6 Automatic feature design for classification

In Chapter 6 we mirror closely the exposition given in the previous chapter on regression, beginning with the approximation of the underlying data generating function itself by bases of features, and going on to finally describing cross-validation in the context of classification. In short we will see that all of the tools from the previous chapter can be applied to the automatic design of features for the problem of classification as well.

6.1 Automatic feature design for the ideal classification scenario

In Fig. 6.1 we illustrate a prototypical dataset on which we perform the general task of two class classification, where the two classes can be effectively separated using a nonlinear boundary. In contrast to those examples given in Section 4.5, where visualization or scientific knowledge guided the fashioning of a feature transformation to capture this nonlinearity, in this chapter we suppose that this cannot be done due to the complexity and/or high dimensionality of the data. At the heart of the two class classification framework is the tacit assumption that the data we receive are in fact noisy samples of some underlying indicator function, a nonlinear generalization of the step function briefly discussed in Section 4.5, like the one shown in the right panel of Fig. 6.1. Akin to regression, our goal with classification is then to approximate this data-generating indicator function as well as we can using the data at our disposal.

In this section we will assume the impossible: that we have clean and complete access to every data point in the space of a two class classification environment, whose labels take on values in $\{-1, 1\}$, and hence access to its associated indicator function $y(\mathbf{x})$. Although an indicator function is *not* continuous, the same bases of continuous features discussed in the previous chapter can be used to represent it (near) perfectly.

6.1.1 Approximation of piecewise continuous functions

In Section 5.1 we saw how fixed and adjustable neural network bases of features can be used to approximate *continuous* functions. These bases can also be used to effectively approximate the broader class of *piecewise continuous* functions, composed of fragments of continuous functions with gaps or jumps between the various pieces. Shown

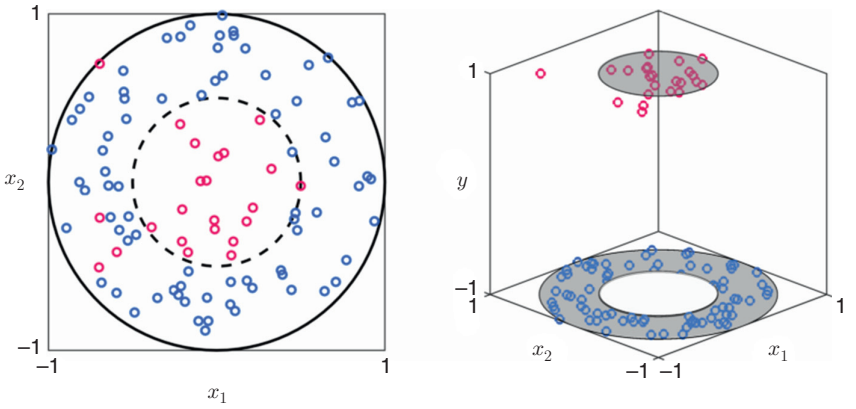

Fig. 6.1 (left panel) A realistic dataset for two class classification shown in the original space from above and made by taking noisy samples from the data generating indicator function $y(\mathbf{x}) = \begin{cases} +1 & \text{if} \quad 0 \leq \|\mathbf{x}\|_2^2 \leq 0.5 \\ -1 & \text{if} \quad 0.5 < \|\mathbf{x}\|_2^2 \leq 1, \end{cases}$ over the unit circle. (right panel) View of the original space from the side with the data-generating indicator function shown in gray.

in Fig. 6.2 are two example piecewise continuous functions[1] (in black) along with their polynomial, Fourier, and single hidden layer neural network basis approximations. For each instance in the figure we have used as many of the respective basis elements as needed to give a visually close approximation.

As with continuous functions, adding more basis elements generally produces a finer approximation of any piecewise function defined over a bounded space (for convenience we will take the domain of y to be the unit hypercube, again denoted as $[0, 1]^N$, but any bounded domain would also suffice, e.g., a hypersphere with finite radius). That is, by increasing the number of basis elements M we generally have that the approximation

$$\sum_{m=0}^{M} f_m(\mathbf{x})\, w_m \approx y(\mathbf{x}) \tag{6.2}$$

improves overall.[2] Note here that once again $f_0(\mathbf{x}) = 1$ is the constant basis element, with the remaining $f_m(\mathbf{x})$ basis elements of any type desired.

[1] The piecewise continuous functions in the top and bottom panels are defined over the unit interval, respectively as

$$y(x) = \begin{cases} +1 & 0.33 \leq x \leq 0.67 \\ -1 & \text{else,} \end{cases} \qquad y(x) = \begin{cases} 0.25\left(\left(1 - 10x^2\right)\cos(10\pi x) + 1\right) & 0 \leq x < 0.33 \\ 0.8 & 0.33 \leq x \leq 0.67 \\ 4\left(x^2 - 0.75\right) + 0.4 & 0.67 < x \leq 1. \end{cases} \tag{6.1}$$

[2] As with continuous function approximation, the details of this statement are quite technical, and we do not dwell on them here. Our goal is to provide an intuitive high level understanding of this sort of function approximation. See Section 5.7 for further information and reading.

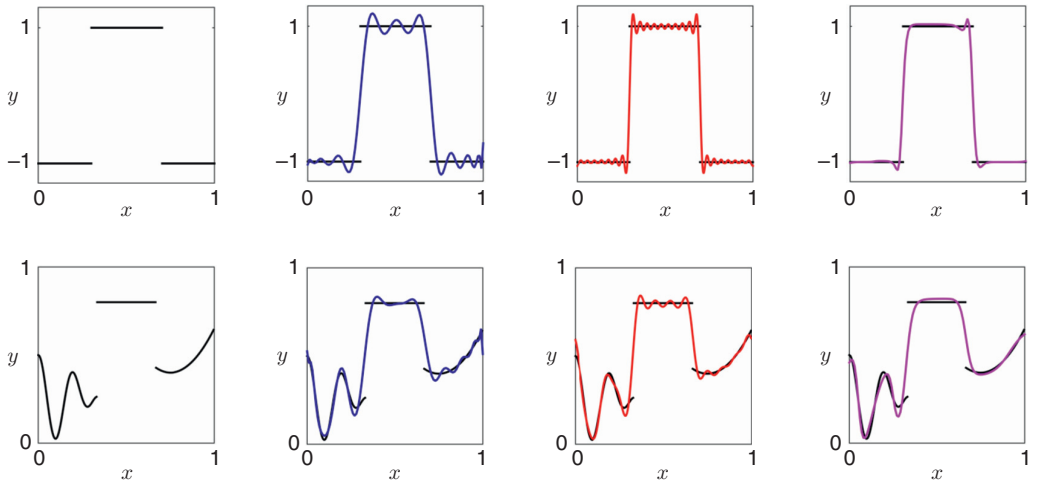

Fig. 6.2 Two piecewise continuous functions (in black). The first function is constant in each piece (top panels) while the second is a more complicated function consisting of several different types of continuous pieces (bottom panels). Also shown are polynomial (in blue), Fourier (in red), and single hidden layer neural network (in purple) basis approximations of each function. Using more basis features in each case will increase the quality of the approximation, just as with continuous function approximation.

6.1.2 The formal definition of an indicator function

Formally speaking an indicator function, like the one shown in Fig. 6.1, is a simple tool for identifying points $\mathbf{x}$ in a set $\mathcal{S}$ by assigning them some unique constant value while assigning all other points that are not in $\mathcal{S}$ a different constant value.[3] Using $+1$ to indicate that a point is in a set $\mathcal{S}$ and -1 otherwise, we can define an indicator function on $\mathcal{S}$ as

$$y(\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{x} \in \mathcal{S} \\ -1 & \text{if } \mathbf{x} \notin \mathcal{S}. \end{cases} \tag{6.3}$$

For instance, the one-dimensional function plotted in the top panels of Fig. 6.2 is such an indicator function on the interval $\mathcal{S} = [0.33, 0.67]$. The two-dimensional function shown at the beginning of the section in Fig. 6.1 is an indicator function on the set $\mathcal{S} = \left\{\mathbf{x} \mid 0 \leq \|\mathbf{x}\|_2^2 \leq 0.5\right\}$. Moreover the general step function discussed in Section 3.3.1 is another example of such an indicator function on the set $\mathcal{S} = \left\{\mathbf{x} \mid b + \mathbf{x}^T\mathbf{w} > 0\right\}$, taking the form

$$y(\mathbf{x}) = \text{sign}\left(b + \mathbf{x}^T\mathbf{w}\right) = \begin{cases} +1 & \text{if } b + \mathbf{x}^T\mathbf{w} > 0 \\ -1 & \text{if } b + \mathbf{x}^T\mathbf{w} < 0. \end{cases} \tag{6.4}$$

[3] Although $\mathcal{S}$ can take any arbitrary shape in general, in the context of classification we are only interested in sets that, loosely speaking, have an *interior* to them. This excludes degenerate cases such as the union of isolated points.

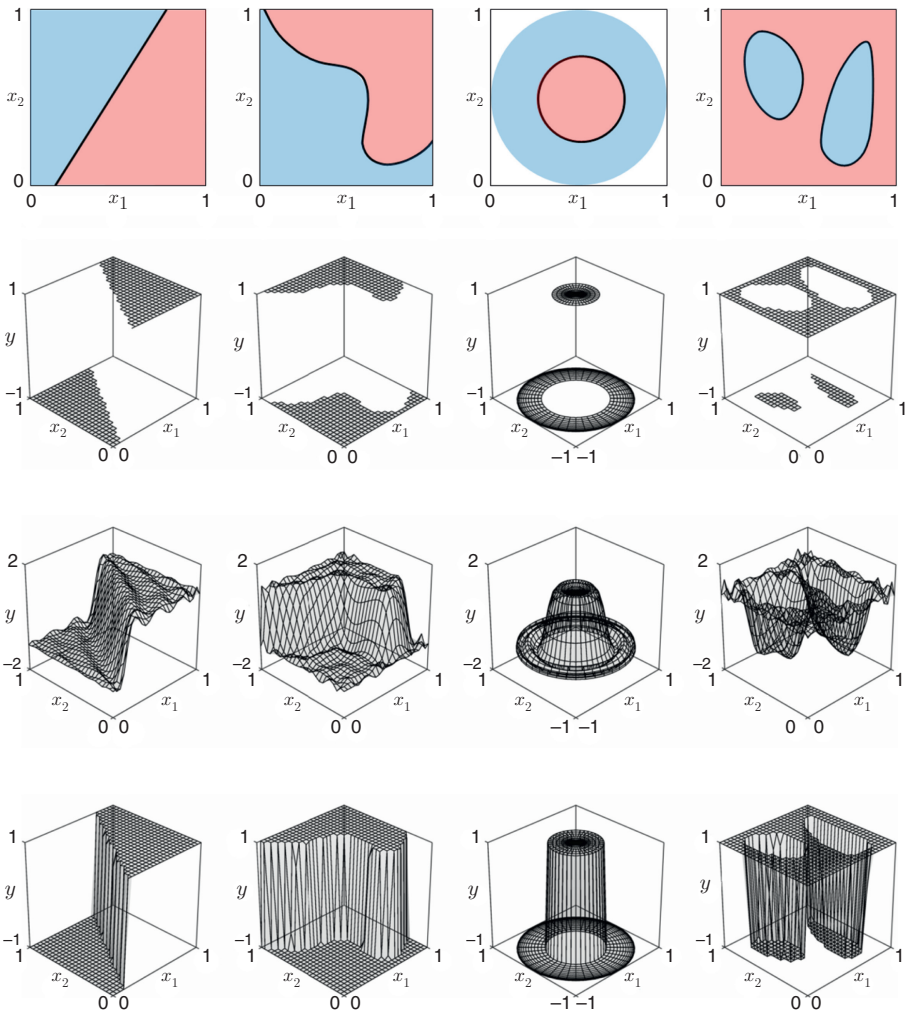

Fig. 6.3 (top row) Four instances of the type of sets $\mathcal{S}$ that concern us in machine learning. Shown from left to right a half space, a general region with nonlinear boundary, a circular region, and the unit square with two ovoid shapes removed from it. For visualization purposes the points that are in the set $\mathcal{S}$ are colored red in each instance, while the rest are colored blue. (second row) The corresponding indicator functions shown in the data space. (third row) Respective polynomial approximations to each indicator function where $M = 30$ in each instance. (bottom row) The associated logistic approximations match the original indicator functions very closely.

