3 Regression

In this chapter we formally describe the regression problem, or the fitting of a representative line or curve (in higher dimensions a hyperplane or general surface) to a set of input/output data points as first broadly detailed in Section 1.2.1. Regression in general may be performed for a variety of reasons: to produce a so-called trend line (or curve) that can be used to help visually summarize, drive home a particular point about the data under study, or to learn a model so that precise predictions can be made regarding output values in the future. Here we also discuss more formally the notion of feature design for regression, in particular focusing on rare low dimensional instances (like the one outlined in Example 1.7) when very specific feature transformations of the data can be proposed. We finally end by discussing regression problems that have non-convex cost functions associated with them and a commonly used approach, called ℓ_2 regularization, for ameliorating some of the problems associated with the minimization of such functions.

3.1 The basics of linear regression

With linear regression we aim to fit a line (or hyperplane in higher dimensions) to a scattering of data. In this section we describe the fundamental concepts underlying this procedure.

3.1.1 Notation and modeling

Data for regression problems comes in the form of a training set of P input/output observation pairs:

$$\left\{\left(\mathbf{x}_1, y_1\right), \left(\mathbf{x}_2, y_2\right), \ldots, \left(\mathbf{x}_P, y_P\right)\right\}, \tag{3.1}$$

or $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ for short, where $\mathbf{x}_p$ and y_p denote the pth input and output respectively. In many instances of regression, like the one discussed in Example 1.1, the input to regression problems is scalar-valued (the output will always be considered scalar-valued here) and hence the linear regression problem is geometrically speaking one of fitting a line to the associated scatter of data points in 2-dimensional space. In general, however, each input $\mathbf{x}_p$ may be a column vector of length N

$$\mathbf{x}_p = \begin{bmatrix} x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{N,p} \end{bmatrix}, \tag{3.2}$$

in which case the linear regression problem is analogously one of fitting a hyperplane to a scatter of points in $N + 1$ dimensional space.

In the case of scalar input, fitting a line to the data (see Fig. 3.1) requires we determine a slope w and bias (or "y-intercept") b so that the approximate linear relationship holds between the input/output data,

$$b + x_p w \approx y_p, \quad p = 1, \ldots, P. \tag{3.3}$$

Note that we have used the *approximately equal* sign in (3.3) because we cannot be sure that all data lies completely on a single line. More generally, when the input dimension is $N \geq 1$, then we have a bias and N associated weights,

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}, \tag{3.4}$$

to tune properly in order to fit a hyperplane (see Fig. 3.1). Likewise the linear relationship in (3.3) is then more generally given as

$$b + \mathbf{x}_p^T \mathbf{w} \approx y_p, \quad p = 1, \ldots, P. \tag{3.5}$$

The elements of an input vector $\mathbf{x}_p$ are referred to as *input features* to a regression problem. For instance the student debt data described in Example 1.1 has only one feature: *year*. Conversely in the GDP growth rate data described in Example 3.1 the first element of the input feature vector might contain the feature *unemployment rate* (that is,

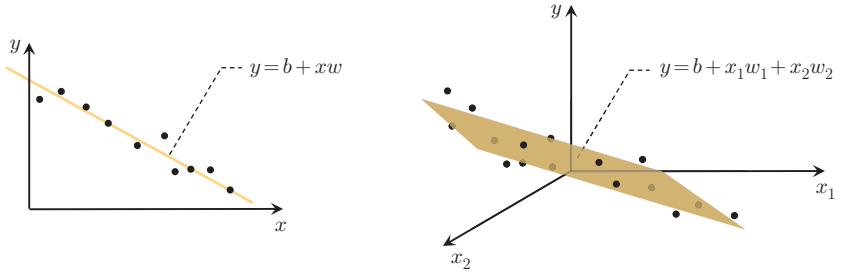

Fig. 3.1 (left panel) A dataset in two dimensions along with a well-fitting line. A line in two dimensions is defined as $b + xw = y$, where b is referred to as the bias and w the weight, and a point (x_p, y_p) lies close to it if $b + x_p w \approx y_p$. (right panel) A simulated 3-dimensional dataset along with a well-fitting hyperplane. A hyperplane is defined as $b + \mathbf{x}^T \mathbf{w} = y$, where again b is called the bias and $\mathbf{w}$ the weight vector, and a point $(\mathbf{x}_p, y_p)$ lies close to it if $b + \mathbf{x}_p^T \mathbf{w} \approx y_p$.

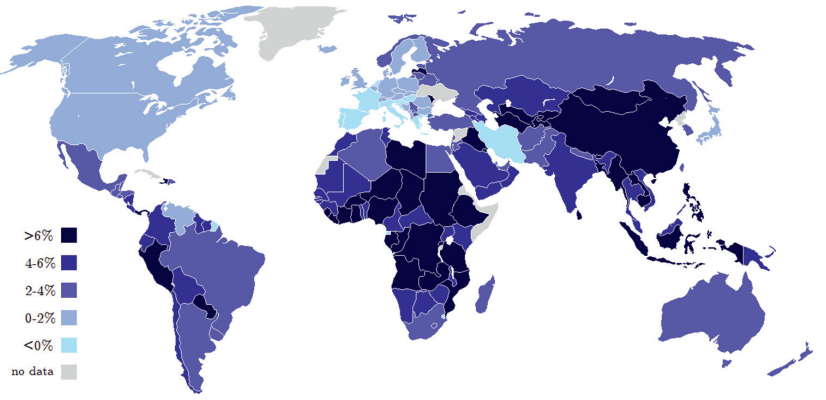

Fig. 3.2 A map of the world where countries are color-coded by their GDP growth rates (the darker the color the higher the growth rate) as reported by the International Monetary Fund (IMF) in 2013.

the unemployment data from each country under study), the second might contain the feature *education level*, and so on.

Example 3.1 Predicting Gross Domestic Product growth rates

As an example of a regression problem with vector-valued input consider the problem of predicting the growth rate of a country's Gross Domestic Product (GDP), which is the value of all goods and services produced within a country during a single year. Economists are often interested in understanding factors (e.g., unemployment rate, education level, population count, land area, income level, investment rate, life expectancy, etc.,) which determine a country's GDP growth rate in order to inform better financial policy making. To understand how these various *features* of a country relate to its GDP growth rate economists often perform linear regression [33, 72].

In Fig. 3.2 we show a heat map of the world where countries are color-coded based on their GDP growth rate in 2013, reported by the International Monetary Fund (IMF) (data used in this figure was taken from [12]).

3.1.2 The Least Squares cost function for linear regression

To find the parameters of the hyperplane which best fits a regression dataset, it is common practice to first form the *Least Squares cost function*. For a given set of parameters $(b, \mathbf{w})$ this cost function computes the total squared error between the associated hyperplane and the data (as illustrated pictorially in Fig. 3.3), giving a good measure of how well the particular linear model fits the dataset. Naturally then the best fitting hyperplane is the one whose parameters minimize this error.

Because we aim to have the system of equations in (3.5) hold as well as possible, to form the desired cost we simply square the difference (or error) between the linear model $b + \mathbf{x}_p^T\mathbf{w}$ and the corresponding output y_p over the entire dataset. This gives the Least Squares cost function

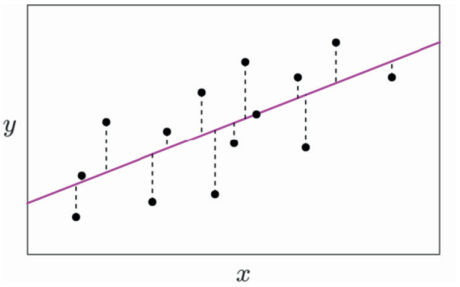

Fig. 3.3 A simulated 2-dimensional training dataset along with a line (in magenta) fit to the data using the Least Squares framework, which aims at recovering the line that minimizes the total squared length of the dashed error bars.

$$g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P} \left(b + \mathbf{x}_p^T \mathbf{w} - y_p\right)^2. \tag{3.6}$$

We of course want to find a parameter pair $(b, \mathbf{w})$ that provides a small value for $g\left(b, \mathbf{w}\right)$ since the larger this value the larger the squared error between the corresponding linear model and the data, and hence the poorer we represent the given data. Therefore we aim to *minimize* g over the bias and weight vector in order to recover the best pair $(b, \mathbf{w})$, which is written formally (see Section 2.2) as

$$\underset{b, \mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \left(b + \mathbf{x}_p^T \mathbf{w} - y_p\right)^2. \tag{3.7}$$

By checking the second order definition of convexity (see Exercise 3.3) we can easily see that the Least Squares cost function in (3.6) is convex. Figure 3.4 illustrates the Least Squares cost associated with the student loan data in Example 1.1, whose "upward bending" shape confirms its convexity visually in the instance of that particular dataset.

3.1.3 Minimization of the Least Squares cost function

Now that we have a minimization problem to solve we can employ the tools described in Chapter 2. To perform calculations it will first be convenient to use the following more compact notation:

$$\tilde{\mathbf{x}}_p = \begin{bmatrix} 1 \\ \mathbf{x}_p \end{bmatrix} \qquad \tilde{\mathbf{w}} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}. \tag{3.8}$$

With this notation we can rewrite the cost function shown in (3.6) in terms of the single vector $\tilde{\mathbf{w}}$ of parameters as

$$g\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P} \left(\tilde{\mathbf{x}}_p^T \tilde{\mathbf{w}} - y_p\right)^2. \tag{3.9}$$

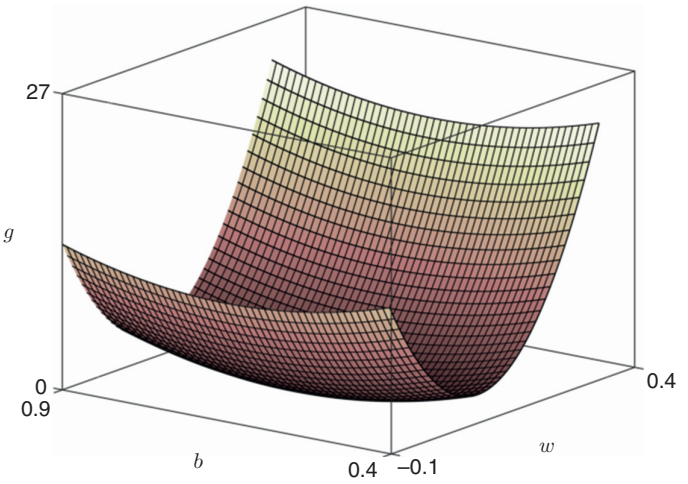

Fig. 3.4 The surface generated by the Least Squares cost function using the student loan debt data shown in Fig. 1.8, is clearly convex. However, regardless of the dataset, the Least Squares cost for linear regression is always convex.

To compute the gradient of this cost we simply apply the chain rule from calculus, which gives

$$\nabla g\left(\tilde{\mathbf{w}}\right) = 2\sum_{p=1}^{P}\tilde{\mathbf{x}}_p\left(\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}} - y_p\right) = 2\left(\sum_{p=1}^{P}\tilde{\mathbf{x}}_p\tilde{\mathbf{x}}_p^T\right)\tilde{\mathbf{w}} - 2\sum_{p=1}^{P}\tilde{\mathbf{x}}_p y_p. \tag{3.10}$$

Using this we can perform gradient descent to minimize the cost. However, in this (rare) instance we can actually solve the first order system directly in order to recover a global minimum. Setting the gradient above to zero and solving for $\tilde{\mathbf{w}}$ gives the system of linear equations

$$\left(\sum_{p=1}^{P}\tilde{\mathbf{x}}_p\tilde{\mathbf{x}}_p^T\right)\tilde{\mathbf{w}} = \sum_{p=1}^{P}\tilde{\mathbf{x}}_p y_p. \tag{3.11}$$

In particular one algebraic solution to this system,[1] if the matrix $\sum_{p=1}^{P}\tilde{\mathbf{x}}_p\tilde{\mathbf{x}}_p^T$ is invertible,[2] may be written as

[1] By setting the input vectors $\tilde{\mathbf{x}}_p$ columnwise to form the matrix $\tilde{\mathbf{X}}$ and by stacking the output y_p into the column vector $\mathbf{y}$ we may write the linear system in Equation (3.11) equivalently as $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T\tilde{\mathbf{w}} = \tilde{\mathbf{X}}\mathbf{y}$.

[2] In instances where the linear system in (3.11) has more than one solution, or in other words when $\sum_{p=1}^{P}\tilde{\mathbf{x}}_p\tilde{\mathbf{x}}_p^T$ is not invertible, one can choose the solution with the smallest length (or ℓ_2 norm), sometimes written as $\tilde{\mathbf{w}}^\star = \left(\sum_{p=1}^{P}\tilde{\mathbf{x}}_p\tilde{\mathbf{x}}_p^T\right)^\dagger \sum_{p=1}^{P}\tilde{\mathbf{x}}_p y_p$, where $(\cdot)^\dagger$ denotes the pseudo-inverse of its input matrix. See Appendix C for further details.

$$\tilde{\mathbf{w}}^{\star} = \left(\sum_{p=1}^{P} \tilde{\mathbf{x}}_p \tilde{\mathbf{x}}_p^T\right)^{-1} \sum_{p=1}^{P} \tilde{\mathbf{x}}_p y_p. \tag{3.12}$$

However, while an algebraically expressed solution is appealing it is typically more computationally efficient in practice to solve the original linear system using numerical linear algebra software.

3.1.4 The efficacy of a learned model

With optimal parameters $\tilde{\mathbf{w}}^{\star} = \begin{bmatrix} b^{\star} \\ \mathbf{w}^{\star} \end{bmatrix}$ we can compute the efficacy of the linear model in representing the training set by computing the mean squared error (or MSE),

$$\text{MSE} = \frac{1}{P}\sum_{p=1}^{P}\left(b^{\star} + \mathbf{x}_p^T\mathbf{w}^{\star} - y_p\right)^2. \tag{3.13}$$

When possible it is also a good idea to compute the MSE of a learned regression model on a set of new testing data, i.e., data that was not used to learn the model itself, to provide some assurance that the learned model will perform well on future data points. This is explored further in Chapter 5 in the context of *cross-validation*.

3.1.5 Predicting the value of new input data

With optimal parameters $(b^{\star}, \mathbf{w}^{\star})$, found by minimizing the Least Squares cost, we can predict the output y_{new} of a new input feature $\mathbf{x}_{\text{new}}$ by simply plugging the new input into the tuned linear model and estimating the associated output as

$$y_{\text{new}} = b^{\star} + \mathbf{x}_{\text{new}}^T\mathbf{w}^{\star}. \tag{3.14}$$

This is illustrated pictorially on a toy dataset for the case when $N = 1$ in Fig. 3.5.

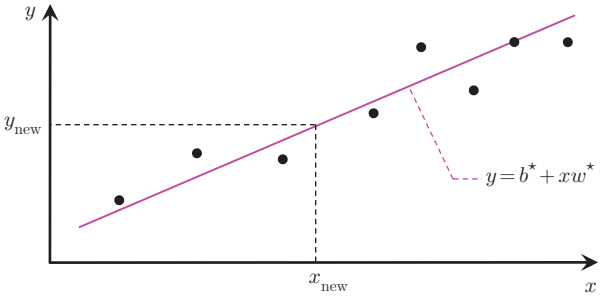

Fig. 3.5 Once a line/hyperplane has been fit to a dataset via minimizing the Least Squares cost function it may be used to predict the output value of future input. Here a line has been fit to a two-dimensional dataset in this manner, giving optimal parameters $b^{\star}$ and $w^{\star}$, and the output value of a new point x_{new} is created using the learned linear model as $y_{\text{new}} = b^{\star} + x_{\text{new}}w^{\star}$.

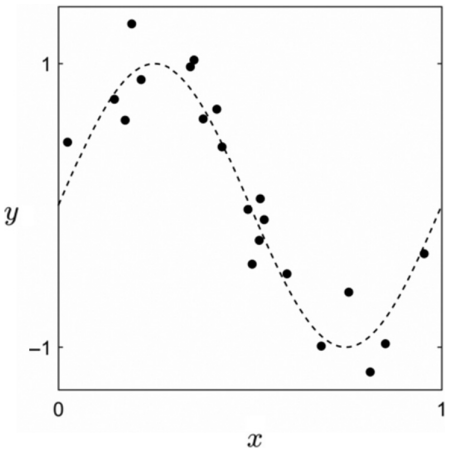

Fig. 3.6 A simulated regression dataset where the relationship between the input feature x and the output y is not linear. However, because we can visualize this dataset we can see that there is clearly a structured nonlinear relationship between its input and output. Our knowledge in this instance, based on our ability to visualize the data, allows us to design a new feature for the data and formulate a corresponding function (shown here in dashed black) that appears to be generating the data.

3.2 Knowledge-driven feature design for regression

In many regression problems the relationship between input feature(s) and output values is nonlinear, as in Fig. 3.6, which illustrates a simulated dataset where the scalar feature x and the output y are related in a nonlinear fashion. In such instances a linear model would clearly fail at representing how the input and output are related. In this brief section we present two simple examples through which we discuss how to fit a nonlinear model to the data when we have significant understanding or *knowledge* about the data itself. This knowledge may originate from our prior understanding or intuition about the phenomenon under study or simply our ability to visualize low dimensional data. As we now see, based on this knowledge we can propose an appropriate nonlinear feature transformation which allows us to employ the linear regression framework as described in the previous section.

Example 3.2 Sinusoidal pattern

In the left panel of Fig. 3.7 we show a simulated regression dataset (first shown in Fig. 3.6) consisting of $P = 21$ data points. Visually analyzing this data it appears to trace out (with some noise) one period of a sine wave over the interval $[0, 1]$. Therefore we can reasonably propose that some weighted version of the sinusoidal function $f(x) = \sin(2\pi x)$, i.e., $y = b + f(x)\,w$ where b and w are respectively a bias and weight to learn, will properly describe this data. In machine learning the function $f(x)$, in this instance a sinusoid, is referred to as a *feature transformation* of the original input. One could of

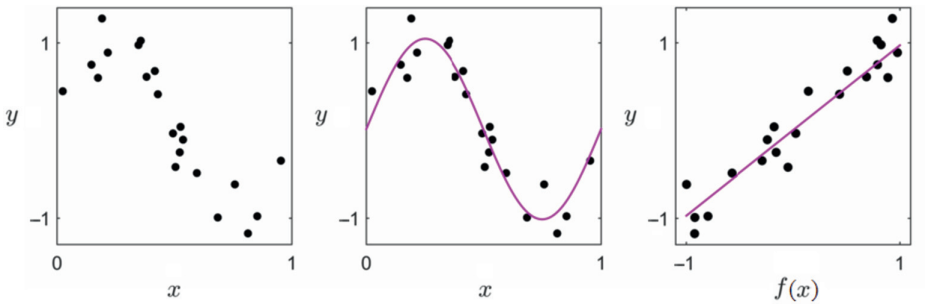

Fig. 3.7 (left panel) A simulated regression dataset. (middle panel) A weighted form of a simple sinusoid $y = b + f(x)\,w$ (in magenta), where $f(x) = \sin(2\pi x)$ and where b and w are tuned properly, describes the data quite well. (right panel) Fitting a sinusoid in the original feature space is equivalent to fitting a line in the transformed feature space where the input feature has undergone feature transformation $x \longrightarrow f(x) = \sin(2\pi x)$.

course propose a more curvy feature, but the sinusoid seems to explain the data fairly well while remaining relatively simple.

