7 Linear Multi-class Classification

7.1 Introduction

In practice many classification problems (e.g., face recognition, hand gesture recognition, recognition of spoken phrases or words, etc.) have more than just two classes we wish to distinguish. However a single linear decision boundary naturally divides the input space only in *two* sub-spaces, and therefore is fundamentally insufficient as a mechanism for differentiating between more than two classes of data. In this Chapter we describe how to generalize what we have seen in the previous Chapter to deal with this multi-class setting.

7.2 One-versus-All multi-class classification

In this Section we explain the fundamental multi-classification scheme called *One-versus-All*, step by step using a single toy dataset with three classes.

7.2.1 Notation and modeling

A multi-class classification dataset $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ consists of C distinct classes of data. As with the two-class case we can in theory use *any* set of C distinct label values to distinguish between the classes. For convenience, here we use label values $y_p \in \{0, 1, \ldots, C-1\}$. In what follows we employ the prototypical toy dataset shown in Figure 7.1 to help us derive our fundamental multi-class classification scheme.

7.2.2 Training C One-versus-All classifiers

A good first step in the development of multi-class classification is to simplify the problem to something we are already familiar with: *two-class classification*. We already know how to distinguish each class of our data from the other $C-1$ classes. This sort of two-class classification problem (discussed in the previous Chapter) is indeed simpler, and is adjacent to the actual problem we want to solve, i.e., learn a classifier that can distinguish between all C classes simultaneously. To solve this adjacent problem we learn C two-class classifiers

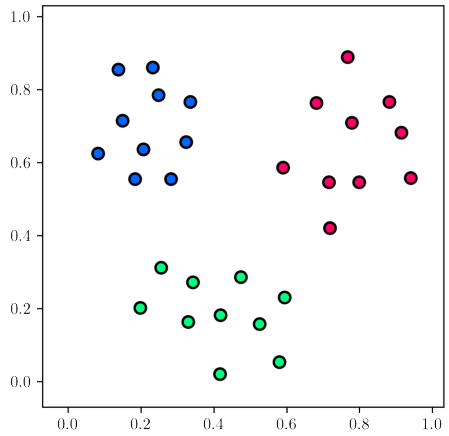

Figure 7.1 A prototypical classification dataset with $C = 3$ classes. Here points with label value $y_p = 0$, $y_p = 1$, and $y_p = 2$ are colored blue, red, and green, respectively.

over the entire dataset, with the c^{th} classifier trained to distinguish the c^{th} class from the remainder of the data, and hence called a *one-versus-rest* or *one-versus-all* classifier.

To solve the c^{th} two-class classification sub-problem we simply assign temporary labels $\tilde{y}_p$ to the entire dataset, giving +1 labels to the c^{th} class and -1 labels to the remainder of the dataset

$$\tilde{y}_p = \begin{cases} +1 & \text{if } y_p = c \\ -1 & \text{if } y_p \neq c \end{cases} \tag{7.1}$$

where y_p is the original label for the p^{th} point of the multi-class dataset. We then run a two-class classification scheme of our choice (by minimizing any classification cost detailed in the previous Chapter). Denoting the optimal weights learned for the c^{th} classifier as $\mathbf{w}_c$ where

$$\mathbf{w}_c = \begin{bmatrix} w_{0,c} \\ w_{1,c} \\ w_{2,c} \\ \vdots \\ w_{N,c} \end{bmatrix} \tag{7.2}$$

we can then express the corresponding decision boundary associated with the c^{th} two-class classification (distinguishing class c from all other datapoints) simply as

$$\mathring{\mathbf{x}}^T \mathbf{w}_c = 0. \tag{7.3}$$

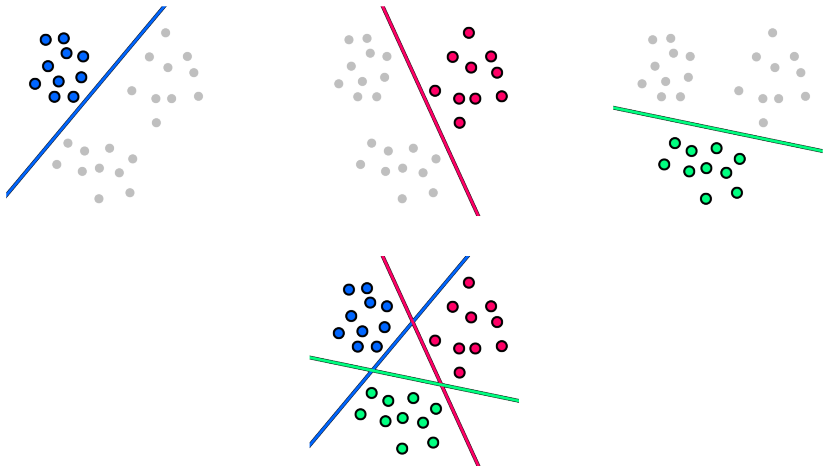

Figure 7.2 (top row) Three one-versus-all linear two-class classifiers learned to the dataset shown in Figure 7.1. (bottom row) All three classifiers shown on top of the original dataset. See text for further details.

In Figure 7.2 we show the result of solving this set of sub-problems on the prototypical dataset shown first in Figure 7.1. In this case we learn $C = 3$ distinct linear two-class classifiers which are shown in the top row of the Figure. Each learned decision boundary is illustrated in the color of the single class being distinguished from the rest of the data, with the remainder of the data colored gray in each instance. In the bottom row of the Figure the dataset is shown once again along with all three learned two-class decision boundaries.

Having solved the adjacent problem by learning C one-versus-all classifiers we can now ask: is it possible to somehow *combine* these two-class classifiers to solve the original multi-class problem? As we will see, it most certainly is. But first it is helpful to break up the problem into three pieces for (i) points that lie on the positive side of a *single* two-class classifier, (ii) points that lie on the positive side of *more than one* classifier, and (iii) points that lie on the positive side of *none* of the classifiers. Notice, these three cases exhaust all the possible ways a point can lie the input space in relation to the C two-class classifiers.

7.2.3 Case 1. Labeling points on the positive side of a single classifier

Geometrically speaking, a point $\mathbf{x}$ that lies on the positive side of the c^{th} classifier but on the negative side of all the rest, satisfies the following inequalities: $\mathring{\mathbf{x}}^T\mathbf{w}_c > 0$, and $\mathring{\mathbf{x}}^T\mathbf{w}_j < 0$ for all $j \neq c$. First, notice that since all classifier evaluations of $\mathbf{x}$ are negative except for the c^{th} one (which is positive), we can write

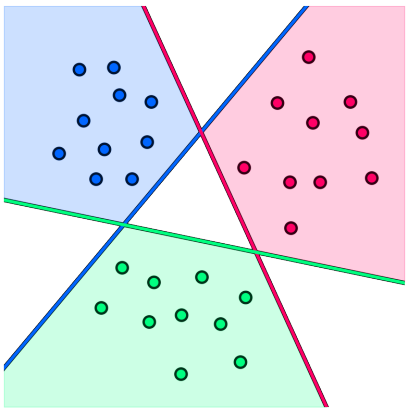

Figure 7.3 One-versus-all classification of points lying on the positive side of a single classifier. See text for details.

$$\mathring{\mathbf{x}}^T\mathbf{w}_c = \max_{j=0,\ldots,C-1} \mathring{\mathbf{x}}^T\mathbf{w}_j \tag{7.4}$$

Next, that the c^{th} classifier evaluates $\mathbf{x}$ positively means that (from its perspective) $\mathbf{x}$ belongs to class c. Similarly, because every other classifier evaluates $\mathbf{x}$ negatively (from their perspective) $\mathbf{x}$ does *not* belong to any class $j \neq c$. Altogether we have all C individual classifiers *in agreement* that $\mathbf{x}$ should receive the label $y = c$. Therefore, using Equation (7.4) we can write the label y as

$$y = \underset{j=0,\ldots,C-1}{\text{argmax}} \ \mathring{\mathbf{x}}^T\mathbf{w}_j \tag{7.5}$$

In Figure 7.3 we show the result of classifying all points in the space of our toy dataset that lie on the positive side of a *single* classifier. These points are colored to match their respective classifier. Notice there still are regions left uncolored in Figure 7.3. These include regions where points are either on the positive side of more than one classifier or on the positive side of none of the classifiers. We discuss these cases next.

7.2.4 Case 2. Labeling points on the positive side of more than one classifier

When a point $\mathbf{x}$ falls on the positive side of more than one classifier it means that, unlike the previous case, more than one classifiers claim $\mathbf{x}$ as one of their own. In a situation like this and as we saw in Section 6.8.2, we can consider the *signed distance* of $\mathbf{x}$ to a two-class decision boundary as a measure of our *confidence* in how the point should be labeled. The *farther* a point is on the *positive* side of

a classifier, the *more* confidence we have that this point should be labeled +1. Intuitively this is a simple geometric concept: the larger a point's distance to the boundary the deeper into one region of a classifier's half-space it lies, and thus we can be much more confident in its class identity than a point closer to the boundary. Another way to think about is imagine what would happen if we slightly perturbed the decision boundary? Those points originally close to its boundary might end up on the other side of the perturbed hyperplane, changing classes, whereas those points farther from the boundary are less likely to be so affected and hence we can be more confident in their class identities to begin with.

In the present context we can intuitively extend this idea. If a point lies on the positive side of multiple two-class classifiers, we should assign it the label corresponding to the decision boundary it is *farthest* from.

