the dataset then one instance of it will have weight $\beta_p = R$ while the others have weight $\beta_p = 0$. Since these weights are fixed (i.e., they are *not* parameters that need to be tuned, like $\mathbf{w}$) we can minimize a weighted classification cost precisely as we would any other e.g,. via a local optimization scheme like gradient descent or Newton's method.

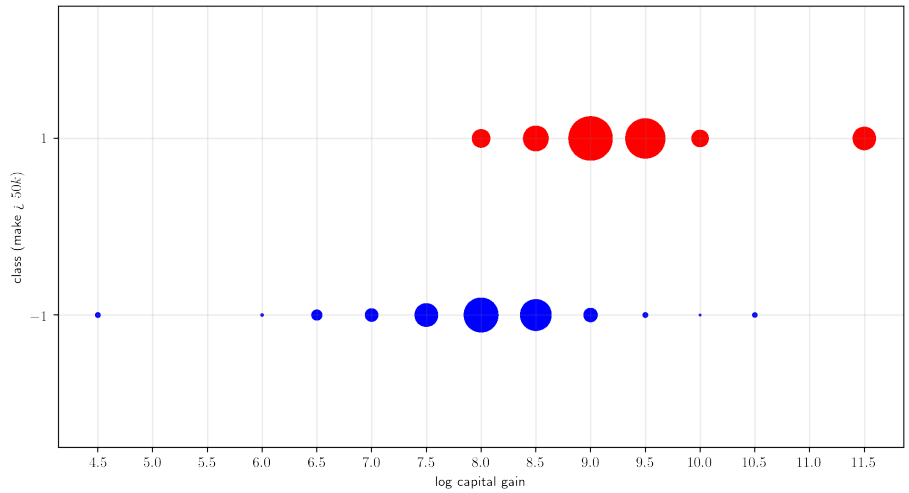

Figure 6.25 An example of duplicate entries in a metadata dataset. Here a single input feature of this dataset is plotted along with the labels for each point. There are many duplicates in this slice of the data, which are visually depicted by the radius of each point (the larger the radius, the more duplicates of that point exist in the dataset).

6.9.2 Weighting points by confidence

Just as with regression (see Section 5.4), we can also think of *assigning* the fixed weight values in Equation 6.9.1 based on our 'confidence' of the legitimacy of a datapoint. If we believe that a point is very trustworthy we can set its corresponding weight β_p closer to 1, and the more untrustworthy we find a point the smaller we set β_p in the range $0 \leq \beta_p \leq 1$ where $\beta_p = 0$ implies we do not trust the point at all. In making these weight selections we of course determine how important each datapoint is in the training of the model.

6.9.3 Dealing with class imbalances

Weighted classification - in the manner detailed above - is often used to deal with imbalanced datasets. These are datasets which contain far more examples of one class of data than the other. With such datasets it is often easy to achieve a high accuracy by misclassifying points from the smaller class (as described in Section 6.8.4).

One way of ameliorating this issue is to use a weighted classification cost to alter the behavior of the learned classifier so that it weights points in the smaller class more, and points in the larger class less. In order to produce this outcome it is common to assign such weights *inversely proportional to the number of members of each class*. This weights members of the majority and minority classes so that - overall - each provides an equal contribution to the weighted classification.

That is if we denote Ω_{+1} and $\Omega-1$ index sets for the points in classes +1 and

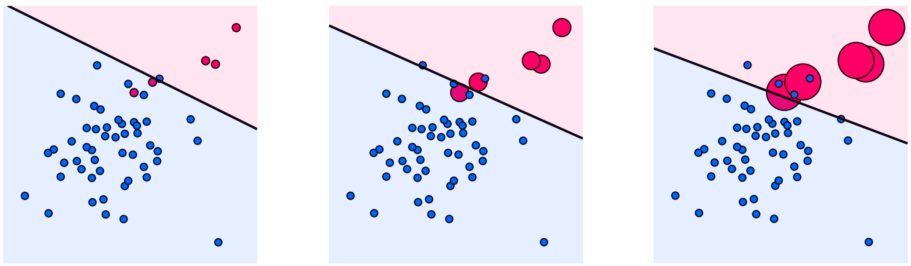

Figure 6.26 Figure associated to Example 6.10. See text for details.