By fitting a simple weighted sinusoid to the data, we would like to find the parameter pair (b, w) so that

$$b + f\left(x_p\right) w = b + \sin\left(2\pi x_p\right) w \approx y_p, \quad p = 1, \ldots, P. \tag{3.15}$$

Note that while this is nonlinear in the input x, it is still linear in both b and w. In other words, the relationship between the output y and the new feature $f(x) = \sin(2\pi x)$ is linear. Plotting $\left\{\left(f\left(x_p\right), y_p\right)\right\}_{p=1}^{P}$ in the *transformed feature space* (i.e., the space whose input is given by the new feature $f(x)$ and whose output is still y) shown in the right panel of Fig. 3.7, we can see that the new feature and given output are now indeed linearly related.

After creating the new features for the data by transforming the input as $f_p = f\left(x_p\right) = \sin\left(2\pi x_p\right)$, we may solve for the parameter pair by minimizing the Least Squares cost function formed by summing the squared error between the model containing each transformed input $b + f_p w$ and the corresponding output value y_p (so that (3.15) holds as well as possible) as

$$\underset{b,w}{\text{minimize}} \sum_{p=1}^{P} \left(b + f_p w - y_p\right)^2. \tag{3.16}$$

The cost function here is still convex, and can be minimized by a numerical scheme like gradient descent or by directly solving its first order system to recover a global minimum. By using the compact notation

$$\tilde{\mathbf{f}}_p = \begin{bmatrix} 1 \\ f_p \end{bmatrix}, \qquad \tilde{\mathbf{w}} = \begin{bmatrix} b \\ w \end{bmatrix}, \tag{3.17}$$

we can rewrite the cost function in terms of the single vector $\tilde{\mathbf{w}}$ of parameters as

$$g\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P}\left(\tilde{\mathbf{f}}_p^T\tilde{\mathbf{w}} - y_p\right)^2. \tag{3.18}$$

Mirroring the discussion in Section 3.1.3, setting the gradient of the above to zero then gives the linear system of equations to solve

$$\left(\sum_{p=1}^{P}\tilde{\mathbf{f}}_p\tilde{\mathbf{f}}_p^T\right)\tilde{\mathbf{w}} = \sum_{p=1}^{P}\tilde{\mathbf{f}}_p y_p. \tag{3.19}$$

In the middle and right panels of Fig. 3.7 we show the resulting fit to the data $y = b^\star + f(x)\,w^\star$ in magenta, where $b^\star$ and $w^\star$ are recovered by solving this system. We refer to this fit as the *estimated data generating function* since it is our estimation of the underlying continuous function generating this dataset (shown in dashed black in Fig. 3.6). Note that this fit is a sinusoid in the original feature space (middle panel), and a line in the transformed feature space (right panel).

Example 3.3 Galileo and uniform acceleration

Recall Galileo's acceleration experiment, first described in Example 1.7. In the left panel of Fig. 3.8 we show data consisting of $P = 6$ data points from a modern reenactment of this experiment [75], where the input x denotes the time and the output y denotes the corresponding portion of the ramp traversed. Several centuries ago Galileo saw data very similar looking to the data shown here. To describe the data he saw, Galileo proposed the relation $y = f(x)\,w$, where $f(x) = x^2$ is a simple quadratic feature and w is some weight to be tuned to the data. Note that in this specific example there is no need to add

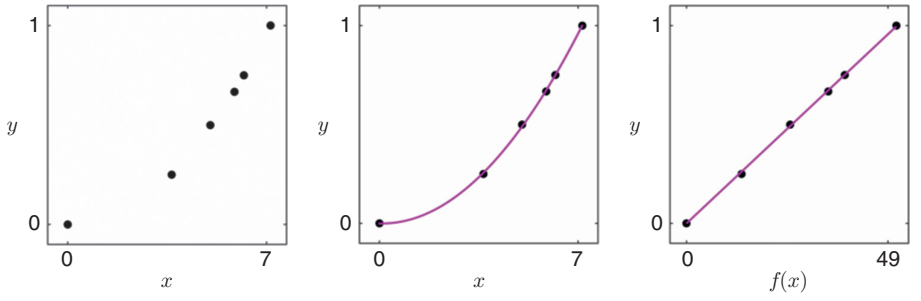

Fig. 3.8 Data from a modern reenactment of Galileo's ramp experiment. (left panel) The raw data seems to reflect a quadratic relationship between the input and output variables. (middle panel) A weighted form of a simple quadratic feature $y = f(x)\,w$ (in magenta) where $f(x) = x^2$ and where w is tuned properly, describes the data quite well. (right panel) Fitting a quadratic to the data in the original feature space is equivalent to fitting a line to the data in a transformed feature space wherein the input feature has undergone feature transformation $x \longrightarrow f(x) = x^2$.

a bias parameter b to the quadratic model since at time zero the ball has not moved at all, and hence the output must be precisely zero.

Looking at how the data is distributed in the left panel of Fig. 3.8, we too can intuit that such a quadratic feature of the input appears to be a reasonable guess at what lies beneath the data shown.

By fitting a simple weighted quadratic to the data, we would like to find parameter w such that

$$f\left(x_p\right) w = x_p^2 w \approx y_p, \quad p = 1, \ldots, P. \tag{3.20}$$

Although the relationship between the input feature x and output y is nonlinear, this model is still linear in the weight w. Thus we may tune w by minimizing the Least Squares cost function after forming the new features. That is, transforming each input as $f_p = f\left(x_p\right) = x_p^2$ we can find the optimal w by solving

$$\underset{w}{\text{minimize}} \sum_{p=1}^{P} \left(f_p w - y_p\right)^2. \tag{3.21}$$

The cost function in (3.21) is again convex and we may solve for the optimal w by simply checking the first order condition. Setting the derivative of the cost function equal to zero and solving for w, after a small amount of algebraic rearrangement, gives

$$w^\star = \frac{\sum_{p=1}^{P} f_p y_p}{\sum_{p=1}^{P} f_p^2}. \tag{3.22}$$

We show in the middle panel of Fig. 3.8 the weighted quadratic fit $y = f(x) w^\star$ (our estimated data generating function) to the data (in magenta) in the original feature space. In the right panel of this figure we show the same fit, this time in the transformed feature space where the fit is linear.

3.2.1 General conclusions

The two examples discussed above are very special. In each case, by using our ability to visualize the data we have been able to design an excellent new feature $f(x)$ explicitly using common algebraic functions. As we have seen in these two examples, a properly designed feature (or set of features more generally) for linear regression is one that provides a good *nonlinear* fit in the original space while, simultaneously, a good *linear* fit in the transformed feature space. In other words, a properly designed set of features for linear regression produces a good linear fit to the feature-transformed data.[3]

[3] Technically speaking there is one subtle yet important caveat to the use of the word "good" in this statement, this being that we do not want to "overfit" the data (an issue we discuss at length in Chapter 5). However, for now this issue will not concern us.

A properly designed feature (or set of features) for linear regression provides a good *nonlinear* fit in the original feature space and, simultaneously, a good *linear* fit in the transformed feature space.

However, just because we can visualize a low dimensional regression dataset does not mean we can easily design a proper feature "by eye" as we have done in Examples 3.2 and 3.3. For instance, in Fig. 3.9 we show a simulated dataset built by randomly taking $P = 30$ inputs x_p on the interval $[0, 1]$, evaluating each through a rather wild function[4] $y(x)$ (shown in dashed black in the figure), and then adding a small amount of noise to each output. Here even though we can clearly see a structured nonlinear relationship in the data, it is not immediately obvious how to formulate a proper feature $f(x)$ to recover the original data generating function $y(x)$. No common algebraic function (e.g., a quadratic, a sine wave, an exponential, etc.,) seems to be a reasonable candidate and hence our knowledge, in this case the fact that we can visualize the data itself, is not enough to form a proper feature (or set of features) for this data.

For vector-valued input we can say something very similar. While we can imagine forming a proper set of features for a dataset with vector-valued input, the fact that we cannot visualize the data prohibits us from "seeing" the right sort of feature(s) to use.

In fact rarely in practice can we use our knowledge of a dataset to construct perfect features. Often we may only be able to make a rough guess at a proper feature transformation given our intuition about the data at hand, or can make no educated guess at all. Thankfully, we can *learn* feature transformations automatically from the data itself that can ameliorate this problem. This process will be described in Chapter 5.

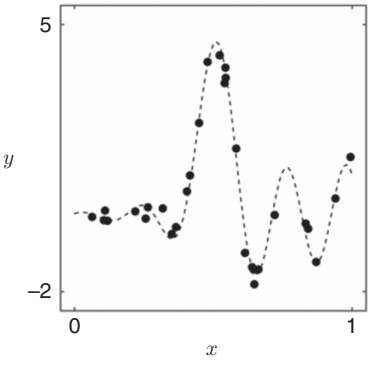

Fig. 3.9 A simulated dataset generated as noisy samples of a data generating function $y(x)$. We show $y(x)$ here in dashed black. Unlike the previous two cases in Fig. 3.7 and 3.8, it is not so clear what sort of function would serve as a proper feature $f(x)$ here.

[4] Here $y(x) = e^{3x} \frac{\sin\left(3\pi^2(x-0.5)\right)}{3\pi^2(x-0.5)}$.

3.3 Nonlinear regression and ℓ_2 regularization

In the previous section we discussed examples of regression where the nonlinear relationship in a given dataset could be determined by intuiting a nonlinear feature transformation, and where (once the data is transformed) the associated cost functions remained convex and linear in their parameters. Because the input/output relationship associated to each of these examples was linear in its parameters (see (3.15) and (3.20)), each is still referred to as a *linear* regression problem. In this section we explore the consequences of employing nonlinear models for regression where the corresponding cost function is non-convex and the input/output relationship nonlinear in its parameters (referred to as instances of *nonlinear* regression). We also introduce a common tool for partially ameliorating the practical inconveniences of non-convex cost functions, referred to as the ℓ_2 regularizer. Specifically we describe how the ℓ_2 regularizer helps numerical optimization techniques avoid poor stationary points of non-convex cost functions. Because of this utility, regularization is often used with non-convex cost functions, as we will see later with e.g., neural networks in Chapters 5–7.[5]

While the themes of this section are broadly applicable, for the purpose of clarity we frame our discussion of nonlinear regression and ℓ_2 regularization around a single classic example referred to as *logistic regression* (which we will also see arise in the context of classification in the next chapter). Further examples of nonlinear regression are explored in the chapter exercises.

3.3.1 Logistic regression

At the heart of the classic logistic regression problem is the so-called *logistic sigmoid function*, illustrated in the left panel of Fig. 3.10, and defined mathematically as

$$\sigma\left(t\right) = \frac{1}{1 + e^{-t}}, \tag{3.23}$$

where t can take on any real value. Invented in the early 19th century by mathematician Pierre François Verhulst [79], this function was designed in his pursuit of modeling how a population (of microbes, animal species, etc.,) grows over time, taking into account the realistic assumption that regardless of the kind of organism under study, the system in which it lives has only a finite amount of resources.[6] Thus, as a result, there should be a strict cap on the total population in any biological system. According to Verhulst's model, the initial stages of growth should follow an exponential trend until a saturation level where, due to lack of required resources (e.g., space, food, etc.), the growth stabilizes and levels off.[7]

[5] Additionally, ℓ_2 regularization also arises in the context of the (convex) support vector machine classifier, as we will see in Section 4.3. Another popular use of ℓ_2 regularization is discussed later in Section 7.3 in the context of 'cross-validation'.

[6] Beyond its classical use in modeling population growth, we will see in the next chapter that logistic regression can also be used for the task of classification.

[7] Like any good mathematician, Verhulst first phrased this ideal population growth model in terms of a differential equation. Denoting the desired function f and the maximum population level of the system as

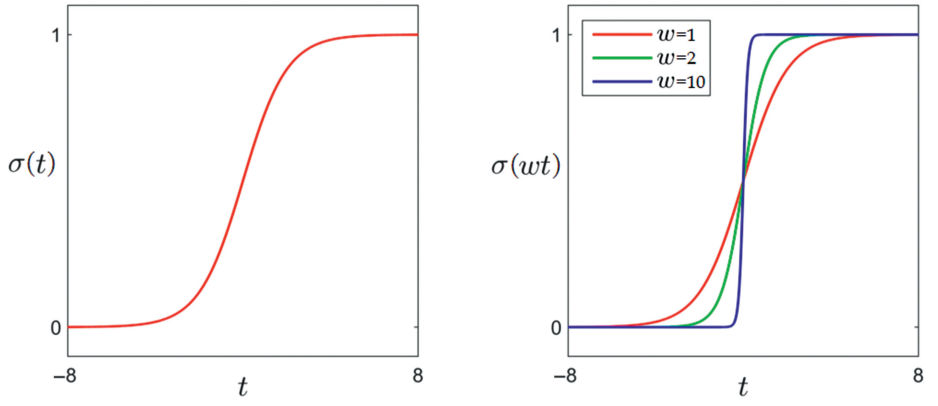

Fig. 3.10 (left panel) Plot of the logistic sigmoid function defined in (3.23). Note that the output of this function is always between 0 and 1. (right panel) By increasing the weight w of the sigmoid function $\sigma\,(wt)$ from $w = 1$ (red) to $w = 2$ (green) and finally to $w = 10$ (blue), the sigmoid becomes an increasingly good approximator of a "step function," that is a function that only takes on the values 0 and 1 with a sharp transition between the two.

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

$$\sigma\left(b + x_p w\right) \approx y_p, \quad p = 1, \ldots, P, \tag{3.24}$$

where b and w are parameters which must be properly tuned. The weight w, as illustrated in the right panel of Fig. 3.10, controls how quickly the system saturates, and the bias term b shifts the curve left and right along the horizontal axis. Likewise when the input is N-dimensional the system of equations given in (3.24) may be written analogously as

$$\sigma\left(b + \mathbf{x}_p^T \mathbf{w}\right) \approx y_p, \quad p = 1, \ldots, P, \tag{3.25}$$

where as usual $\mathbf{x}_p = \begin{bmatrix} x_{1,p} & x_{2,p} & \ldots & x_{N,p} \end{bmatrix}^T$ and $\mathbf{w} = \begin{bmatrix} w_1 & w_2 & \ldots & w_N \end{bmatrix}^T$. Note that unlike the analogous set of equations with linear regression given in Equation (3.5), each of these equations is nonlinear[8] in b and $\mathbf{w}$. These nonlinearities lead to a non-convex Least Squares cost function which is formed by summing the squared differences of Equation (3.25) over all p,

$$g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P} \left(\sigma\left(b + \mathbf{x}_p^T \mathbf{w}\right) - y_p\right)^2. \tag{3.26}$$

1, he supposed that the population growth rate $\frac{df}{dt}$ should, at any time t, be proportional to both the current population level f as well as the remaining capacity left in the system $1 - f$. Together this gives the differential equation $\frac{df}{dt} = f\,(1 - f)$. One can check by substitution that the logistic sigmoid function satisfies this relationship with initial condition $f\,(0) = 1/2$.

[8] In certain circumstances this system may be transformed into one that is linear in its parameters. See Exercise 3.10 for further details.

Using the compact notation $\tilde{\mathbf{x}}_p = \begin{bmatrix} 1 \\ \mathbf{x}_p \end{bmatrix}$ and $\tilde{\mathbf{w}} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}$, the fact that the derivative of the sigmoid is given as $\sigma'(t) = \sigma(t)(1 - \sigma(t))$ (see footnote 7), and the chain rule from calculus, the gradient of this cost can be calculated as

$$\nabla g(\tilde{\mathbf{w}}) = 2\sum_{p=1}^{P} \left(\sigma\left(\tilde{\mathbf{x}}_p^T \tilde{\mathbf{w}}\right) - y_p\right) \sigma\left(\tilde{\mathbf{x}}_p^T \tilde{\mathbf{w}}\right) \left(1 - \sigma\left(\tilde{\mathbf{x}}_p^T \tilde{\mathbf{w}}\right)\right) \tilde{\mathbf{x}}_p. \tag{3.27}$$

Due to the many nonlinearities involved in the above system of equations, solving the first order system directly is a fruitless venture, instead a numerical technique (i.e., gradient descent or Newton's method) must be used to find a useful minimum of the associated cost function.

Example 3.4 Bacterial growth

In the left panel of Fig. 3.11 we show a real dataset consisting of $P = 9$ data points corresponding to the normalized cell concentration[9] of a particular bacteria, *Lactobacillus delbrueckii*, in spatially constrained laboratory conditions over the period of 24 hours. Also shown in this panel are two sigmoidal fits (shown in magenta and green) found via minimizing the Least Squares cost in (3.26) using gradient descent. In the middle panel we show the surface of the cost function which is clearly non-convex, having stationary points in the large flat region colored orange as well as a global minimum in the long narrow valley highlighted in dark blue. Two paths taken by initializing gradient descent at different values are shown in magenta and green, respectively, on the surface itself. While the initialization of the magenta path in the yellow-green area of the surface

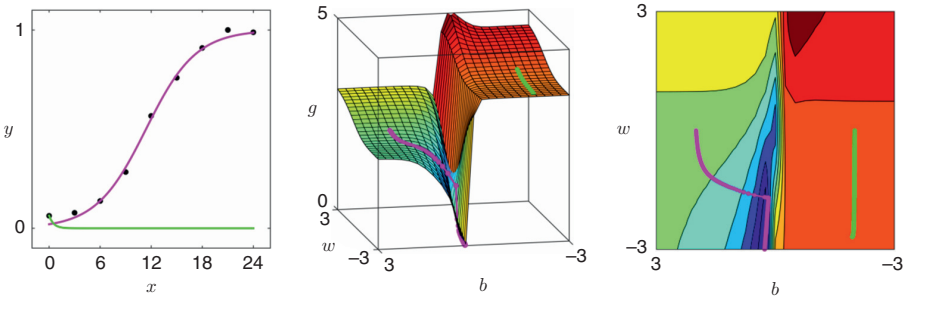

Fig. 3.11 (left panel) A dataset along with two sigmoidal fits (shown in magenta and green), each found via minimizing the Least Squares cost in (3.26) using gradient descent with a different initialization. A surface (middle) and contour (right) plot of this cost function, along with the paths taken by the two runs of gradient descent. Each path has been colored to match the resulting sigmoidal fit produced in the left panel (see text for further details). Data in this figure is taken from [48].

[9] Cell concentration is measured as the mass of organism per unit volume. Here we have normalized the cell concentration values so that they lie strictly in the interval $(0, 1)$.

leads to the global minimum, which corresponds with the good sigmoidal fit in magenta shown in the left panel, the initialization of the green path in the large flat orange region leads to a poor solution, with corresponding poor fit shown in green in the left panel. In the right panel we show the contour plot of the same surface (along with the two gradient descent paths) that more clearly shows the long narrow valley containing the desired global minimum of the surface.