Let us now see how this simple idea plays out when used on our toy dataset. In the left and middle panels of Figure 7.4 we show two example points (drawn in black) in a region of our toy dataset that lie on the *positive* side of more than one two-class classifier. In each case we highlight the distance from the new point to both decision boundaries in dashed black, with the projection of the point onto each decision boundary shown as an 'x' in the same color as its respective classifier. Beginning with the point shown in the left panel, notice that it lies on the positive side of both the red and blue classifiers. However we can be more confident that the point should belong to the blue class since the point lies a greater distance from the blue decision boundary than the red (as we can tell by examining the length of the dashed lines emanating from the point to each boundary). By the same logic the point shown in the middle panel is best assigned to the red class, being at a greater distance from the red classifier than the blue. If we repeat this logic for every point in this region (as well as those points in the other two triangular regions where two or more classifiers are positive) and color each point the color of its respective class, we will end up shading each such region as shown in the right panel of Figure 7.4.

Decomposing the weights $\mathbf{w}_j$ of our j^{th} classifier by separating the bias weight from feature-touching weights as

$$\text{(bias): } b_j = w_{0,j} \quad \text{(feature-touching weights): } \quad \boldsymbol{\omega}_j = \begin{bmatrix} w_{1,j} \\ w_{2,j} \\ \vdots \\ w_{N,j} \end{bmatrix} \tag{7.6}$$

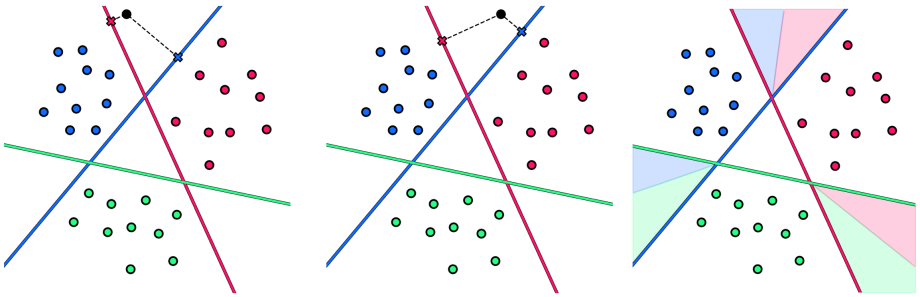

Figure 7.4 One-versus-all classification of points lying on the positive side of more than one classifier. Here a point (shown in black) lies on the positive side of both the red and blue classifiers with its distance to each decision boundary noted by dashed lines connecting it to each classifier. See text for further details.

allows us to write the *signed distance* from a point $\mathbf{x}$ to its decision boundary as

$$\text{signed distance of } \mathbf{x} \text{ to } j^{th} \text{ boundary} = \frac{\mathring{\mathbf{x}}^T \mathbf{w}_j}{\left\| \boldsymbol{\omega}_j \right\|_2} \tag{7.7}$$

Now, if we *normalize* the weights of each linear classifier by the length of its normal vector (containing all feature-touching weights) as

$$\mathbf{w}_j \longleftarrow \frac{\mathbf{w}_j}{\left\| \boldsymbol{\omega}_j \right\|_2} \tag{7.8}$$

which we will assume to do from here on, then this distance is simply written as the raw evaluation of the point via the decision boundary

$$\text{signed distance of } \mathbf{x} \text{ to } j^{th} \text{ boundary} = \mathring{\mathbf{x}}^T \mathbf{w}_j. \tag{7.9}$$

To assign a point in one of our current regions we seek out the classifier which *maximizes* this quantity. Expressed algebraically, the label y assigned to this point is given (after weight-normalization) as

$$y = \underset{j=0,\ldots,C-1}{\text{argmax}} \ \mathring{\mathbf{x}}^T \mathbf{w}_j. \tag{7.10}$$

This is precisely the same rule we found in Equation (7.5) for regions of the space where only a single classifier is positive. Note that normalizing the weights of each classifier (by the magnitude of their feature-touching weights) does not affect the validity of this labeling scheme for points lying on the positive side of just a single classifier as dividing a set of raw evaluations by a positive number does not change the mathematical *sign* of those raw evaluations: the

sole classifier that positively evaluates a point will remain the sole classifier whose signed distance to the point in question is positive.

7.2.5 Case 3. Labeling points on the positive side of no classifier

When a point $\mathbf{x}$ lies on the positive side of *none* of our C one-versus-all classifiers (or in other words, on the *negative* side of all of them), it means that each of our classifiers designates it as *not* in their respective class. Thus we cannot argue (as we did in the previous case) that one classifier is more confident in its class identity. What we can do instead is ask which classifier is the *least* confident about $\mathbf{x}$ *not* belonging to their class? The answer is *not* the decision boundary it is farthest from as was the case previously, but the one it is *closest* to. Here again the notion of *signed distance to decision boundary* comes in handy, noticing that assigning $\mathbf{x}$ to the decision boundary that it is closest to means assigning it to the boundary that it has the largest signed distance to (even though this signed distance, as well as other signed distances, are all now negative). Formally, again assuming that the weights of each classifier have been normalized, we can write

$$y = \underset{j=0,\ldots,C-1}{\text{argmax}} \ \mathring{\mathbf{x}}^T\mathbf{w}_j. \tag{7.11}$$

In the left and middle panels of Figure 7.5 we show two example points in a region of our toy dataset that lie on the *negative* side of all three of our two-class classifiers. This is the white triangular region in the middle of data. Again we highlight the distance from each point to all three decision boundaries in dashed black, with the projection of the point onto each decision boundary shown as an 'x' in the same color as its respective classifier. Starting with the point in the left panel, since it lies on the negative side of all three classifiers the best we can do is assign it to the class that it is closest to (or 'least offends'), here the blue class. Likewise the point shown in the middle panel can be assigned to the green class as it lies closest to its boundary. If we repeat this logic for every point in the region and color each point the color of its respective class, we will end up shading this central region as shown in the right panel of Figure 7.5.

7.2.6 Putting it all together

We have now deduced, by breaking down the problem of how to combine our C two-class classifiers to perform multi-class classification into three exhaustive cases, that a *single* rule can be used to assign a label y to any point $\mathbf{x}$ as

$$y = \underset{j=0,\ldots,C-1}{\text{argmax}} \ \mathring{\mathbf{x}}^T\mathbf{w}_j \tag{7.12}$$

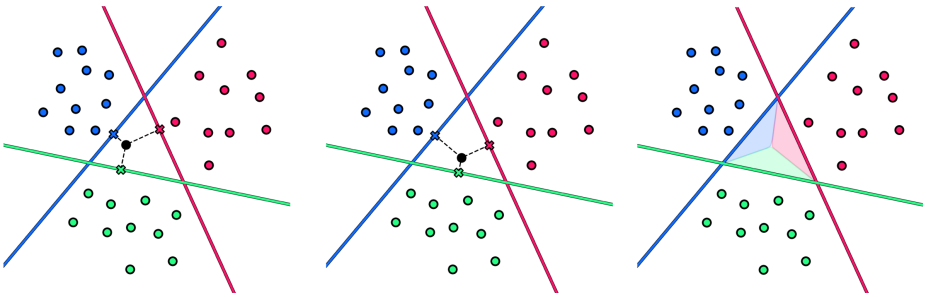

Figure 7.5 One-versus-all classification of points lying on the negative side of all classifiers. See text for further details.

assuming that the (feature-touching) weights of each classifier have been normalized as in Equation (7.8). We call this the *fusion rule* since we can think of it as a rule that fuses our C two-class classifiers together to determine the label that must be assigned to a point based on the notion of signed distance to decision boundary.

In Figure 7.6 we show the result of applying the fusion rule in Equation (7.12) to the entire input space of our toy dataset we have used throughout this Section. This includes the portions of the space where points lie (i) on the positive side of a single classifier (as shown in Figure 7.3), (ii) on the positive side of more than one classifier (as shown in the right panel of Figure 7.4), and (iii) on the positive side of none of the classifiers (as shown in the right panel of Figure 7.5). In the left panel of Figure 7.6 we show the entire spaced colored appropriately according to the fusion rule, along with the three original two-class classifiers. In the right panel we highlight (in black) the line segments that define borders between the regions of the space occupied by each class. This is indeed our *multi-class classifier* or *decision boundary* provided by the fusion rule. In general when using linear two-class classifiers the boundary resulting from the fusion rule will always be *piecewise* linear (as with our simple example here). While the fusion rule explicitly defines this piecewise linear boundary, it does not provide us with a nice, closed form formula for it (as with two class frameworks like logistic regression or SVMs).

7.2.7 The One-versus-All (OvA) algorithm

When put together, the two-step process we have now seen (i.e., learn C one-versus-all two-class classifiers and then combine them using the fusion rule) is generally referred to as the *one-versus-all multi-class classification algorithm* or OvA algorithm for short. In practice it is common to see implementations of the OvA algorithm that skip the normalization step shown in Equation (7.8). This

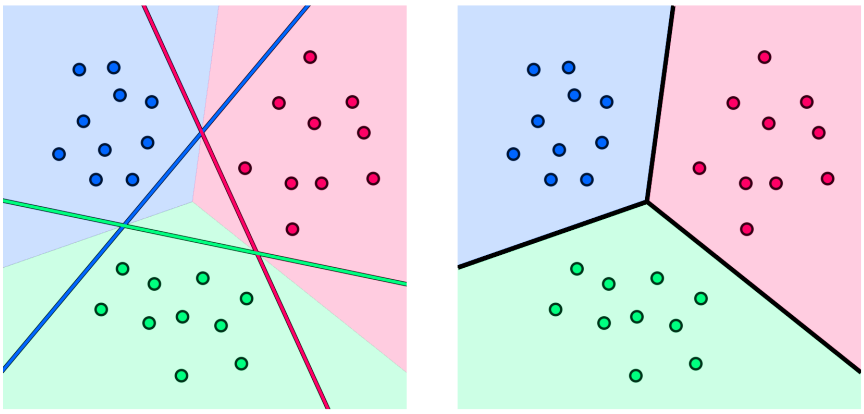

Figure 7.6 The result of applying the fusion rule in Equation (7.12) to the input space of our toy dataset, with regions colored according to their predicted label along with the original two-class decision boundaries (left panel) and fused multi-class decision boundary in black (right panel). See text for further details.

can theoretically lead to poor classification accuracy due to differently-sized normal vectors, creating out of scale distance-to-classifier measurements. However because often each classifier is trained using the same local optimization scheme the resulting magnitude of each trained normal vector can end up being around the same magnitude, hence reducing the difference between normalized and unnormalized evaluation of input points by each classifier.