We show four instances of such sets in the top row of Fig. 6.3 where the points that are in the set $\mathcal{S}$ are colored red while the rest are colored blue in each instance. From a classification perspective, the red and blue regions indicate the domain of class $+1$ and class -1 in the input space, respectively. Plotted in the second row of this figure are the corresponding indicator functions.

Since indicator functions are piecewise constant, a special subclass of piecewise continuous functions, they can be approximated effectively using any of the familiar bases

previously discussed and large enough M. In particular we show in the third row of Fig. 6.3 polynomial approximations to each respective indicator function, where the number of basis features in each case is set to $M = 30$.

6.1.3 Indicator function approximation

Given the fact that an indicator function $y(\mathbf{x})$ takes on values ± 1 we can refine our approximation by passing the basis sum in Equation (6.2) through the sign $(\cdot)$ function giving

$$\text{sign}\left(\sum_{m=0}^{M} f_m(\mathbf{x})\, w_m\right) \approx y(\mathbf{x}). \tag{6.5}$$

Fundamental to the notion of logistic regression (and to two class classification more broadly), as discussed in Section 4.2.2, is that the smooth logistic function $\tanh(\alpha t)$ can be made to approximate $\text{sign}(t)$ as finely as desired by increasing α. By absorbing a large constant α into each weight w_m in Equation (6.5) we can write the *logistic approximation* to the indicator function $y(\mathbf{x})$ as

$$\tanh\left(\sum_{m=0}^{M} f_m(\mathbf{x})\, w_m\right) \approx y(\mathbf{x}). \tag{6.6}$$

In the bottom row of Fig. 6.3 we show the result of learning weights so that Equation (6.6) holds, again using a polynomial basis with $M = 30$ for each of the four indicator functions. As can be seen, the logistic approximation provides a better resemblance to the actual indicator compared to the direct polynomial approximation seen in the third row.

6.1.4 Recovering weights

Since $y(\mathbf{x})$ takes on values in $\{\pm 1\}$ at each $\mathbf{x}$ and the approximation in Equation (6.6) is linear in the weights w_m, using precisely the argument given in Section 4.2.2 (in deriving the softmax cost function in the context of logistic regression) we may rewrite Equation (6.6) equivalently as

$$\log\left(1 + e^{-y(\mathbf{x})\sum_{m=0}^{M} f_m(\mathbf{x}) w_m}\right) \approx 0. \tag{6.7}$$

Therefore in order to properly tune the weights $\{w_m\}_{m=0}^{M}$, as well as any internal parameters Θ when employing a neural network basis, we can formally minimize the logistic approximation in Equation (6.7) over all $\mathbf{x}$ in the unit hypercube as

$$\underset{w_0 \ldots w_M, \Theta}{\text{minimize}} \int_{\mathbf{x} \in [0,1]^N} \log\left(1 + e^{-y(\mathbf{x})\sum_{m=0}^{M} f_m(\mathbf{x}) w_m}\right) d\mathbf{x}. \tag{6.8}$$

As with the Least Squares problem with continuous function approximation previously discussed in Section 5.1.5, this is typically not solvable in closed form due to the

intractability of the integrals involved. Instead, once again by discretizing the functions involved we will see how this problem reduces to a general problem for two class classification, and how this framework is directly applicable to real classification datasets.

6.2 Automatic feature design for the real classification scenario

In this section we discuss how fixed and neural network feature bases are applied to the automatic design of features for real two-class classification datasets. Analogous to their incorporation into the framework of regression discussed in Section 5.2, here we will see that the concept of feature bases transfers quite easily from the ideal scenario discussed in the previous section.

6.2.1 Approximation of discretized indicator functions

As with the discussion in Section 5.1.2 for continuous functions, *discretizing* an indicator function $y(\mathbf{x})$ finely over its domain gives a close facsimile of the true indicator. For example, shown in Fig. 6.4 is the circular indicator previously shown in Fig. 6.1 and 6.3, along with a closely matching discretized version made by sampling the function over a fine grid of evenly spaced points in its input domain.

Formally, by taking a fine grid of P evenly spaced points $\left\{\left(\mathbf{x}_p, y\left(\mathbf{x}_p\right)\right)\right\}_{p=1}^{P}$ over the input domain of an indicator function $y(\mathbf{x})$ we can then say for a given number M of basis features that the condition in (6.7) essentially holds for each p as

$$\log\left(1 + e^{-y(\mathbf{x}_p)\sum_{m=0}^{M} f_m(\mathbf{x}_p)w_m}\right) \approx 0. \tag{6.9}$$

By denoting $y_p = y\left(\mathbf{x}_p\right)$, using the feature vector notation $\mathbf{f}_p = \left[\begin{array}{ccc} f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots \quad f_M\left(\mathbf{x}_p\right) \end{array}\right]^T$, and reintroducing the bias notation $b = w_0$ we may write Equation (6.9) more conveniently as

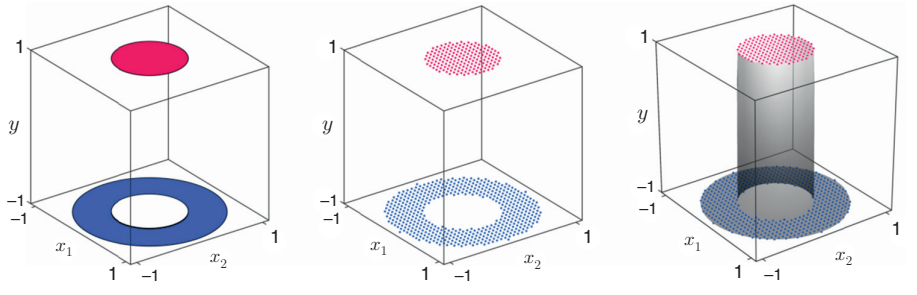

Fig. 6.4 (left panel) An indicator function $y(\mathbf{x})$ defined over the unit circle, and (middle panel) a discretized facsimile made by evaluating y over a fine grid of evenly spaced points. (right panel) We can fit a smooth approximation to the discretized indicator via minimizing the softmax cost function.

$$\log\left(1 + e^{-y_p\left(b+\mathbf{f}_p^T\mathbf{w}\right)}\right) \approx 0. \tag{6.10}$$

Similarly to the discussion of logistic regression in Section 4.2.2, by minimizing the sum of these terms,

$$g\left(b, \mathbf{w}, \Theta\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b+\mathbf{f}_p^T\mathbf{w}\right)}\right), \tag{6.11}$$

we can learn parameters $(b, \mathbf{w}, \Theta)$ to make Equation (6.10) hold as well as possible for all p. Note how this is precisely the logistic regression problem discussed in Section 4.2.2, only here each original data point $\mathbf{x}_p$ has been replaced with its feature transformed version $\mathbf{f}_p$. Stating this minimization formally,

$$\underset{b,\,\mathbf{w},\,\Theta}{\text{minimize}} \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b+\mathbf{f}_p^T\mathbf{w}\right)}\right), \tag{6.12}$$

we can see that it is in fact a discrete form of the original learning problem in Equation (6.8) for the ideal classification scenario. In addition, recall from Section 4.1 that there are many highly related costs to the softmax function that recover similar weights when properly minimized. Therefore we can use e.g., the squared margin cost in place of the softmax, and instead solve

$$\underset{b,\,\mathbf{w},\,\Theta}{\text{minimize}} \sum_{p=1}^{P} \max^2\left(0, 1 - y_p\left(b + \mathbf{f}_p^T\mathbf{w}\right)\right) \tag{6.13}$$

to determine optimal parameters. Regardless of the cost function and feature basis used, the corresponding problem can be minimized via numerical techniques like gradient descent (for details see the next section). Also note that regardless of the cost function or feature basis used in producing a *nonlinear* boundary in the original feature space, we are simultaneously determining a *linear* boundary in the transformed feature space in the bias b and weight vector $\mathbf{w}$, as we showed visually with the elliptical dataset in Example 4.7.

6.2.2 The real classification scenario

Very rarely in practice can we acquire large quantities of noiseless data which span the entire input space evenly like a finely discretized indicator function. On the contrary, often we have access to a limited amount of data that is not so evenly distributed and, due to errors in its acquisition, is noisy. In the case of classification, noisy means that some data points have been assigned the wrong labels. For example, in the right column of Fig. 6.5 we show three simulated realistic classification datasets each composed of P noisy samples[4] $\left\{\left(\mathbf{x}_p, y\left(\mathbf{x}_p\right)\right)\right\}_{p=1}^{P}$ of the three indicator functions shown discretized in the figure's left column.

[4] In each instance we take $P = 99$ points randomly from the respective input domain, and evaluate each input $\mathbf{x}_p$ in its indicator function giving the associated label $y\left(\mathbf{x}_p\right)$. We then add noise to the first two datasets by randomly switching the output (or label) $y\left(\mathbf{x}_p\right)$ of 4 and 6 points respectively, that is, for these points we replace $y\left(\mathbf{x}_p\right)$ with $-y\left(\mathbf{x}_p\right)$, and then refer to the (potentially noisy) label of each $\mathbf{x}_p$ as simply y_p.