3.3.2 Non-convex cost functions and ℓ_2 regularization

The problematic flat areas posed by non-convex cost functions like the one shown in Fig. 3.11 can be ameliorated by the addition of a *regularizer*. A regularizer is a simple convex function that is often added to such a cost function, slightly convexifying it and thereby helping numerical optimization techniques avoid poor solutions in its flat areas. One of the most common regularizers used in practice is the squared ℓ_2 norm of the weights $\|\mathbf{w}\|_2^2 = \sum_{n=1}^{N} w_n^2$, referred to as the ℓ_2 regularizer. To regularize a cost function $g(b, \mathbf{w})$ with this regularizer we simply add it to g giving the regularized cost function

$$g(b, \mathbf{w}) + \lambda \|\mathbf{w}\|_2^2 . \tag{3.28}$$

Here $\lambda \geq 0$ is a parameter (set by the user in practice) that controls the strength of each term, the original cost function and the regularizer, in the final sum. For example, if $\lambda = 0$ we have our original cost. On the other hand, if λ is set very large then the regularizer drowns out the cost and we have $g(b, \mathbf{w}) + \lambda \|\mathbf{w}\|_2^2 \approx \lambda \|\mathbf{w}\|_2^2$. Typically in practice λ is set fairly small (e.g., $\lambda = 0.1$ or smaller).

In Fig. 3.12 we show two simple examples of non-convex cost functions which exemplify how the ℓ_2 regularizer can help numerical techniques avoid many (but not all) poor solutions in practice.

The first non-convex cost function,[10] shown in the top left panel of Fig. 3.12, is defined over a symmetric interval about the origin and has three large flat areas containing undesirable stationary points. This kind of non-convex function is highly problematic because if an algorithm like gradient descent or Newton's method is initialized at any point lying in these flat regions it will immediately halt. In the top right panel we show an ℓ_2 regularized version of the same cost. Note how regularization slightly convexifies the entire cost function, and in particular how it forces the flat regions to curve upwards. Now if e.g., gradient descent is initialized in either of the two flat regions on the left or right sides it will in fact travel downwards and reach a minimum. Note that both minima have slightly changed position from the original cost, but as long as λ is set relatively small this small change does not typically make a difference in practice. Note in this instance, however, that regularization has not helped with the

[10] Here the cost is defined as $g(w) = \max^2\left(0,\, e^{-w}\sin\left(4\pi\,(w - 0.1)\right)\right)$, and λ has been set fairly high at $\lambda = 1$ for illustrative purposes only.

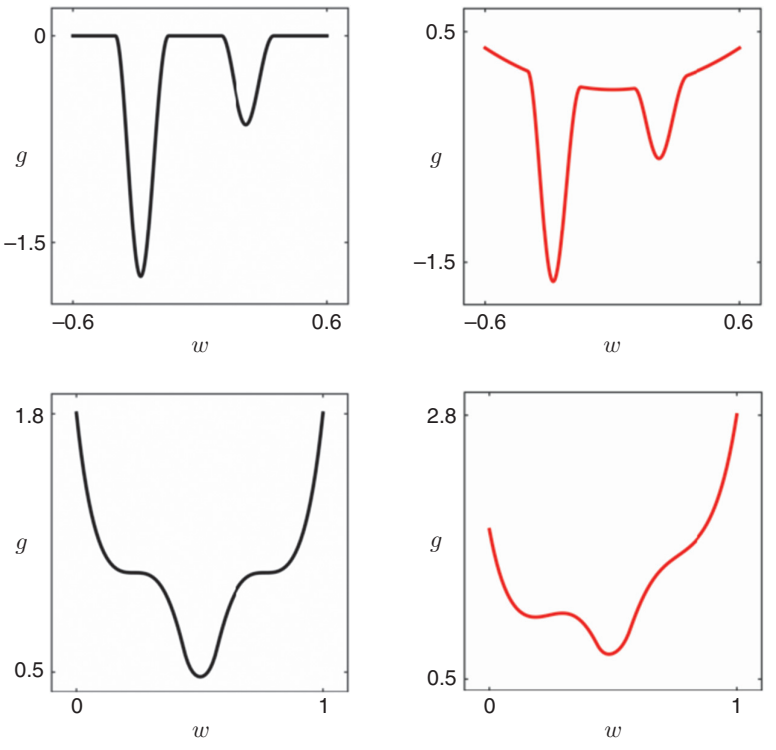

Fig. 3.12 Two examples of non-convex functions with flat regions (top left panel) and saddle points (bottom left panel) where numerical optimization methods can halt undesirably. Using the ℓ_2 regularizer we can slightly convexify each (right panels), which can help avoid some of these undesirable solutions. See text for further details.

problem of gradient descent halting at a poor solution if initialized in the middle flat region of the original cost. That is, by regularizing we have actually created a local minimum near the middle of the original flat region in the regularized cost function, and so if gradient descent is initialized in this region it will halt at this undesirable solution.

The second non-convex cost function,[11] shown in the bottom left panel of Fig. 3.12, is defined over the unit interval and has two saddle points at which the derivative is zero and so at which e.g., gradient descent, will halt undesirably if initialized at a point corresponding to any region on the far left or right. In the lower right panel we show the ℓ_2 regularized cost which no longer has an issue with the saddle point on the right, as the region surrounding it has been curved upwards. However the saddle point on the left is still problematic, as regularizing the original cost has created a local minimum near the point that will cause gradient descent to continue to halt at an undesirable solution.

[11] Here the cost is defined as $g(w) = \max^2\left(0,\ (3w - 2.3)^3 + 1\right) + \max^2\left(0,\ (-3w + 0.7)^3 + 1\right)$, and λ has been set fairly high at $\lambda = 1$ for illustrative purposes only.

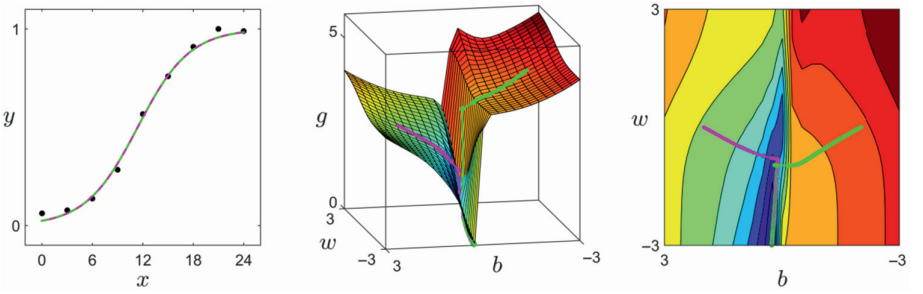

Fig. 3.13 A regularized version of Fig. 3.11. (left panel) Plot of the bacterial growth dataset along with two overlapping sigmoidal fits (shown in magenta and green) found via minimizing the ℓ_2 regularized Least Squares cost for logistic regression in (3.29) using gradient descent. (middle and right panels) The surface and contour plot of the regularized cost function along with the paths (in magenta and green) of gradient descent with same two initializations as shown in Fig. 3.11. While the surface is still non-convex, the large flat region that originally led the initialization of the green path to a poor solution with the unregularized cost has been curved upwards by the regularizer, allowing the green run of gradient descent to reach the global minimum of the problem. Data in this figure is taken from [48].

Example 3.5 ℓ_2 **regularized Least Squares for logistic regression**

We saw in Fig. 3.11 that the initialization of gradient descent in the flat orange region resulted in a poor fit to the bacterial growth data. A second version of all three panels from this figure is duplicated in Fig. 3.13, only here we add the ℓ_2 regularizer with $\lambda = 0.1$ to the original Least Squares logistic cost in (3.26). Formally, this ℓ_2 regularized Least Squares cost function is written as

$$g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P} \left(\sigma\left(b + \mathbf{x}_p^T \mathbf{w}\right) - y_p\right)^2 + \lambda \left\|\mathbf{w}\right\|_2^2. \tag{3.29}$$

Once again in order to minimize this cost we can employ gradient descent (see Exercise 3.13). Comparing the regularized surface in Fig. 3.13 to the original shown in Fig. 3.11 we can see that regularizing the original cost curves the flat regions of the surface upwards, helping gradient descent avoid poor solutions when initialized in these areas. Now both initializations first shown in Fig. 3.11 lead gradient descent to the global minimum of the surface.

3.4 Summary

Linear regression is a fundamental predictive learning problem which aims at determining the relationship between continuous-valued input and output data via the fitting of an appropriate model that is linear in its parameters. In this chapter we first saw how

to fit a linear model (i.e., a line or hyperplane in higher dimensions) to a given dataset, culminating in the minimization of the Least Squares cost function at the end of Section 3.1. Due to the parameters being linearly related, this cost function may be minimized by solving the associated first order system.

We then saw in Section 3.2 how in some rare instances our understanding of a phenomenon, typically due to our ability to visualize a low dimensional dataset, can permit us to accurately suggest an appropriate feature transformation to describe our data. This provides a proper nonlinear fit to the original data while simultaneously fitting linearly to the data in an associated transformed feature space.

Finally, using the classical example of logistic regression, we saw in Section 3.3 how a nonlinear regression model typically involves the need to minimize an associated non-convex cost function. We then saw how ℓ_2 regularization is used as a way of "convexifying" a non-convex cost function to help gradient descent avoid some undesirable stationary points of such a function.

3.5 Exercises

Section 3.1 exercises

Exercises 3.1 Fitting a regression line to the student debt data

Fit a linear model to the U.S. student loan debt dataset shown in Fig. 1.8, called *student_debt_data.csv*, by solving the associated linear regression Least Squares problem. If this linear trend continues what will the total student debt be in 2050?

Exercises 3.2 Kleiber's law and linear regression

After collecting and plotting a considerable amount of data comparing the body mass versus metabolic rate (a measure of at rest energy expenditure) of a variety of animals, early 20th century biologist Max Kleiber noted an interesting relationship between the two values. Denoting by x_p and y_p the body mass (in kg) and metabolic rate (in kJ/day) of a given animal respectively, treating the body mass as the input feature Kleiber noted (by visual inspection) that the natural logs of these two values were linearly related. That is,

$$w_0 + \log\left(x_p\right) w_1 \approx \log\left(y_p\right). \tag{3.30}$$

In Fig. 3.14 we show a large collection of transformed data points $\left\{\left(\log\left(x_p\right), \log\left(y_p\right)\right)\right\}_{p=1}^{P}$, each representing an animal ranging from a small black-chinned hummingbird in the bottom left corner to a large walrus in the top right corner.

a) Fit a linear model to the data shown in Fig. 3.14 (called *kleibers_law_data.csv*). Make sure to take the log of both arguments!

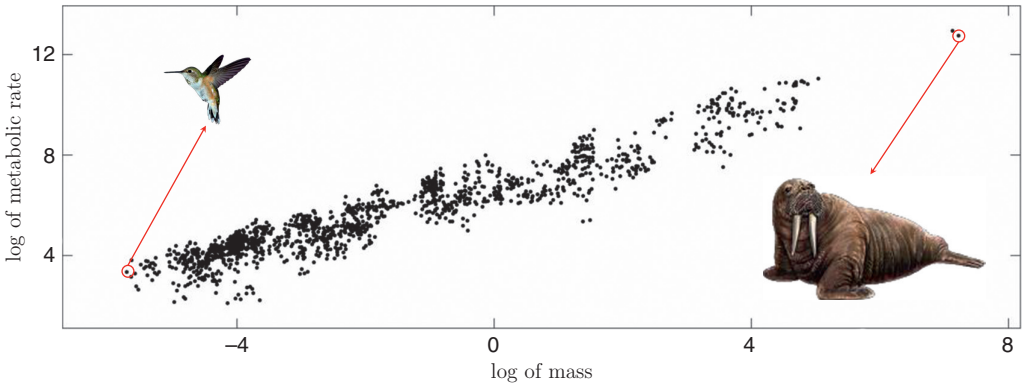

Fig. 3.14 A large set of body mass/metabolic rate data points, transformed by taking the log of each value, for various animals over a wide range of different masses.

b) Use the optimal parameters you found in part (a) along with the properties of the log function to write the nonlinear relationship between the body mass x and the metabolic rate y.

c) Use your fitted line to determine how many calories an animal weighing 10 kg requires (note each calorie is equivalent to 4.18 J).

Exercises 3.3 The Least Squares cost for linear regression is convex

Show that the Least Squares cost function for linear regression written compactly as in Section 3.1.3,

$$g\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P}\left(\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}} - y_p\right)^2, \tag{3.31}$$

is a convex quadratic function by completing the following steps.

a) Show that $g\left(\tilde{\mathbf{w}}\right)$ can be written as a quadratic function of the form

$$g\left(\tilde{\mathbf{w}}\right) = \frac{1}{2}\tilde{\mathbf{w}}^T\mathbf{Q}\tilde{\mathbf{w}} + \mathbf{r}^T\tilde{\mathbf{w}} + d \tag{3.32}$$

by determining proper $\mathbf{Q}$, $\mathbf{r}$, and d.

b) Show that $\mathbf{Q}$ has all nonnegative eigenvalues (*hint: see Exercise 2.10*).

c) Verify that $\nabla^2 g\left(\tilde{\mathbf{w}}\right) = \mathbf{Q}$ and so that g satisfies the second order definition of convexity, and is therefore convex.

d) Show that applying a single Newton step (see Section 2.2.4) to minimize the Least Squares cost function leads to precisely the first order system of linear equations discussed in Section 3.1.3, i.e., to the system $\left(\sum_{p=1}^{P}\tilde{\mathbf{x}}_p\tilde{\mathbf{x}}_p^T\right)\tilde{\mathbf{w}} = \sum_{p=1}^{P}\tilde{\mathbf{x}}_p y_p$. (This is because the Least Squares cost is a quadratic function, as in Example 2.7.)

Section 3.2 exercises

Exercises 3.4 Reproduce Galileo's example

Use the value for the optimal weight shown in Equation (3.22) to reproduce the fits shown in Fig. 3.8. The data shown in this figure is located in the file *Galileo_data.csv*.

Exercises 3.5 Reproduce the sinusoidal example

a) Set up the first order system associated with the Least Squares cost function being minimized in Equation (3.16).

b) Reproduce the sinusoidal and associated linear fit shown in the middle and right panels of Fig. 3.7 by solving for the proper weights via the first order system you determined in part **a)**. The dataset shown in this figure is called *sinusoid_example_data.csv*.

Exercises 3.6 Galileo's extended ramp experiment

In this exercise we modify Galileo's ramp experiment, discussed in Example 3.3, to explore the relationship between the angle x of the ramp and the distance y that the ball travels during a certain fixed amount of time. In Fig. 3.15 we plot six simulated measurements corresponding to six different angle values x (measured in degrees).

a) Propose a suitable nonlinear feature transformation for this dataset such that the relationship between the new feature you form and the distance traveled is linear in its weights. *Hint: there is no need for a bias parameter b here.*

b) Formulate and minimize a Least Squares cost function using your new feature for a proper weight w, the data (located in the file *another_ramp_experiment.csv*). Plot the resulting fit in the data space.

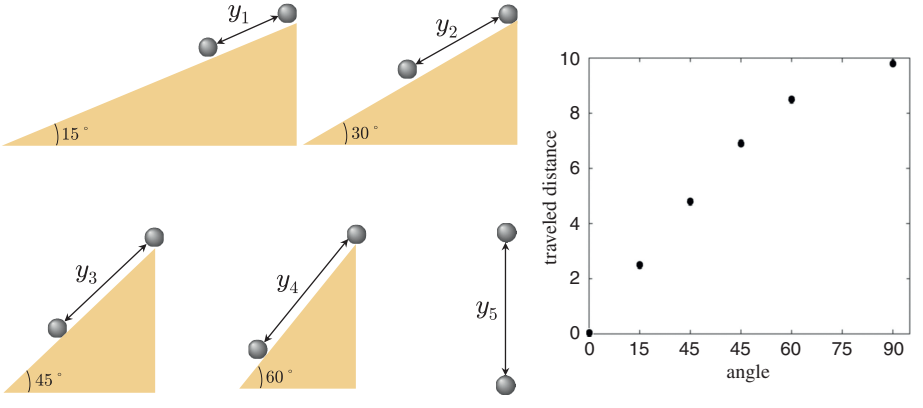

Fig. 3.15 An extended set of data from Galileo's ramp experiment, first described in Example 3.3. See text for details.

Exercises 3.7 Moore's law and the power of future computers

Gordon Moore, co-founder of Intel corporation, predicted in a 1965 paper[12] that the number of transistors on an integrated circuit would double approximately every two years. This conjecture, referred to nowadays as Moore's law, has proven to be sufficiently accurate over the past five decades. Since the processing power of computers is directly related to the number of transistors in their CPUs, Moore's law provides a trend model to predict the computing power of future microprocessors. Figure 3.16 plots the transistor counts of several microprocessors versus the year they were released, starting from Intel 4004 in 1971 with only 2300 transistors, to Intel's Xeon E7 introduced in 2014 with more than 4.3 billion transistors.

a) Propose an exponential-based transformation of the Moore's law dataset shown in Fig. 3.16 so that the transformed input/output data is related linearly. *Hint: to produce a linear relationship you will end up having to transform the output, not the input*.

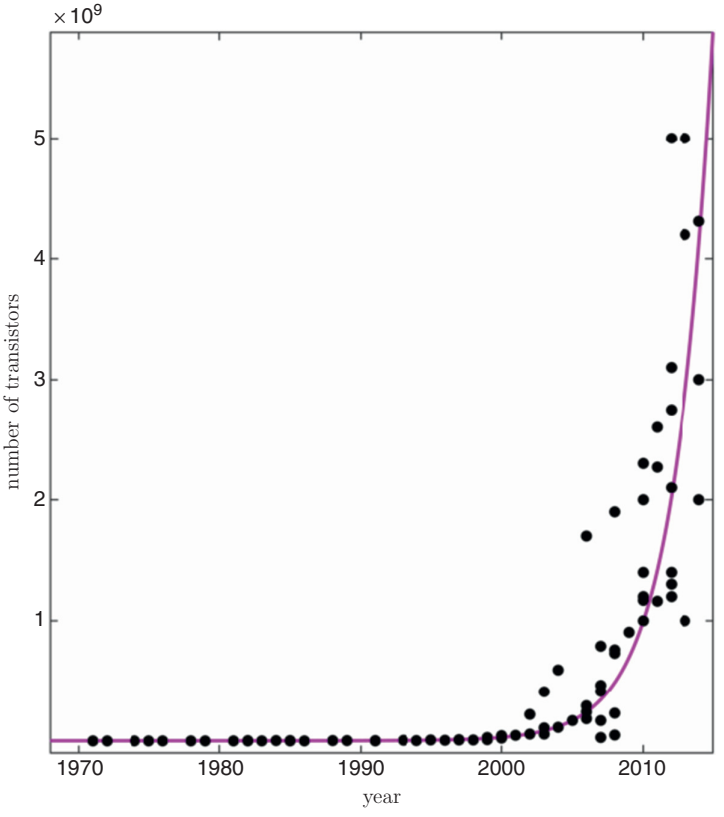

Fig. 3.16 As Moore proposed 50 years ago, the number of transistors in microprocessors versus the year they were invented follows an exponential pattern.

[12] One can find a modern reprinting of this paper in e.g., [57].

b) Formulate and minimize a Least Squares cost function for appropriate weights, and fit your model to the data in the original data space. The data shown here is located in the file *transistor_counts.csv*.

