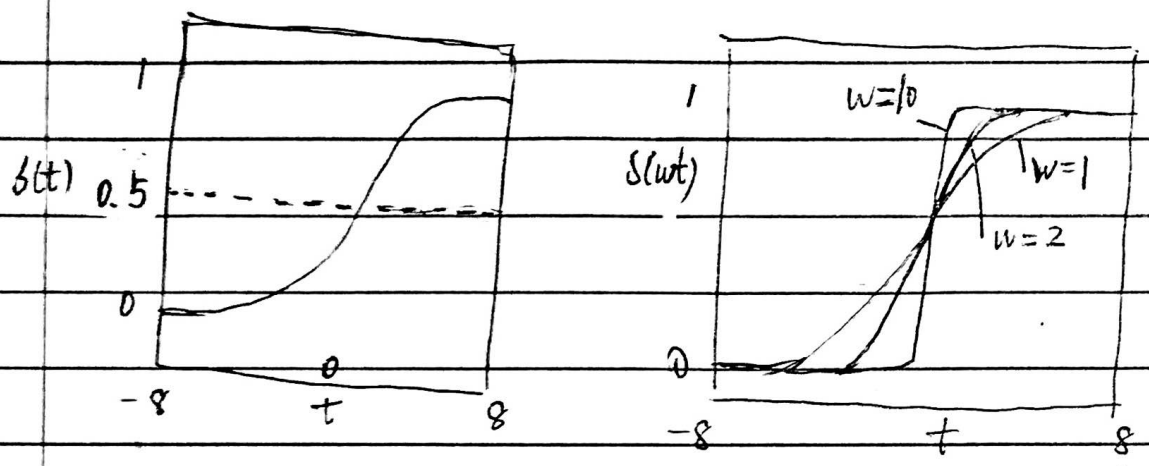
The illustration below is from *MACHINE LEARNING REFINED Foundations, Algorithms, and Applications* written by Jeremy Watt, Reza Borhani and Aggelos K. Katsaggelos.

Logistic sigmoid function

$$\delta(t) = \frac{1}{1+e^{-t}}$$



If a dataset of P points $\{(\bar{x}_p, y_p)\}_{p=1}^{P}$ is roughly distributed like a sigmoid function, then this data satisfies

$$\delta(b + \bar{x}_p^T \bar{w}) \approx y_p \quad p = 1, \ldots, P$$

where $\bar{x}_p = [x_{1,p} \;\; x_{2,p} \;\; \ldots \;\; x_{N,p}]^T$ and $\bar{w} = [w_1 \;\; w_2 \;\; \ldots \;\; w_N]^T$

∴ Least squares cost function which is formed by summing the squared differences:

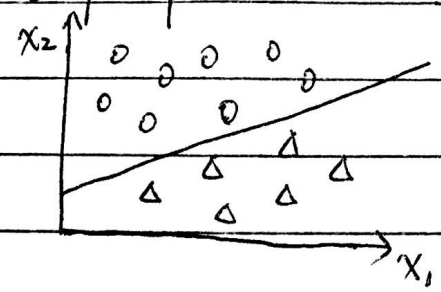
$$g(b, w) = \sum_{p=1}^{P} (\delta(b + \bar{x}_p^T \bar{w}) - y_p)^2$$

if $\tilde{x}_p = \begin{bmatrix} 1 \\ \bar{x}_p \end{bmatrix}$ and $\tilde{w} = \begin{bmatrix} b \\ \bar{w} \end{bmatrix}$ and $\delta'(t) = \delta(t)(1 - \delta(t))$

$$\nabla g(\tilde{w}) = 2 \sum_{p=1}^{P} (\delta(\tilde{x}_p^T \tilde{w}) - y_p)\, \delta(\tilde{x}_p^T \tilde{w})\, (1 - \delta(\tilde{x}_p^T \tilde{w}))\, \tilde{x}_p$$

Classification

① The basic perceptron model.



We want to learn a hyperplane $b + \bar{x}^T \bar{w} = 0$ that separates that two classes of points. If a given hyperplane places the point $\bar{x}_p$ on its correct side, then we have precisely that

$$b + \bar{x}_p^T \bar{w} > 0 \quad \text{if } y_p = +1$$

$$b + \bar{x}_p^T \bar{w} < 0 \quad \text{if } y_p = -1$$

$$\Rightarrow \quad -y_p (b + \bar{x}_p^T \bar{w}) < 0$$

If a hyperplane correctly classifies the point $\bar{x}_p$, it means

$$\max(0, -y_p(b + \bar{x}_p^T \bar{w})) = 0$$

max cost function:

$$g_1(b, \bar{w}) = \sum_{p=1}^{P} \max(0, -y_p (b + \bar{x}_p^T \bar{w}))$$

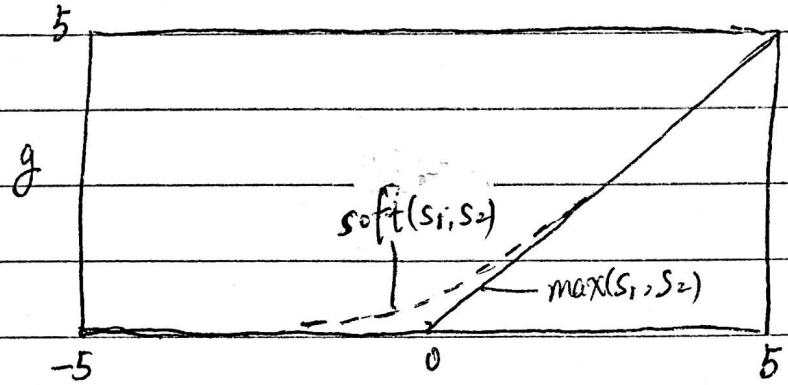
Solving the minimization problem,

$$\underset{b, \bar{w}}{\text{minimize}} \sum_{p=1}^{P} \max(0, -y_p(b + \bar{x}_p^T \bar{w}))$$

Two technical issues: 1. One minimum of g_1 always presents itself at the trivial and undesirable values $b=0$ and $\bar{w} = \bar{0}_{n \times 1}$ (which gives $g_1 = 0$). 2. Note that while g_1 is continuous, it is not everywhere differentiable.

The softmax cost function.

$$soft(s_1, s_2) = \log(e^{s_1} + e^{s_2})$$



$$soft(0, -y_p(b + \bar{x}_p \bar{w})) = \log(1 + e^{-y_p(b + \bar{x}_p^T \bar{w})})$$

cost function: $g_2(b, \bar{w}) = \sum_{p=1}^{P} \log(1 + e^{-y_p(b + \bar{x}_p^T \bar{w})})$

∴ The softmax minimization problem is written as

$$\underset{b, \bar{w}}{\text{minimize}} \sum_{p=1}^{P} \log(1 + e^{-y_p(b + \bar{x}_p^T \bar{w})})$$

Using the compact notation $\tilde{x}_p = \begin{bmatrix} 1 \\ \bar{x}_p \end{bmatrix}$ and $\tilde{w} = \begin{bmatrix} b \\ \bar{w} \end{bmatrix}$ we can rewrite the softmax cost function as

$$g_2(\tilde{w}) = \sum_{p=1}^{P} \log(1 + e^{-y_p \tilde{x}_p^T \tilde{w}})$$

$$\therefore \boxed{\nabla g_2(\tilde{w}) = -\sum_{p=1}^{P} \sigma(-y_p \tilde{x}_p^T \tilde{w}) y_p \tilde{x}_p = \bar{0}_{(N+1)\times 1}}$$

$$\nabla^2 g_2(\tilde{w}) = \sum_{p=1}^{P} \sigma(-y_p \tilde{x}_p^T \tilde{w})(1 - \sigma(-y_p \tilde{x}_p^T \tilde{w})) \tilde{x}_p \tilde{x}_p^T$$

Looking closely at the soft-margin cost we can see that, practically speaking, it is just the margin perceptron cost given in (4.16) with the addition of an ℓ_2 regularizer (as described in Section 3.3.2).

Practically speaking, the soft-margin SVM cost is just an ℓ_2 regularized form of the margin perceptron cost.

As with the original margin perceptron cost described in Section 4.1, differentiable approximations of the same sort we have seen before (e.g., squaring the "max" function or using the softmax approximation) are typically used in place of the margin perceptron component of the soft-margin SVM cost function. For example, using the softmax approximation (see Section 4.1.2) the soft-margin SVM cost may be written as

$$g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)}\right) + \lambda \left\|\mathbf{w}\right\|_2^2 . \tag{4.44}$$

With this approximation the soft-margin SVM cost is sometimes referred to as *log-loss SVM* (see e.g., [21]). However, note that, using the softmax approximation, we can also think of log-loss SVM as an ℓ_2 regularized form of logistic regression. ℓ_2 regularization, first described in Section 3.3 in the context of nonlinear regression, can be analogously applied to classification cost functions as well.