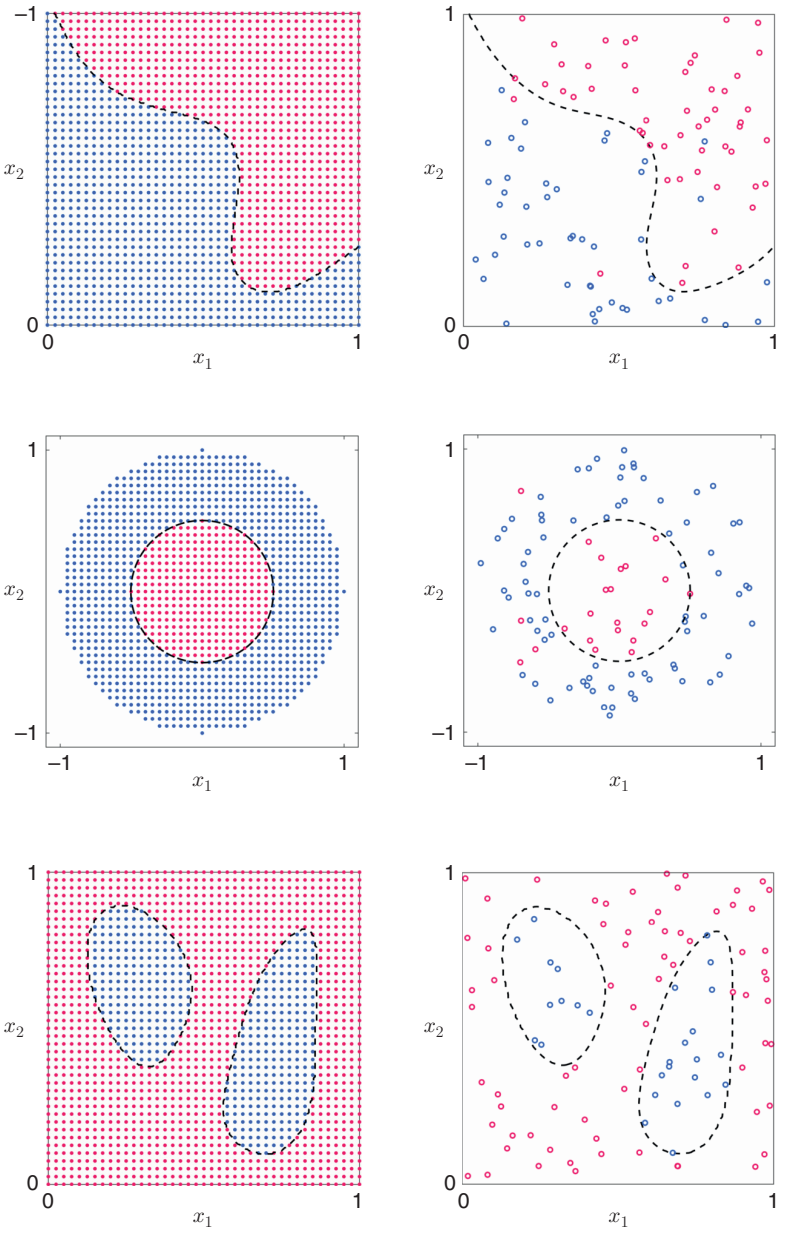

Fig. 6.5 (left column) Three ideal datasets for general two class classification shown in the original feature space (bird's-eye view). (right column) Examples of realistic datasets for two class classification, as noisy samples of each respective indicator.

With this in mind, generally speaking we can think about classification datasets encountered in practice as noisy samples of some unknown indicator function. In other words, analogous to regression (as discussed in Section 5.2.2), general two class classification is an (indicator) function approximation problem based on noisy samples.

> The general case of two class classification is an (indicator) function approximation problem based on noisy samples of the underlying function.

As with the data consisting of a discretized indicator function, for real classification datasets we may also approximate the underlying indicator function/determine class separating boundaries by leveraging fixed and neural network feature bases. For example, in Fig. 6.6 we show the result of employing degree[5] 3, 2, and 5 polynomial basis features to approximate the underlying indicator functions of the noisy datasets originally shown in the top, middle, and bottom right panels of Fig. 6.5. In each case we learn proper parameters by minimizing the softmax cost function as in Equation (6.12), the details of which we describe (for both fixed and neural network feature bases) following this discussion. As can be seen in the left and right columns of Fig. 6.6, this procedure determines nonlinear boundaries and approximating indicator functions[6] that closely mimic those of the true indicators shown in Fig. 6.3.

Example 6.1 Classification with fixed bases of features

The minimization of the softmax or squared margin perceptron cost functions using a fixed feature transformation follows closely the details first outlined in Examples 4.1 and 4.2 respectively. Foremost when employing M fixed basis features, the associated cost, being a function only of the bias $b = w_0$ and weight vector $\mathbf{w} = \begin{bmatrix} w_1 & w_2 & \dots & w_M \end{bmatrix}^T$, is convex regardless of the cost function used. Hence both gradient descent and Newton's method can be readily applied. The form of the gradients and Hessians are entirely the same, with the only cosmetic difference being the use of the M-dimensional feature vector $\mathbf{f}_p$ in place of the N-dimensional input $\mathbf{x}_p$ (see Exercises 6.1 and 6.2). For example, using the compact notation $\tilde{\mathbf{f}}_p = \begin{bmatrix} 1 \\ \mathbf{f}_p \end{bmatrix}$ and $\tilde{\mathbf{w}} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}$, the softmax cost in Equation (6.11) can be written as $g\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p \tilde{\mathbf{f}}_p^T \tilde{\mathbf{w}}}\right)$, whose gradient is given by

$$\nabla_{\tilde{\mathbf{w}}} g\left(\tilde{\mathbf{w}}\right) = -\sum_{p=1}^{P} \sigma\left(-y_p \tilde{\mathbf{f}}_p^T \tilde{\mathbf{w}}\right) y_p \tilde{\mathbf{f}}_p. \tag{6.14}$$

[5] Note that a set of degree D polynomial features for input of dimension $N > 1$ consists of all monomials of the form $f_m(\mathbf{x}) = x_1^{m_1} x_2^{m_2} \cdots x_N^{m_N}$ (see footnote 5 of Chapter 5 where this was first introduced) where $0 \leq m_1 + m_2 + \cdots + m_N \leq D$. There are a total of $M = \frac{(N+D)!}{N!D!} - 1$ such terms excluding the constant feature and hence this is the length of the corresponding feature vector.

[6] The general equations defining both the learned boundaries and corresponding indicator functions shown in the figures of this section are explicitly defined in Section 6.2.3.

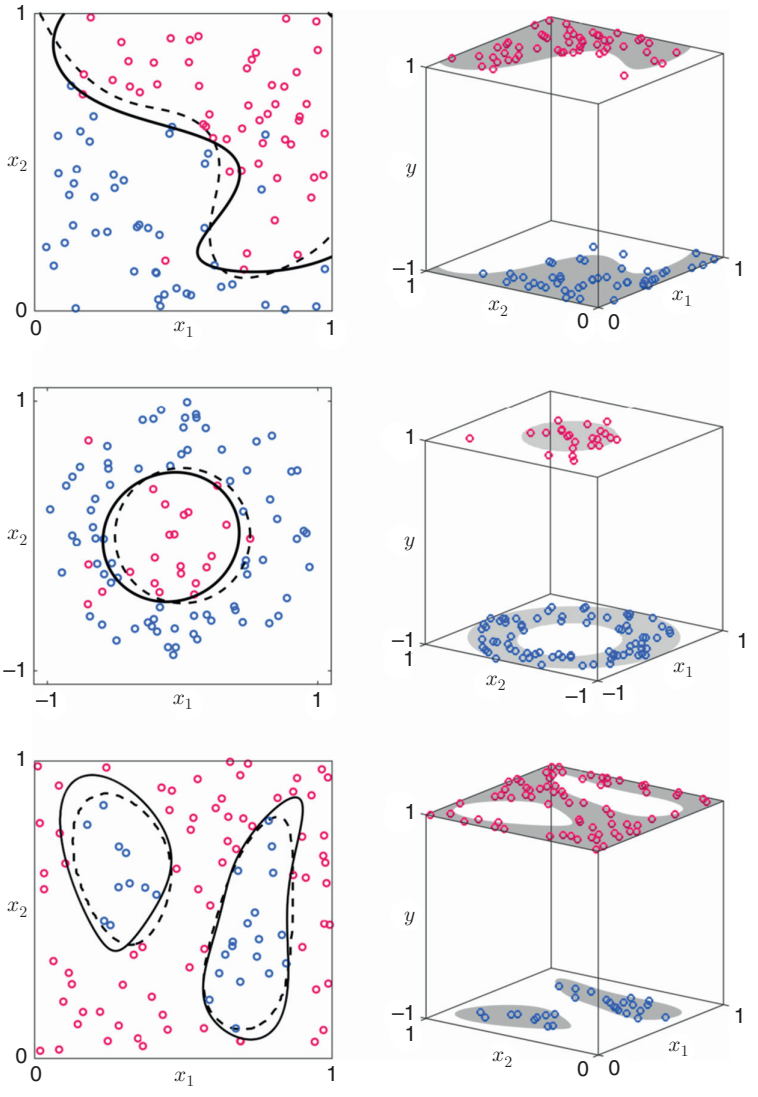

Fig. 6.6 Three datasets first shown in Fig. 6.5. Each dataset is shown in the original feature space from above (left panels) where both the original boundaries (dashed black) and learned boundaries (black) using polynomial basis features are shown, and from the side (right panels) where the indicator functions corresponding to each learned boundary are shown (in gray). The general equations defining both the learned boundaries and corresponding indicator functions are defined in Section 6.2.3.

In Fig. 6.7 we show two additional datasets along with nonlinear separators formed using fixed feature bases, in particular degree 2 polynomial and Fourier features[7] respectively, which in each case provides perfect classification. As was the case with regression

[7] Note that a degree D Fourier set of basis features for input $\mathbf{x}_p$ of dimension $N > 1$ consists of all monomial features of the form $f(\mathbf{x}) = e^{2\pi i m_1 x_1} e^{2\pi i m_2 x_2} \cdots e^{2\pi i m_N x_N}$ (see footnote 5 of Chapter 5) where $-D \leq m_1, m_2, \cdots, m_N \leq D$. There are a total of $M = (2D+1)^N - 1$ such terms excluding the constant feature, and hence this is the length of the corresponding feature vector.

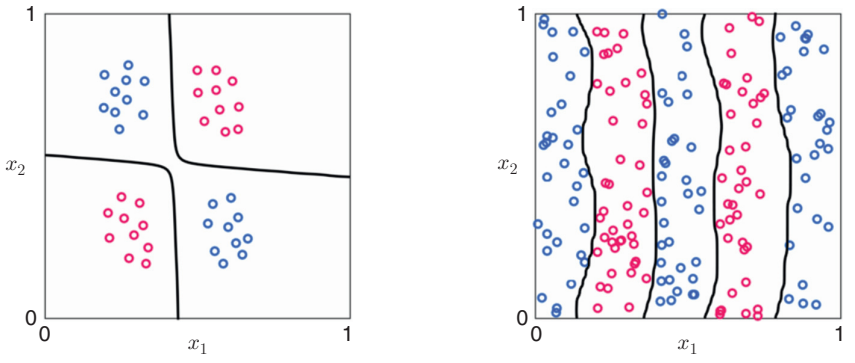

Fig. 6.7 (left panel) A version of the so-called XOR dataset with learned nonlinear boundary using the squared margin cost and degree 2 polynomial features. (right panel) Degree 2 Fourier features used for classification of a dataset consisting of consecutive bands of differing class points. In each case the data is perfectly separated.

in Example 5.1, the number of fixed bases for classification again grows combinatorially with the dimension of the input. As we will see in Section 7.1, this problem can once again be dealt with via the notion of a "kernel," however, this again introduces a serious numerical optimization problem as the size of the data-set grows.