Example 7.1 **Classification of a dataset with $C = 4$ classes using OvA**

In this example we apply the OvA algorithm to classify a toy dataset with $C = 4$ classes shown in the left panel of Figure 7.7. Here blue, red, green, and yellow points have label values 0, 1, 2, and 3, respectively. In the middle panel of Figure 7.7 we show the input space colored according to the fusion rule, along with the $C = 4$ individually-learned two-class classifiers. Notice that none of these two-class classifiers perfectly separates its respective class from rest of the data. Nevertheless, the final multi-class decision boundary shown in the right panel of Figure 7.7 does a fine job of distinguishing the four classes.

Example 7.2 **Regression view of the fusion rule**

In deriving the fusion rule in Equation(7.12) we viewed the problem of multi-class classification from the perceptron perspective, meaning that we viewed our data in the input feature space alone, coloring the value of each label instead of plotting our output as a dimension of the data. However if we view our multi-class data from the side, in what we call the regression perspective, we

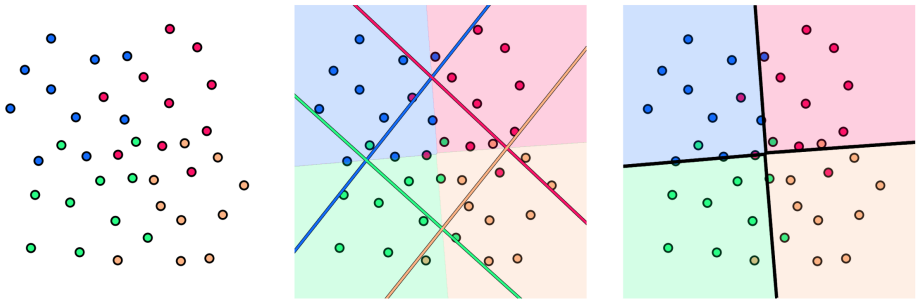

Figure 7.7 Figure associated with Example 7.1. See text for details.

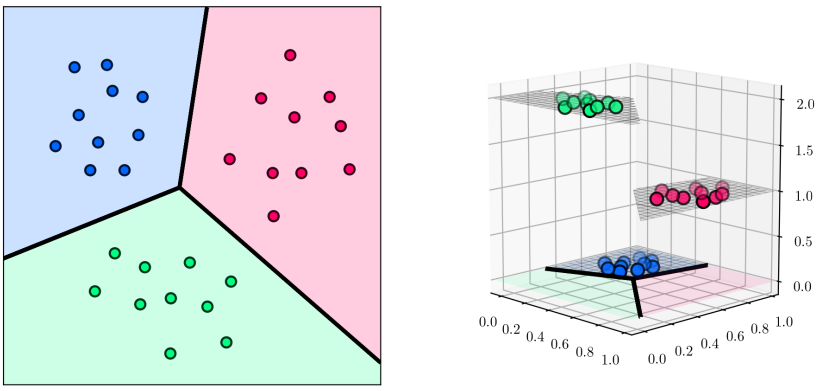

Figure 7.8 Fusion rule for a toy classification dataset with $C = 3$ classes, shown from the perceptron perspective (left panel) and from the regression perspective (right panel). See Example 7.2 for further details.

can indeed view the fusion rule as a *multi-level step function*. This is shown in Figure 7.8 for the primary toy dataset used throughout this Section. The left panel shows the data and fusion rule 'from above', while in the right panel the same setup shown 'from the side' with the fusion rule displayed as a discrete step function. Note that the jagged edges on some of the steps in the right panel are merely an artifact of the plotting mechanism used to generate the three dimensional plot. In reality the edges of each step are smooth like the fused decision boundary shown in the input space.

7.3 Multi-class classification and the Perceptron

In this Section we discuss a natural alternative to one-versus-all (OvA) multi-class classification detailed in the previous Section. Instead of training C two

class classifiers *first* and *then* fusing them into a single decision boundary (via the *fusion rule*), we train all C classifiers *simultaneously* to directly satisfy the fusion rule. In particular we derive the *multi-class Perceptron* cost for achieving this feat, which can be thought of as a direct generalization of the two-class Perceptron described in Section 6.4.

7.3.1 The multi-class Perceptron cost function

In the previous Section on OvA multi-class classification we saw how the fusion rule in Equation (7.12) defined class ownership for *every* point $\mathbf{x}$ in the input space of the problem. This, of course, includes all (input) points $\mathbf{x}_p$ in our training dataset $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$. Ideally, with all two-class classifiers properly tuned, we would like the fusion rule to hold true for as many of these points as possible

$$y_p = \underset{j=0,\ldots,C-1}{\text{argmax}} \; \mathring{\mathbf{x}}_p^T \mathbf{w}_j. \tag{7.13}$$

Instead of tuning our C two-class classifiers one-by-one and then combining them in this way, we can learn the weights of all C classifiers *simultaneously* so as to satisfy this ideal condition as often as possible.

To get started in constructing a proper cost function whose minimizers satisfy this ideal, first note that *if* Equation (7.13) is to hold for our p^{th} point then we can say that the following must be true as well

$$\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p} = \max_{j=0,\ldots,C-1} \mathring{\mathbf{x}}_p^T \mathbf{w}_j. \tag{7.14}$$

In words, Equation (7.14) simply says that the (signed) distance from the point $\mathbf{x}_p$ to its class decision boundary is *greater* than (or equal to) its distance to every other two-class decision boundary. This is what we ideally want for all of our training datapoints.

Subtracting $\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}$ from the right hand side of Equation (7.14) then gives us a good candidate for a point-wise cost function that is always non-negative and minimal at zero, defined as

$$g_p\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \left(\max_{j=0,\ldots,C-1} \mathring{\mathbf{x}}_p^T \mathbf{w}_j\right) - \mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}. \tag{7.15}$$

Notice that if our weights $\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}$ are set ideally, $g_p\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right)$ should be zero for as many points as possible. With this in mind, we can then form a cost function by taking the average of the point-wise cost in Equation (7.15) over the entire dataset, as

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \frac{1}{P} \sum_{p=1}^{P} \left[\left(\max_{j=0,\ldots,C-1} \mathring{\mathbf{x}}_p^T \mathbf{w}_j\right) - \mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}\right] \tag{7.16}$$

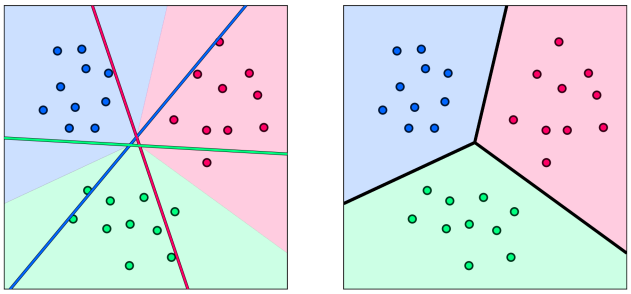

Figure 7.9 Figure associated with Example 7.3. See text for details.

This cost function, which we refer to hereafter as the *multi-class Perceptron cost*, provides a way to tune all classifier weights *simultaneously* in order to recover weights that satisfy the fusion rule as well as possible.

7.3.2 Minimizing the multi-class Perceptron cost

Like its two-class analog discussed in Section 6.4, the multi-class Perceptron is always convex regardless of the dataset employed (see Chapter's exercises). It also has a trivial solution at zero. That is, when $\mathbf{w}_j = \mathbf{0}$ for all $j = 0, ..., C-1$ the cost is minimal. This undesirable behavior can often be avoided by initializing any local optimization scheme used to minimize it away from the origin. Note also that we are restricted to using zero and first order optimization methods to minimize the multi-class Perceptron cost as its second derivative is zero (wherever it is defined).

Example 7.3 **Minimizing the multi-class Perceptron**

In this Example we minimize the multi-class Perceptron over the toy multi-class dataset originally shown in Figure 7.1, minimizing the cost via the standard gradient descent procedure (see Section 3.6).

In the left panel of Figure 7.9 we plot the dataset, the final classification over the entire space, and each individual two-class classifier. In the right panel we show the fused multi-class decision boundary formed by combining these individual classifiers via the fusion rule. Note in the left panel that because we did *not* train each individual classifier in a one-versus-all fashion, each individual learned two-class classifier performs quite poorly in separating its class from rest of the data. This is just fine as it is the two-class classifiers' combination (via the fusion rule) that provides the final decision boundary, here with zero misclassifications, as illustrated in the right panel.

7.3.3 Alternative formulations of the multi-class Perceptron cost

The multi-class Perceptron cost in Equation (7.16) can also be derived as a *direct generalization* of its two-class version introduced in Section 6.4. Using the following simple property of the max function

$$\max\left(s_0,\, s_1, ..., s_{C-1}\right) - z = \max\left(s_0 - z,\, s_1 - z, ..., s_{C-1} - z\right) \tag{7.17}$$

where $s_0,\, s_1, ..., s_{C-1}$ and z are scalar values, we can write each summand on the right hand side of Equation (7.16) as

$$\max_{j=0,...,C-1} \mathring{\mathbf{x}}_p^T \left(\mathbf{w}_j - \mathbf{w}_{y_p}\right) \tag{7.18}$$

Notice that for $j = y_p$ we have $\mathring{\mathbf{x}}_p^T \left(\mathbf{w}_j - \mathbf{w}_{y_p}\right) = 0$. This allows us to re-write the quantity in Equation (7.18) equivalently as

$$\max_{\substack{j=0,...,C-1 \\ j \neq y_p}} \left(0, \mathring{\mathbf{x}}_p^T \left(\mathbf{w}_j - \mathbf{w}_{y_p}\right)\right) \tag{7.19}$$

and hence the entire multi-class Perceptron cost as

$$g\left(\mathbf{w}_0,\, ...,\, \mathbf{w}_{C-1}\right) = \frac{1}{P} \sum_{p=1}^{P} \max_{\substack{j=0,...,C-1 \\ j \neq y_p}} \left(0, \mathring{\mathbf{x}}_p^T \left(\mathbf{w}_j - \mathbf{w}_{y_p}\right)\right). \tag{7.20}$$

In this form it is easy to see that when $C = 2$ the multi-class Perceptron cost reduces to the two-class version detailed in Section 6.4.