4.3.4 Support vector machines and logistic regression

While the motives for formally deriving the SVM and logistic regression classifiers differ significantly, due to the fact that their cost functions are so similar (or the same if the softmax cost is employed for SVM as in (4.44)) both perform similarly well in practice (as first discussed in Section 4.1.7). Unsurprisingly, as we will see later in Chapters 5 through 7, both classifiers can be extended (using so-called "kernels" and "feed-foward neural networks") in precisely the same manner to perform nonlinear classification.

While the motives for formally deriving the SVM and logistic regression classifiers differ, due to their similar cost functions (which in fact can be entirely similar if the softmax cost is employed for SVM) both perform similarly well in practice.

4.4 Multiclass classification

In practice many classification problems have more than two classes we wish to distinguish, e.g., face recognition, hand gesture recognition, recognition of spoken phrases or

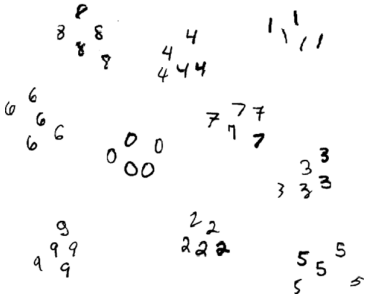


Fig. 4.16 Various handwritten digits in a feature space. Handwritten digit recognition is a common multiclass classification problem. The goal here is to determine regions in the feature space where current (and future) instances of each type of handwritten digit are present.

words, etc. Such a multiclass dataset $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ consists of C distinct classes of data, where each label y_p now takes on a value between 1 and C, i.e., $y_p \in \{1, 2, \ldots, C\}$. In this section we discuss two popular generalizations of the two class framework, namely, *one-versus-all* and *multiclass softmax classification* (sometimes referred to as *softmax regression*). Each scheme learns C two class linear separators to deal with the multiclass setting, differing only in how these linear separators are learned. Both methods are commonly used and perform similarly in practice, as we discuss further in Section 4.4.4.

Example 4.4 Handwritten digit recognition

Recognizing handwritten digits is a popular multiclass 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 versions of a single digit in the range $0 - 9$, giving a total of ten classes. Using the methods discussed in this section, as well as their nonlinear extensions described in Section 6.3, we aim to learn a separator that distinguishes each of the ten classes from each other (as illustrated in Fig. 4.16). You can perform this task on a large dataset of handwritten digits by completing Exercise 4.16.

4.4.1 One-versus-all multiclass classification

Because it has only two sides, a single linear separator is fundamentally insufficient as a mechanism for differentiating between more than two classes of data. To overcome this shortcoming when dealing with $C > 2$ classes we can instead learn C linear classifiers (one per class), each distinguishing one class from the rest of the data. We illustrate this idea for a particular toy dataset with $C = 3$ classes in Fig. 4.17. By properly fusing these C learned linear separators, we can then form a classification rule for the entire dataset. This approach is called one-versus-all (OvA) classification.