Exercises 3.8 Ohm's law and linear regression

Ohm's law, proposed by the German physicist Georg Simon Ohm following a series of experiments made by him in the 1820s, connects the magnitude of the current in a galvanic circuit to the sum of all the exciting forces in the circuit, as well as the length of the circuit. Although he did not publish any account of his experimental results, it is easy to verify his law using a simple experimental setup, shown in the left panel of Fig. 3.17, that is very similar to what he then utilized (the data in this figure is taken from [56]). The spirit lamp heats up the circuit, generating an electromotive force which creates a current in the coil deflecting the needle of the compass. The tangent of the deflection angle is directly proportional to the magnitude of the current passing through the circuit. The magnitude of this current, denoted by I, varies depending on the length of the wire used to close the circuit (dashed curve). In the right panel of Fig. 3.17 we plot the readings of the current I (in terms of the tangent of the deflection angle) when the circuit is closed with a wire of length x (in cm), for five different values of x.

a) Suggest a suitable nonlinear transformation of the original data to fit (located in the file *ohms_data.csv*) so that the transformed input/output data is related linearly. *Hint: to produce a linear relationship you will likely end up having to transform the output.*

b) Formulate a proper Least Squares cost function using your transformed data and minimize it to recover ideal parameters for your model.

c) Fit your proposed model to the data and display it in the original data space.

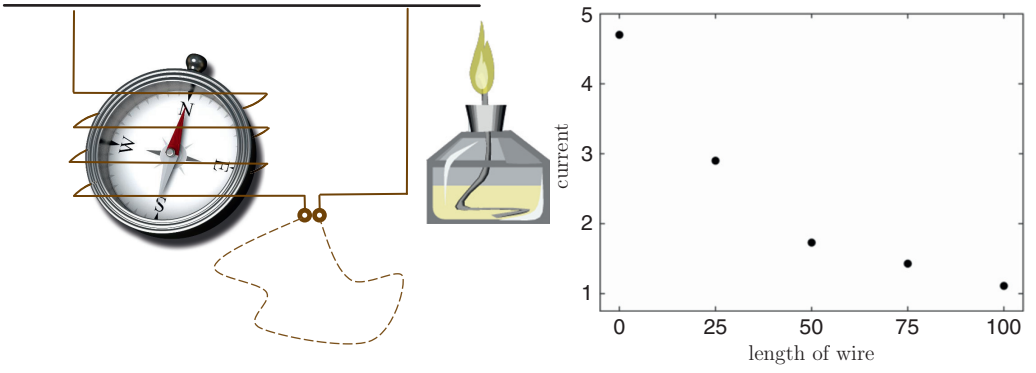

Fig. 3.17 (left panel) Experimental setup for verification of Ohm's law. Black and brown wires are made up of constantan and copper, respectively. (right panel) Current measurements for five different lengths of closing wire.

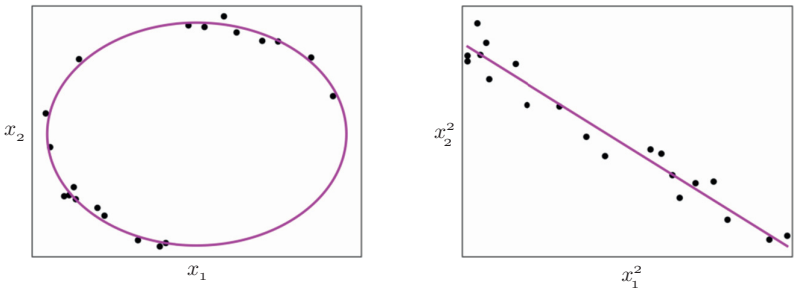

Fig. 3.18 Simulated observation data for the location of the asteroid Pallas on its orbital plane. The ellipsoidal curve fit to the data approximates the true orbit of Pallas. (right panel) Fitting an ellipsoid to the data in the original data space is equivalent to fitting a line to the data in a new space where both dimensions are squared.

Exercises 3.9 Determining the orbit of celestial bodies

One of the first recorded uses of regression via the Least Squares approach was made by Carl Frederich Gauss, a German mathematician, physicist, and all round polymath, who was interested in calculating the orbit of the asteroid Pallas by leveraging a dataset of recorded observations. Although Gauss solved the problem using ascension and declination data observed from the earth (see [61] and references therein), here we modify the problem so that the simulated data shown in the left panel of Fig. 3.18 simulates Cartesian coordinates of the location of the asteroid on its orbital plane. With this assumption, and according to Kepler's laws of planetary motion, we need to fit an ellipse to a series of observation points in order to recover the true orbit.

In this instance the data comes in the form of $P = 20$ noisy coordinates $\left\{\left(x_{1,p},\, x_{2,p}\right)\right\}_{p=1}^{P}$ taken from an ellipsoid with the standard form of

$$\left(\frac{x_{1,p}}{v_1}\right)^2 + \left(\frac{x_{2,p}}{v_2}\right)^2 \approx 1 \quad \text{for all } p = 1 \ldots P, \tag{3.33}$$

where v_1 and v_2 are tunable parameters. By making the substitutions $w_1 = \left(\frac{1}{v_1}\right)^2$ and $w_2 = \left(\frac{1}{v_2}\right)^2$ this can be phrased equivalently as a set of approximate linear equations

$$x_{1,p}^2 w_1 + x_{2,p}^2 w_2 \approx 1 \quad \text{for all } p = 1 \ldots P. \tag{3.34}$$

a) Reformulate the equations shown above using vector notation as

$$\mathbf{f}_p^T \mathbf{w} \approx y_p \quad \text{for all } p = 1 \ldots P \tag{3.35}$$

by determining the appropriate $\mathbf{f}_p$ and y_p where $\mathbf{w} = \begin{bmatrix} w_1 & w_2 \end{bmatrix}^T$.

b) Formulate and solve the associated Least Squares cost function to recover the proper weights **w** and plot the ellipse with the data shown in the left panel of Fig. 3.18 located in the file *asteroid_data.csv*.

Section 3.3 exercises

Exercises 3.10 Logistic regression as a linear system

In this exercise you will explore particular circumstances that allow one to transform the nonlinear system of equations in (3.24) into a system which is linear in the parameters b and w. In order to do this recall that a function f has an *inverse* at t if another function f^{-1} exists such that $f^{-1}(f(t)) = t$. For example, the exponential function $f(t) = e^t$ has the inverse $f^{-1}(t) = \log(t)$ for every t since we always have $f^{-1}(f(t)) = \log\left(e^t\right) = t$.

a) Show that the logistic sigmoid has an inverse for each t where $0 < t < 1$ of the form $\sigma^{-1}(t) = \log\left(\frac{t}{1-t}\right)$ and check that indeed $\sigma^{-1}(\sigma(t)) = t$ for all such values of t.

b) Suppose for a given dataset $\left\{\left(x_p, y_p\right)\right\}_{p=1}^{P}$ that $0 < y_p < 1$ for all p. Apply the sigmoid inverse to the system shown in Equation (3.24) to derive the equivalent set of linear equations

$$b + x_p w \approx \log\left(\frac{y_p}{1 - y_p}\right) \quad p = 1, \ldots, P. \tag{3.36}$$

Since the equations in (3.36) are now linear in both b and w we may solve for these parameters by simply checking the first order condition for optimality.

c) Using the dataset *bacteria_data.csv* solve the Least Squares cost function based on the linear system of equations from part **b)** and plot the data, along with the logistic sigmoid fit to the data as shown in Fig. 3.19.

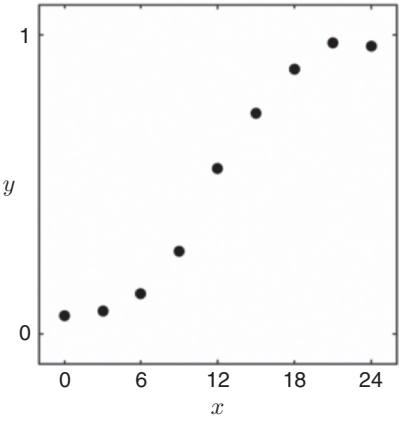

Fig. 3.19 The normalized cell concentration of Lactobacillus delbrueckii in a constrained laboratory environment over the period of 24 hours. Data in this figure is taken from [48].

Exercises 3.11 Code up gradient descent for logistic regression

In this exercise you will reproduce the gradient descent paths shown in Fig. 3.11.

a) Verify that the gradient descent step shown in Equation (3.27) is correct.

Note that this gradient can be written more compactly by denoting $\sigma_p^{k-1} = \sigma\left(\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}^{k-1}\right)$, $r_p^{k-1} = 2\left(\sigma_p^{k-1} - y_p\right)\sigma_p^{k-1}\left(1-\sigma_p^{k-1}\right)$ for all $p = 1, ..., P$, and $\mathbf{r}^{k-1} = \begin{bmatrix} r_1^{k-1} & r_2^{k-1} & \cdots & r_P^{k-1} \end{bmatrix}^T$, and stacking the $\tilde{\mathbf{x}}_p$ column-wise into the matrix $\widetilde{\mathbf{X}}$. Then the gradient can be written as $\nabla g\left(\tilde{\mathbf{w}}^{k-1}\right) = \widetilde{\mathbf{X}}\mathbf{r}^{k-1}$. For programming languages like Python and MATLAB/OCTAVE that have especially efficient implementations of matrix/vector operations this can be much more efficient than explicitly summing over the P points as in Equation (3.27).

b) The surface in this figure was generated via the wrapper *nonconvex_logistic_growth* with the dataset *bacteria_data.csv*, and inside the wrapper you must complete a short gradient descent function to produce the descent paths called

$$[\text{in, out}] = \text{grad\_descent}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \tilde{\mathbf{w}}^0\right), \tag{3.37}$$

where "in" and "out" contain the gradient steps $\tilde{\mathbf{w}}^k = \tilde{\mathbf{w}}^{k-1} - \alpha_k \nabla g\left(\tilde{\mathbf{w}}^{k-1}\right)$ taken and corresponding objective value $g\left(\tilde{\mathbf{w}}^k\right)$ respectively, $\widetilde{\mathbf{X}}$ is the input data matrix, $\mathbf{y}$ the output values, and $\tilde{\mathbf{w}}^0$ the initial point.

Almost all of this function has already been constructed for you. For example, the step length is fixed at $\alpha_k = 10^{-2}$ for all iterations, etc., and you must only enter the gradient of the associated cost function. Pressing "run" in the editor will run gradient descent and will reproduce Fig. 3.11.

Exercises 3.12 A general sinusoid model nonlinear in its parameters

Recall the periodic sinusoidal regression discussed in Example 3.2. There we chose a model $b + \sin\left(2\pi x_p w\right) \approx y_p$ that fit the given data which was linear in the weights b and w, and we saw that the corresponding Least Squares cost function was therefore convex. This allowed us to solve for the optimal values for these weights in closed form via the first order system, with complete assurance that they represent a global minimum of the associated Least Squares cost function. In this exercise you will investigate how a simple change to this model leads to a comparably much more challenging optimization problem to solve.

Figure 3.20 shows a set of $P = 75$ data points $\left\{\left(x_p, y_p\right)\right\}_{p=1}^P$ generated via the model

$$w_1 \sin\left(2\pi x_p w_2\right) + \epsilon = y_p \quad \text{for all } p = 1 \ldots P, \tag{3.38}$$

where $\epsilon > 0$ is a small amount of noise. This dataset may be located in the file *extended_sinusoid_data.csv*. Unlike the previous instance here the model is nonlinear

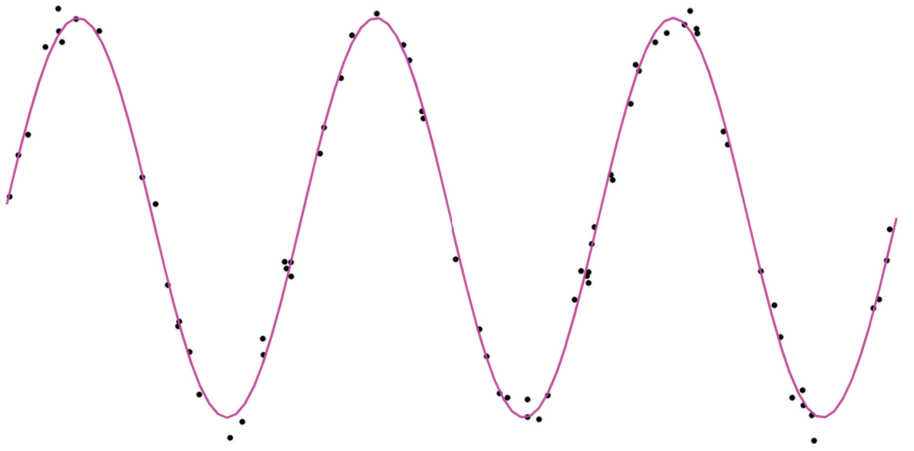

Fig. 3.20 A set of $P = 75$ periodic data points along with the underlying sinusoidal model used to generate the data in magenta.

in both the weights. Here w_1 controls the amplitude (i.e., stretches the model in the vertical direction) and w_2 controls the frequency (i.e., how quickly the sinusoid completes a single period) of the sinusoidal model.

We can then attempt to recover optimal weights of a representative curve for this data-set by minimizing the associated Least Squares cost function with respect to the dataset

$$g\left(\mathbf{w}\right) = \sum_{p=1}^{P}\left(w_1 \sin\left(2\pi x_p w_2\right) - y_p\right)^2. \tag{3.39}$$

a) Plot the surface of g over the region defined by $-3 \leq w_1, w_2 \leq 3$.

b) Discuss the approach you would take to find the best possible stationary point of this function, along with any potential difficulties you foresee in doing so.

Exercises 3.13 Code up gradient descent for ℓ_2 regularized logistic regression

In this exercise you will reproduce Fig. 3.13 by coding up gradient descent to minimize the regularized logistic regression Least Squares cost function shown in Equation (3.29).

a) Verify that the gradient of the cost function can be written as

$$\nabla g\left(\tilde{\mathbf{w}}\right) = 2\sum_{p=1}^{P}\left(\sigma\left(\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}\right) - y_p\right)\sigma\left(\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}\right)\left(1 - \sigma\left(\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}\right)\right)\tilde{\mathbf{x}}_p + 2\lambda\begin{bmatrix}0\\ \mathbf{w}\end{bmatrix}. \tag{3.40}$$

b) The surface in this figure was generated via the wrapper *l2reg_nonconvex_logistic_growth* with the dataset *bacteria_data.csv*, and inside the wrapper you must complete a short gradient descent function to produce the descent paths called

$$[\text{in, out}] = \text{grad\_descent}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \tilde{\mathbf{w}}^0\right), \tag{3.41}$$

where "in" and "out" contain the gradient steps $\tilde{\mathbf{w}}^k = \tilde{\mathbf{w}}^{k-1} - \alpha_k \nabla g\left(\tilde{\mathbf{w}}^{k-1}\right)$ taken and corresponding objective value $g\left(\tilde{\mathbf{w}}^k\right)$ respectively, $\widetilde{\mathbf{X}}$ is the input data matrix whose pth column is the input data $\tilde{\mathbf{x}}_p$, $\mathbf{y}$ the output values stacked into a column vector, and $\tilde{\mathbf{w}}^0$ the initial point.

Almost all of this function has already been constructed for you. For example, the step length is fixed at $\alpha_k = 10^{-2}$ for all iterations, etc., and you must only enter the gradient of the associated cost function. Pressing "run" in the editor will run gradient descent and will reproduce Fig. 3.13.

Exercises 3.14 The ℓ_2 regularized Newton's method

Recall from Section 2.2.4 that when applied to minimizing non-convex cost functions, Newton's method can climb to local maxima (or even diverge) due to the concave shape of the quadratic second order Taylor series approximation at concave points of the cost function (see Fig. 2.11). One very common way of dealing with this issue, which we explore formally in this exercise, is to add an ℓ_2 regularizer (centered at each step) to the quadratic approximation used by Newton's method in order to ensure that it is convex at each step. This is also commonly done when applying Newton's method to convex functions as well since, as we will see, the addition of a regularizer increases the eigenvalues of the Hessian and therefore helps avoid numerical problems associated with solving linear systems with zero (or near-zero) eigenvalues.

a) At the kth iteration of the regularized Newton's method we add an ℓ_2 regularizer centered at $\mathbf{w}^{k-1}$, i.e., $\frac{\lambda}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2$ where $\lambda \geq 0$, to the second order Taylor series approximation in (2.18), giving

$$\begin{aligned} h(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2}\left(\mathbf{w} - \mathbf{w}^{k-1}\right)^T \\ \nabla^2 g\left(\mathbf{w}^{k-1}\right)\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{\lambda}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2. \end{aligned} \tag{3.42}$$

Show that the first order condition for optimality leads to the following adjusted Newton's system for a stationary point of the above quadratic:

$$\left[\nabla^2 g\left(\mathbf{w}^{k-1}\right) + \lambda \mathbf{I}_{N\times N}\right]\mathbf{w} = \left[\nabla^2 g\left(\mathbf{w}^{k-1}\right) + \lambda \mathbf{I}_{N\times N}\right]\mathbf{w}^{k-1} - \nabla g\left(\mathbf{w}^{k-1}\right). \tag{3.43}$$

b) Show that the eigenvalues of $\nabla^2 g\left(\mathbf{w}^{k-1}\right) + \lambda \mathbf{I}_{N\times N}$ in the system above can all be made to be positive by setting λ large enough. What is the smallest value of λ that will make this happen? This is typically the value used in practice. *Hint: see Exercise* 2.9.

c) Using the value of λ determined in part **b)**, conclude that the ℓ_2 regularized second order Taylor series approximation centered at $\mathbf{w}^{k-1}$ in (3.42) is convex. *Hint: see Exercise 2.11.*

For a non-convex function, λ is typically adjusted at each step so that it just forces the eigenvalues of $\nabla^2 g\left(\mathbf{w}^{k-1}\right) + \lambda \mathbf{I}_{N\times N}$ to be all positive. In the case of a convex cost, since the eigenvalues of $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$ are always nonnegative (via the second order definition of convexity) *any* positive value of λ will force the eigenvalues of $\nabla^2 g\left(\mathbf{w}^{k-1}\right) + \lambda \mathbf{I}_{N\times N}$ to be all positive. Therefore often for convex functions λ is set fixed for all iterations at some small value like $\lambda = 10^{-3}$ or $\lambda = 10^{-4}$.

4 Classification

In this chapter we discuss the problem of classification, where we look to distinguish between different types of distinct things. Beyond the crucial role such an ability plays in contributing to what we might consider as "intelligence," modern applications of classification arise in a wide range of fields including computer vision, speech processing, and digital marketing (see e.g., Sections 1.2.2 and 4.6). We begin by introducing the fundamental model for two class classification: the *perceptron*. As described pictorially in Fig. 1.10, the perceptron works by finding a line/hyperplane (or more generally a curve/surface) that separates two classes of data. We then describe two equally effective approximations to the basic perceptron known as the softmax and margin perceptrons, followed by a description of popular perspectives on these approximations where they are commonly referred to as *logistic regression* and *support vector machines*, respectively. In Section 4.4 we see how the two class framework can be easily generalized to deal with multiclass classification problems that have arbitrary numbers of distinct classes. Finally, we end the chapter by discussing knowledge-driven feature design methods for classification. This includes a description of basic histogram-based features commonly used for text, image, and speech classification problems.