Example 6.2 Classification with a basis of single hidden layer neural network features

The feature vector of the input $\mathbf{x}_p$ made by using a basis of single hidden layer neural network features takes the form

$$\mathbf{f}_p = \begin{bmatrix} a\left(c_1 + \mathbf{x}_p^T\mathbf{v}_1\right) & a\left(c_2 + \mathbf{x}_p^T\mathbf{v}_2\right) & \cdots & a\left(c_M + \mathbf{x}_p^T\mathbf{v}_M\right) \end{bmatrix}^T, \tag{6.15}$$

where $a(\cdot)$ is any activation function as detailed in Section 5.1.4. However, unlike the case with fixed basis features, when using neural networks a cost like the softmax is non-convex, and thus effectively solving e.g.,

$$\underset{b,\mathbf{w},\Theta}{\text{minimize}} \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\mathbf{w}\right)}\right) \tag{6.16}$$

requires running gradient descent several times (using a different initialization in each instance) in order to find a good local minimum (see Exercise 6.4).

This issue is illustrated in Fig. 6.8, where we have used a single hidden layer basis with the tanh activation to classify the datasets originally shown in Fig. 6.7. In Fig. 6.8 we show the result of running gradient descent, with (top) $M = 2$ and (bottom) $M = 4$ basis features respectively, three times with three random initializations. Note that in each case one of these initializations leads gradient descent to a bad stationary point

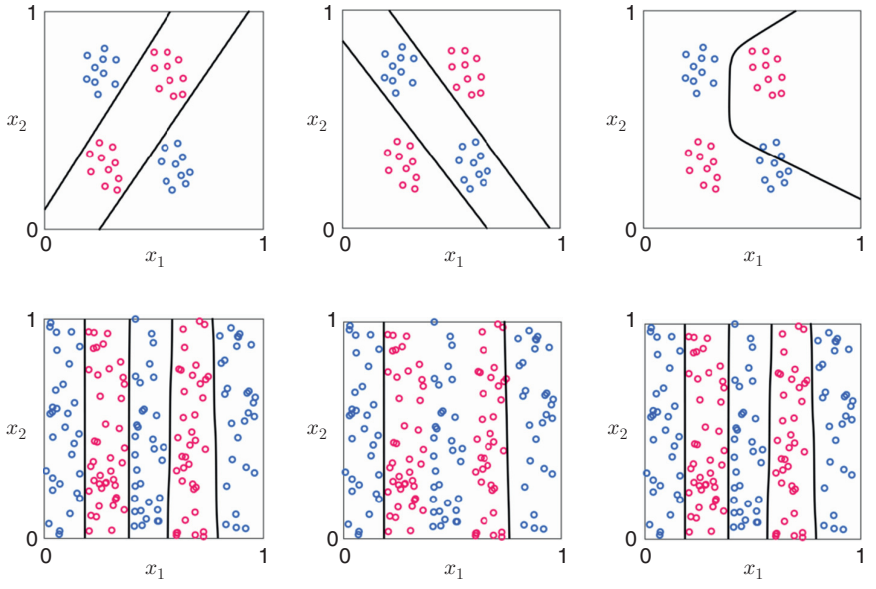

Fig. 6.8 The single hidden layer feature basis with tanh activation applied to classifying "XOR" and "stripe" datasets first shown in Fig. 6.7. For each dataset gradient descent is run three times with a different random initialization in each instance. (top panels) The first two resulting learned boundaries for the XOR dataset classify the data perfectly, while the third fails. (bottom panels) The first and last run of gradient descent provide a perfect separating boundary for the dataset consisting of consecutive class stripes, while the second does not.

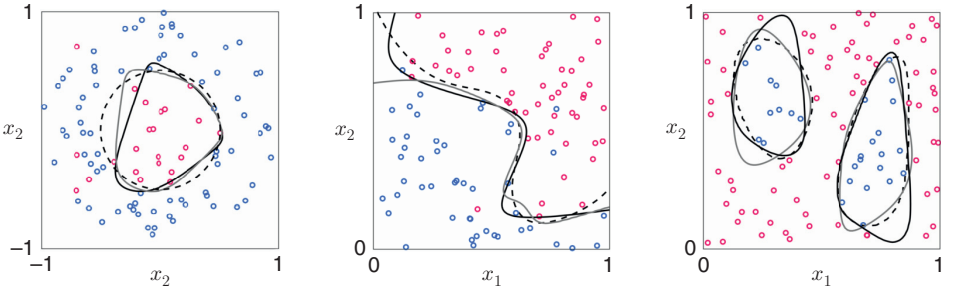

Fig. 6.9 The datasets first shown in Fig. 6.6 classified using a single hidden layer basis with (left panel) $M = 3$, (middle panel) $M = 4$, and (right panel) $M = 6$ basis features. For each case the boundaries of two successful runs of gradient descent, with random initializations, are shown in black and gray. Also plotted in dashed black are the true boundaries.

resulting in a poor fit to the respective dataset. In Fig. 6.9 we show the result of applying the same basis type with $M = 3$ (left panel), $M = 4$ (middle panel), and $M = 6$ (right panel) basis features respectively to the three datasets shown in Fig. 6.6.

Example 6.3 Classification with multilayer neural network features

To construct multilayer neural network bases we can sum and compose simple functions as described in Section 5.1.4. Doing so creates the same practical tradeoff as with the use of deep net basis features for regression, as discussed in Example 5.3. This tradeoff being that adding more hidden layers, while making each adjustable basis feature more flexible, comes at the cost of making the corresponding minimization problem more non-convex and thus more challenging to solve. However, the same ideas, i.e., regularization and the use of particularly helpful activation functions like the rectified linear unit $a(t) = \max(0, t)$, can also be used to mitigate this non-convexity issue when employing deep net basis features for classification.

Unfortunately, implementing gradient descent (often referred to in the machine learning community as *the backpropagation algorithm*) remains a tedious task due to the careful book-keeping required to correctly compute the gradient of a cost function incorporating deep net features. Because of this we provide the interested reader with an organized presentation of gradient computation for cost functions employing deep net basis features in Section 7.2. To avoid potential errors in computing the derivatives of a deep net cost function by hand, computational techniques like automatic differentiation [59] are often utilized when using deep nets in practice.

6.2.3 Classifier accuracy and boundary definition

Regardless of the cost function used, once we have learned proper parameters $(b^\star, \mathbf{w}^\star, \Theta^\star)$ using a given set of basis features the accuracy of a learned classifier with these parameters is defined almost precisely, as in Section 4.1.5, by replacing each input $\mathbf{x}_p$ with its corresponding feature representation $\mathbf{f}_p$. We first compute the counting cost

$$g_0\left(b^\star, \mathbf{w}^\star, \Theta^\star\right) = \sum_{p=1}^{P} \max\left(0,\ \text{sign}\left(-y_p\left(b^\star + \mathbf{f}_p^T\mathbf{w}^\star\right)\right)\right), \tag{6.17}$$

which gives the number of misclassifications of the learned model, and this defines the final accuracy of the classifier as

$$\text{accuracy} = 1 - \frac{g_0}{P}. \tag{6.18}$$

The learned nonlinear boundary (like those shown in each figure of this section) is then defined by the set of $\mathbf{x}$ where

$$b^\star + \sum_{m=1}^{M} f_m(\mathbf{x})\, w_m^\star = 0. \tag{6.19}$$

Likewise the final approximation of the true underlying indicator function (like those shown in the right column of Fig. 6.6) is given as

$$y(\mathbf{x}) = \text{sign}\left(b^\star + \sum_{m=1}^{M} f_m(\mathbf{x})\, w_m^\star\right), \tag{6.20}$$

which is itself an indicator function.

6.3 Multiclass classification

In this section we briefly describe how feature bases may be used to automate the design of features for both popular methods of multiclass classification first introduced in Section 4.4. Note that throughout this section we will suppose that we have a dataset $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ consisting of C distinct classes, where $y_p \in \{1, 2, \ldots, C\}$.

6.3.1 One-versus-all multiclass classification

Generalizing the one-versus-all (OvA) approach (introduced in Section 4.4.1) to multiclass classification is quite straightforward. Recall that with OvA we decompose the multiclass problem into C individual two class subproblems, and having just discussed how to incorporate feature bases into such problems in the previous section we can immediately employ them with each OvA subproblem.

To learn the classifier distinguishing class c from all other classes we assign temporary labels to all the points: points in classes c and "not-c" are assigned temporary labels $+1$ and -1, respectively. Choosing a type of feature basis we transform each $\mathbf{x}_p$ into an M_c length feature (note this length can be chosen independently for each subproblem) vector $\mathbf{f}_p^{(c)}$ and solve the associated two class problem, as described in the previous section, giving parameters $(b_c, \mathbf{w}_c, \Theta_c)$.

To determine the class of a point $\mathbf{x}$ we combine the resulting C individual classifiers (via the fusion rule in (4.47) properly generalized) as

$$y = \underset{j=1\ldots C}{\text{argmax}} \left(b_j + \left(\mathbf{f}_p^{(j)}\right)^T \mathbf{w}_j\right). \tag{6.21}$$

Example 6.4 One-versus-all classification using fixed feature bases

In Fig. 6.10 we show the result of applying OvA to two multiclass datasets each containing $C = 3$ classes. For the first dataset (shown in the top left panel) we use a degree 4 polynomial for each subproblem, and likewise for the second dataset (shown in the bottom left panel) we use a degree 2 for all subproblems. The following three panels in each example show the resulting fit on each individual subproblem, with the final panel displaying the final combined boundary using Equation (6.21). Note in the second example especially that one of the subproblem classifiers is quite poor, but nonetheless the combined classifier perfectly separates all classes.