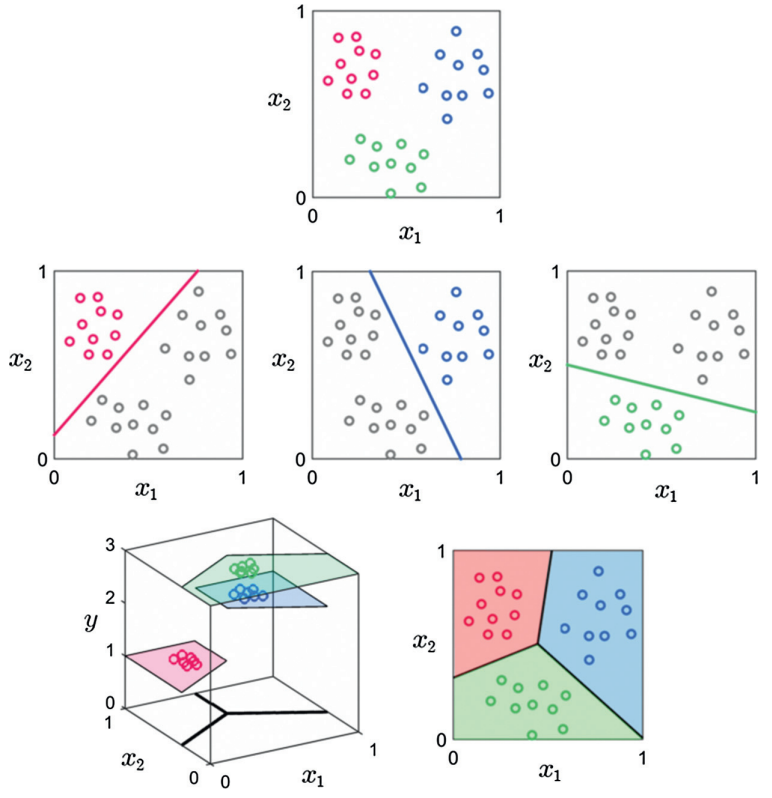


Fig. 4.17 One-versus-all multiclass scheme applied to (top panel) a toy classification dataset with $C = 3$ classes consisting of $P = 30$ data points in total (10 per class). (middle panels) The three classifiers learned to distinguish each class from the rest of the data. In each panel we have temporarily colored all data points not in the primary class gray for visualization purposes. (bottom panels) By properly fusing these $C = 3$ individual classifiers we determine a classification rule for the entire space, allowing us to predict the label value of every point. These predictions are illustrated as the colored regions shown from "the side" and "from above" in the left and right panels respectively.

Beginning, we first learn C individual linear separators in the manner described in previous sections (using any desired cost function and minimization technique). In learning the cth classifier we treat all points not in class c as a single "not-c" class by lumping them all together. To learn a two class classifier we then assign temporarily labels to the P training points: points in classes c and "not-c" are assigned temporary labels $+1$ and -1, respectively. With these temporary labels we can then learn a linear classifier distinguishing the points in class c from all other classes. This is illustrated in the middle panels of Fig. 4.17 for a $C = 3$ class dataset.

Having done this for all C classes we then have C linear separators of the form

$$b_c + \mathbf{x}^T \mathbf{w}_c = 0, \quad c = 1, \ldots, C. \tag{4.45}$$

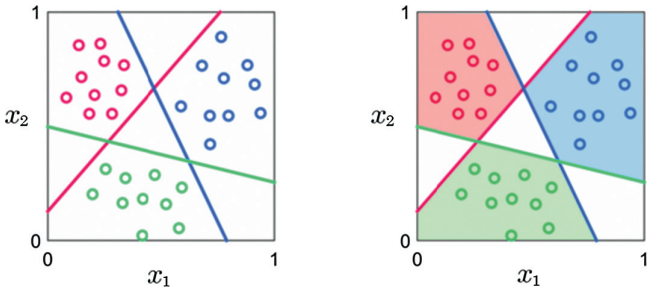


Fig. 4.18 (left panel) Linear separators from the middle panel of Fig. 4.17. (right panel) Regions of the space are colored according to the set of rules in (4.46). White regions do not satisfy these conditions, meaning that points in these areas cannot be assigned to any class/color. In the case shown here those points lying in the three white regions between any two classes are positive with respect to both classes' hyperplanes, while the white triangular region in the middle is negative with respect to all three classifiers.

In the ideal situation shown in Fig. 4.17 each classifier perfectly separates its class from the remainder of the points. In other words, all data points from class c lie on the *positive side* of its associated separator, while the points from other classes lie on its *negative side*. Stating this formally, a known point $\mathbf{x}_p$ belongs to class c if it satisfies the following set of inequalities

$$\begin{aligned} b_c + \mathbf{x}_p^T\mathbf{w}_c &> 0 \\ b_j + \mathbf{x}_p^T\mathbf{w}_j &< 0 \quad j = 1, \ldots, C,\ j \neq c. \end{aligned} \tag{4.46}$$

While this correctly describes the labels of the current set of points in an ideal scenario, using this criterion more generally to assign labels to other points in the space would be a very poor idea, as illustrated in Fig. 4.18 where we show the result of using the set of rules in (4.46) to assign labels to all points $\mathbf{x}$ in the feature space of our toy dataset from Fig. 4.17. As can be seen in the figure there are entire regions of the space for which the inequalities in (4.46) do not simultaneously hold, meaning that points in these regions cannot be assigned a class at all. These regions, left uncolored in the figure, include those areas lying on the positive side of more than one classifier (the three white regions lying between each pair of classes), and those lying on the negative side of all the classifiers (the triangular region in the middle of all three).

However, by generalizing the criteria in (4.46) we can in fact produce a useful rule that assigns labels to every point in the entire space. For a point $\mathbf{x}$ the rule is generalized by determining not the classifier that provides a positive evaluation $b_c + \mathbf{x}^T\mathbf{w}_c > 0$ (if there even is one such classifier), but by assigning $\mathbf{x}$ the label according to whichever classifier produces the largest evaluation (even if this evaluation is negative). In other words, we generalize (4.46) by assigning the label y to a point $\mathbf{x}$ by taking

$$y = \underset{j=1,\ldots,C}{\text{argmax}}\ b_j + \mathbf{x}^T\mathbf{w}_j. \tag{4.47}$$

This criterion, which we refer to as the *fusion rule*,[14] was used to assign labels[15] to the entire space of the toy dataset shown in the bottom panel of Fig. 4.17. Although devised in the context of an ideal scenario where the classes are not overlapping, the fusion rule is effective in dealing with overlapping multiclass datasets as well (see Example 4.5). As we will see in Section 4.4.2, the fusion rule is also the basis for the second multiclass method described here, multiclass softmax classification.