4.1 The perceptron cost functions

In the most basic instance of a classification problem our data consists of just two classes. Common examples of two class classification problems include face detection, with classes consisting of facial versus non-facial images, textual sentiment analysis where classes consist of written product reviews ascribing a positive or negative opinion, and automatic diagnosis of medical conditions where classes consist of medical data corresponding to patients who either do or do not have a specific malady (see Sections 1.2.2 and 4.6 for further descriptions of these problems). In this section we introduce the most foundational tool for two class classification, the *perceptron*, as well as a popular variation called the *margin perceptron*. Both tools are commonly used and perform similarly in practice, as we discuss further in Section 4.1.7.

4.1.1 The basic perceptron model

Recall from the previous chapter that in a linear regression setting, given a training set of P continuous-valued input/output data points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$, we aim to learn a hyperplane $b + \mathbf{x}^T\mathbf{w}$ with parameters b and $\mathbf{w}$ such that

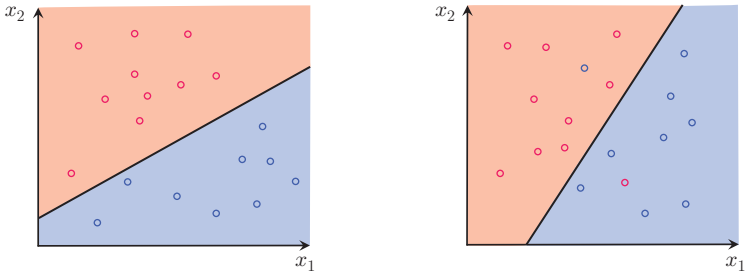

Fig. 4.1 With linear classification we aim to learn a hyperplane $b + \mathbf{x}^T\mathbf{w} = 0$ (shown here in black) to separate feature representations of the two classes, colored red (class "+1") and blue (class "−1"), by dividing the feature space into a red half-space where $b + \mathbf{x}^T\mathbf{w} > 0$, and a blue half-space where $b + \mathbf{x}^T\mathbf{w} < 0$. (left panel) A linearly separable dataset where it is possible to learn a hyperplane to perfectly separate the two classes. (right panel) A dataset with two overlapping classes. Although the distribution of data does not allow for perfect linear separation, we can still find a hyperplane that minimizes the number of misclassified points that end up in the wrong half-space.

$$b + \mathbf{x}_p^T\mathbf{w} \approx y_p \tag{4.1}$$

holds for $p = 1, \ldots, P$. In the case of linear classification a disparate yet simple motivation leads to the pursuit of a different sort of ideal hyperplane. As opposed to linear regression, where our aim is to *represent* a dataset, with classification our goal is to *separate* two distinct classes of the input/output data with a learned hyperplane. In other words, we want to learn a hyperplane $b + \mathbf{x}^T\mathbf{w} = 0$ that separates the two classes of points as much as possible, with one class lying "above" the hyperplane in the half-space given by $b + \mathbf{x}^T\mathbf{w} > 0$ and the other "below" it in the half-space $b + \mathbf{x}^T\mathbf{w} < 0$, as illustrated in Fig. 4.1.

More formally, with two class classification we still have a training set of P input/output data points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ where each input $\mathbf{x}_p$ is N-dimensional (with each entry representing an input feature, just as with regression). However, the output data no longer takes on continuous but two discrete values or *labels* indicating class membership, i.e., points belonging to each class are assigned a distinct label. While one can choose any two values for this purpose, we will see that the values ± 1 are particularly useful and therefore will assume that $y_p \in \{-1, +1\}$ for $p = 1, \ldots, P$.

We aim to learn the parameters b and $\mathbf{w}$ of a hyperplane, so that the first class (where $y_p = +1$) lies largely *above* the hyperplane in the half-space defined by $b + \mathbf{x}^T\mathbf{w} > 0$, and the second class (where $y_p = -1$) lies mostly *below*[1] it in the half-space defined by $b + \mathbf{x}^T\mathbf{w} < 0$. If a given hyperplane places the point $\mathbf{x}_p$ on its correct side (or we say that it correctly classifies the point), then we have precisely that

[1] The choice of which class we assume lies "above" and "below" the hyperplane is arbitrary, i.e., if we instead suppose that those points with label $y_p = -1$ lie above and those with label $y_p = +1$ lie below, similar calculations can be made which lead to the perceptron cost function in Equation (4.5).

$$\begin{aligned} b + \mathbf{x}_p^T\mathbf{w} &> 0 \quad \text{if } \ y_p = +1 \\ b + \mathbf{x}_p^T\mathbf{w} &< 0 \quad \text{if } y_p = -1. \end{aligned} \tag{4.2}$$

Because we have chosen the labels ± 1 we can express (4.2) compactly by multiplying the two expressions by minus their respective label value $-y_p$, giving one equivalent expression

$$-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right) < 0. \tag{4.3}$$

By taking the maximum of this quantity and zero we can then write this condition, which states that a hyperplane correctly classifies the point $\mathbf{x}_p$, equivalently as

$$\max\left(0,\, -y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) = 0. \tag{4.4}$$

Note that the expression $\max\left(0,\, -y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)$ returns zero if $\mathbf{x}_p$ is classified correctly, but it returns a *positive* value if the point is classified incorrectly. This is useful not only because it characterizes the sort of hyperplane we wish to have, but more importantly by simply summing this expression over all the points we have the non-negative cost function

$$g_1(b, \mathbf{w}) = \sum_{p=1}^{P} \max\left(0,\, -y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right), \tag{4.5}$$

referred to as the *perceptron* or *max* cost function.[2] Solving the minimization problem

$$\underset{b,\,\mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \max\left(0,\, -y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right), \tag{4.6}$$

then determines the optimal parameters for our separating hyperplane. However, while this problem is fine in principle, there are two readily apparent technical issues regarding the minimization itself. First, one minimum of g_1 always presents itself at the trivial and undesirable values $b = 0$ and $\mathbf{w} = \mathbf{0}_{N\times 1}$ (which indeed gives $g_1 = 0$). Secondly, note that while g_1 is continuous (and it is in fact convex) it is not everywhere differentiable (see Fig. 4.2), thus prohibiting the use of gradient descent and Newton's method.[3] One simple work-around for both of these issues is to make a particular smooth approximation to the perceptron function, which we discuss next.

4.1.2 The softmax cost function

One popular way of approximating the perceptron cost is to replace the non-differentiable "max" function $\max(s_1, s_2)$ (which returns the maximum of the two scalar inputs s_1 and s_2) in (4.5) with the smooth *softmax function* defined as

$$\text{soft}(s_1, s_2) = \log\left(e^{s_1} + e^{s_2}\right). \tag{4.7}$$

[2] The perceptron is also referred to as the *hinge* (as it is shaped like a hinge, see Fig. 4.2 for an illustration) or *rectified linear unit*.

[3] While specialized algorithms can be used to tune the perceptron (see e.g., [19]) differentiable approximations (that permit the use of gradient descent and/or Newton's method) are typically preferred over these options due to their superior efficacy and speed.

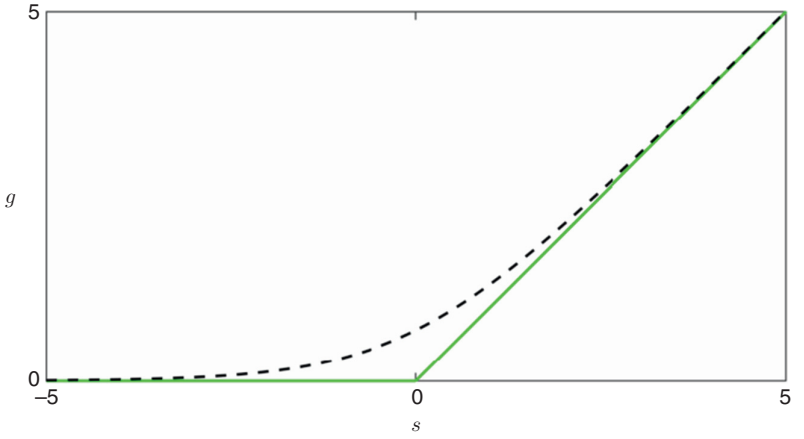

Fig. 4.2 Plots of the non-differentiable perceptron or hinge cost $g\,(s) = \max\,(0,\,s)$ (shown in green) as well as its smooth softmax approximation $g\,(s) = \text{soft}\,(0,\,s) = \log\left(1 + e^{s}\right)$ (shown in dashed black).

That $\text{soft}\,(s_1, s_2) \approx \max\,(s_1, s_2)$, or in words that the softmax approximates the max function, can be verified formally[4] and intuited visually in the particular example shown in Fig. 4.2.

Replacing the "max" function in the pth summand of g_1 in (4.5) with its softmax approximation,

$$\text{soft}\left(0,\,-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) = \log\left(1 + e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}\right), \tag{4.8}$$

we have a smooth approximation of the perceptron cost given by

$$g_2\,(b,\,\mathbf{w}) = \sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}\right), \tag{4.9}$$

which we will refer to as the *softmax cost function.* Note that this cost function does not have a trivial minimum at $b = 0$ and $\mathbf{w} = \mathbf{0}_{N\times 1}$ as was the case with the original perceptron. It also has the benefit of being smooth and hence we may apply gradient descent or Newton's method for its minimization as detailed in Section 2.2, the latter

[4] The fact that the softmax function provides a good approximation to the max function can be shown formally by the following simple argument. Suppose momentarily that $s_1 \leq s_2$, so that $\max\,(s_1,\,s_2) = s_2$. Therefore $\max\,(s_1,\,s_2)$ can be written as $\max\,(s_1,\,s_2) = s_1 + (s_2 - s_1)$, or equivalently as $\max\,(s_1,\,s_2) = \log\left(e^{s_1}\right) + \log\left(e^{s_2-s_1}\right)$ since $s = \log\left(e^{s}\right)$ for any s. Written in this way we can see that $\log\left(e^{s_1}\right) + \log\left(1 + e^{s_2-s_1}\right) = \log\left(e^{s_1} + e^{s_2}\right) = \text{soft}\,(s_1, s_2)$ is always larger than $\max\,(s_1,\,s_2)$ but not by much, especially when $e^{s_2-s_1} \gg 1$. Since the same argument can be made if $s_1 \geq s_2$ we can say generally that $\text{soft}\,(s_1, s_2) \approx \max\,(s_1, s_2)$.

Note also that the softmax approximation to the max function applies more generally for C inputs, as

$$\max\,(s_1, \ldots, s_C) \approx \text{soft}\,(s_1, \ldots, s_C) = \log\left(\sum_{c=1}^{C} e^{s_c}\right).$$

of which we may safely use as the softmax cost is indeed convex (see Exercise 4.2). Formally, the softmax minimization problem is written as

$$\underset{b,\,\mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}\right). \tag{4.10}$$

This approximation to the perceptron cost is very commonly used in practice, most often referred to as the *logistic regression* for classification (see Section 4.2) or *log-loss support vector machines* (see Section 4.3). Due to its immense popularity as the logistic regression, we will at times refer to the minimization of the softmax cost as the learning of the softmax or logistic regression classifier.

Example 4.1 Optimization of the softmax cost

Using the compact notation $\tilde{\mathbf{x}}_p = \begin{bmatrix} 1 \\ \mathbf{x}_p \end{bmatrix}$ and $\tilde{\mathbf{w}} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}$ we can rewrite the softmax cost function in (4.9) more conveniently as

$$g_2\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}}\right). \tag{4.11}$$

Using the chain rule[5] we can then compute the gradient, and setting it equal to zero we check the first order condition (see Section 2.1.2),

$$\nabla g_2\left(\tilde{\mathbf{w}}\right) = -\sum_{p=1}^{P} \sigma\left(-y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}\right) y_p\tilde{\mathbf{x}}_p = \mathbf{0}_{(N+1)\times 1}. \tag{4.12}$$

Note here that $\sigma(-t) = \frac{1}{1+e^t}$ denotes the logistic sigmoid function[6] evaluated at $-t$ (see Section 3.3.1). However (4.12) is an unwieldy and highly nonlinear system of

[5] To see how to employ the chain rule let us briefly rewrite the pth summand in (4.11) explicitly as a composition of functions

$$\log\left(1 + e^{-y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}}\right) = f\left(r\left(s\left(\tilde{\mathbf{w}}\right)\right)\right),$$

where $f(r) = \log(r)$, $r(s) = 1 + e^{-s}$, and $s\left(\tilde{\mathbf{w}}\right) = y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}$. To compute the derivative of this with respect to a single entry $\tilde{w}_n$ the chain rule gives

$$\frac{\partial}{\partial\tilde{w}_n} f\left(r\left(s\left(\tilde{\mathbf{w}}\right)\right)\right) = \frac{df}{dr}\cdot\frac{dr}{ds}\cdot\frac{\partial}{\partial\tilde{w}_n}s\left(\tilde{\mathbf{w}}\right) = \frac{1}{r}\left(-e^{-s}\right)y_p\tilde{x}_{n,p} = \frac{1}{1+e^{-y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}}}\left(-e^{-y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}}\right)y_p\tilde{x}_{n,p},$$

which can be written more compactly as $-\sigma\left(-y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}\right)y_p\tilde{x}_{p,n}$ using the fact that $\frac{1}{1+e^{-t}}\left(-e^{-t}\right)$ $= \frac{1}{1+e^{-t}}\cdot\frac{-1}{e^t} = \frac{-1}{1+e^t} = -\sigma(-t)$, where $\sigma(t)$ is the logistic sigmoid function. By combining the result for all entries in $\tilde{\mathbf{w}}$ and summing over all P summands we then get the gradient as shown in (4.12).

[6] Writing the derivative in this way also helps avoid numerical problems associated with using the exponential function on a modern computer. This is due to the exponential "overflowing" with large exponents, like e.g., e^{1000}, as these numbers are too large to store explicitly on the computer and so are represented symbolically as ∞. This becomes a problem when dividing two exponentials like e.g., $\frac{e^{1000}}{1+e^{1000}}$ which, although basically equal to the value 1, is thought of by the computer to be a NaN (not a

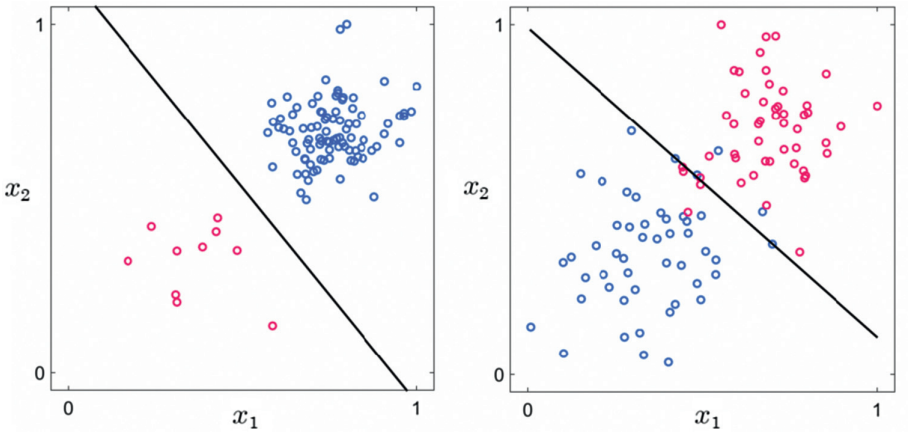

Fig. 4.3 (left panel) A two-dimensional toy dataset with linearly separable classes consisting of $P = 100$ points in total (90 in the "+1" class and 10 in the "−1" class), along with the softmax classifier learned using gradient descent. (right panel) A two-dimensional toy dataset with overlapping classes consisting of $P = 100$ points in total (50 points in each class), with the softmax classifier learned again using gradient descent. In both cases the learned classifier does a good job separating the two classes.

$N + 1$ equations which must be solved numerically by applying e.g., gradient descent or Newton's method. By again employing the chain rule, and noting that we always have that $y_p^2 = 1$ since $y_p \in \{-1, +1\}$, one may additionally compute the Hessian of the softmax as the following sum of weighted outer product matrices (see Exercise 2.10):

$$\nabla^2 g_2\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P} \sigma\left(-y_p \tilde{\mathbf{x}}_p^T \tilde{\mathbf{w}}\right)\left(1 - \sigma\left(-y_p \tilde{\mathbf{x}}_p^T \tilde{\mathbf{w}}\right)\right) \tilde{\mathbf{x}}_p \tilde{\mathbf{x}}_p^T. \tag{4.13}$$

Figure 4.3 illustrates the classification of two toy datasets, one linearly separable (left panel) and the other non-separable or overlapping (right panel), using the softmax classifier. In both cases a gradient descent scheme is used to learn the hyperplanes' parameters.

4.1.3 The margin perceptron

Here we discuss an often used variation of the original perceptron, called the margin perceptron, that is once again based on analyzing the geometry of the classification problem

number) as it thinks $\frac{e^{1000}}{1+e^{1000}} = \frac{\infty}{\infty}$ which is undefined. By writing each summand of the gradient such that it has an exponential in its denominator only we avoid the problem of dividing two overflowing exponentials. The overflowing exponential issue is discussed further in the exercises, as it is also something to keep in mind when both choosing an initial point for gradient descent/Newton's method as well as recording the value of the softmax cost at each iteration.

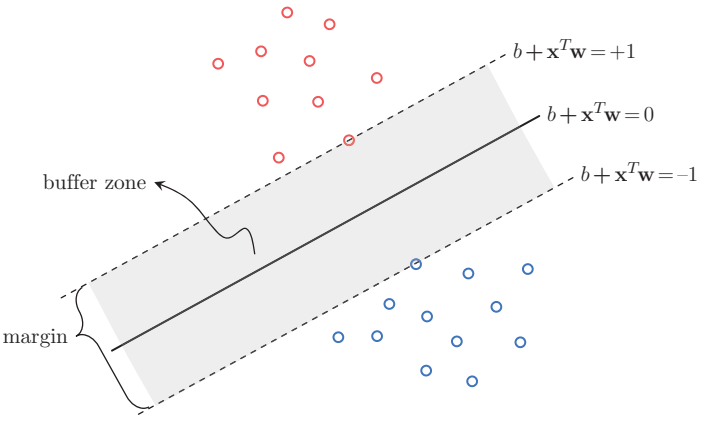

Fig. 4.4 For linearly separable data the width of the buffer zone confined between two evenly spaced translates of a separating hyperplane that just touches each respective class, defines the margin of that separating hyperplane.

where a line (or hyperplane in higher dimensions) is used to separate two classes of data. Due to the great similarity between the two perceptron concepts, what follows closely mirrors Sections 4.1.1 and 4.1.2.