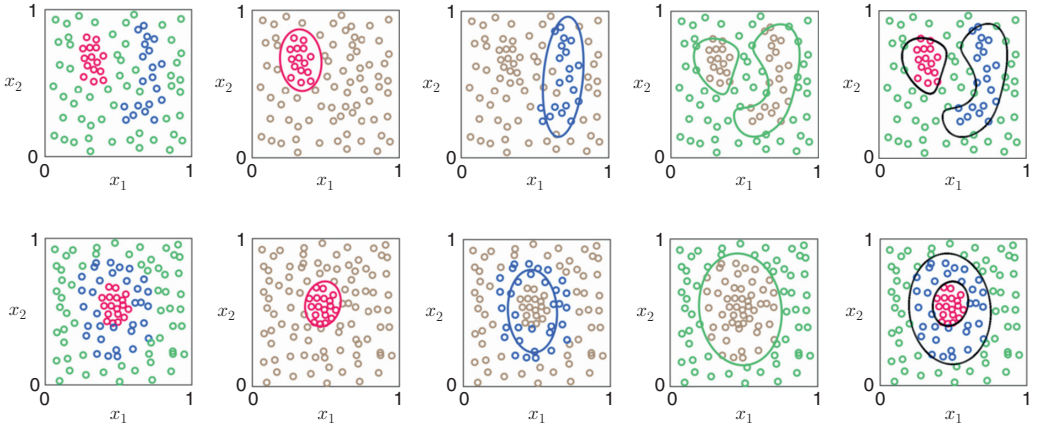

Fig. 6.10 Result of applying the OvA framework to two $C = 3$ class problems (shown in the top and bottom left panels respectively). In each case the following three panels show the result for the red class versus all, blue class versus all, and green class versus all subproblems. A degree 4 (top) and 2 (bottom) polynomial was used respectively for each subproblem. The panels on the extreme right show the combined boundary determined by Equation (6.21), which in both cases perfectly separates the three classes (even though the blue versus all classifier in the second example performs quite poorly).

6.3.2 Multiclass softmax classification

To apply the multiclass softmax framework discussed in Section 4.4 we transform all input data via a single fixed or neural network feature basis, and denote by $\mathbf{f}_p$ the resulting M length feature map of the point $\mathbf{x}_p$. We then minimize the corresponding version of the multiclass softmax cost function, first shown in Equation (4.53), on the transformed data as

$$g\left(b_1, \ldots, b_C, \mathbf{w}_1, \ldots, \mathbf{w}_C, \Theta\right) = -\sum_{c=1}^{C}\sum_{p\in\Omega_c}\left[\left(b_c + \mathbf{f}_p^T\mathbf{w}_c\right) - \log\left(\sum_{j=1}^{C} e^{b_j + \mathbf{f}_p^T\mathbf{w}_j}\right)\right]. \tag{6.22}$$

Note here that each $\mathbf{w}_c$ now has length M, and that the parameter set Θ as always contains internal parameters of a neural network basis feature if it is used (and is otherwise empty if using a fixed feature map). When employing a fixed feature basis this can then be minimized precisely as described for the original in Example 4.6, i.e., the gradient is given precisely as in Equation (4.57) replacing each $\mathbf{x}_p$ with $\mathbf{f}_p$. Computing the gradient is more complicated when employing a neural network feature basis (requiring careful bookkeeping and many uses of the chain rule) and additionally the corresponding cost function above becomes non-convex (while it remains convex when using any fixed feature basis).

6.4 Cross-validation for classification

In the previous chapter we saw how using more basis elements generally results in a better approximation of a continuous function. However, as we saw with regression in

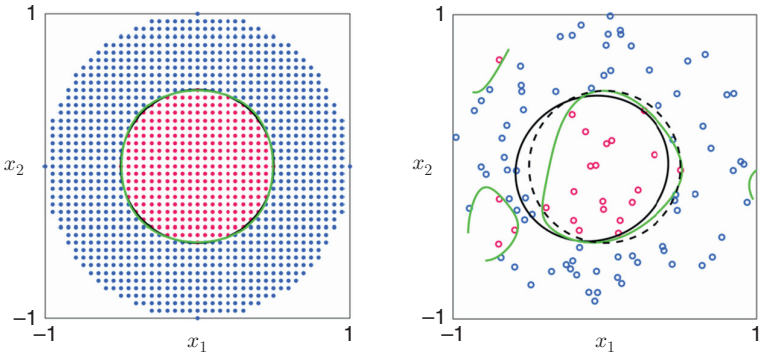

Fig. 6.11 Discretized (left panel) and noisy samples (right) from a generating function with class boundary shown (the circle of radius one-half) in dashed black, along with fits of degree 2 (in black) and degree 5 (in green) polynomial features. Like regression, while increasing the number of basis elements produces a better fit in the discretized case, for more realistic cases like the dataset on the right this can lead to overfitting. In the right panel the lower degree polynomial produces a classifier that matches the true boundary fairly well, while the higher degree polynomial leads to a classifier that overfits the data encapsulating falsely labeled red points outside the true boundary, thus leading to a poorer representation of the underlying generating function (see text for further details).

Section 5.3, while it is true that our approximation of a dataset itself improves as we add more basis features, this can substantially decrease our estimation of the underlying data generating function (a phenomenon referred to as *overfitting*). Unfortunately, the same overfitting problem presents itself in the case of classification as well. Similarly to regression, in the ideal classification scenario discussed in Section 6.1 using more basis elements generally improves our approximation. However, in general instances of classification, analogous to what we saw with regression, adding more basis features (increasing M) can result in fitting closely to the data we have while poorly to the underlying function (a phenomenon once again referred to as overfitting).

We illustrate the overfitting issue with classification using a particular dataset in Fig. 6.11, where we show the discretized indicator (left panel) along with the related noisy dataset (right panel) originally shown together in Fig. 6.5. For each dataset we show the resulting fit provided by both a degree 2 and a degree 5 polynomial (shown in black and green respectively). While the degree 2 features produce a classifier in each case that closely matches the true boundary, the higher degree 5 polynomial creates an overfitting classifier which encapsulates mislabeled points outside of the half circle boundary of the true function, leading to a poorer representation.

In this section we outline the use of cross-validation, culminating once again with the *k-fold cross-validation* method, for the intelligent automatic choice of M for both two class and multiclass classification problems. As with regression, here once again the k-fold method[8] provides a way of determining a proper value of M, however, once again this comes at significant computational cost.

[8] Readers particularly interested in using fixed bases with high dimensional input, deep network features, as well as multiclass softmax classification using feature bases should also see Section 7.3, where a variation of k-fold cross-validation is introduced that is more appropriate for these instances.

6.4.1 Hold out cross-validation

Analogous to regression, in the case of classification ideally we would like to choose a number M of basis features so that the corresponding learned representation matches the true data generating indicator function as well as possible. Of course because we only have access to this true function via (potentially) noisy samples, this goal must be pursued based solely on the data. Therefore once again we aim to choose M such that the corresponding model fits both the data we currently have, as well as the data we might receive in the future. Because we do not have access to any future data points this intuitively directs us to employ cross-validation, where we tune M, so that the corresponding model fits well to an unseen portion of our original data (i.e., the testing set).

Thus we can do precisely what was described for regression in Section 5.3 and perform k-fold cross-validation to determine M. In other words, we can simulate this desire by splitting our original data into k evenly sized pieces and merge $k - 1$ of them into a training set and use the remaining piece as a testing set. Furthermore the same intuition for choosing k introduced for regression also holds here, with common values in practice being in the range $k = 3$–10.

Example 6.5 Hold out for classification using polynomial features

For clarity we first show an example of the hold out method, followed by explicit computations, which are then simply repeated on each fold (averaging the results) in performing the k-fold method. In Fig. 6.12 we show the result of applying hold out cross-validation to the dataset first shown in the bottom panels of Fig. 6.5. Here we use $k = 3$, use the softmax cost, and M in the range $M = 2, 5, 9, 14, 20, 27, 35, 44$ which corresponds (see footnote 5) to polynomial degrees $D = 1, 2, \ldots, 8$ (note that for clarity panels in the figure are indexed by D).

Based on the models learned for each value of M (see the middle set of eight panels of the figure) we plot training and testing errors (in the panel second to the right), measuring how well each model fits the training and testing data respectively, over the entire range of values. Note that unlike the testing error, the training error always decreases as we increase M (which occurs more generally regardless of the dataset/feature basis used). The model that provides the smallest testing error ($M^\star = 14$ or equivalently $D^\star = 4$) is then trained again on the entire dataset, giving the final classification model shown in black in the rightmost panel of the figure.

6.4.2 Hold out calculations

Here we give a complete set of hold out cross-validation calculations in a general setting, which closely mirrors the version given for regression in Section 5.3.3. We denote the collection of points belonging to the training and testing sets respectively by their indices as

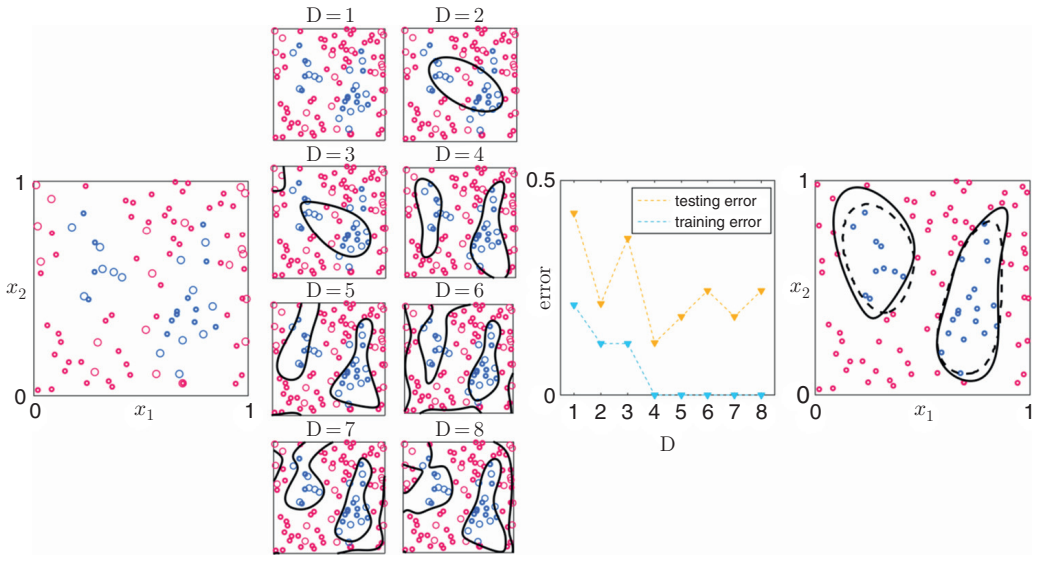

Fig. 6.12 An example of hold out cross-validation applied using polynomial features. (left panel) The original data split into training and testing sets, with the points belonging to each set drawn as smaller thick and larger thin points respectively. (middle eight panels) The fit resulting from each set of degree D polynomial features in the range $D = 1, 2, \ldots, 8$ shown in black in each panel. Note how the lower degree fits underfit the data, while the higher degree fits overfit the data. (second from right panel) The training and testing errors, in blue and yellow respectively, of each fit over the range of degrees tested. From this we see that $D^\star = 4$ (or $M^\star = 14$) provides the best fit. Also note how the training error always decreases as we increase the degree/number of basis elements, which will always occur regardless of the dataset/feature basis type used. (right panel) The final model using $M^\star = 14$, trained on the entire dataset (shown in black), fits the data well and closely matches the boundary of the underlying data generating function (shown in dashed black).