-1 respectively, then first note that $P = |\Omega_{+1}| + |\Omega_{-1}|$. Then denoting β_{+1} and β_{-1} the weight for each member of class $+1$ and -1 respectively we can set these class-wise weights inversely proportional to the number of points in each class as

$$\begin{array}{l} \beta_{+1} \propto \frac{1}{|\Omega_{+1}|} \\ \beta_{-1} \propto \frac{1}{|\Omega_{-1}|}. \end{array} \tag{6.84}$$

Example 6.10 **Class imbalance and weighted classification**

In the left panel of Figure 6.26 we show a toy dataset with severe class imbalance. Here we also show the linear decision boundary resulting from properly minimizing the Softmax cost over this dataset, and color each region of the space based on how this trained classifier labels points. As we can see, almost the entire minority class is misclassified here.

In the middle and right panels we show the result of increasing the weights of each member of the minority class from $\beta = 1$ to $\beta = 5$ (middle panel) and $\beta = 10$ (right panel). These weights are denoted visually in the Figure by increasing the radius of the points in proportion to the value of β used (thus their radius increases from left to right). Also shown in the middle and right panels is the result of properly minimizing the weighted Softmax cost in Equation 6.9.1. As the value of β is increased on the minority class, we encourage fewer misclassifications of its members (at the expense here of additional misclassifications of the majority class).

6.10 Exercises

Section 6.1 exercises

1. **Implementing sigmoidal Least Squares cost**

Repeat the experiment described in Example 6.3 by coding up the Least Squares cost function shown in Equation 6.9 and the normalized gradient descent algorithm detailed in Section 3.9. You need not re-produce the contour plot shown in the right panel of Figure 6.5, however you can verify that your implementation is working properly by re-producing the final fit shown in the left panel of that Figure. Alternatively show that your final result produces zero misclassifications (see Section 6.8.3).

2. **Show the equivalence of the Log Error and Cross Entropy pointwise cost**

Show that - with label values $y_p \in \{0, 1\}$ - that the Log Error in Equation 6.10 is equivalent to the Cross-Entropy pointwise cost in Equation 6.2.5.

3. **Implementing the Cross Entropy cost**

Repeat the experiment described in Example 6.4 by coding up the Cross Entropy cost function shown in Equation 6.2.5 as detailed in Section 6.2.7. You need not re-produce the contour plot shown in the right panel of Figure 6.7, however you can verify that your implementation is working properly by re-producing the final fit shown in the left panel of that Figure. Alternatively show that your final result produces zero misclassifications (see Section 6.8.3).

4. **Compute the Lipschitz constant of the Cross Entropy cost**

Compute the Lipschitz constant (Section 3.12.4) of the Cross Entropy cost shown in Equation 6.2.5.

5. **Confirm gradient and Hessian calculations**

Confirm that the gradient and Hessian of the Cross-Entropy cost are as shown in Section 6.2.7.

Section 6.2 exercises

6. **Show the equivalence of the Log Error and Softmax pointwise cost**

Show that - with label values $y_p \in \{-1, +1\}$ - that the Log Error in Equation 6.22 is equivalent to the Softmax pointwise cost in Equation 6.3.1.

7. **Implementing the Softmax cost**

   Repeat the experiment described in Example 6.5 by coding up the Softmax cost function shown in Equation 6.3.1. You need not re-produce the contour plot shown in the right panel of Figure 6.10, however you can verify that your implementation is working properly by re-producing the final fit shown in the left panel of that Figure. Alternatively show that your final result produces zero misclassifications (see Section 6.8.3).

8. **Implementing the Log Error version of Softmax**

   Repeat the experiment described in Section 6.3.3 and shown in Figure 6.11 using the Log Error based Softmax cost shown in Equation 6.21 and any local optimization scheme you wish. You need not re-produce the plots shown in the Figure to confirm your implementation works properly, but should be able to achieve a result that has less than 5 misclassifications (see Section 6.8.3)..

Section 6.3 exercises

9. **Using gradient descent to minimize the Perceptron cost**

   Use the standard gradient descent scheme to minimize the Perceptron cost function in Equation 6.4.1 over the dataset shown in Figure 6.11. Make two runs of gradient descent using fixed steplength values $\alpha = 10^{-1}$ and 10^{-2}, with 50 steps each (and random initializations). Produce a cost function history plot and history of misclassification (see Section 6.8.3) at each step of the run. Which run achieves perfect classification first?

10. **The perceptron cost is convex**

    Show that the pPerceptron cost given in Equation 6.4.1 is convex (see Section 7.13).