7.3.4 Regularizing the multi-class Perceptron

In deriving the fusion rule and sunsequently the multi-class Perceptron cost function we have assumed that the normal vector for each two-class classifier has unit length, so that we can fairly compare the (signed) distance of each input $\mathbf{x}_p$ to each of our two-class decision boundaries. This means that in minimizing the multi-class Perceptron cost in Equation (7.16) we should (at least formally) subject it to the constraints that all normal vectors have unit length, giving the *constrained* minimization problem

$$\begin{aligned} \underset{b_0, \boldsymbol{\omega}_0, ..., b_{C-1}, \boldsymbol{\omega}_{C-1}}{\text{minimize}} \quad & \frac{1}{P} \sum_{p=1}^{P} \left[\left(\max_{j=0,...,C-1} b_j + \mathbf{x}_p^T \boldsymbol{\omega}_c\right) - \left(b_{y_p} + \mathbf{x}_p^T \boldsymbol{\omega}_{y_p}\right)\right] \\ \text{subject to} \quad & \left\|\boldsymbol{\omega}_j\right\|_2^2 = 1, \quad j = 0, ..., C-1 \end{aligned} \tag{7.21}$$

where we have used the bias/feature-touching weight notation, allowing us to decompose each weight vector $\mathbf{w}_j$ as shown in Equation (7.2.4).

While this problem can be solved in its constrained form, it is more commonplace (in the machine learning community) to *relax* such a problem (as we have seen previously with the two-class Perceptron in Section 6.4.6 and the Support Vector Machine in Section 6.5.4), and solve a *regularized* version.

While in theory we could provide a distinct penalty (or regularization) parameter for each of the C constraints in Equation (7.21), for simplicity one can choose a single regularization parameter $\lambda \geq 0$ to penalize the magnitude of all normal vectors simultaneously. This way we need only provide one regularization value instead of C distinct regularization parameters, giving the regularized version of the multi-class Perceptron problem as

$$\underset{b_0, \boldsymbol{\omega}_0, \ldots, b_{C-1}, \boldsymbol{\omega}_{C-1}}{\text{minimize}} \quad \frac{1}{P}\sum_{p=1}^{P}\left[\left(\max_{j=0,\ldots,C-1} b_j + \mathbf{x}_p^T\boldsymbol{\omega}_j\right) - \left(b_{y_p} + \mathbf{x}_p^T\boldsymbol{\omega}_{y_p}\right)\right] + \lambda\sum_{j=0}^{C-1}\left\|\boldsymbol{\omega}_j\right\|_2^2. \tag{7.22}$$

This regularized form does not quite match the original constrained formulation as regularizing all normal vectors together will not necessarily guarantee that $\left\|\boldsymbol{\omega}_j\right\|_2^2 = 1$ for all j. However it will generally force the magnitude of all normal vectors to 'behave well' by, for instance, disallowing (the magnitude of) one normal vector to grow arbitrarily large while one shrinks to almost zero. As we see many times in machine learning, it is commonplace to make such compromises to get something that is 'close enough' to the original as long as it does work well in practice. This is indeed the case here, with λ typically set to a small value (e.g., 10^{-3} or smaller).

7.3.5 The multi-class softmax cost function

As with the two-class Perceptron (see Section 6.4.3), we are often willing to sacrifice a small amount of modeling precision, by forming a closely matching smoother cost function to the one we already have, in order to make optimization easier or expand the optimization tools we can bring to bear. As was the case with the two class Perceptron, here too we can *smooth* the multi-class Perceptron cost employing the *softmax* function.

Replacing the max function in each summand of the multi-class Perceptron in Equation (7.16) with its softmax approximation in Equation (6.4.3) gives the following cost function

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{j=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T\mathbf{w}_j}\right) - \mathring{\mathbf{x}}_p^T\mathbf{w}_{y_p}\right]. \tag{7.23}$$

This is referred to as the *multi-class softmax cost function*, both because it is built by smoothing the multi-class Perceptron using the softmax function, and because it can be shown to be the direct multi-class generalization of the two-class Softmax function (see e.g., Equation (6.4.3)). The multi-class Softmax cost

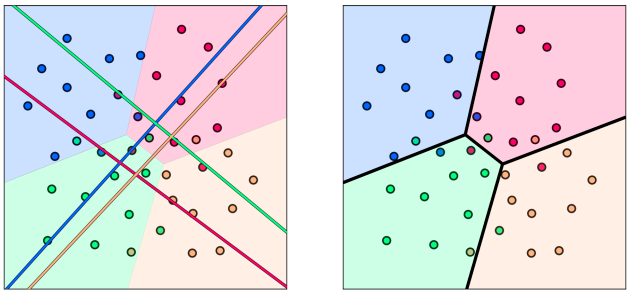

Figure 7.10 Figure associated with Example 7.4. See text for details.

function in Equation (7.23 also goes by *many* other names including the *multi-class cross entropy cost*, the *softplus cost*, and the *multi-class logistic* cost.

7.3.6 Minimizing the multi-class Softmax

Not only is the multi-class Softmax cost function convex (see Chapter's exercises) but (unlike the multi-class Perceptron) it also has infinitely many smooth derivatives, enabling us to use second order methods (in addition to zero and first order methods) in order to properly minimize it. Notice also that it no longer has a trivial solution at zero, akin to its two-class Softmax analog that removes this deficiency from the two-class Perceptron.

Example 7.4 **Minimizing the multi-class Softmax cost using Newton's method**

In Figure 7.10 we show the result of applying Newton's method to minimize the multi-class Softmax cost to a toy dataset with $C = 4$ classes, first shown in Figure 7.7.

7.3.7 Alternative formulations of the multi-class Softmax

Smoothing the formulation of the multi-class Perceptron given in Equation (7.16) by replacing the max with the softmax function gives an equivalent but different formulation of the multi-class Softmax as

$$g\left(\mathbf{w}_0, ..., \mathbf{w}_{C-1}\right) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1 + \sum_{\substack{j=0 \\ j\neq y_p}}^{C-1} e^{\mathring{\mathbf{x}}_p^T\left(\mathbf{w}_j - \mathbf{w}_{y_p}\right)}\right). \tag{7.24}$$

Visually, this formulation appears more similar to the two-class Softmax cost, and indeed does reduce to it when $C = 2$ and $y_p \in \{-1, +1\}$.

This cost function is also referred to as the *multi-class cross entropy cost* because it is, likewise, a natural generalization of the two-class version seen in Section

6.2. To see that this is indeed the case, first note that we can rewrite the p^{th} summand of the multi-class Softmax cost in Equation (7.23), using the fact that $\log\left(e^s\right) = s$, as

$$\log\left(\sum_{j=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_j}\right) - \mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p} = \log\left(\sum_{j=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_c}\right) - \log\left(e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}}\right). \tag{7.25}$$

Next, we can use the log property that $\log(s) - \log(t) = \log\left(\frac{s}{t}\right)$ to rewrite Equation (7.25) as

$$\log\left(\sum_{j=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_j}\right) - \log e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}} = \log\left(\frac{\sum_{j=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_j}}{e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}}}\right). \tag{7.26}$$

Finally, since $\log(s) = -\log\left(\frac{1}{s}\right)$ this can be rewritten equivalently as

$$\log\left(\frac{\sum_{j=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_j}}{e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}}}\right) = -\log\left(\frac{e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}}}{\sum_{j=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_j}}\right) \tag{7.27}$$

Altogether we can then express the multi-class Softmax in Equation (7.23) equivalently as

$$g\left(\mathbf{w}_0, ..., \mathbf{w}_{C-1}\right) = -\frac{1}{P}\sum_{p=1}^{P} \log\left(\frac{e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}}}{\sum_{j=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_j}}\right). \tag{7.28}$$

Visually, this formulation appears more similar to the two-class cross entropy cost in Equation (6.2.5), and indeed does reduce to it in quite a straightforward manner when $C = 2$ and $y_p \in \{0, 1\}$.

7.3.8 Regularization and the multi-class Softmax

As with the multi-class Percpetron cost (in Section 7.3.4), it is common to *regularize* the multi-class Softmax as

$$\frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{j=0}^{C-1} e^{b_j + \mathbf{x}_p^T \boldsymbol{\omega}_j}\right) - \left(b_{y_p} + \mathbf{x}_p^T \boldsymbol{\omega}_{y_p}\right)\right] + \lambda \sum_{j=0}^{C-1} \left\|\boldsymbol{\omega}_j\right\|_2^2 \tag{7.29}$$

where, once again, we have used the bias/feature-touching weight notation allowing us to decompose each weight vector $\mathbf{w}_c$ as shown in Equation (7.2.4). Regularization can also help prevent local optimization methods like Newton's method (which take large steps) from diverging when dealing with perfectly separable data (see Section 6.4).