To perform one-versus-all classification on a dataset with C classes:

(1) Learn C individual classifiers using any approach (e.g., logistic regression, support vector machines, etc.), each distinguishing one class from the remainder of the data.

(2) Combine the learned classifiers using the fusion rule in (4.47) to make final assignments.

Example 4.5 OvA classification for overlapping data

In Fig. 4.19 we show the results of applying the OvA framework to a toy dataset with $C = 4$ overlapping classes. In this example we use the logistic regression classifier (i.e., softmax cost) and Newton's method for minimization, as described in Section 4.1.2. After learning each of the four individual classifiers (shown in the middle panels) they are fused using the rule in (4.47) to form the final partitioning of the space as shown in the bottom panels of this figure.

4.4.2 Multiclass softmax classification

As we have just seen, in the OvA framework we learn C linear classifiers separately and fuse them afterwards to create a final assignment rule for the entire space. A popular

[14] One might smartly suggest that we should first normalize the learned hyperplanes by the length of their respective normal vectors as $\frac{b_j+\mathbf{x}^T\mathbf{w}_j}{\|\mathbf{w}_j\|_2}$ prior to fusing them as in (4.47) in order to put all the classifiers "on equal footing." Or, in other words, so that no classifier is given an unwanted advantage or disadvantage in fusing due to the size of its learned weight pair $(b_j, \mathbf{w}_j)$, as this size is arbitrary (since the hyperplane $b + \mathbf{x}^T\mathbf{w} = 0$ remains unchanged when multiplied by a positive scalar γ as $\gamma \cdot (b + \mathbf{x}^T\mathbf{w}) = \gamma \cdot 0 = 0$. While this is rarely done in practice it is certainly justified, and one should feel free to normalize each hyperplane in practice prior to employing the fusion rule if desired.

[15] Note that while the boundary resulting from the fusion rule is always piecewise-linear, as in the toy examples shown here, the fusion rule itself does *not* explicitly define this boundary, i.e., it does not provide us with a nice formula for it (although one may work out a somewhat convoluted formula describing the boundary in general). This is perfectly fine since remember that our goal is not to find a formula for some separating boundary, but rather a reliable rule for accurately predicting labels (which the fusion rule provides). In fact the piecewise-linear boundaries shown in the figures of this section were drawn *implicitly* by labeling (and appropriately coloring) every point in the region shown using the fusion rule.

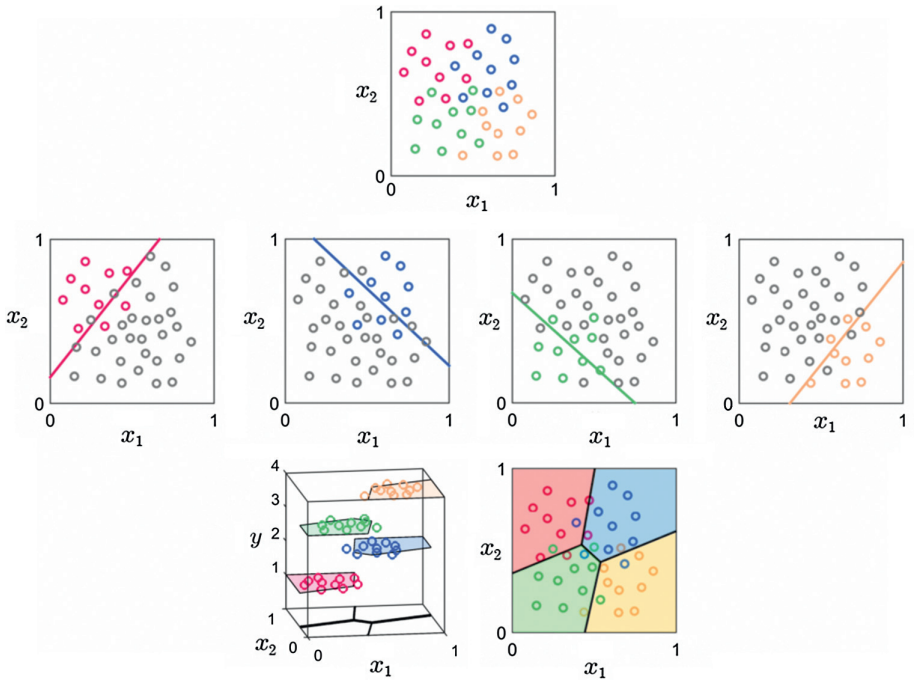


Fig. 4.19 One-versus-all multiclass scheme applied to (top panel) a toy classification dataset with $C = 4$ classes consisting of $P = 40$ data points in total (10 per class). (middle panels) The four classifiers learned to distinguish each class from the rest of the data. (bottom panels) Having determined proper linear separators for each class, we use the fusion rule in (4.47) to form the final partitioning of the space. The left and right panels illustrate the predicted labels (shown as colored regions) from both "the side" and "from above." These regions implicitly define the piecewise linear boundary shown in the right panel.