11. **The Softmax cost is convex**

    Show that the Softmax cost function given in Equation is convex by verifying that it satisfies the second order definition of convexity. *Hint: the Hessian, already given in Equation (6.3.2), is a weighted outer product matrix like the one described in Exercise 2.*

12. **The regularized Softmax**

Repeat the experiment described in Example 6.7 and shown in Figure 6.17. You need not re-produce the plots shown in the Figure to confirm your implementation works properly, but should be able to achieve a result that has less than 5 misclassifications.

Section 6.4 exercises

13. **The Margin-Perceptron cost function is convex**

Show that the Margin-Perceptron cost function is convex. You can do this by following the same steps outlined in Exercise 10.

Section 6.7 exercises

14. **Compare the efficacy of two-class cost functions I**

Compare the efficacy of the Softmax and the Perceptron cost functions in terms of the minimal number of misclassifications each can achieve by proper minimization via gradient descent on a **A breast cancer dataset**. This dataset consists of $P = 699$ datapoints, each point consisting of $N = 9$ input of attributes of a single individual and output label indicating whether or not the individual does or does not have breast cancer.

15. **Compare the efficacy of two-class cost functions II**

Compare the efficacy of the Softmax and the Perceptron cost functions in terms of the minimal number of misclassifications each can achieve by proper minimization via gradient descent on a **A spam detection dataset**. This spam email consists of $P = 4601$ datapoints datapoints, each consisting of various input features for an email and a label indicating whether or not the email is spam (an unwanted solicitation or advertisement) or regular email.

6.11 Endnotes

Proof that the Cross Entropy cost is convex

To show that the Cross Entropy cost function is convex we can use the second order definition of convexity, by which we must show that the eigenvalues of this cost function's Hessian matrix are all always nonnegative. Studying the Hessian of the cros-entropy - which was defined algebraically in Example 4 above - we have

$$\nabla^2 g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T. \tag{6.85}$$

We know that the smallest eigenvalue of any square symmetric matrix is given as the minimum of the *Rayleigh quotient* (as detailed Section 6.1), i.e., the smallest value taken by

$$\mathbf{z}^T\nabla^2 g\left(\mathbf{w}\right)\mathbf{z} \tag{6.86}$$

for any unit-length vector $\mathbf{z}$ and any possible weight vector $\mathbf{w}$. Substituting in the particular form of the Hessian here, denoting σ

To show that the Softmax cost function is convex we can use the second order definition of convexity, by which we must show that the eigenvalues of the Softmax's Hessian matrix are all always nonnegative. Studying the Hessian of the Softmax - which was defined algebraically in Example 4 above - we have

$$\nabla^2 g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\frac{1}{1+e^{y_p\mathbf{x}_p^T\mathbf{w}}}\right)\left(1-\frac{1}{1+e^{y_p\mathbf{x}_p^T\mathbf{w}}}\right)\mathbf{x}_p\mathbf{x}_p^T. \tag{6.87}$$

We know that the smallest eigenvalue of any square symmetric matrix is given as the minimum of the *Rayleigh quotient* (as detailed Section 6.1), i.e., the smallest value taken by

$$\mathbf{z}^T\nabla^2 g\left(\mathbf{w}\right)\mathbf{z} \tag{6.88}$$

for any unit-length vector $\mathbf{z}$ and any possible weight vector $\mathbf{w}$. Substituting in the particular form of the Hessian here, denoting $\sigma\_p = \left(\frac{1}{1+e^{y_p\mathbf{x}_p^T\mathbf{w}}}\right)\left(1-\frac{1}{1+e^{y_p\mathbf{x}_p^T\mathbf{w}}}\right)$ for each p for short, we have

$$\mathbf{z}^T\nabla^2 g\left(\mathbf{w}\right)\mathbf{z} = \mathbf{z}^T\left(\frac{1}{P}\sum_{p=1}^{P}\sigma_p\mathbf{x}_p\mathbf{x}_p^T\right)\mathbf{z} = \frac{1}{P}\sum_{p=1}^{P}\sigma_p\left(\mathbf{z}^T\mathbf{x}_p\right)\left(\mathbf{x}_p^T\mathbf{z}\right) = \frac{1}{P}\sum_{p=1}^{P}\sigma_p\left(\mathbf{z}^T\mathbf{x}_p\right)^2 \tag{6.89}$$

Since it is always the case that $\left(\mathbf{z}^T\mathbf{x}_p\right)^2 \geq 0$ and $\sigma_p \geq 0$, it follows that the smallest value the above can take is 0, meaning that this is the smallest possible

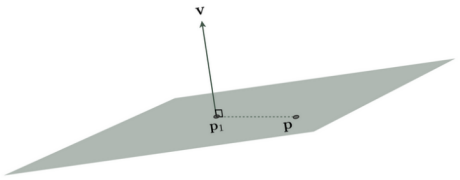

Figure 6.27 The normal vector $\mathbf{v}$ is perpendicular to all objects inside a hyperplane, including the line segment connecting its intersection with the hyperplane at $\mathbf{p}_1$ to any point $\mathbf{p}$ in the hyperplane.

eigenvalue of the Softmax cost's Hessian. Since this is the case, the Softmax cost must be convex.