7.3.9 Python implementation

To implement either of the multi-class cost functions detailed in this Section it is first helpful to re-write our model using the matrix notation first introduced in Section 5.5. In other words we first stack the weights from our C classifiers together into a single $(N+1) \times C$ array of the form

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & \cdots & w_{0,C-1} \\ w_{1,0} & w_{1,1} & w_{1,2} & \cdots & w_{1,C-1} \\ w_{2,0} & w_{2,1} & w_{2,2} & \cdots & w_{2,C-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ w_{N,0} & w_{N,1} & w_{N,2} & \cdots & w_{N,C-1} \end{bmatrix} \tag{7.30}$$

.

Here the bias and normal vector of the c^{th} classifier have been stacked on top of one another and make up the c^{th} column of the array. We likewise lets extend our `model` notation to also denote the evaluation of our C individual linear models together as

$$\text{model}\left(\mathbf{x}, \mathbf{W}\right) = \mathring{\mathbf{x}}^T\mathbf{W} = \begin{bmatrix} \mathring{\mathbf{x}}^T\mathbf{w}_0 & \mathring{\mathbf{x}}^T\mathbf{w}_1 & \cdots & \mathring{\mathbf{x}}^T\mathbf{w}_{C-1} \end{bmatrix}. \tag{7.31}$$

This is precisely the *same* condensed linear we used to implement multi-output regression in Section 5.5.3,which we repeat below.

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

With this `model` notation we can more conveniently implement essentially any formula derived from the fusion rule like e.g., the multi-class Perceptron. For example, we can write the fusion rule in Equation 7.12 itself equivalently as

$$y = \text{argmax}\left[\,\text{model}\left(\mathbf{x}, \mathbf{W}\right)\right]. \tag{7.32}$$

Likewise we can write the p^{th} summand of the multi-class Perceptron compactly as

$$\left(\max_{c=0,\ldots,C-1} \mathring{\mathbf{x}}_p^T\mathbf{w}_c\right) - \mathring{\mathbf{x}}_p^T\mathbf{w}_{y_p}. = \max\left[\text{model}\left(\mathbf{x}_p, \mathbf{W}\right)\right] - \text{model}\left(\mathbf{x}_p, \mathbf{W}\right)_{y_p} \tag{7.33}$$

where here the term $\text{model}\left(\mathbf{x}_p, \mathbf{W}\right)_{y_p}$ refers to the y_p^{th} entry of $\text{model}\left(\mathbf{x}_p, \mathbf{W}\right)$.

`Python` code often runs much faster when `for` loops - or equivalently `list comprehensions` - are written equivalently using matrix-vector `numpy` operations

(this has been a constant theme in our implementations since linear regression in Section 5.1.4).

Below we show an example implementation of the multi-class Perceptron in Equation 7.16 that takes in the `model` function provided above. Note that `np.linalg.fro` denotes the *Frobenius* matrix norm (see Section 7.13.3).

One can implement the multi-class Softmax in Equation 7.23 in an entirely similar manner.

```
# multiclass perceptron
lam = 10**-5  # our regularization paramter
def multiclass_perceptron(w):
    # pre-compute predictions on all points
    all_evals = model(x,w)

    # compute maximum across data points
    a = np.max(all_evals,axis = 0)

    # compute cost in compact form using numpy broadcasting
    b = all_evals[y.astype(int).flatten(),np.arange(np.size(y))]
    cost = np.sum(a - b)

    # add regularizer
    cost = cost + lam*np.linalg.norm(w[1:,:],'fro')**2

    # return average
    return cost/float(np.size(y))
```

7.4 Which approach produces the best results?

In the previous two Sections we have seen two fundamental approaches to linear multi-class classification: the one-versus-all (OvA) and the multi-class Perceptron/Softmax. Both approaches are commonly used in practice and often (depending on the dataset) produce similar results. However the latter approach is (at least in principle) capable of achieving higher accuracy on a broader range of datasets. This is due to the fact that with OvA we solve a sequence of C two-class sub-problems (one per class), tuning the weights of our classifiers *independently* of each other. Only afterward do we combine all classifiers together to form the fusion rule. Thus the weights we learn satisfy the fusion rule *indirectly*. On the other hand, with the multi-class Perceptron or Softmax cost function minimization we are tuning all parameters of all C classifiers *simultaneously* to *directly* satisfy fusion rule over our training dataset. This joint minimization permits potentially valuable interactions to take place in-between the two-class classifiers in the tuning of their weights that cannot take place in the OvA approach.

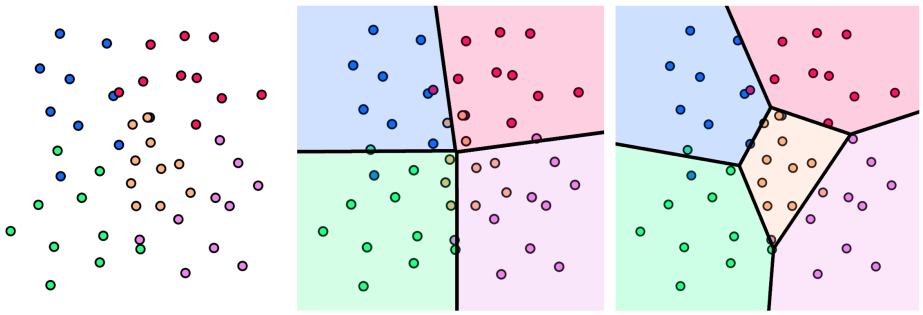

Figure 7.11 Figure associated with Example 7.5. See text for details.

Example 7.5 **Comparison of OvA and multi-class Perceptron**

We illustrate this principal superiority of the multi-class Perceptron approach over OvA using a toy dataset with $C = 5$ classes, shown in the left panel of Figure 7.11 where points colored red, blue, green, yellow, and violet have label values $y_p = 0, 1, 2, 3$, and 4, respectively.

Think for a moment how the OvA approach will perform in terms of the yellow colored class concentrated in the middle of the dataset, particularly how the sub-problem in which we distinguish between members of this class and all others will be solved. Because this class of data is surrounded by members of the other classes, and there fewer members of the yellow class then all other classes combined, the optimal linear classification rule for this sub-problem is to classify all points as non-yellow (or in other words, to misclassify the entire yellow class). This implies that the linear decision boundary will lie outside the range of the points shown, with all points in the training data lying on its *negative side*. Since the weights of decision boundary associated with the yellow colored class are tuned solely based on this sub-problem, this will lead to the entire yellow class being misclassified in the final OvA solution provided by the fusion rule, as shown in the middle panel of Figure 7.11.

On the other hand, if we employ the multi-class Perceptron or Softmax approach we will not miss this class since all $C = 5$ two-class classifiers are learned *simultaneously*, resulting in a final fused decision boundary that is far superior to the one provided by OvA, as shown in the right panel of Figure 7.11. We misclassify far fewer points and, in particular, do not misclassify the entire yellow class of data.

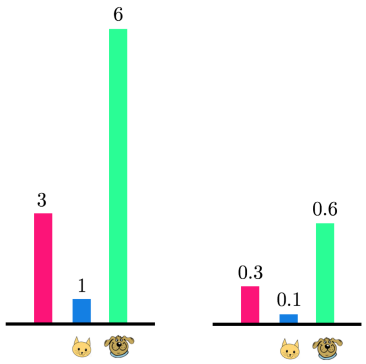

Figure 7.12 (left panel) Visual representation of a sample data vector $\mathbf{s} = \begin{bmatrix} 3, 1, 6 \end{bmatrix}$, resulting from a pet ownership survey and shown as a histogram. (right panel) Normalizing this vector by dividing it off by the sum of its entries allows us to treat it as a discrete probability distribution.

7.5 The categorical cross entropy cost function

In the previous Sections we employed by default the numerical label values $y_p \in \{0, 1, ..., C-1\}$. However, as in the case of two-class classification (see Section 6.7), the choice of label values with multi-class classification is also *arbitrary*. Regardless of how we define label values we still end up with precisely the same multi-class scheme we saw in Section 7.3 and cost functions like the multi-class Softmax.

In this Section we see how to use *categorical* labels, that is labels that have no intrinsic numerical order, to perform multi-class classification. This perspective introduces the notion of a *discrete probability distribution* as well as the notion of a *categorical cross entropy cost*, which (as we will see) is completely equivalent to the multi-class Softmax/cross entropy cost function we saw in Section 7.3.

7.5.1 Discrete probability distributions

Suppose you took a poll of 10 friends or work colleagues inquiring if they owned a pet cat or dog. From this group you learned that 3 people owned neither a cat or a dog, 1 person owned a cat, and 6 people owned dogs. Building a corresponding data vector $\mathbf{s} = \begin{bmatrix} 3, 1, 6 \end{bmatrix}$ from this survey response, $\mathbf{s}$ can be represented visually as a *histogram* where the value of each entry is represented by a vertical bar whose height is made proportional to its respective value. A histogram of this particular vector is shown in the left panel of Figure 7.12.

It is quite common to *normalize* data vectors like this one so that they can be interpreted as a *discrete probability distribution*. To do so, the normalization must be done in a way to ensure that (i) the numerical ordering of its values (from smallest to largest) is retained, (ii) its values are non-negative, and (iii) its values

sum exactly to 1. For a vector of all non-negative entries this can be done by simply dividing it by the sum of its values. With the toy example we use here since its (non-negative) values sum to 3 + 1 + 6 = 10 this requires dividing all entries of $\mathbf{s}$ by 10 as

$$\mathbf{s} = \left[0.3,\ 0.1,\ 0.6\right]. \tag{7.34}$$

This *normalized histogram* (sometimes referred to as a probability mass function) is shown in the right panel of Figure 7.12.