alternative, referred to as *multiclass softmax classification*, determines the C classifiers jointly by learning all of their parameters together using a cost function based on the fusion rule in (4.47). According to the fusion rule if we want a point $\mathbf{x}_p$ belonging to class c (i.e., $y_p = c$) to be classified correctly we must have that

$$c = \underset{j=1,\ldots,C}{\text{argmax}} \left(b_j + \mathbf{x}_p^T \mathbf{w}_j\right). \tag{4.48}$$

This means that we must have that

$$b_c + \mathbf{x}_p^T \mathbf{w}_c = \max_{j=1,\ldots,C} \left(b_j + \mathbf{x}_p^T \mathbf{w}_j\right), \tag{4.49}$$

or equivalently

$$\max_{j=1,\ldots,C} \left(b_j + \mathbf{x}_p^T \mathbf{w}_j\right) - \left(b_c + \mathbf{x}_p^T \mathbf{w}_c\right) = 0. \tag{4.50}$$

Indeed we would like to tune the weights so that (4.50) holds for all points in the dataset (with their respective class label). Because the quantity on the left hand side of (4.50) is always nonnegative, and is exactly zero if the point $\mathbf{x}_p$ is classified correctly, it makes

sense to form a cost function using this criterion that we then minimize in order to determine proper weights. Summing the expression in (4.50) over all P points in the dataset, denoting Ω_c the index set of points belonging to class c, we have a nonnegative cost function

$$g\left(b_1,\ldots,b_C,\mathbf{w}_1,\ldots,\mathbf{w}_C\right) = \sum_{c=1}^{C}\sum_{p\in\Omega_c}\left[\max_{j=1,\ldots,C}\left(b_j + \mathbf{x}_p^T\mathbf{w}_j\right) - \left(b_c + \mathbf{x}_p^T\mathbf{w}_c\right)\right]. \tag{4.51}$$

Note that there are only P summands in this sum, one for each point in the dataset. However, the problem here, which we also encountered when deriving the original perceptron cost function for two class classification in Section 4.1, is that the max function is continuous but not differentiable and that the trivial solution ($b_j = 0$ and $\mathbf{w}_j = \mathbf{0}_{N\times 1}$ for all j) successfully minimizes the cost. One useful work-around approach we saw there for dealing with this issue, which we will employ here as well, is to approximate $\max_{j=1,\ldots,C}\left(b_j + \mathbf{x}_p^T\mathbf{w}_j\right)$ using the smooth *softmax* function.

Recall from Section 4.1.2 that the softmax function of C scalar inputs $s_1,\ldots,s_C$, written as soft $(s_1,\ldots,s_C)$, is defined as

$$\text{soft}\left(s_1,\ldots,s_C\right) = \log\left(\sum_{j=1}^{C} e^{s_j}\right), \tag{4.52}$$

and provides a good approximation to $\max(s_1,\ldots,s_C)$ for a wide range of input values. Substituting the softmax function in (4.51) we have a smooth approximation to the original cost, given as

$$g\left(b_1,\ldots,b_C,\mathbf{w}_1,\ldots,\mathbf{w}_C\right) = \sum_{c=1}^{C}\sum_{p\in\Omega_c}\left[\log\left(\sum_{j=1}^{C} e^{b_j+\mathbf{x}_p^T\mathbf{w}_j}\right) - \left(b_c + \mathbf{x}_p^T\mathbf{w}_c\right)\right]. \tag{4.53}$$

Using the facts that $s = \log\left(e^s\right)$ and that $\log\left(\frac{s}{t}\right) = \log(s) - \log(t)$ and $\frac{e^a}{e^b} = e^{a-b}$, the above may be written equivalently as

$$g\left(b_1,\ldots,b_C,\mathbf{w}_1,\ldots,\mathbf{w}_C\right) = \sum_{c=1}^{C}\sum_{p\in\Omega_c}\log\left(1 + \sum_{\substack{j=1\\ j\neq c}}^{C} e^{\left(b_j-b_c\right)+\mathbf{x}_p^T\left(\mathbf{w}_j-\mathbf{w}_c\right)}\right). \tag{4.54}$$