$$\begin{aligned} \Omega_{\text{train}} &= \left\{p \,|\, \left(\mathbf{x}_p,\, y_p\right) \text{ belongs to the training set}\right\} \\ \Omega_{\text{test}} &= \left\{p \,|\, \left(\mathbf{x}_p,\, y_p\right) \text{ belongs to the testing set}\right\} \end{aligned} . \tag{6.23}$$

We then choose a basis type (e.g., polynomial, Fourier, neural network) and choose a range for the number of basis features over which we search for an ideal value for M. To determine the training and testing error of each value of M tested we first form the corresponding feature vector $\mathbf{f}_p = \left[\, f_1\left(\mathbf{x}_p\right) \;\; f_2\left(\mathbf{x}_p\right) \;\; \cdots \;\; f_M\left(\mathbf{x}_p\right) \,\right]^T$ and fit a corresponding model to the training set by minimizing e.g., the softmax or squared margin cost. For example employing the softmax we solve

$$\underset{b,\,\mathbf{w},\Theta}{\text{minimize}} \sum_{p \in \Omega_{\text{train}}} \log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\mathbf{w}\right)}\right). \tag{6.24}$$

Denoting a solution to the problem above as $\left(b_M^\star,\, \mathbf{w}_M^\star,\, \Theta_M^\star\right)$ we find the training and testing errors for the current value of M using these parameters over the training and testing sets using the counting cost (see Section 6.2.3), respectively:

$$\begin{aligned}\text{Training error} &= \frac{1}{|\Omega_{\text{train}}|}\sum_{p\in\Omega_{\text{train}}}\max\left(0,\ \text{sign}\left(-y_p\left(b^\star+\mathbf{f}_p^T\mathbf{w}^\star\right)\right)\right)\\ \text{Testing error} &= \frac{1}{|\Omega_{\text{test}}|}\sum_{p\in\Omega_{\text{test}}}\max\left(0,\ \text{sign}\left(-y_p\left(b^\star+\mathbf{f}_p^T\mathbf{w}^\star\right)\right)\right),\end{aligned} \tag{6.25}$$

where the notation $|\Omega_{\text{train}}|$ and $|\Omega_{\text{test}}|$ denotes the cardinality or number of points in the training and testing sets, respectively. Once we have performed these calculations for all values of M we wish to test, we choose the one that provides the lowest testing error, denoted by $M^\star$.

Finally we form the feature vector $\mathbf{f}_p = \left[\begin{array}{ccc} f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots \quad f_{M^\star}\left(\mathbf{x}_p\right)\end{array}\right]^T$ for all the points in the entire dataset, and solve the following optimization problem over the entire dataset to form the final model

$$\underset{b,\mathbf{w},\Theta}{\text{minimize}}\ \sum_{p=1}^{P}\log\left(1+e^{-y_p\left(b+\mathbf{f}_p^T\mathbf{w}\right)}\right). \tag{6.26}$$

6.4.3 k-fold cross-validation

As introduced in Section 5.3.4, *k-fold cross-validation* is a robust extension of the hold out method whereby the procedure is repeated k times where in each instance (or *fold*) we treat a different portion of the split as a testing set and the remaining $k-1$ portions as the training set. The hold out calculations are then made, as detailed previously, on each fold and the value of M with the lowest *average* testing error is chosen. This produces a more robust choice of M, because potentially poor hold out choices on individual folds can be averaged out, producing a stronger model.

Example 6.6 k-fold cross-validation for classification using polynomial features

In Fig. 6.13 we illustrate the result of applying k-fold cross-validation to choose the ideal number M of polynomial features for the dataset shown in Example 6.5, where it was originally used to illustrate the hold out method. As in the previous example, here we set $k = 3$, use the softmax cost, and try M in the range $M = 2, 5, 9, 14, 20, 27, 35, 44$ which corresponds (see footnote 5) to polynomial degrees $D = 1, 2, \ldots, 8$ (note that for clarity panels in the figure are indexed by D).

In the top three rows of Fig. 6.13 we show the result of applying hold out on each fold. In each row we show a fold's training and testing data in the left panel, the training/testing errors for each M on the fold (as computed in Equation (6.25)) in the middle panel, and the final model (learned to the entire dataset) provided by the choice of M with lowest testing error. As can be seen, the particular split leads to an overfitting result on the first two folds and an underfitting result on the third fold. In the middle panel of the final row we show the result of averaging the training/testing errors over all $k = 3$ folds, and in the right panel the result of choosing the overall best $M^\star = 14$ (or equivalently $D^\star = 4$) providing the lowest average

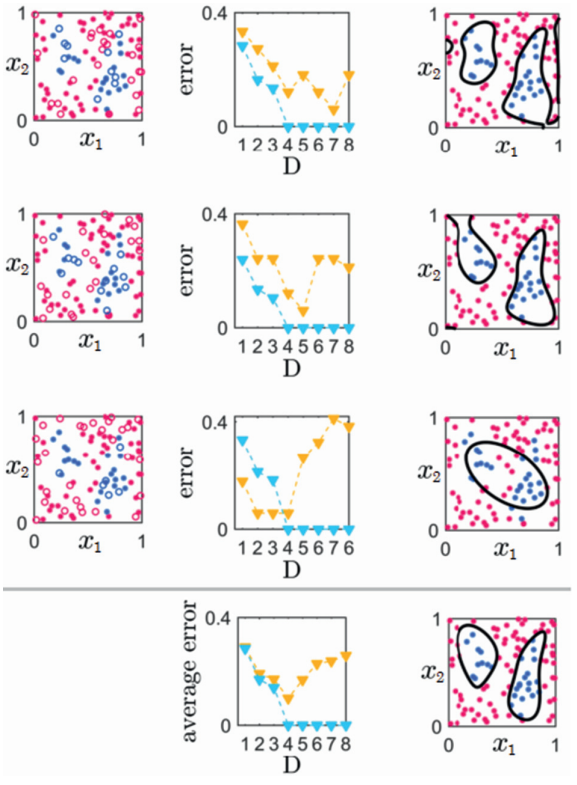

Fig. 6.13 Result of performing k-fold cross-validation with $k = 3$ (see text for further details). The top three rows display the result of performing the hold out method on each fold. The left, middle, and right columns show each fold's training/testing sets (drawn as thick and thin points respectively), training and testing errors over the range of M tried, and the final model (fit to the entire dataset) chosen by picking the value of M providing the lowest testing error. Due to the split of the data, performing hold out on each fold results in a poor overfitting (first two folds) or underfitting (final fold) model for the data. However, as illustrated in the final row, by averaging the testing errors (bottom middle panel) and choosing the model with minimum associated average test error we average out these problems (finding that $D^\star = 4$ or $M^\star = 14$) and determine an excellent model for the phenomenon (as shown in the bottom right panel).

testing error. By taking this value we average out the poor choices determined on each fold, and end up with a model that fits both the data and underlying function quite well.

Example 6.7 Warning examples

When a k-fold determined set of features performs poorly this is almost always indicative of a poorly structured dataset (i.e., there is little relationship between the input/output data), like the one shown in the left panel of Fig. 6.14. However, there are

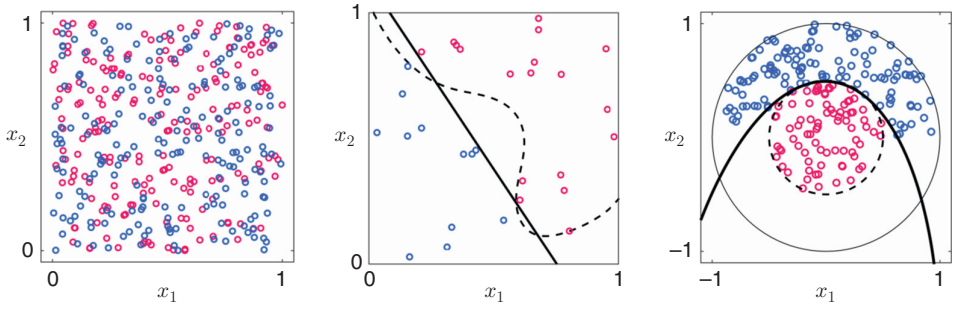

Fig. 6.14 (left panel) A low accuracy k-fold fit to a dataset indicates that it has little structure (i.e., that there is little to no relationship between the input and output). It is possible that a high accuracy k-fold fit fails to capture the true nature of an underlying function, as when (middle panel) we have too little data (the k-fold linear separator is shown in black, and the true nonlinear separator is shown dashed) and (right panel) when we have poorly distributed data (again the k-fold separator is shown in black, the original separator dashed). See text for further details.

also instances, when we have too little or too poorly distributed data, when a *high* performing k-fold model can be misleading as to how well we understand a phenomenon. In the middle and right panels of the figure we show two such instances that the reader should keep in mind when using k-folds, where we either have too little (middle panel) or poorly distributed data (right panel).

In the first instance we have generated a small sample of points based on the second indicator function shown in Fig. 6.3, which has a nonlinear boundary in the original feature space. However, the sample of data is so small that it is perfectly linearly separable, and thus applying e.g., k-fold cross-validation with polynomial basis features will properly (due to the small selection of data) recover a line to distinguish between the two classes. However, clearly data generated via the same underlying process in the future will violate this linear boundary, and thus our model will perform poorly. This sort of problem arises in applications such as automatic medical diagnosis (see Example 1.6) where access to data is limited. Unless we can gather additional data to fill out the space (making the nonlinear boundary more visible) this problem is unavoidable.

In the second instance shown in the right panel of the figure, we have plenty of data (generated using the indicator function originally shown in Fig. 6.4) but it is poorly distributed. In particular, we have no samples from the blue class in the lower half of the space. In this case the k-fold method (again here using polynomial features) properly determines a separating boundary that perfectly distinguishes the two classes. However, many of the blue class points we would receive in the future in the lower half of the space will be misclassified given the learned k-fold model. This sort of issue can arise in practice, e.g., when performing face detection (see Example 1.4), if we do not collect a thorough dataset of blue (e.g., "non-face") examples. Again, unless we can gather further data to fill out the space this problem is unavoidable.

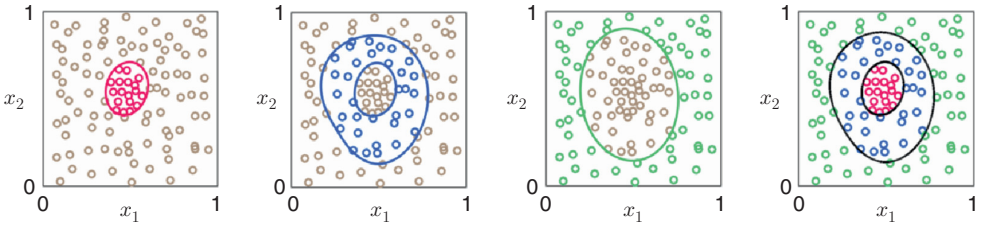

Fig. 6.15 Result of performing $k = 3$ fold cross-validation on the $C = 3$ class dataset first shown in Fig. 6.10 using OvA (see text for further details). The left three panels show the result for the red class versus all, blue class versus all, and green class versus all subproblems. For the red/green versus all problems the optimal degree found was $D^\star = 2$, while for the blue versus all $D^\star = 4$ (note how this produces a better fit than the $D = 2$ fit shown originally in Fig. 6.10). The right panel shows the combined boundary determined by Equation (6.21), which perfectly separates the three classes.