7.5.2 Exponential normalization

This kind of normalization can be done in general for any length C vector $\mathbf{s} = \left[s_0,\ s_1,\ ...,s_{C-1}\right]$ with potentially negative elements. Exponentiating each element in $\mathbf{s}$ gives the vector

$$\left[e^{s_0},\ e^{s_1},\ ...,e^{s_{C-1}}\right] \tag{7.35}$$

where all entries are now guaranteed to be non-negative. Notice also that exponentiation maintains the ordering of values in $\mathbf{s}$ from small to large[1]. If we now divide off the sum of this exponentiated version of $\mathbf{s}$ from each of its entries, as

$$\sigma\left(\mathbf{s}\right) = \left[\frac{e^{s_0}}{\sum_{c=0}^{C-1} e^{s_c}},\ \frac{e^{s_1}}{\sum_{c=0}^{C-1} e^{s_c}},\ \cdots,\ \frac{e^{s_{C-1}}}{\sum_{c=0}^{C-1} e^{s_c}}\right] \tag{7.36}$$

we not only maintain the two aforementioned properties (i.e., non-negativity and numerical order structure) but, in addition, satisfy the third property of a valid discrete probability distribution: all entries now sum to 1.

The function $\sigma\left(\cdot\right)$ defined in Equation (7.36) is called a *normalized exponential function*[2]. It is often used so that we may *interpret* an arbitrary vector $\mathbf{s}$ (possibly containing negative as well as positive entries) as a discrete probability distribution (see Figure 7.13), and can be thought of as a generalization of the *logistic sigmoid* introduced in Section 6.2.3.

7.5.3 Exponentially normalized signed distances

In the previous two Sections we relied on a point $\mathbf{x}$'s *signed distance* to each of the C individual decision boundaries (or something very close to it if we do not

[1] This is because the exponential function $e^{(\cdot)}$ is always monotonically increasing.

[2] This function is sometimes referred to as *softmax activation* in the context of neural networks. This naming convention is unfortunate, as the normalized exponential is *not* a soft version of the max function as the rightly named *softmax function* detailed in Section 7.2.5 is, and should not be confused with it. While it is a transformation that does preserve the index of the largest entry of its input, it is *not* a soft version of the argmax function either as it is sometimes erroneously claimed to be.

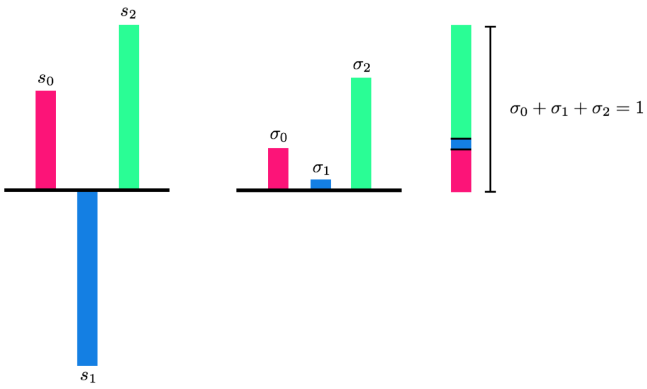

Figure 7.13 (left panel) A vector of length $C = 3$ shown as a histogram. (right panel) Taking the normalized exponential of this vector, as defined in Equation (7.36), produces a new vector of all non-negative entries whose numerical order is preserved, and whose total value sums to 1.

normalize feature-touching weights) to properly determine class membership. This is codified directly in the fusion rule (given in e.g., Equation 7.12) itself.

For a given setting of our weights for all C two-class classifiers the evaluation of $\mathbf{x}$ through all decision boundaries produces C signed distance measurements

$$\mathbf{s} = \begin{bmatrix} \mathring{\mathbf{x}}^T\mathbf{w}_0 & \mathring{\mathbf{x}}^T\mathbf{w}_1 & \cdots & \mathring{\mathbf{x}}^T\mathbf{w}_{C-1} \end{bmatrix} \tag{7.37}$$

which we can think of as a *histogram*.

Because - as we have seen above - the normalized exponential function preserves numerical order we can likewise consider the exponentially normalized signed distance in determining proper class ownership. Denoting the $\sigma(\cdot)$ the normalized exponential our generic histogram of signed distances becomes

$$\sigma(\mathbf{s}) = \begin{bmatrix} \frac{e^{\mathring{\mathbf{x}}^T\mathbf{w}_0}}{\sum_{c=0}^{C-1} e^{\mathring{\mathbf{x}}^T\mathbf{w}_c}} & \frac{e^{\mathring{\mathbf{x}}^T\mathbf{w}_1}}{\sum_{c=0}^{C-1} e^{\mathring{\mathbf{x}}^T\mathbf{w}_c}} & \cdots & \frac{e^{\mathring{\mathbf{x}}^T\mathbf{w}_{C-1}}}{\sum_{c=0}^{C-1} e^{\mathring{\mathbf{x}}^T\mathbf{w}_c}} \end{bmatrix}. \tag{7.38}$$

Transforming the histogram of signed distance measurements also gives us a way of considering class ownership *probabilistically*. For example, if for a particular setting of the entire set of weights gave for a particular point $\mathbf{x}_p$

$$\sigma(\mathbf{s}) = \begin{bmatrix} 0.1, & 0.7, & 0.2 \end{bmatrix} \tag{7.39}$$

then while we would still *assign a label based on the fusion rule* (see Equation 7.12) - here assigning the label $y_p = 1$ since the second entry 0.7 of this vector is largest - we could also add a note of confidence that "$y_p = 1$ with 70% probability".

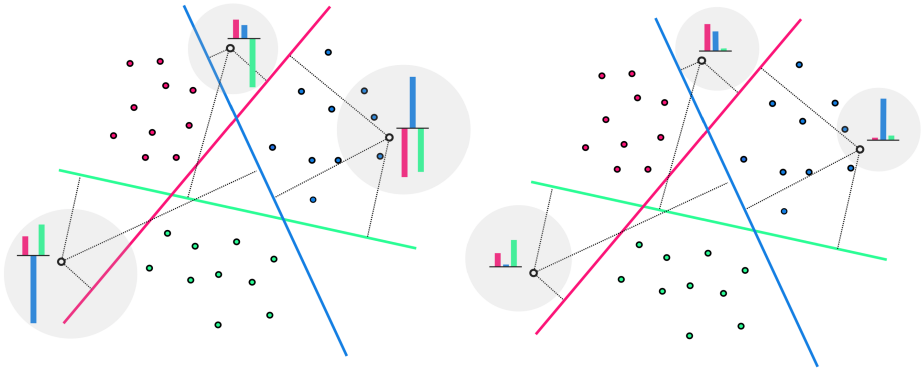

Figure 7.14 (left panel) Histogram visualizations of signed distance measurements of three exemplar points in a $C = 3$ class dataset. (right panel) Exponentially normalized signed distance measurements visualized as histograms for the same three points.

Example 7.6 **Signed-distances as a probability distribution**

In Figure 7.14 we use the prototype $C = 3$ class dataset (shown in Figure 7.1) and visualize both signed distance vectors as histograms for several points (left panel) as well as their normalized exponential versions (right panel). Note how the largest positive entry in each original histogram shown on the left panel remains the largest positive entry in the normalized version shown on the right panel.

7.5.4 Categorical classification and the categorical Cross-Entropy cost

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

$$\begin{aligned} y_p = 0 \longleftarrow \mathbf{y}_p &= \left[1, 0, \cdots 0, 0\right] \\ y_p = 1 \longleftarrow \mathbf{y}_p &= \left[0, 1, \cdots 0, 0\right] \\ &\vdots \\ y_p = C-1 \longleftarrow \mathbf{y}_p &= \left[0, 0, \cdots 0, 1\right]. \end{aligned} \tag{7.40}$$

Here each one-hot encoded *categorical label* is a length C vector and contains all zeros except a 1 in the index equal to the value of y_p (note that the first entry in this vector has index 0 and the last $C-1$).

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 $\mathbf{y}_p$ are one-hot encoded labels defined as above. Our goal, however, remains the same: to properly tune the weights of our C one-versus-all two-class classifiers to learn the best correspondence between the N dimensional input and C dimensional output of our training dataset.

With vector output, instead of scalar numerical values, we can phrase multi-class classification as an instance of *multi-output regression* (see Section 5.5). In other words, denoting $\mathbf{W}$ the $(N + 1 \times C)$ matrix of weights for all C classifiers (see Section 7.3.9) we can aim at tuning $\mathbf{W}$ so that the approximate linear relationship holds for all our points

$$\mathring{\mathbf{x}}_p^T \mathbf{W} \approx \mathbf{y}_p. \tag{7.41}$$

However since know our output $\mathbf{y}_p$ does not consist of continuous values but are one-hot encoded vectors, a linear relationship would not represent such vectors very well at all. Entries of the left hand side for a given p can be non-negative, less than zero or greater than one, etc, However taking the *normalized exponential* transform of our linear model normalizes it in such a way (by forcing all its entries to be non-negative and sum exactly to one) so that we could reasonably propose to tune $\mathbf{W}$ so that

$$\sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{W}\right) \approx \mathbf{y}_p \tag{7.42}$$

holds as tightly as possible over our training dataset (where $\sigma(\cdot)$ is the normalized exponential). Interpreting $\sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{W}\right)$ as a discrete probability distribution, this is saying that we want to tune the weights of our model so that this distribution concentrates completely at index y_p i.e., the only non-zero entry of the one-hot encoded output $\mathbf{y}_p$.