Suppose for a moment that we are dealing with a two class dataset that is linearly separable with a known hyperplane $b+\mathbf{x}^T\mathbf{w} = 0$ passing evenly between the two classes as illustrated in Fig. 4.4. This separating hyperplane creates a buffer zone between the two classes confined between two evenly shifted versions of itself: one version that lies *above* the separator and just touches the class having labels $y_p = +1$ taking the form $b + \mathbf{x}^T\mathbf{w} = +1$, and one lying below it just touching the class with labels $y_p = -1$ taking the form $b + \mathbf{x}^T\mathbf{w} = -1$. The width of this buffer zone is commonly referred to as the *margin* of such a hyperplane.[7]

The fact that all points in the "+1" class lie on or above $b+\mathbf{x}^T\mathbf{w} = +1$, and all points in the "−1" class lie on or below $b+\mathbf{x}^T\mathbf{w} = -1$ can be written formally as the following conditions:

$$\begin{array}{ll} b + \mathbf{x}_p^T\mathbf{w} \geq 1 & \text{if } y_p = +1 \\ b + \mathbf{x}_p^T\mathbf{w} \leq -1 & \text{if } y_p = -1. \end{array} \tag{4.14}$$

We can combine these conditions into a single statement by multiplying each by their respective label values, giving the single inequality $y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right) \geq 1$, which can be equivalently written as

$$\max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) = 0. \tag{4.15}$$

[7] The translations above and below the separating hyperplane are more generally defined as $b + \mathbf{x}^T\mathbf{w} = +\beta$ and $b + \mathbf{x}^T\mathbf{w} = -\beta$ respectively, where $\beta > 0$. However, by dividing off β in both equations and reassigning the variables as $\mathbf{w} \longleftarrow \frac{\mathbf{w}}{\beta}$ and $b \longleftarrow \frac{b}{\beta}$, we can leave out the redundant parameter β and have the two translations as stated, $b + \mathbf{x}^T\mathbf{w} = \pm 1$.

Dropping the assumption that we know the parameters of the hyperplane we can propose, as we did in devising the perceptron cost in (4.5), to *learn* them by minimizing the cost function formed by summing the criterion in (4.15) over all points in the dataset. Referred to as a *margin perceptron* or *hinge cost* this function takes the form

$$g_3(b, \mathbf{w}) = \sum_{p=1}^{P} \max\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right). \tag{4.16}$$

Note the striking similarity between the original perceptron cost in (4.5) and the margin perceptron cost in (4.16): naively we have just "added a 1" to the nonzero input of the "max" function in each summand. However this additional "1" prevents the issue of a trivial zero solution with the original perceptron discussed in Section 4.1.1, which simply does not arise here.

If the data is indeed linearly separable, any hyperplane passing between the two classes will have a parameter pair $(b, \mathbf{w})$ where $g_3(b, \mathbf{w}) = 0$. However, the margin perceptron is still a valid cost function even if the data is not linearly separable. The only difference is that with such a dataset we cannot make the criteria in (4.14) hold for all points in the dataset. Thus a violation for the pth point adds the positive value of $1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)$ to the cost function in (4.16).

Regardless of whether the two classes are linearly separable or not, by minimizing the margin perceptron cost stated formally as

$$\underset{b,\,\mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \max\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right), \tag{4.17}$$

we can learn the parameters for the margin perceptron classifier. However, like the original perceptron, the margin cost is still not everywhere differentiable due to presence of the "max" function. Again it is common practice to make simple differentiable approximations to this cost so that descent methods, such as gradient descent and Newton's method, may be employed.

4.1.4 Differentiable approximations to the margin perceptron

To produce a differentiable approximation to the margin perceptron cost in (4.16) we can of course employ the softmax function first introduced in Section 4.1.2, replacing each summand's $\max(\cdot)$ function with $\text{soft}(\cdot)$. Specifically, taking the softmax approximation of the pth summand of (4.16) gives $\text{soft}\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) \approx \max\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)$, where

$$\text{soft}\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) = \log\left(1 + e^{1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)}\right). \tag{4.18}$$

Summing over $p = 1, \ldots, P$ we can produce a cost function that approximates the margin perceptron, with the added benefit of differentiability. However, note that,

as illustrated in Fig. 4.7, in fact the softmax approximation of the original perceptron summand $\text{soft}\left(0, -y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) = \log\left(1 + e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}\right)$ provides, generally speaking, just as good approximation of the margin perceptron.[8] Therefore the original softmax cost in (4.9) also provides a useful differentiable approximation to the margin perceptron as well!

Another perhaps more straightforward way of making a differentiable approximation to the margin perceptron cost is simply to square each of its summands, giving the *squared margin perceptron* cost function

$$g_4(b, \mathbf{w}) = \sum_{p=1}^{P} \max^2\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right), \tag{4.19}$$

where $\max^2(s_1, s_2)$ is a brief way of writing $(\max(s_1, s_2))^2$. Note that when the two classes are linearly separable, solutions to the corresponding minimization problem,

$$\underset{b,\mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \max^2\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right), \tag{4.20}$$

are precisely those of the original problem in (4.17). Moreover its differentiability permits easily computed gradient for use in gradient descent and Newton's method.

Example 4.2 Optimization of the squared margin perceptron

Using the compact notation $\tilde{\mathbf{x}}_p = \begin{bmatrix} 1 \\ \mathbf{x}_p \end{bmatrix}$ and $\tilde{\mathbf{w}} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}$ we can compute the gradient of the squared margin perceptron cost using the chain rule, and form the first order system of $N + 1$ equations

$$\nabla g_4\left(\tilde{\mathbf{w}}\right) = -2\sum_{p=1}^{P} \max\left(0,\ 1 - y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}\right) y_p\tilde{\mathbf{x}}_p = \mathbf{0}_{(N+1)\times 1}. \tag{4.21}$$

Because once again it is impossible to solve this system for $\tilde{\mathbf{w}}$ in closed form, a solution must be found iteratively by applying gradient descent. Since g_4 is convex (see Exercise 4.6) it is also possible to apply Newton's method, with the Hessian easily computable (noting that we always have that $y_p^2 = 1$ since $y_p \in \{-1, +1\}$) as[9]

$$\nabla^2 g_4\left(\tilde{\mathbf{w}}\right) = 2\sum_{p\in\Omega_{\tilde{\mathbf{w}}}} \tilde{\mathbf{x}}_p\tilde{\mathbf{x}}_p^T, \tag{4.22}$$

where $\Omega_{\tilde{\mathbf{w}}}$ is the index set defined as $\Omega_{\tilde{\mathbf{w}}} = \left\{p \middle| 1 - y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}} > 0\right\}$.

[8] As shown in Fig. 4.7 while the function $\text{soft}(0, 1 - t)$ better approximates $\max(0, 1 - t)$ for values of $t \leq 0$, $\text{soft}(0, -t)$ provides a better approximation for $t > 0$.

[9] This is actually a "generalized" Hessian since the "max" function in the gradient is not everywhere differentiable. Nevertheless, it still makes a highly effective Newton's method for the squared margin cost (see e.g., [27]).

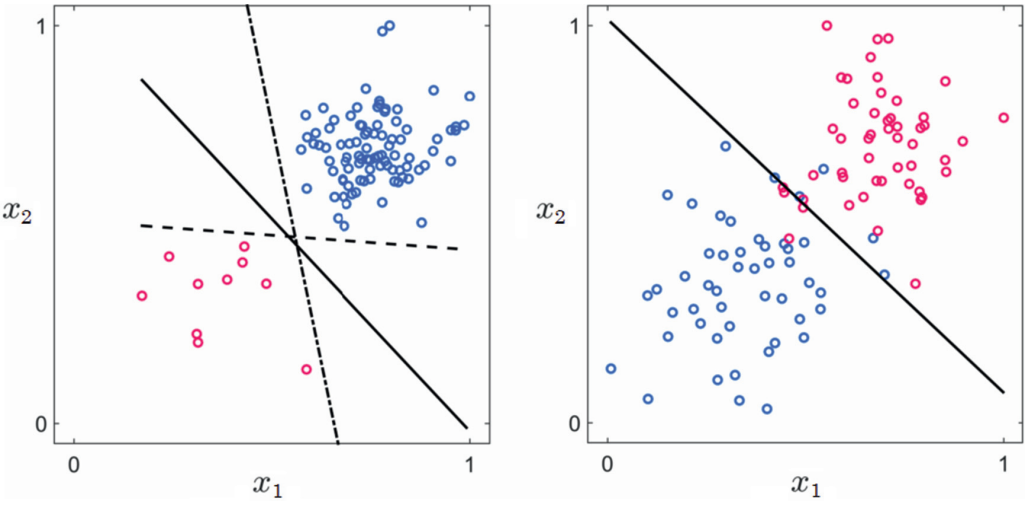

Fig. 4.5 Classification of two toy datasets, first shown in Fig. 4.3, using gradient descent for minimizing the squared margin perceptron cost function. Initializing gradient descent with three different starting points results in three different classifiers for the linearly separable dataset in the left panel, each perfectly separating the two classes.

In Fig. 4.5 we show the resulting linear classifiers learned by minimizing the squared margin perceptron cost for the two toy datasets first shown in Fig. 4.3. As is the case with the dataset in the left panel of this figure, when the two classes of data are linearly separable there are infinitely many distinct separating hyperplanes, and correspondingly infinitely many distinct minima of g_4. Thus on such a dataset initializing gradient descent or Newton's method with a random starting point means we may reach a different solution at each run, along with a distinct separating hyperplane.

4.1.5 The accuracy of a learned classifier

From our discussion of the original perceptron in Section 4.1.1, note that given any parameter pair $(b, \mathbf{w})$ (learned by any of the cost functions described in this section) we can determine whether a point $\mathbf{x}_p$ is classified correctly or not via the following simple evaluation:

$$\text{sign}\left(-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) = \begin{cases} +1 & \text{if } \mathbf{x}_p \text{ incorrectly classified} \\ -1 & \text{if } \mathbf{x}_p \text{ correctly classified,} \end{cases} \tag{4.23}$$

where $\text{sign}(\cdot)$ takes the mathematical sign of the input. Also note that by taking the maximum of this value and 0,

$$\max\left(0, \text{sign}\left(-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)\right) = \begin{cases} +1 & \text{if } \mathbf{x}_p \text{ incorrectly classified} \\ 0 & \text{if } \mathbf{x}_p \text{ correctly classified,} \end{cases} \tag{4.24}$$

we can count the precise number of misclassified points for a given set of parameters $(b, \mathbf{w})$ by summing (4.24) over all p. This observation naturally leads to a fundamental *counting cost* function, which precisely counts the number of points from the training data classified incorrectly as

$$g_0(b, \mathbf{w}) = \sum_{p=1}^{P} \max\left(0,\ \text{sign}\left(-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)\right). \tag{4.25}$$

By plugging in any learned weight pair $(b^\star, \mathbf{w}^\star)$, the value of this cost function provides a metric for evaluating the performance of the associated linear classifier, i.e., the number of misclassifications for the given weight pair. This can be used to define the *accuracy* of a classifier with the weights $(b^\star, \mathbf{w}^\star)$ on the training data as

$$\text{accuracy} = 1 - \frac{g_0(b^\star, \mathbf{w}^\star)}{P}. \tag{4.26}$$

This metric ranges from 0 to 1, with an ideal classification corresponding to an accuracy of 1 or 100%. If possible it is also a good idea to compute the accuracy of a learned classifier on a set of new testing data, i.e., data that was not used to learn the model itself, 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*.

4.1.6 Predicting the value of new input data

As illustrated in Fig. 4.6, to predict the label y_{new} of a new point $\mathbf{x}_{\text{new}}$ we simply check which side of the learned hyperplane it lies on as

$$y_{\text{new}} = \text{sign}\left(b^\star + \mathbf{x}_{\text{new}}^T\mathbf{w}^\star\right), \tag{4.27}$$

where this hyperplane has parameters $(b^\star,\ \mathbf{w}^\star)$ learned over the current dataset via any of the cost functions described in this section. In other words, if the new point lies above the learned hyperplane $(b^\star + \mathbf{x}_{\text{new}}^T\mathbf{w}^\star > 0)$ it is given the label $y_{\text{new}} = 1$, and likewise if

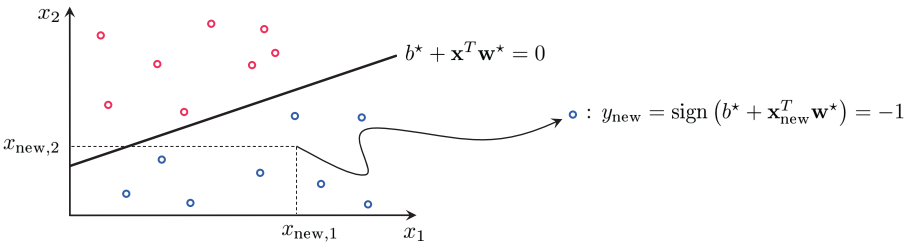

Fig. 4.6 Once a hyperplane has been learned to the current dataset with optimal parameters $\left(b^\star, \mathbf{w}^\star\right)$, the label y_{new} of a new point $\mathbf{x}_{\text{new}}$ can be determined by simply checking which side of the boundary it lies on. In the illustration shown here $\mathbf{x}_{\text{new}}$ lies below the learned hyperplane $(b^\star + \mathbf{x}_{\text{new}}^T\mathbf{w}^\star < 0)$ and so is given the label $y_{\text{new}} = \text{sign}\left(b^\star + \mathbf{x}_{\text{new}}^T\mathbf{w}^\star\right) = -1$.

the point lies below the boundary ($b^\star + \mathbf{x}_{\text{new}}^T\mathbf{w}^\star < 0$) it receives the label $y_{\text{new}} = -1$. If on the off chance the point lies on the boundary itself (i.e., $b^\star + \mathbf{x}_{\text{new}}^T\mathbf{w}^\star = 0$) then $\mathbf{x}_{\text{new}}$ may be assigned to either class.

4.1.7 Which cost function produces the best results?

In terms of accuracy, which (differentiable) cost function works the best in practice, the softmax or squared margin perceptron? Nothing we have seen so far seems to indicate one cost function's superiority over the other. In fact, the various geometric derivations given so far have shown how both are intimately related to the original perceptron cost in (4.5). Therefore it should come as little surprise that while they can differ from dataset to dataset in terms of their performance, in practice both differentiable costs typically produce very similar results.

> The softmax and squared margin costs perform similarly well in practice.

Thus one should feel comfortable using either one or, if resources allow, apply both and keep the higher performer on a case by case basis. Figure 4.7 shows a visual comparison of all classification cost functions we have discussed so far.

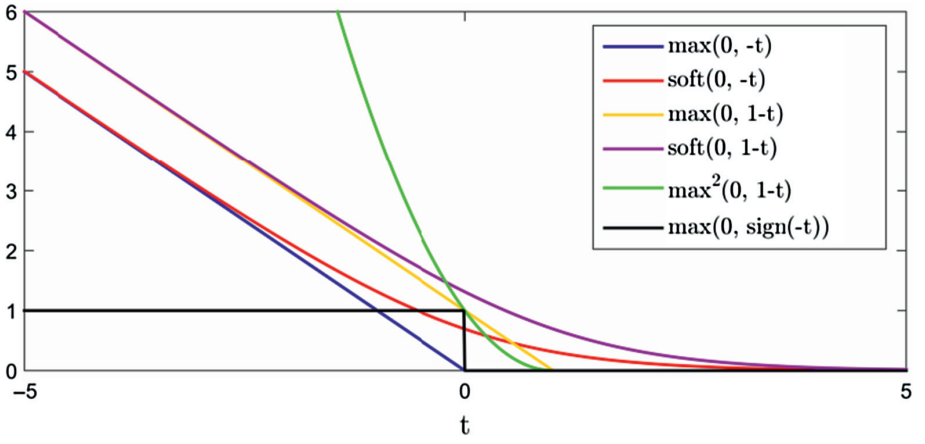

Fig. 4.7 Comparison of various classification cost functions. For visualization purposes we show here only one summand of each cost function plotted versus $t = b + \mathbf{x}_p^T\mathbf{w}$ with the label y_p assumed to be 1. The softmax cost (red) is a smooth approximation to the non-differentiable perceptron or hinge cost (blue), which can be thought of itself as a continuous surrogate for the discontinuous counting loss (black). The margin cost (yellow) is a shifted version of the basic perceptron, and is non-differentiable at its corner point. The squared margin cost (green) resolves this issue by taking its square (as does the softmax cost). Note that all the cost functions (except for the counting cost) are convex.

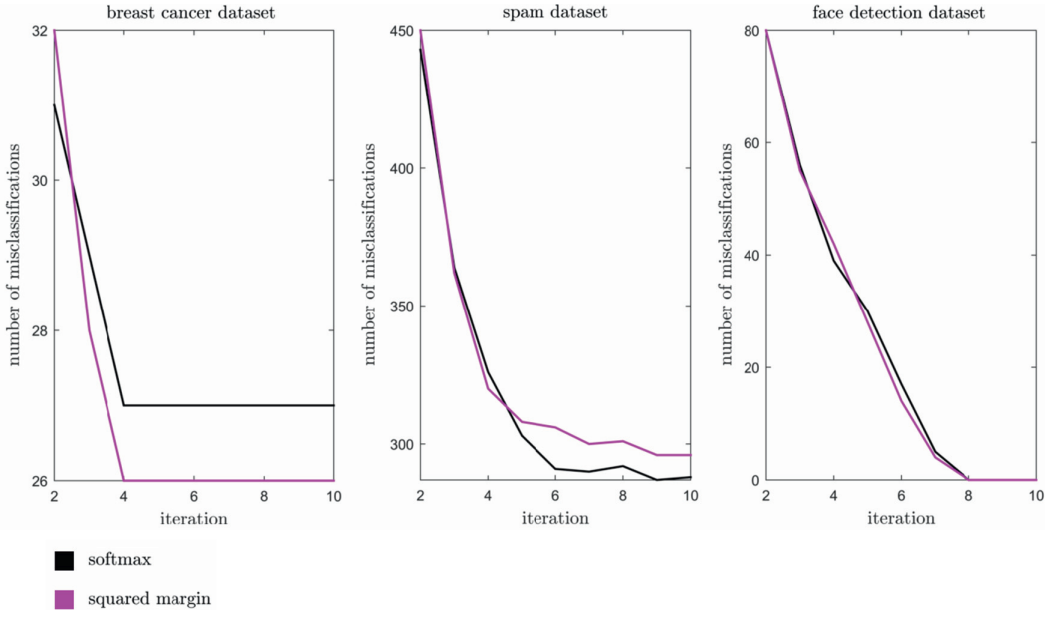

Fig. 4.8 A comparison of the softmax and margin costs on three real training datasets. Shown in each panel is the number of misclassifications per iteration of Newton's method (only ten iterations were required for convergence in all instances) applied to minimizing the softmax cost (shown in black) and squared margin cost (shown in magenta) over (left panel) breast cancer, (middle panel) spam email, and (right panel) face detection datasets respectively. While the performance of each cost function differs from case to case, generally they perform similarly well.

Example 4.3 Real dataset comparison of the softmax and squared margin costs

In Fig. 4.8 we illustrate the similar efficacy of the softmax and squared margin costs on three real training datasets. For each dataset we show the number of misclassifications resulting from the use of ten iterations of Newton's method (as only ten iterations were required for the method to converge in all cases), by evaluating the counting cost in (4.25) at each iteration, to minimize both cost functions over the data.

The left, middle, and right panels of the figure display these results on breast cancer (consisting of $P = 569$), spam email (with $P = 4601$ points), and face detection datasets (where $P = 10\,000$) respectively. While their performance differs from case to case the softmax and margin costs perform similarly well in these examples. For more information about the datasets used here see Exercise 4.9, as well as Examples 4.9 and 4.10.