6.4.4 *k*-fold cross-validation for one-versus-all multiclass classification

Employing the one-versus-all (OvA) framework for multiclass classification, we can immediately apply the k-fold method described previously. For a C class problem we simply apply the k-fold method to each of the C two class classification problems, and combine the resulting classifiers as shown in Equation (6.21). We show the result of applying $k = 3$ fold cross-validation with OvA on two datasets with $C = 3$ and $C = 5$ classes respectively in Fig. 6.15 and 6.16, where we have used polynomial features with $M = 2, 5, 9, 14, 20, 27, 35, 44$ or equivalently of degree $D = 1, 2, \ldots, 8$ for each two class subproblem. Displayed in each figure are the nonlinear boundaries determined for each fold, as well as the combined result in the right panel of each figure. In both instances the combined boundaries separate the different classes of data very well.

6.5 Which basis works best?

For an arbitrary classification dataset we cannot say whether a particular feature basis will provide better results than others. However, as with the case of regression discussed in Section 5.4, we can say something about the choice of bases in particular instances. For example, in the instance where data is plentiful and well distributed throughout the input space we can expect comparable performance among different feature bases (this was illustrated for the case of regression in Fig. 5.20). Practical considerations can again guide the choice of basis as well.

Due to the nature of classification problems it is less common (than with regression) that domain knowledge leads to a particular choice of basis. Rather, in practice it is more common to employ knowledge in the design of a feature transformation (like those discussed for text, image, or audio data in Section 4.6), and then determine possible nonlinear boundaries in this transformed data using feature bases as described in this chapter. For certain data types such as image data one can incorporate a parameterized transformation that outlines the sort of edge detection/histogramming

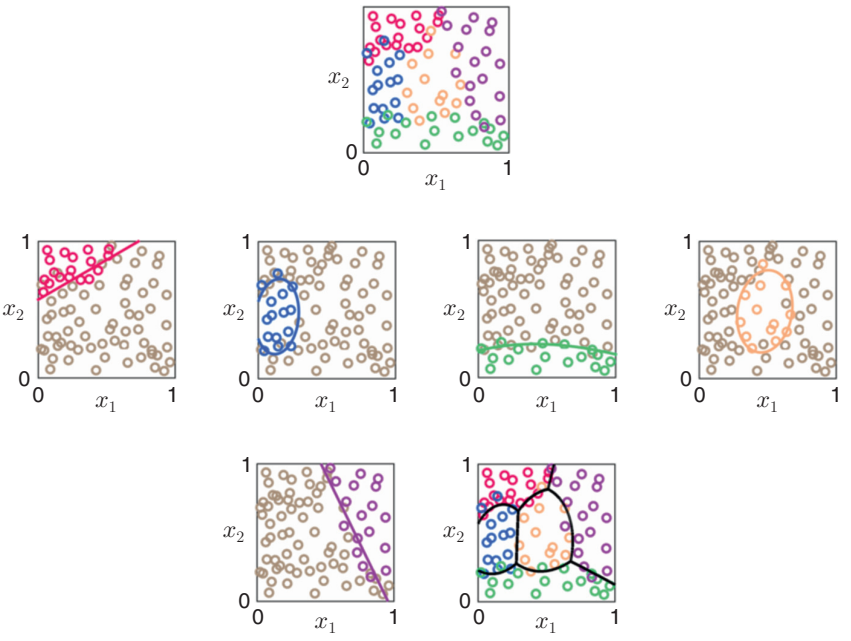

Fig. 6.16 Result of performing $k = 3$ fold cross-validation on an overlapping $C = 5$ class classification dataset (top panel) using OvA. The middle four panels show the result for the red, blue, green, and yellow class versus all subproblems respectively. The bottom two panels show the (left) purple class versus all and (right) the final combined boundary. For the red/purple versus all problems the k-fold found degree was $D^\star = 1$, while for the remaining subproblems $D^\star = 2$.

operations outlined in Section 4.6.2 directly into basis elements themselves. Parameters of this transformation are then learned simultaneously with those of the weighted basis sum itself. A popular example of this sort of approach is the *convolutional network* (see e.g., [42–44] and references therein), which incorporates such (parameterized) knowledge-driven features into a standard feed forward neural network basis.

Regardless of how knowledge is integrated, having some understanding of a phenomenon can significantly lessen the amount of data required to produce a k-fold representation that properly traces out a data generating function. On the other hand, broadly speaking if we have no understanding of a phenomenon we will typically require a significant amount of data in order to ensure that the features we have designed through the k-fold process are truly representative. What constitutes a "significant amount"? There is no precise formula in general, but due to the curse of dimensionality (see Fig. 5.2) we can say that the higher the dimension of the input the exponentially more data we will need to properly understand the underlying function.

This data–knowledge tradeoff is illustrated symbolically in Fig. 6.17.

6.6 Summary

We have seen that, analogous to regression, the general problem of classification is one of function approximation based on noisy samples. In the instance of classification,

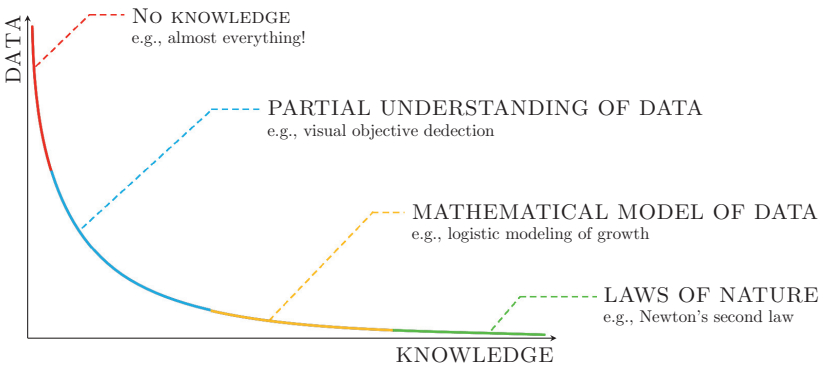

Fig. 6.17 A symbolic representation of the data–knowledge spectrum. The more knowledge we can incorporate into the design of features the less data is required to determine a strong k-fold cross-validated set of features. At the other end of the spectrum, if we know nothing regarding the underlying phenomenon we are modeling we will need a significant amount of data in order to forge strong cross-validated features.

however, the underlying data generating function is a piecewise continuous indicator function. As in the previous chapter, we began in the first section by investigating how to approximate such a data generating function itself, leading to both the familiar fixed and adjustable neural network bases we have seen previously.

In the second and third sections we described how real instances of classification datasets can be thought of as noisy samples from a true underlying indicator. We then saw how we can use a tractable minimization problem to learn the parameters of a weighted sum of basis features to fit general classification datasets. This idea was also shown to be easily integrated into both one-versus-all and multiclass softmax classification schemes, leading to natural nonlinear extensions of both.

In Section 6.4 we saw (as with regression) how overfitting is a problem when using too many features for classification. Cross-validation, culminating with the k-fold method, was then reintroduced in the context of classification as a way of preventing overfitting. Again, as with regression, it is (k-fold) cross-validation that often uses the bulk of computational resources in practice when solving general classification problems.

6.7 Exercises

Section 6.1 exercises

Exercises 6.1 Softmax cost gradient/Hessian calculations with fixed basis features

a) Assuming a fixed feature basis verify, using the compact notation $\tilde{\mathbf{f}}_p = \begin{bmatrix} 1 \\ \mathbf{f}_p \end{bmatrix}$ and $\tilde{\mathbf{w}} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}$, that the gradient of the softmax cost given in Equation (6.14) is correct. Furthermore, verify that the Hessian of the softmax in this case is given by

$$\nabla^2_{\tilde{\mathbf{w}}} g\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P} \sigma\left(-y_p \tilde{\mathbf{f}}_p^T \tilde{\mathbf{w}}\right)\left(1 - \sigma\left(-y_p \tilde{\mathbf{f}}_p^T \tilde{\mathbf{w}}\right)\right) \tilde{\mathbf{f}}_p \tilde{\mathbf{f}}_p^T. \tag{6.27}$$

Both gradient and Hessian here are entirely similar to the originals given in Example 4.1, replacing each $\mathbf{x}_p$ with its corresponding feature vector $\mathbf{f}_p$.

b) Show that the softmax cost using M elements of any fixed feature basis is still convex by verifying that it satisfies the second order condition for convexity. *Hint: the Hessian is a weighted sum of outer product matrices like the one described in Exercise 2.10.*

Exercises 6.2 Squared margin cost gradient/Hessian calculations with fixed feature basis

a) Assuming a fixed feature basis verify, using the compact notation $\tilde{\mathbf{f}}_p = \begin{bmatrix} 1 \\ \mathbf{f}_p \end{bmatrix}$ and $\tilde{\mathbf{w}} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}$, that the gradient and Hessian of the squared margin cost

$$g\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P} \max^2\left(0, 1 - y_p \tilde{\mathbf{f}}_p^T \tilde{\mathbf{w}}\right), \tag{6.28}$$

are given as

$$\begin{aligned} \nabla_{\tilde{\mathbf{w}}} g\left(\tilde{\mathbf{w}}\right) &= -2 \sum_{p=1}^{P} y_p \tilde{\mathbf{f}}_p \max\left(0,\ 1 - y_p \tilde{\mathbf{f}}_p^T \tilde{\mathbf{w}}\right) \\ \nabla^2_{\tilde{\mathbf{w}}} g\left(\tilde{\mathbf{w}}\right) &= 2 \sum_{p \in \Omega_{\tilde{\mathbf{w}}}} \tilde{\mathbf{f}}_p \tilde{\mathbf{f}}_p^T, \end{aligned} \tag{6.29}$$

where $\Omega_{\tilde{\mathbf{w}}}$ is the index set $\Omega_{\tilde{\mathbf{w}}} = \left\{p \middle| 1 - y_p \tilde{\mathbf{f}}_p^T \tilde{\mathbf{w}} > 0\right\}$. These are entirely similar to the calculations given in Example 4.2 except for using the feature map $\mathbf{f}_p$ in place of the input $\mathbf{x}_p$.

b) Show that the squared margin cost using M elements of any fixed feature basis is convex. *Hint: see Exercise 4.6.*

Exercises 6.3 Polynomial basis features and the softmax cost

In this exercise you will explore how various degree D polynomial basis features fit using the softmax cost and the dataset shown in the bottom panel of Fig. 6.6. For this exercise you will need the wrapper *poly_classification_hw* and the data located in *2eggs_data.csv*.

a) Use the description of the two-dimensional polynomial basis features given in footnote 5 to transform the input using a general degree D polynomial. Write this feature transformation in the module

$$\mathbf{F} = \text{poly\_features}\left(\mathbf{X},\ D\right) \tag{6.30}$$

located in the wrapper. Here $\mathbf{X}$ is the input data, D the degree of the polynomial features, and $\mathbf{F}$ the corresponding degree D feature transformation of the input (note your code should be able to transform the input to any degree D desired).

b) With your module complete you may run the wrapper. Two figures will be generated: the first shows the data along with various degree D polynomial fits, and the second shows the average number of misclassifications of each fit to the data-set. Discuss the results shown in these two figures. In particular, describe the relationship between a model's average number of misclassifications and how well it seems to represent the phenomenon generating the data as D increases over the range shown.