To learn our weights properly we could employ a standard pointwise regression cost like e.g., the Least Squares (see Section 5.1)

$$g_p(\mathbf{W}) = \left\|\sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{W}\right) - \mathbf{y}_p\right\|_2^2. \tag{7.43}$$

However, as we discussed in Section 6.2 a more appropriate pointwise cost when dealing with binary output is the Log Error since it more heavily penalizes error in such instances. Here the Log Error of $\sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{W}\right)$ and $\mathbf{y}_p$ can be written as

$$g_p(\mathbf{W}) = -\sum_{c=0}^{C} y_{p,c} \log \sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{W}\right)_c. \tag{7.44}$$

Note that this formula simplifies considerably because $\mathbf{y}_p$ is a one-hot encoded vector, and so all but one summand on the right hand side above equals zero. This is precisely the index y_p, where the one-hot encoded vector $\mathbf{y}_p$ equals one, meaning that the above simplifies too

$$g_p\left(\mathbf{W}\right) = -\log\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{W}\right)_{y_p} \tag{7.45}$$

and from the definition of the normalized exponential this is precisely

$$g_p\left(\mathbf{W}\right) = -\log\left(\frac{e^{\mathring{\mathbf{x}}_p^T\mathbf{w}_{y_p}}}{\sum_{c=0}^{C-1}e^{\mathring{\mathbf{x}}_p^T\mathbf{w}_c}}\right). \tag{7.46}$$

If we then form a cost function by taking the average of the above over all P training datapoints we have

$$g\left(\mathbf{W}\right) = -\frac{1}{P}\sum_{p=1}^{P}\log\left(\frac{e^{\mathring{\mathbf{x}}_p^T\mathbf{w}_{y_p}}}{\sum_{c=0}^{C-1}e^{\mathring{\mathbf{x}}_p^T\mathbf{w}_c}}\right). \tag{7.47}$$

This is precisely a form of the standard Multi-class Cross Entropy / Softmax cost function we saw in Section 7.3.7 in Equation 7.3.7 where we used numerical label values $y_p \in \{0, 1, ..., C-1\}$.

7.6 Making predictions and measuring the quality of a trained model

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

7.6.1 Making predictions using a trained model

If we denote the optimal set of weights for the c^{th} two-class classifier found by minimizing a multi-class classification cost function in Section 7.3 (or via performing OvA as in Section 7.2) as $\mathbf{w}_c^\star$, then to predict the label y of an input $\mathbf{x}$ we employ the fusion rule as

$$y = \underset{c=0,...,C-1}{\text{argmax}}\ \mathbf{x}^T\mathbf{w}_c^\star \tag{7.48}$$

where any point lying *exactly* on the decision boundary should be assigned a label randomly based on the index of those classifiers providing maximum evaluation. This concept is illustrated in Figure 7.15 using the toy $C = 3$ class dataset first shown in Figure 7.1.

7.6.2 Confidence scoring

Once a proper decision boundary is learned we can describe the *confidence* we have in any point based on *the point's distance to the decision boundary*, in the same way we can with two-class data (see Section 6.8.2). More specifically we can its

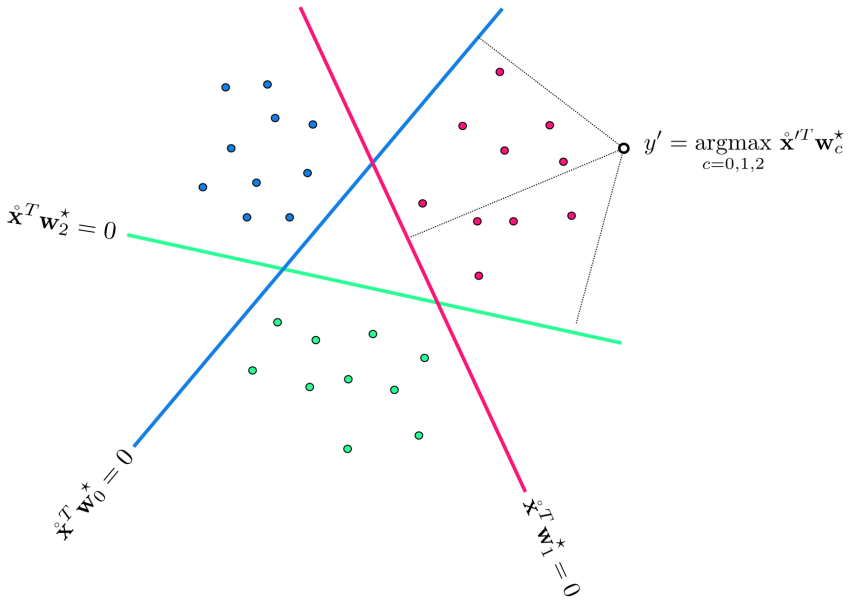

Figure 7.15 With all properly weights tuned we use the fusion rule to predict the label of new input. Here (using a $C = 3$ class dataset) the label of a new point, shown as a hollow circle in the upper-right, is predicted.

the exponentially normalized distance to score our confidence in the prediction, as described in Section 7.5.3.

7.6.3 Judging the quality of a trained model using accuracy

To count the number of misclassifications a trained multi-class classifier forms over our training dataset we simply take a raw count of the number of training datapoints $\mathbf{x}_p$ whose true label y_p is predicted *incorrectly*. To compare the point $\mathbf{x}_p$'s predicted label $\hat{y}_p = \underset{j=0,...,C-1}{\text{argmax}}, , \mathbf{x}_p^T\mathbf{w}_j^\star$ and true true label y_p we can use an identity function $\mathcal{I}(\cdot)$ and compute

$$\mathcal{I}\left(\hat{y}_p,\, y_p\right) = \begin{cases} 0 & \text{if } \hat{y}_p, = y_p \\ 1 & \text{if } \hat{y}_p, \neq y_p. \end{cases} \tag{7.49}$$

Taking a sum of the above over all P points gives the total number of misclassifications of our trained model

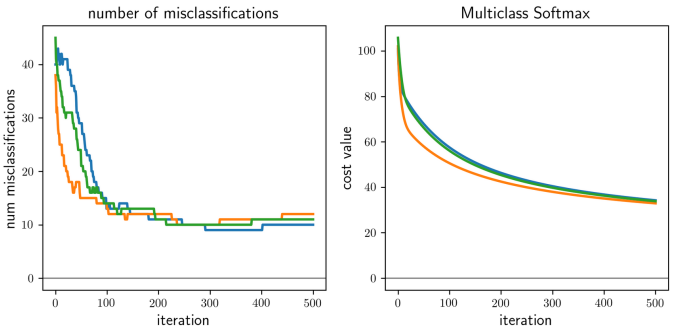

Figure 7.16 Figure associated with Example 7.7. See text for further details.

$$\text{number of misclassifications} = \sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p,\, y_p\right). \tag{7.50}$$

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.

$$\mathcal{A} = 1 - \frac{1}{P}\sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p,\, y_p\right). \tag{7.51}$$

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

Example 7.7 **Comparing cost function and counting cost values**

In Figure 7.16 we show compare the number of misclassifications versus the value of the multi-class Softmax cost in classifying the $C = 5$ class dataset shown in Figure 7.11 over three runs of standard gradient descent using a steplength parameter $\alpha = 10^{-2}$ for all three 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 (very much akin to the two-class case, see Example 7.7). This occurs because our Softmax cost is only an approximation of the true quantity we would like to minimize, i.e., the number of misclassifications.

This simple example has an extremely practical implication: after a running a local optimization to minimize a multi-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.

$$\begin{array}{c} \\ \text{red} \\ \text{blue} \\ \text{green} \\ \text{yellow} \end{array} \begin{array}{c} \begin{array}{cccc} \text{red} & \text{blue} & \text{green} & \text{yellow} \end{array} \\ \begin{pmatrix} 8 & 1 & 1 & 0 \\ 1 & 7 & 1 & 1 \\ 1 & 1 & 7 & 1 \\ 0 & 1 & 1 & 8 \end{pmatrix} \end{array}$$

Figure 7.17 Figure associated with Example 7.8. See text for details.

7.6.4 Advanced quality metrics for dealing with unbalanced classes

The advanced quality metrics we saw in the case of two-class classification to deal with severe class imbalance, including balanced accuracy (see Section 6.8.4) as well as further metrics defined by a confusion matrix (see Section 6.8.5), can be directly extended to deal with class-imbalance issues in multi-class context. These direct extensions are explored further in this Chapter's exercises.

Example 7.8 A multi-class confusion matrix

In the left panel of Figure 7.17 we show the result of a fully tuned multi-class classifier trained to the dataset first shown in Figure 7.7. The *confusion matrix* corresponding to this classifier is shown in the right panel of the Figure. The $(i, j)^{\text{th}}$ of this matrix counts the entry is the number of training data points that have *true* label $y = i$ and *predicted* label $\hat{y} = j$.

7.7 Weighted multi-class classification

Weighted multi-class classification arises for precisely the same reasons described for two-class classification in Section 6.9, that is as a way of including a notion of confidence in datapoints and for dealing with severe class imbalances. One can easily derive weighted versions of the mult-class Perceptron / Softmax cost functions that completely mirror the two-class analog detailed in this earlier Section (see Exercise 9).

7.8 Stochastic and mini-batch learning

In Section 3.11 we discussed the concept of stochastic and more mini-batch extensions of local optimization algorithms in order to accelerate the minimization of cost functions that consist of P terms of the form

$$g\left(\mathbf{w}\right) = \sum_{p=1}^{P} g_p\left(\mathbf{w}\right) \tag{7.52}$$

where $g_1\, g_2$, ..., g_P are all functions of the same kind that take in the same parameters. As we have seen, every supervised learning cost function looks like this - including those used for regression, two-class, and mult-class classification. Each g_p is what we have generally referred to as a pointwise cost that measures the error of a particular model on the p^{th} point of a dataset. For example, with the Least Squares cost we saw in Section 5.1 the pointwise cost took the form $g_p\left(\mathbf{w}\right) = \left(\mathring{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2$, with the two-class Softmax (in Section 6.4.3 it took the form $g_p\left(\mathbf{w}\right) = -\log\left(\sigma\left(y_p\,\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)$, and the mult-class Softmax (Section 7.3.5) the form $g_p\left(\mathbf{W}\right) = \left[\log\left(\sum_{c=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T\mathbf{w}_c}\right) - \mathring{\mathbf{x}}_p^T\mathbf{w}_{y_p}\right]$. More generally, as we will see moving forward (like with e.g., the Linear Autoencoder described in Section **??**), *every machine learning cost function takes this form* (because they always decompose over their training data) where g_p is a pointwise cost of the p^{th} point of a dataset. Because of this we can directly apply mini-batch optimization in tuning their parameters.