This is referred to as the *multiclass softmax cost function*, or because the softmax cost for two class classification can be interpreted through the lens of surface fitting as logistic regression (as we saw in Section 4.2), for similar reasons multiclass softmax classification is often referred to as *softmax regression*.[16] When $C = 2$ one can show that this cost

[16] When thought about in this way the multiclass softmax cost is commonly written as

$$g\left(b_1,\ldots,b_C,\mathbf{w}_1,\ldots,\mathbf{w}_C\right) = -\sum_{c=1}^{C}\sum_{p\in\Omega_c}\log\left(\frac{e^{b_c+\mathbf{x}_p^T\mathbf{w}_c}}{\sum_{j=1}^{C} e^{b_j+\mathbf{x}_p^T\mathbf{w}_j}}\right), \tag{4.55}$$

which is also equivalent to (4.53).

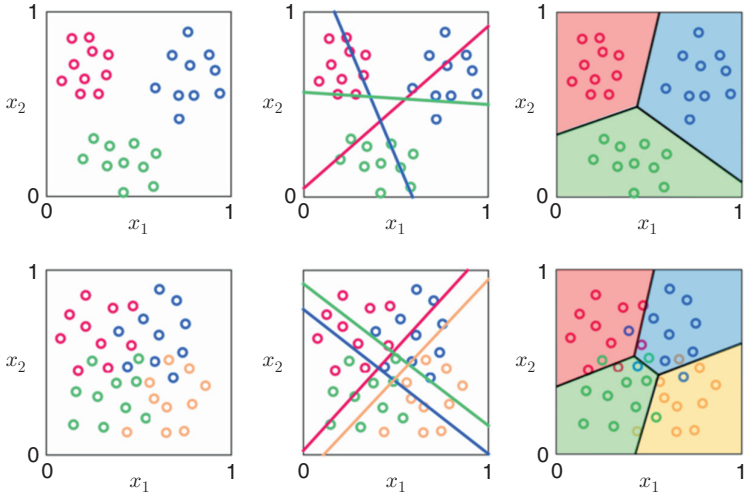


Fig. 4.20 (top left panel) Toy dataset from Fig. 4.17 with $C = 3$ classes. (top middle panel) Individual linear classifiers learned by the multiclass softmax scheme. (top right panel) Final partitioning of the feature space resulting from the application of the fusion rule in (4.47). (bottom left panel) Toy dataset from Fig. 4.19 with $C = 4$ classes. (bottom middle panel) Individual linear classifiers learned by the multiclass softmax scheme. (bottom right panel) Final partitioning of the feature space.

function reduces to the two class softmax cost originally given in (4.9). Furthermore, because the multiclass softmax cost function is convex[17] we can apply either gradient descent or Newton's method to minimize it and recover optimal weights for all C classifiers simultaneously.

In the top and bottom panels of Fig. 4.20 we show multiclass softmax classification applied to the toy datasets previously shown in the context of OvA in Fig. 4.17 and 4.19, respectively. Note that unlike the OvA separators shown in the middle panel of Fig. 4.17, the linear classifiers learned by the multiclass softmax scheme do not individually create perfect separation between one class and the remainder of the data. However, when combined according to the fusion rule in (4.47), they still perfectly partition the three classes of data. Also note that similar to OvA, the multiclass softmax scheme still produces a very good classification of the data even with overlapping classes. In both instances shown in Fig. 4.20 we used gradient descent for minimization of the multiclass softmax cost function, as detailed in Example 4.6.

Example 4.6 Optimization of the multiclass softmax cost

To calculate the gradient of the multiclass softmax cost in (4.54), we first rewrite it more compactly as

[17] This is perhaps most easily verified by noting that it is the composition of linear terms $b_j + \mathbf{x}_p^T\mathbf{x}_j$ with the convex nondecreasing softmax function. Such a composition is always guaranteed to be convex [24].