6.11.1 Representing hyperplanes via normal vectors

The constructive perspective on hyperplanes discussed in the previous Section led to formation of equation (9) for a hyperplane in $N + 1$ dimensions. Here we adopt a new geometric perspective to represent a hyperplane in a different (but equivalent) way via its *normal vector*, that is, a vector perpendicular to the hyperplane. Figure 1 shows a hyperplane along with its normal vector $\mathbf{v}$ piercing the hyperplane at

$$\mathbf{p}_1 = \begin{bmatrix} \mathbf{w}_1 \\ g_1 \end{bmatrix} \tag{6.90}$$

The normal vector, by definition, is perpendicular to every 'object' living inside the hyperplane: this includes every line that connects $\mathbf{p}_1$ to any other point in the hyperplane. Let

$$\mathbf{p} = \begin{bmatrix} \mathbf{w} \\ g(\mathbf{w}) \end{bmatrix} \tag{6.91}$$

be a generic point in the hyperplane. The fact that $\mathbf{p} - \mathbf{p}_1$ is perpendicular to $\mathbf{v}$ (as shown in Figure 1) allows us to write

$$(\mathbf{p} - \mathbf{p}_1)^T \mathbf{v} = 0 \tag{6.92}$$

or

$$\mathbf{p}^T \mathbf{v} = \mathbf{p}_1^T \mathbf{v} \tag{6.93}$$

Denoting the first N entries of $\mathbf{v}$ by $\mathbf{v}_{1:N}$, this can be rewritten as

$$\mathbf{w}^T \mathbf{v}_{1:N} + g(\mathbf{w})\, v_{N+1} = \mathbf{w}_1^T \mathbf{v}_{1:N} + g_1 v_{N+1} \tag{6.94}$$

Notice, by setting

$$a = \frac{\mathbf{w}_1^T \mathbf{v}_{1:N} + v_{N+1} g_1}{v_{N+1}} \tag{6.95}$$

and

$$\mathbf{b} = -\frac{\mathbf{v}_{1:N}}{v_{N+1}} \tag{6.96}$$

we arrive at the equation of the hyperplane in (9), with a small caveat: a and $\mathbf{b}$ in (19) and (20) cannot be defined if $v_{N+1} = 0$. This only happens when the hyperplane is in parallel with the $g(\mathbf{w})$ axis. Therefore - in the most general case - we can represent the equation of a hyperplane via parameters $\boldsymbol{\alpha}$, β, and γ, as

$$\mathbf{w}^T \boldsymbol{\alpha} + g(\mathbf{w})\beta + \gamma = 0 \tag{6.97}$$

where $\begin{bmatrix} \boldsymbol{\alpha} \\ \beta \end{bmatrix}$ is the normal vector.

6.11.2 More information on the relu cost

Note however that in examining a partial derivative of just one summand of the cost

With respect to weights in $\mathbf{w}$ we have

$$\frac{\partial}{\partial w_n} \max\left(0, -y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right) = \begin{cases} -y_p x_{p,n} & \text{if } -y_p \mathring{\mathbf{x}}_p^T \mathbf{w} > 0 \\ 0 & \text{else} \end{cases} \tag{6.98}$$

Similarly for w_0 we can write

$$\frac{\partial}{\partial w_0} \max\left(0, -y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right) = \begin{cases} -y_p & \text{if } -y_p \mathring{\mathbf{x}}_p^T \mathbf{w} > 0 \\ 0 & \text{else} \end{cases} \tag{6.99}$$

we can then conclude the magnitude of the full cost function's gradient will not necessarily diminish to zero close to global minima and could stay fixed (in magnitude) based on the dataset. Thus, it is possible for gradient descent with a fixed steplength value α to oscillate and 'zig-zag' around, never going to a minimum (as in Exercise 9). In this case we need to either tune a fixed steplength or choose a diminishing one.