7.8.1 Mini-batch optimization and online learning

As detailed in Section 3.11, the heart of the mini-batch idea is to minimize such a cost *sequentially* over small mini-batches of its summands, one mini-batch of summands at a time, as opposed to the standard local optimization step where minimize in the entire set of summands at once. Now, since machine learning cost summands are inherently tied to training datapoints, in the context of machine learning we can think about mini-batch optimization *equivalently in terms of mini-batches of training data* as well. Thus as opposed to a standard (also called *full batch*) local optimization that takes individual steps by sweeping through an *entire* set of training data *simultaneously*, the mini-batch approach has us take smaller steps sweeping through training data *sequentially* (with one complete sweep through the data being referred to as an *epoch*). This machine learning / data driven interpretation of mini-batch optimization is illustrated schematically in the Figure 7.18.

As with generic costs, mini-batch learning often greatly accelerates the minimization of machine learning cost functions (and thus the corresponding learning taking place) - and is most popularly paired with gradient descent (Section 3.6) or one of its advanced analogs (see Section 3.10). This is particularly true when dealing with *very large datasets*, i.e., when P is large. With very large datasets the mini-batch approach can also help limit the amount of active memory consumed in storing data by loading in - at each step in a mini-batch epoch - *only the data included in the current mini-batch*. The mini-batch approach can also be used (or interpreted) as a so-called *online learning* technique, wherein data ac-

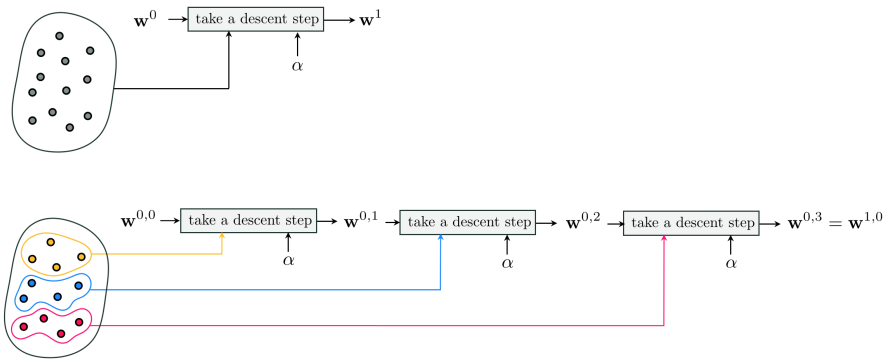

Figure 7.18 Schematic comparison of first iteration of (top panel) full batch and (bottom) stochastic gradient descent, through the lens of machine learning using (for the sake of simplicity) a small dataset of points. In the full batch sweep we take a step in all points *simultaneously*, whereas (bottom panel) in the mini-batch approach we sweep through these points *sequentially* as if we received the data in an *online fashion*.

tually *arises* in small mini-batches and is directly used to update the parameters of the associated model (see e.g., Exercise 10).

7.9 Exercises

Section 7.1 exercises

1. **One-versus-All classification pseudo-code**

   Write down a psuedo-code that summarizes the One-versus-All (OvA) algorithm described in Section 7.2.

2. **One-versus-All classification**

   Repeat the experiment described in Example 7.1. You need not reproduce the illustrations shown in Figure 7.7, however you should make sure your trained model achieves a similar result (in terms of the number of misclassifications) as the one shown there (having less than 10 misclassifications).

Section 7.2 exercises

3. **Multi-class Perceptron**

Repeat the experiment described in Example 7.3. You need not reproduce the illustrations shown in Figure 7.9, however you should make sure your trained model achieves zero misclassifications. You can use the Python implementation outlined in Section 7.3.9 of the multi-class Perceptron cost.

4. **The multi-class and two-class Perceptrons**

Finish the argument started in Section 7.3.3 to show that the multi-class Perceptron cost in Equation 7.16 reduces to the two-class Perceptron cost in Equation 6.4.1.

5. **Multi-class Softmax**

Repeat the experiment described in Example 7.4 using any local optimization method. You need not reproduce the illustrations shown in Figure 7.10, however you should make sure your trained model achieves a small number of misclassifications (10 or fewer). You can use the Python implementation outlined in Section 7.3.9 as a basis for you implementation of the multi-class Softmax cost.

6. **Show the multi-class Softmax reduces to two-class Softmax when** $C = 2$

Finish the argument started in Section 7.3.7 to show that the multi-class Softmax cost in Equation 7.23 reduces to the two-class Softmax cost in Equation 6.4.3.

7. **Hand-calculations with the multi-class Softmax cost**

Show that the Hessian of the multi-class Softmax cost function can be computed block-wise as follows. For $s \neq c$ we have $\nabla^2_{\mathbf{w}_c\mathbf{w}_s} g = -\sum_{p=1}^{P} \frac{e^{\mathbf{x}_p^T\mathbf{w}_c + \mathbf{x}_p^T\mathbf{w}_s}}{\left(\sum_{d=1}^{C} e^{\mathbf{x}_p^T\mathbf{w}_d}\right)^2} \mathbf{x}_p\mathbf{x}_p^T$ and the second derivative block in $\mathbf{w}_c$ is given as $\nabla^2_{\mathbf{w}_c\mathbf{w}_c} g = \sum_{p=1}^{P} \frac{e^{\mathbf{x}_p^T\mathbf{w}_c}}{\sum_{d=1}^{C} e^{\mathbf{x}_p^T\mathbf{w}_d}} \left(1 - \frac{e^{\mathbf{x}_p^T\mathbf{w}_c}}{\sum_{d=1}^{C} e^{\mathbf{x}_p^T\mathbf{w}_d}}\right) \mathbf{x}_p\mathbf{x}_p^T$.

Section 7.5 exercises

8. **Balanced accuracy in the multi-class setting**

   Extend the notion of balanced accuracy, detailed in the context of two-class classification (see Section 6.8.4), to the multi-class setting. In particular give an equation for the balanced accuracy in the context of multi-class classification that is analogous to Equation 6.8.4 for the two-class case.

Section 7.6 exercises

9. **Weighted multi-class Softmax**

   A general weighted version of the two-class Softmax cost is given in Equation 6.9.1. What will the analogous weighted multi-class Softmax cost function look like? If we set the weights of cost function to deal with class imbalance - as detailed for the two-class case in Section 6.9.3 - how should we set the weight values?

Section 7.7 exercises

10. **Recognizing handwritten digits**

    Recognizing handwritten digits is a popular multi-class classification problem commonly built into the software of mobile banking applications, as well as more traditional Automated Teller Machines, to give users e.g., the ability to automatically deposit paper checks. Here each class of data consists of (images of) several handwritten version of a single digit in the range $0 - 9$, giving a total of 10 classes.

    In Figure 7.20 we illustrate the accelerated convergence of mini-batch gradient descent over the standard method using $P = 50,000$ random training points from the MNIST dataset, a popular collection of handwritten images like those described above, and the multi-class Softmax cost. In particular we show a comparison of the first 10 steps / epochs of both methods, using a batch of size 200 for the mini-batch size and the same steplength for both runs, where we see that the mini-batch run drastically accelerates minimization in terms of both the cost function (left panel) and number of misclassifications (right panel).

    Re-create this Figure by implementing mini-batch gradient descent.Re-create this Figure by implementing mini-batch gradient descent. You may not get precisely the same results based on your implementation, initialization of the algorithm, etc., however you should be able to re-create the general effect.

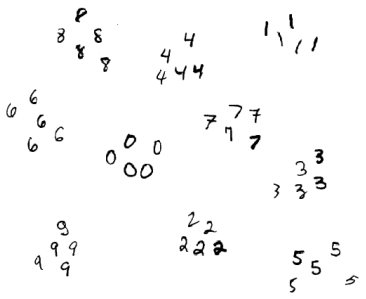

Figure 7.19 Figure associated with Exercise 10) An illustration of various handwritten digits in a feature space. Handwritten digit recognition is a common multi-class classification problem. The goal here is to determine regions in the feature space where current (and future) instances of each type of handwritten digit are present.

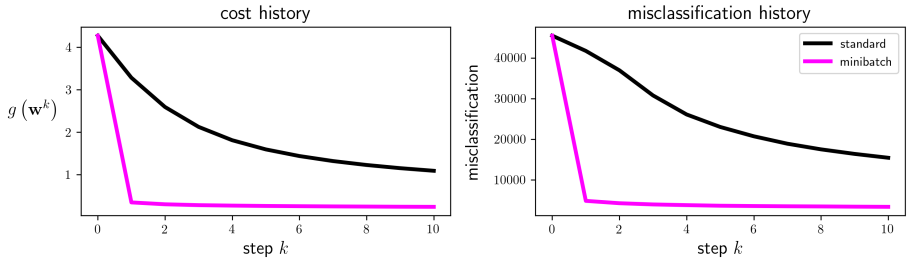

Figure 7.20 Figure associated with Exercise 10. A comparison of the progress made by standard (in black) and mini-batch (in magenta) gradient descent in over the MNIST training dataset in terms of cost function value (left panel) and number of misclassifications (right panel). The mini-batch approaches makes significantly faster progress than standard gradient descent in both measures.

7.10 16.1 Vectors and vector operations

7.10.1 16.1.1 The vector

A vector is another word for an ordered listing of numbers. For example, the following

$$[-3,\ 4,\ 1] \tag{7.53}$$

is a vector of three *elements* or *entries*, also referred to as a vector of *length* or *dimension* three. In general a vector can have an arbitrary number of elements, and can contain numbers, variables, or both. For example,

$$[x_1,\ x_2,\ x_3,\ x_4] \tag{7.54}$$