$$g\left(\tilde{\mathbf{w}}_1,\ldots,\tilde{\mathbf{w}}_C\right) = \sum_{c=1}^{C}\sum_{p\in\Omega_c}\log\left(1+\sum_{\substack{j=1\\ j\neq c}}^{C} e^{\tilde{\mathbf{x}}_p^T\left(\tilde{\mathbf{w}}_j-\tilde{\mathbf{w}}_c\right)}\right), \tag{4.56}$$

where we have used the compact notation $\tilde{\mathbf{w}}_j = \begin{bmatrix} b_j \\ \mathbf{w}_j \end{bmatrix}$ and $\tilde{\mathbf{x}}_p = \begin{bmatrix} 1 \\ \mathbf{x}_p \end{bmatrix}$ for all $c = 1,\ldots,C$ and $p = 1,\ldots,P$. In this form, the gradient of g with respect to $\tilde{\mathbf{w}}_c$ may be computed[18] as

$$\nabla_{\tilde{\mathbf{w}}_c} g = \sum_{p=1}^{P}\left(\frac{1}{1+\sum_{\substack{j=1\\ j\neq c}}^{C} e^{\tilde{\mathbf{x}}_p^T\left(\tilde{\mathbf{w}}_j-\tilde{\mathbf{w}}_c\right)}} - \mathbb{1}_{p\in\Omega_c}\right)\tilde{\mathbf{x}}_p, \tag{4.57}$$

for $c = 1,\ldots,C$, where $\mathbb{1}_{p\in\Omega_c} = \begin{cases} 1 & \text{if } p\in\Omega_c \\ 0 & \text{else} \end{cases}$ is an indicator function on the set Ω_c. Concatenating all individual classifiers' parameters into a single weight vector $\tilde{\mathbf{w}}_{\text{all}}$ as

$$\tilde{\mathbf{w}}_{\text{all}} = \begin{bmatrix} \tilde{\mathbf{w}}_1 \\ \tilde{\mathbf{w}}_2 \\ \vdots \\ \tilde{\mathbf{w}}_C \end{bmatrix}, \tag{4.58}$$

the gradient of g with respect to $\tilde{\mathbf{w}}_{\text{all}}$ is formed by stacking block-wise gradients found in (4.57) into

$$\nabla g = \begin{bmatrix} \nabla_{\tilde{\mathbf{w}}_1} g \\ \nabla_{\tilde{\mathbf{w}}_2} g \\ \vdots \\ \nabla_{\tilde{\mathbf{w}}_C} g \end{bmatrix}. \tag{4.59}$$

4.4.3 The accuracy of a learned multiclass classifier

To calculate the accuracy of both the OvA and multiclass softmax classifiers we use the labeling mechanism in (4.47). That is, denoting $\left(b_j^\star, \mathbf{w}_j^\star\right)$ the learned parameters for the jth boundary, we assign the predicted label $\hat{y}_p$ to the pth point $\mathbf{x}_p$ as

$$\hat{y}_p = \underset{j=1\ldots C}{\text{argmax}}\ b_j^\star + \mathbf{x}_p^T\mathbf{w}_j^\star. \tag{4.60}$$

[18] Writing the gradient in this way helps avoid potential numerical problems posed by the "overflowing" exponential problem described in footnote 6.

We then compare each predicted label to its true label using an indicator function

$$\mathcal{I}\left(y_p, \hat{y}_p\right) = \begin{cases} 1 & \text{if } y_p \neq \hat{y}_p \\ 0 & \text{if } y_p = \hat{y}_p, \end{cases} \tag{4.61}$$

which we use towards computing the accuracy of the multiclass classifier on our training set as

$$\text{accuracy} = 1 - \frac{1}{P}\sum_{p=1}^{P}\mathcal{I}\left(y_p, \hat{y}_p\right). \tag{4.62}$$

This quantity ranges between 1 when every point is classified correctly, and 0 when no point is correctly classified. When possible it is also recommended to compute the accuracy of the learned model on a new testing dataset (i.e., data not used to train the model) in order to provide some assurance that the learned model will perform well on future data points. This is explored further in Chapter 6 in the context of *cross-validation*.

4.4.4 Which multiclass classification scheme works best?

As we have now seen, both OvA and multiclass softmax approaches are built using the fusion rule given in Equation (4.47). While the multiclass softmax approach more directly aims at optimizing this criterion, both OvA and softmax multiclass perform similarly well in practice (see e.g., [70, 77] and references therein).

> One-versus-all (OvA) and multiclass softmax classifiers perform similarly well in practice, having both been built using the fusion rule in (4.47).

The two methods largely differ in how they are applied in practice as well as their computational burden. In learning each of the C linear separators individually the computation required for the OvA classifier is naturally parallelizable, as each linear separator can be learned independently of the rest. On the other hand, while both OvA and multiclass softmax may be naturally extended for use with nonlinear multiclass classification (as we will discuss in Chapter 6), the multiclass softmax scheme provides a more commonly used framework for performing nonlinear multiclass classification using neural networks.

4.5 Knowledge-driven feature design for classification

Often with classification we observe not linear separability between classes but some sort of nonlinear separability. As with regression (detailed in Section 3.2), here we formulate *feature transformations* of the input data to capture this nonlinearity and use