4.1.8 The connection between the perceptron and counting costs

Note that with the cost function defined in (4.25) as our true desired criterion for linear classification, we could have begun our discussion by trying to minimize it formally as

$$\underset{b,\mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \max\left(0,\ \text{sign}\left(-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)\right). \tag{4.28}$$

Unfortunately this problem is not only non-convex but is highly *discontinuous* due to the presence of the "sign" function in each summand of the objective. Therefore it is extremely difficult to attempt to minimize it directly. However, note that the original perceptron cost derived in (4.5) can be thought of simply as a relaxation of this fundamental counting cost, where we remove the discontinuous "sign" function from each summand (or in other words, approximate $\text{sign}\left(-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)$ linearly as $-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)$). Thus while the original perceptron cost, as well as its relatives including the softmax[10] and margin costs, are intimately related to this counting cost they are still *approximations* of the true criterion we wish to minimize.

In Fig. 4.9 we illustrate this point by showing both the number of misclassifications and objective value of gradient descent applied to minimizing the softmax cost over the toy datasets shown first in Fig. 4.3. Specifically, we show results from three runs of gradient descent applied to both the linearly separable (top panels) and overlapping (bottom panels) datasets. In the left panels of Fig. 4.9 we show the number of misclassifications per iteration calculated by evaluating the counting cost in (4.25), while in the right panels we show the corresponding softmax cost values from each run per iteration. In other words, the left and right panels show the value of the counting cost function from (4.25) and the softmax cost from (4.9) per iteration of gradient descent, respectively.

Comparing the left and right panels for each dataset note that, in both instances, the per iteration counting and softmax values do not perfectly match. Further note how with the second dataset, shown in the lower two panels, the counting cost value actually fluctuates (by a small amount) as we increase the number of iterations while the corresponding softmax cost value continues to fall. Both of these phenomena are caused by the fact that we are directly minimizing an approximation of the counting cost, and not the counting cost itself. While neither effect is ideal, they are examples of the tradeoff we must accept for working with cost functions we can actually minimize properly in practice.

4.2 The logistic regression perspective on the softmax cost

This section describes a common way of both deriving and thinking about the softmax cost function first introduced in Section 4.1.2. Here we will see how the softmax cost naturally arises as a direct approximation of the fundamental counting cost discussed in Section 4.1.5. However the major benefit of this new perspective is in adding a useful geometric viewpoint,[11] that of regression/surface-fitting, to the classification framework in general, and the softmax cost in particular.

[10] We will also see in Section 4.2 how the softmax cost can be thought of as a direct approximation of the counting cost.

[11] Logistic regression can also be interpreted from a *probabilistic* perspective (see Exercise 4.12).

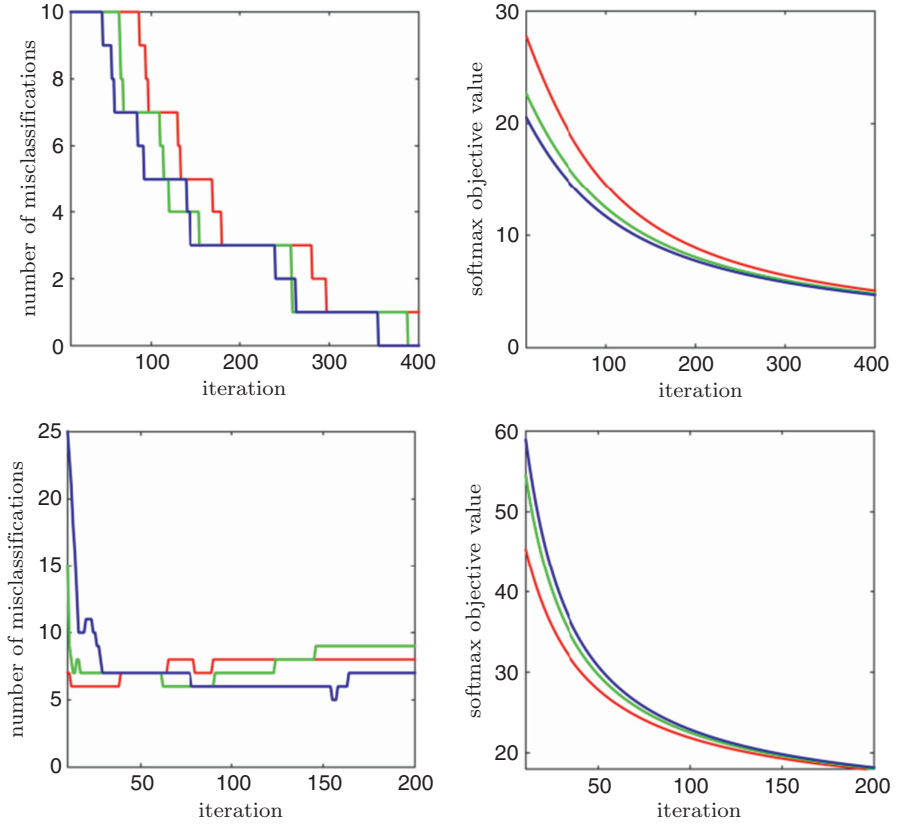

Fig. 4.9 The number of misclassifications (left panels) and objective value (right panels) plotted versus number of iterations of three runs of gradient descent applied to minimizing the softmax cost over two toy datasets, one linearly separable (top panels) and the other overlapping (bottom panels), both shown originally in Fig. 4.3.

4.2.1 Step functions and classification

Two class classification can be fruitfully considered as a particular instance of regression or surface-fitting, wherein the output of a dataset of P points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ is no longer continuous but takes on two fixed values, $y_p \in \{-1, +1\}$, corresponding to the two classes. As illustrated in Fig. 4.10, an ideal *data generating function* for classification (i.e., a function that can be assumed to generate the data we receive) is a discontinuous step function (shown in yellow). When the step function is viewed "from above", as also illustrated in this figure, we return to viewing classification from the "separator" point of view described in the previous section, and the linear boundary separating the two classes is defined exactly by the hyperplane where the step function transitions from its lower to higher step, defined by

$$b + \mathbf{x}^T\mathbf{w} = 0. \tag{4.29}$$

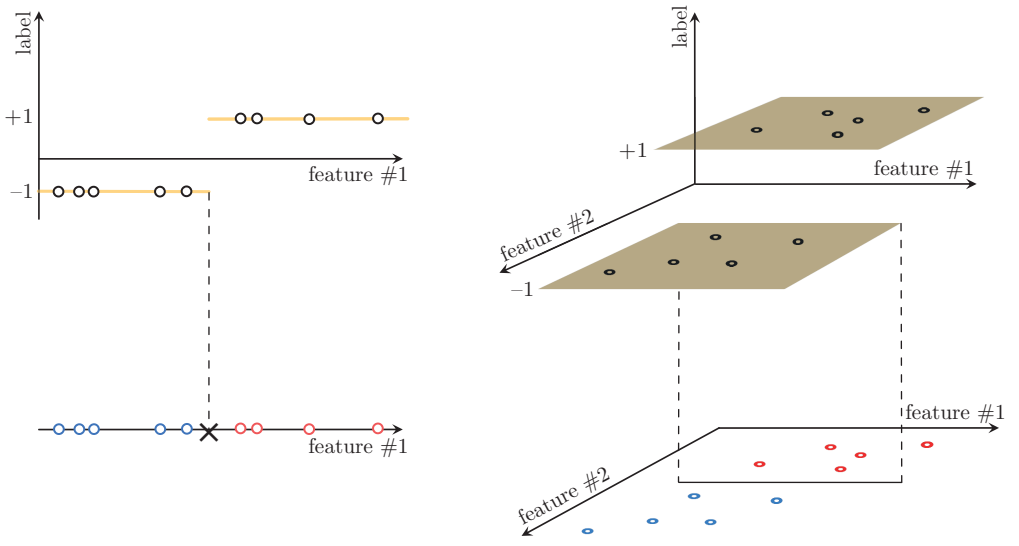

Fig. 4.10 Classification from a regression/surface-fitting perspective for 1-dimensional (left panel) and 2-dimensional (right panel) toy datasets. This surface-fitting view is equivalent to the "separator" perspective described in Section 4.1, where the separating hyperplane is precisely where the step function (shown here in yellow) transitions from its lower to higher step. In the separator view the actual y value (or label) is represented by coloring the points red or blue to denote their respective classes.

With this, the equation for any step function taking values on $\{-1, +1\}$ can be written explicitly as

$$\text{sign}\left(b + \mathbf{x}^T\mathbf{w}\right) = \begin{cases} +1 & \text{if } b + \mathbf{x}^T\mathbf{w} > 0 \\ -1 & \text{if } b + \mathbf{x}^T\mathbf{w} < 0. \end{cases} \tag{4.30}$$

We ideally would like to find a set of parameters $(b, \mathbf{w})$ for a hyperplane so that data points having label $y_p = +1$ lie on the top step, and those having label $y_p = -1$ lie on the bottom step. To say then that a particular parameter choice places a point $\mathbf{x}_p$ on its correct step means that $\text{sign}\left(b + \mathbf{x}_p^T\mathbf{w}\right) = y_p$, and because $y_p \in \{-1, +1\}$ this can be written equivalently as

$$\text{sign}\left(y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) = 1. \tag{4.31}$$

In what follows we will make a smooth approximation to the step function, in particular deriving a smoothed equivalent of the criterion in (4.31) for the parameters of a desired hyperplane. This will quickly lead us to the minimization of the softmax cost function first described in Section 4.1.2 in order to properly fit a smoothed step function to our labeled data.

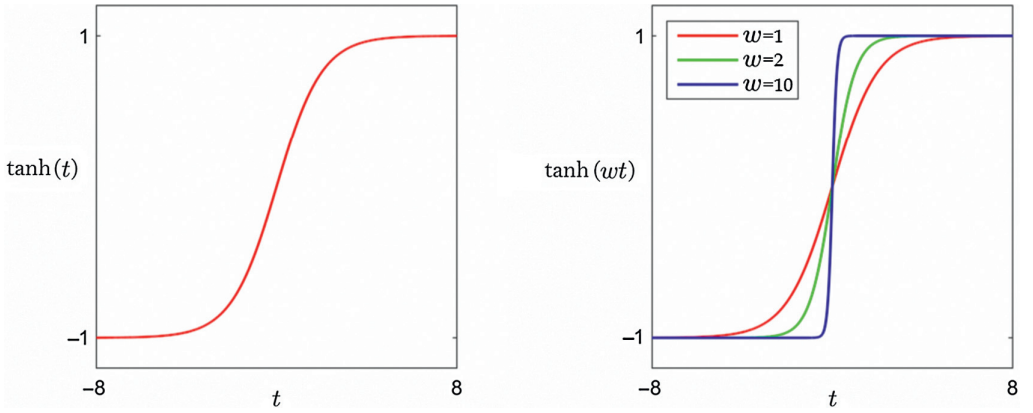

Fig. 4.11 (left panel) Plot of the tanh function defined as $\tanh(t) = 2\sigma(t) - 1$. (right panel) By increasing the weight w of the function $\tanh(wt)$ from $w = 1$ (red) to $w = 2$ (green) and finally to $w = 10$ (blue), it becomes an increasingly good approximator of a step function taking on values -1 and $+1$.

4.2.2 Convex logistic regression

We have actually already seen an excellent smooth approximator of a step function, i.e., the sigmoid function

$$\sigma\left(b + \mathbf{x}^T\mathbf{w}\right) = \frac{1}{1 + e^{-(b+\mathbf{x}^T\mathbf{w})}}, \tag{4.32}$$

introduced in Section 3.3.1 in its original context as a model for population growth. As shown in Fig. 3.10, by adjusting the parameters $(b, \mathbf{w})$ the sigmoid can be made to approximate a step function taking on the values $\{0, 1\}$. By simply multiplying the sigmoid by 2 and then subtracting 1 we can stretch it so that it approximates a step function taking on the values $\{-1, +1\}$. This stretched sigmoid is referred to as the "tanh" function

$$\tanh\left(b + \mathbf{x}^T\mathbf{w}\right) = 2\sigma\left(b + \mathbf{x}^T\mathbf{w}\right) - 1. \tag{4.33}$$

As shown in the left panel of Fig. 4.11, the tanh function retains the desired property of the sigmoid by being a fine approximator to the step function, this time one that takes on values $\{-1, +1\}$.

Thus we have for any pair $(b, \mathbf{w})$ that any desired step function of the form given in (4.30) may be roughly approximated as $\text{sign}\left(b + \mathbf{x}^T\mathbf{w}\right) \approx \tanh\left(b + \mathbf{x}^T\mathbf{w}\right)$, or in other words

$$\tanh\left(b + \mathbf{x}^T\mathbf{w}\right) \approx \begin{cases} +1 & \text{if } b + \mathbf{x}^T\mathbf{w} > 0 \\ -1 & \text{if } b + \mathbf{x}^T\mathbf{w} \leq 0. \end{cases} \tag{4.34}$$

To make this approximation finer we can, as illustrated in the right panel of Fig. 4.11, multiply the input argument of the tanh by a large positive constant.

Now with tanh as a smooth approximation of the "sign" function, we can approximate the criterion in (4.31) as

$$\tanh\left(y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)\right)\approx 1, \tag{4.35}$$

which can be written, using the definition of tanh in (4.33), as

$$1+e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}\approx 1. \tag{4.36}$$

Taking the log of both sides[12] then leads to

$$\log\left(1+e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}\right)\approx 0. \tag{4.37}$$

Since we want a hyperplane that forces the condition in (4.37) to hold for all $p = 1,\ldots,P$, a reasonable way of learning associated parameters is to simply minimize the sum of these expressions over the entire dataset as

$$\underset{b,\mathbf{w}}{\text{minimize}}\sum_{p=1}^{P}\log\left(1+e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}\right). \tag{4.38}$$

This is precisely the minimization of the softmax cost first introduced in Section 4.1.2 as a smooth approximation to the original perceptron. Here, however, our interpretation has changed: we think of the minimization of the softmax cost in the current section in the context of logistic regression surface-fitting (where the output takes on only the values ± 1), determining ideal parameters for a smoothed step function to fit to our labeled data.

Through the perspective of logistic regression, we can think of classification simultaneously as:

(1) finding a hyperplane that best separates the data; and

(2) finding a step-like surface that best places the positive and negative classes on its top and bottom steps, respectively.

In Fig. 4.12 we show an example of both the resulting linear separator and surface fit corresponding to minimizing the softmax cost via Newton's method as described in Example 4.1 on a toy dataset first shown in the left panel of Fig. 4.3. The resulting

[12] Without taking the log on both sides one can deduce instead a desired approximation $e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}\approx 0$ to hold, leading to the analogous conclusion that we should minimize the cost function $\sum_{p=1}^{P} e^{-y_p\left(b+\mathbf{x}_p^T\mathbf{w}\right)}$. This approximation, however, is less useful for classification as it is much more sensitive to the presence of *outliers* in the data (see Exercise 4.11). Regardless, it is used for instance as the objective function in a greedy classification method called *boosting* [34].

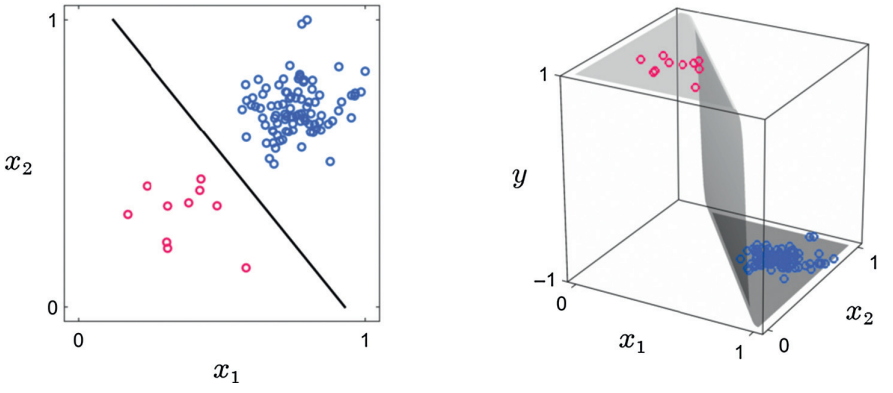

Fig. 4.12 Minimizing the softmax cost in (4.38) gives an optimal weight pair $\left(b^\star, \mathbf{w}^\star\right)$ that define the linear separator $b^\star + \mathbf{x}^T\mathbf{w}^\star = 0$ shown in the left panel (in black), as well as the surface $y\left(\mathbf{x}\right) = \tanh\left(b^\star + \mathbf{x}^T\mathbf{w}^\star\right)$ shown in the right panel (in gray).

parameters found $(b^\star, \mathbf{w}^\star)$ define both the linear separator $b^\star + \mathbf{x}^T\mathbf{w}^\star = 0$, as well as the surface $y\left(\mathbf{x}\right) = \tanh\left(b^\star + \mathbf{x}^T\mathbf{w}^\star\right)$.

4.3 The support vector machine perspective on the margin perceptron

In deriving the margin perceptron in Section 4.1.3 we introduced the concept of a margin for a hyperplane as the width of the buffer zone it creates between two linearly separable classes. We now extend this idea to its natural conclusion, leading to the so-called support vector machine (SVM) classifier. While an intriguing notion in the ideal case where data is perfectly separable, we will see by the end of this section that practically speaking the SVM classifier is a margin perceptron with the addition of an ℓ_2 regularizer (ℓ_2 regularization was first introduced in Section 3.3.2).

4.3.1 A quest for the hyperplane with maximum margin

As discussed in Section 4.1.3, when two classes of data are linearly separable, infinitely many hyperplanes could be drawn to separate the data. In Fig. 4.5 we displayed three such hyperplanes for a given synthetic dataset, each derived by starting the gradient descent procedure for minimizing the squared margin perceptron cost with a different initialization. Given that all these three classifiers (as well as any other separating hyperplane derived from this procedure) would perfectly classify the data, is there one that we can say is the "best" of all possible separating hyperplanes? One reasonable standard for judging the quality of these hyperplanes is via their margin lengths, that is the distance between the evenly spaced translates that just touch each class. The larger this distance is, the intuitively better the associated hyperplane separates the entire space, given the particular distribution of the data. This idea is illustrated in Fig. 4.13.

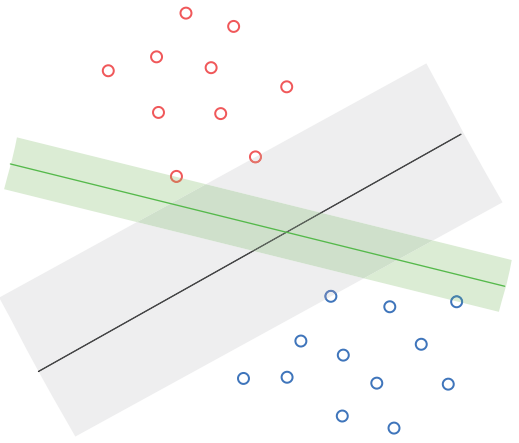

Fig. 4.13 Of the infinitely many hyperplanes that exist between two classes of linearly separable data the one with maximum margin does an intuitively better job than the rest at distinguishing between classes because it more equitably partitions the entire space based on how the data is distributed. In this illustration two separators are shown along with their respective margins. While both perfectly distinguish between the two classes the green separator (with smaller margin) divides up the space in a rather awkward fashion given how the data is distributed, and will therefore tend to more easily misclassify future data points. On the other hand, the black separator (having a larger margin) divides up the space more evenly with respect to the given data, and will tend to classify future points more accurately.