Exercises 6.4 Calculate the gradient using a single hidden layer basis

When employing M single hidden layer basis features (using any activation $a(\cdot)$) the full gradient of a cost g (e.g., the softmax) is a vector of length $Q = M(N+2)+1$ containing the derivatives of the cost with respect to each variable,

$$\nabla g = \begin{bmatrix} \frac{\partial}{\partial b} g & \frac{\partial}{\partial w_1} g & \cdots & \frac{\partial}{\partial w_M} g & \frac{\partial}{\partial c_1} g \cdots & \frac{\partial}{\partial c_M} g & \nabla_{\mathbf{v}_1}^T g & \cdots & \nabla_{\mathbf{v}_M}^T g \end{bmatrix}^T, \quad (6.31)$$

where the derivatives are easily calculated using the chain rule.

a) Using the chain rule verify that the derivatives of this gradient (using the softmax cost) are given by

$$\begin{aligned}
\frac{\partial}{\partial b} g &= -\sum_{p=1}^{P} \sigma\left(-y_p\left(b + \sum_{m=1}^{M} w_m a\left(c_m + \mathbf{x}_p^T \mathbf{v}_m\right)\right)\right) y_p \\
\frac{\partial}{\partial w_n} g &= -\sum_{p=1}^{P} \sigma\left(-y_p\left(b + \sum_{m=1}^{M} w_m a\left(c_m + \mathbf{x}_p^T \mathbf{v}_m\right)\right)\right) a\left(c_n + \mathbf{x}_p^T \mathbf{v}_n\right) y_p \\
\frac{\partial}{\partial c_n} g &= -\sum_{p=1}^{P} \sigma\left(-y_p\left(b + \sum_{m=1}^{M} w_m a\left(c_m + \mathbf{x}_p^T \mathbf{v}_m\right)\right)\right) a'\left(c_n + \mathbf{x}_p^T \mathbf{v}_n\right) w_n y_p \\
\nabla_{\mathbf{v}_n} g &= -\sum_{p=1}^{P} \sigma\left(-y_p\left(b + \sum_{m=1}^{M} w_m a\left(c_m + \mathbf{x}_p^T \mathbf{v}_m\right)\right)\right) a'\left(c_n + \mathbf{x}_p^T \mathbf{v}_n\right) \mathbf{x}_p w_n y_p.
\end{aligned} \quad (6.32)$$

b) This gradient can be written more efficiently for programming languages like Python and MATLAB/OCTAVE that have especially good implementations of matrix/vector operations by writing it more compactly. Supposing that $a = \tanh(\cdot)$ is the activation function (meaning $a' = \text{sech}^2(\cdot)$ is the hyperbolic secant function squared), verify that the derivatives from part a) may be written more compactly as

$$\begin{aligned}
\frac{\partial}{\partial b} g &= -\mathbf{1}_{P\times 1}^T \mathbf{q} \odot \mathbf{y} \\
\frac{\partial}{\partial w_n} g &= -\mathbf{1}_{P\times 1}^T (\mathbf{q} \odot \mathbf{t}_n \odot \mathbf{y}) \\
\frac{\partial}{\partial c_n} g &= -1_{P\times 1}^T (\mathbf{q} \odot \mathbf{s}_n \odot \mathbf{y}) w_n \\
\nabla_{\mathbf{v}_n} g &= -\mathbf{X} \cdot \mathbf{q} \odot \mathbf{s}_n \odot \mathbf{y} w_n,
\end{aligned} \quad (6.33)$$

where $\odot$ denotes the component-wise product and denoting $q_p = \sigma\left(-y_p\left(b + \sum_{m=1}^{M} w_m \tanh\left(c_m + \mathbf{x}_p^T\mathbf{v}_m\right)\right)\right)$, $t_{np} = \tanh\left(c_n + \mathbf{x}_p^T\mathbf{v}_n\right)$, $s_{np} = \text{sech}^2\left(c_n + \mathbf{x}_p^T\mathbf{v}_n\right)$, and $\mathbf{q}$, $\mathbf{t}_n$, and $\mathbf{s}_n$ the P length vectors containing these entries.

Exercises 6.5 Code up gradient descent using single hidden layer bases

In this exercise you will reproduce the classification result using a single hidden layer feature basis with tanh activation shown in the middle panel of Fig. 6.9.

a) Plug the gradient from Exercise 6.4 into the gradient descent function

$$\mathbf{T} = \text{tanh\_softmax}\left(\mathbf{X}, \mathbf{y}, M\right) \tag{6.34}$$

located in the wrapper *single_layer_classification_hw* and the dataset *genreg_data.csv*, both of which may be downloaded from the book website. Here $\mathbf{T}$ is the set of optimal weights learned via gradient descent, $\mathbf{X}$ is the input data matrix, $\mathbf{y}$ contains the associated labels, and M is the number of basis features to employ.

Almost all of this function has already been constructed for you, e.g., various initializations, step length, etc., and you need only enter the gradient of the associated cost function. All of the additional code necessary to generate the associated plot is already provided in the wrapper. Due to the non-convexity of the associated cost function when using neural network features, the wrapper will run gradient descent several times and plot the result of each run.

b) Try adjusting the number of basis features M in the wrapper and run it several times. Is there a value of M other than $M = 4$ that seems to produce a good fit to the underlying function?

Exercises 6.6 Code up the *k*-nearest neighbors (*k*-NN) classifier

The *k-nearest neighbors* (*k*-NN) is a local classification scheme that, while differing from the more global feature basis approach described in this chapter, can produce nonlinear boundaries in the original feature space as illustrated for some particular examples in Fig. 6.18.

With the *k*-NN approach there is no training phase to the classification scheme. We simply use the training data directly to classify any new point $\mathbf{x}_{\text{new}}$ by taking the average of the labels of its k-nearest neighbors. That is, we create the label y_{new} for a point $\mathbf{x}_{\text{new}}$ by simply calculating

$$y_{\text{new}} = \text{sign}\left(\sum_{i\in\Omega} y_i\right), \tag{6.35}$$

where Ω is the set of indices of the k closest training points to $\mathbf{x}_{\text{new}}$. To avoid tie votes (i.e., a value of zero above) typically the number of neighbors k is chosen to be odd (however, in practice the value of k is typically set via cross-validation).

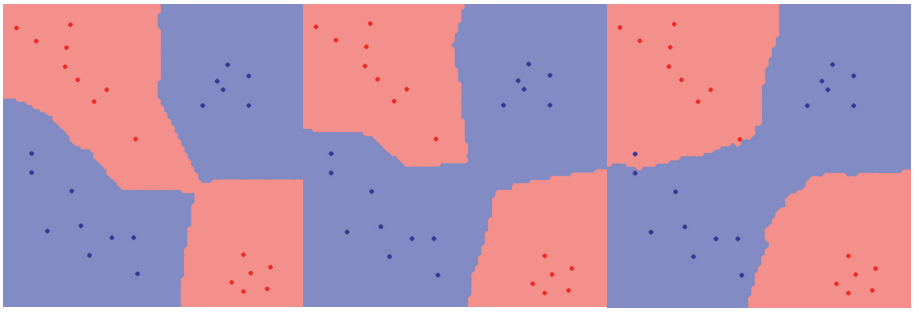

Fig. 6.18 The k-NN classifier applied to a two class dataset (the blue and red points) where (left panel) $k = 1$, (middle panel) $k = 5$, and (right panel) $k = 10$. All points in the space have been colored according to the rule given in Equation (6.35) where the red and blue classes have labels $+1$ and -1 respectively.

Code up the k-NN algorithm and reproduce the results shown in Fig. 6.18 using the dataset located in *knn_data.csv*.

Section 6.2 exercises

Exercises 6.7 One-versus-all using a polynomial basis

In this exercise you will reproduce the one-versus-all classification on the $C = 3$ class dataset shown in the bottom panels of Fig. 6.10 using polynomial features. Note that for this exercise you will need to have completed the poly_features module for polynomial features described in Exercise 6.3.

a) Place the *poly_features* module in the wrapper *ova_fixed_basis* and use the dataset *bullseye_data.csv*, both of which may be downloaded from the book website. After installing the module try running the wrapper to reproduce the results shown in Fig. 6.10.

b) Try adjusting the degree D in the wrapper and run it several times. Is there a value of D other than $D = 2$ that seems to produce a good fit to the data?

Section 6.3 exercises

Exercises 6.8 Code up hold out cross-validation

In this exercise you will perform hold out cross-validation on the dataset shown in Fig. 6.12 as described in Example 6.5 of the text. Start by randomly splitting the dataset, located in *2eggs_data.csv*, into $k = 3$ equal sized folds (keeping 2 folds as training, and 1 fold as testing data). Use the polynomial basis features with M in the range $M = 2, 5, 9, 14, 20, 27, 35, 44$ (or likewise D in the range $D = 1, 2, \ldots, 8$) and produce a graph showing the training and testing error for each D like the one shown in Fig. 6.12, as well as the best (i.e., the lowest test error) model fit to the data.

Note: your results may appear slightly different than those of the figure, given that you will likely use a different random partition of the data. Note: you may find it very useful here to re-use code from previous exercises e.g., functions that compute polynomial features, plot curves, etc.

Exercises 6.9 Code up *k*-fold cross-validation

In this exercise you will perform k-fold cross-validation on the dataset shown in Fig. 6.13 as described in Example 6.6 of the text. Start by randomly splitting the dataset, located in *2eggs_data.csv*, into $k = 3$ equal sized folds (keeping 2 folds as training, and 1 fold as testing data). Use the polynomial basis features with M in the range $M = 2, 5, 9, 14, 20, 27, 35, 44$ (or likewise D in the range $D = 1, 2, \ldots, 8$) and produce a graph showing the training and testing error for each D like the one shown in Fig. 6.13, as well as the best (i.e., the lowest average test error) model fit to the data.

Note: your results may appear slightly different than those of the figure given that you will likely use a different random partition of the data. Note: you may find it very useful here to re-use code from previous exercises e.g., functions that compute polynomial features, plot curves, etc.