To find the separating hyperplane with maximum margin, first recall from Section 4.1.3 that the margin of a hyperplane $b + \mathbf{x}^T\mathbf{w} = 0$ is the width of the buffer zone confined between two symmetric translations of itself, written conveniently as $b + \mathbf{x}^T\mathbf{w} = \pm 1$, each just touching one of the two classes. As shown in Fig. 4.14, the margin can be determined by calculating the distance between any two points (one from each translated hyperplane) both lying on the normal vector $\mathbf{w}$. Denoting by $\mathbf{x}_1$ and $\mathbf{x}_2$ the points on vector $\mathbf{w}$ belonging to the *upper* and *lower* translated hyperplanes, respectively, the margin is computed simply as the length of the line segment connecting $\mathbf{x}_1$ and $\mathbf{x}_2$, i.e., $\|\mathbf{x}_1 - \mathbf{x}_2\|_2$.

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

$$\left(b + \mathbf{x}_1^T\mathbf{w}\right) - \left(b + \mathbf{x}_2^T\mathbf{w}\right) = (\mathbf{x}_1 - \mathbf{x}_2)^T\,\mathbf{w} = 2. \tag{4.39}$$

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

$$\|\mathbf{x}_1 - \mathbf{x}_2\|_2 = \frac{2}{\|\mathbf{w}\|_2}. \tag{4.40}$$

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

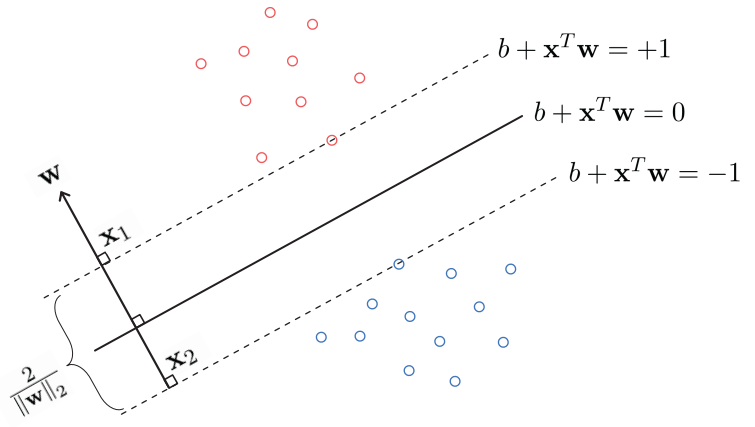

Fig. 4.14 The margin of a separating hyperplane can be calculated by measuring the distance between the two points of intersection of the normal vector $\mathbf{w}$ and the two equidistant translations of the hyperplane. This distance can be shown to have the value of $\frac{2}{\|\mathbf{w}\|_2}$ (see text for further details).

4.3.2 The hard-margin SVM problem

In order to find a separating hyperplane for the data with minimum length normal vector we can simply combine this with our desire to minimize $\|\mathbf{w}\|_2^2$ subject to the constraint that the hyperplane perfectly separates the data (given by the margin criterion in (4.14)). This gives the so-called *hard-margin SVM* constrained optimization problem

$$\begin{aligned} &\underset{b,\,\mathbf{w}}{\text{minimize}} \;\; \|\mathbf{w}\|_2^2 \\ &\text{subject to} \;\; \max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) = 0, \quad p = 1, \ldots, P. \end{aligned} \tag{4.41}$$

Unlike the minimization problems we have seen so far, here we have a set of constraints on the permissible values of $(b, \mathbf{w})$ that guarantee that the hyperplane we recover separates the data perfectly. Problems of this sort can be solved using a variety of optimization techniques (see e.g., [23, 24, 50]) that we do not discuss here.

Figure 4.15 shows the SVM hyperplane learned for a toy dataset along with the buffer zone confined between the separating hyperplane's translates. The points from each class lying on either boundary of the buffer zone are called *support vectors*, hence the name "support vector machines," and are highlighted in green.

4.3.3 The soft-margin SVM problem

Because a priori we can never be entirely sure in practice that our data is perfectly linearly separable, the hard-margin SVM problem in (4.41) is of mostly theoretical interest. This is because if the data is not perfectly separable by a hyperplane, the hard-margin problem in (4.41) is "ill-defined," meaning that it has no solution (as the constraints can never be satisfied). As a result the hard-margin SVM problem, which again was designed on the assumption of perfect linear separability between the two classes, is not

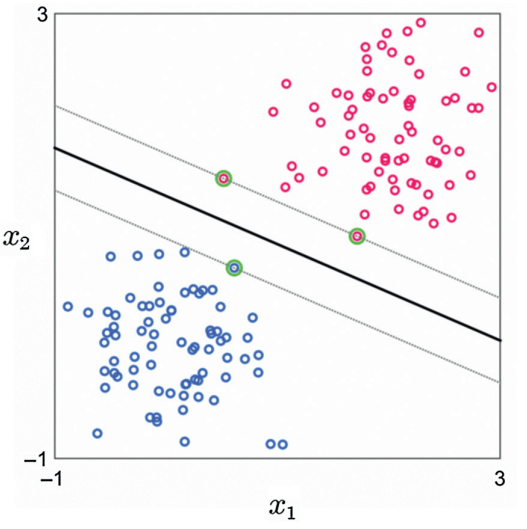

Fig. 4.15 A linearly separable toy dataset consisting of $P = 150$ points in total (75 per class) with the SVM classifier (in black) learned by solving the hard-margin SVM problem. Also shown are the buffer zone boundaries (dotted) and support vectors (highlighted in green).

commonly used in practice. Instead, its constraints are typically "relaxed" in order to allow for possible violations of linear separability. To relax the constraints[13] we make them part of a single cost function, which includes the original objective $\|\mathbf{w}\|_2^2$ as well, so that they are not all forced to hold exactly. This gives the *soft-margin SVM* cost function

$$g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P} \max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) + \lambda\left\|\mathbf{w}\right\|_2^2, \tag{4.42}$$

where the parameter $\lambda \geq 0$ controls the trade-off between how well we satisfy the original constraints in (4.41) while seeking a large margin classifier. The smaller we set λ the more pressure we put on satisfying the constraints of the original problem, and the less emphasis we put on the recovered hyperplane having a large margin (and vice versa). While λ is often set to a small value in practice, we discuss methods for automatically choosing the value of λ in Chapter 7. Formally, minimization of the soft-margin SVM cost function is written as

$$\underset{b,\,\mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) + \lambda\left\|\mathbf{w}\right\|_2^2. \tag{4.43}$$

[13] Generally speaking any relaxed version of the SVM, which allows for violations of perfect linear separability of the data, is referred to as a *soft-margin SVM* problem. While there is another popular relaxation of the basic SVM problem used in practice (see e.g., [22, 23]) it has no theoretical or practical advantage over the one presented here [21, 27].

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

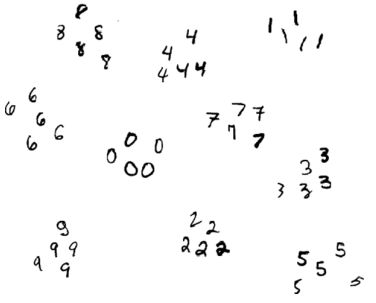

Fig. 4.16 Various handwritten digits in a feature space. Handwritten digit recognition is a common multiclass classification problem. The goal here is to determine regions in the feature space where current (and future) instances of each type of handwritten digit are present.

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.

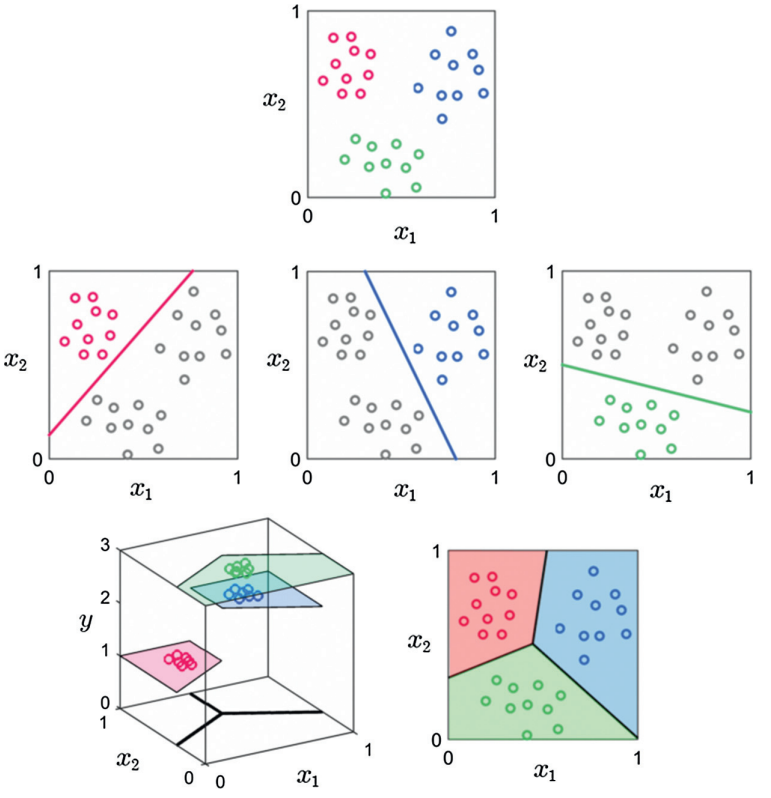

Fig. 4.17 One-versus-all multiclass scheme applied to (top panel) a toy classification dataset with $C = 3$ classes consisting of $P = 30$ data points in total (10 per class). (middle panels) The three classifiers learned to distinguish each class from the rest of the data. In each panel we have temporarily 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

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

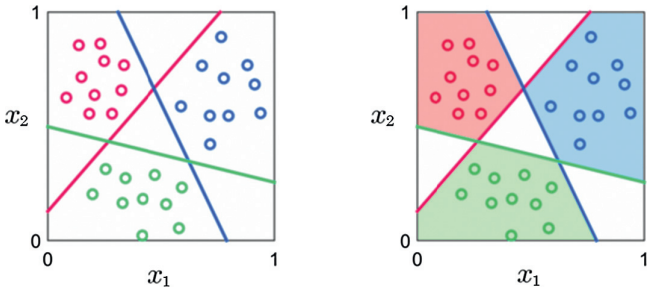

Fig. 4.18 (left panel) Linear separators from the middle panel of Fig. 4.17. (right panel) Regions of the space are colored according to the set of rules in (4.46). White regions do not satisfy these conditions, meaning that points in these areas cannot be assigned to any class/color. In the case shown 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.

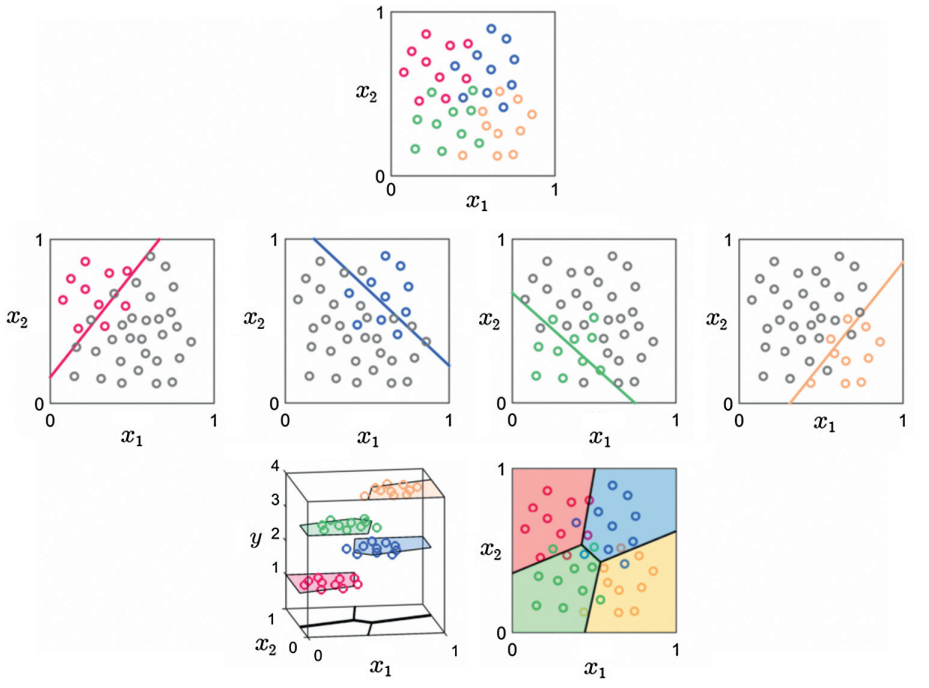

Fig. 4.19 One-versus-all multiclass scheme applied to (top panel) a toy classification dataset with $C = 4$ classes consisting of $P = 40$ data points in total (10 per class). (middle panels) The four classifiers learned to distinguish each class from the rest of the data. (bottom panels) Having determined 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).

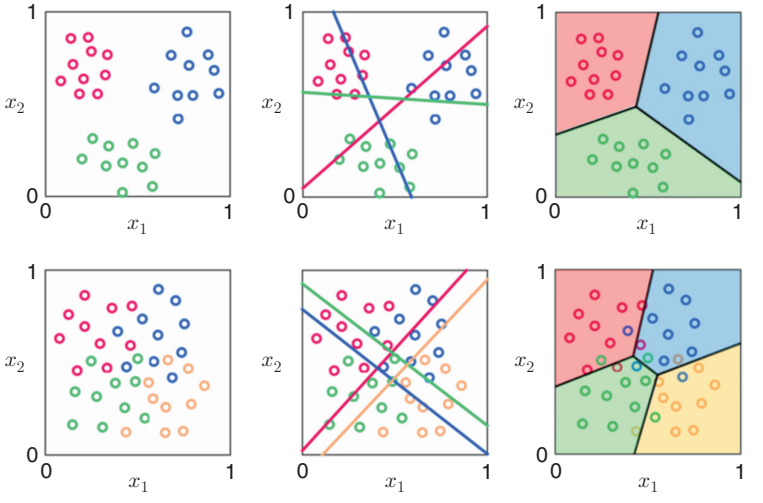

Fig. 4.20 (top left panel) Toy dataset from Fig. 4.17 with $C = 3$ classes. (top middle panel) Individual linear classifiers learned by the multiclass softmax scheme. (top right panel) Final partitioning of the feature space resulting from the application of the fusion rule in (4.47). (bottom left panel)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

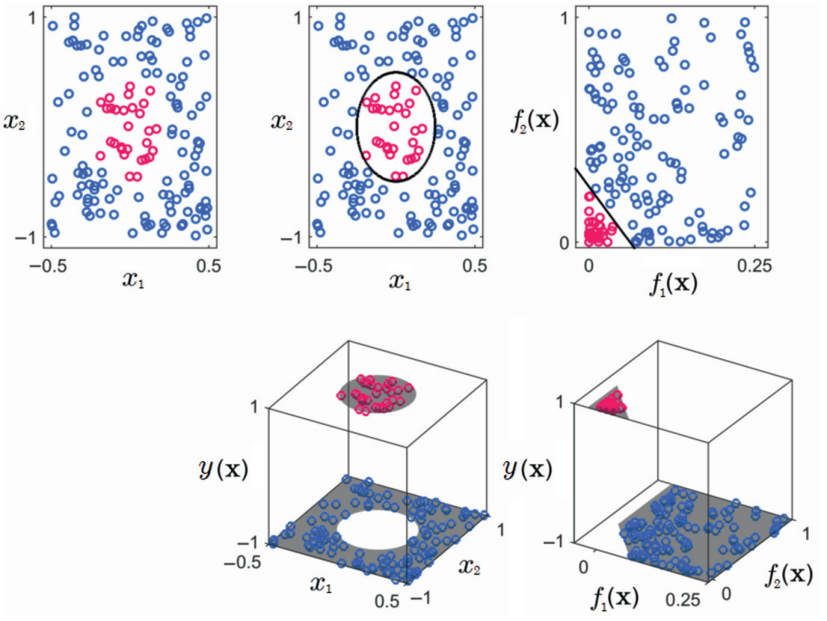

Fig. 4.21 (top left panel) A toy classification dataset where the two classes are separable via an elliptical boundary. (top middle panel) A proper learned boundary given as $1 + x_1^2 w_1^\star + x_2^2 w_2^\star = 0$ can perfectly separate the two classes. (top right panel) Finding this elliptical boundary in the original feature space is equivalent to finding a line to separate the data in the transformed space where both input features have undergone a feature transformation $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T \longrightarrow \begin{bmatrix} f_1(\mathbf{x}) & f_2(\mathbf{x}) \end{bmatrix}^T = \begin{bmatrix} x_1^2 & x_2^2 \end{bmatrix}^T$. (bottom panels) The estimated data generating function (in gray) corresponding to each learned boundary, which is a "step function" in the transformed space.

these to construct an *estimated data generating function* (i.e., a function that appears to generate the data at hand). In very rare instances, when the dimension of the input data is low and the distribution of data is "nice," we can visualize the data and determine features by inspecting the data itself. We begin this brief section with such an example in order to practice the concept of feature design in a simple setting, and conclude by making some general points about feature design for classification. In the section following this one we then give a high level overview of common features used for classification problems involving high dimensional text, image, and audio data.

Example 4.7 Data separable by an ellipse

In the top left panel of Fig. 4.21 we show a toy dataset where, by visual inspection, it appears that a nonlinear elliptical boundary can perfectly separate the two classes of data. Recall that the equation of a standard ellipse (i.e., one aligned with the horizontal and vertical axes and centered at the origin) can be written as $1 + x_1^2 w_1 + x_2^2 w_2 = 0$, where w_1 and w_2 determine how far the ellipse stretches in the x_1 and x_2 directions, respectively.

Here we would like to find weights w_1 and w_2 so that the red class (which have label $y_p = +1$) lie inside the ellipse, and the blue class (having label $y_p = -1$) lie outside it. In other words, if the pth point of the dataset is written as $\mathbf{x}_p = \begin{bmatrix} x_{1,p} & x_{2,p} \end{bmatrix}^T$ we would like the weight vector $\mathbf{w} = \begin{bmatrix} w_1 & w_2 \end{bmatrix}^T$ to satisfy

$$\begin{aligned} 1 + x_{1,p}^2 w_1 + x_{2,p}^2 w_2 &< 0 \text{ if } y_p = +1 \\ 1 + x_{1,p}^2 w_1 + x_{2,p}^2 w_2 &> 0 \text{ if } y_p = -1, \end{aligned} \tag{4.63}$$

for $p = 1, \ldots, P$. Note that these equations are *linear* in their weights, and so we can interpret the above as precisely a linear separation criterion for the original perceptron (as in (4.2)) where the bias has been fixed at $b = 1$. In other words, denoting the feature transformations $f_1(\mathbf{x}) = x_1^2$ and $f_2(\mathbf{x}) = x_2^2$, we can combine the above two conditions as $y_p\left(1 + f_1\left(\mathbf{x}_p\right) w_1 + f_2\left(\mathbf{x}_p\right) w_2\right) < 0$ or $\max\left(0,\, y_p\left(1 + f_1\left(\mathbf{x}_p\right) w_1 + f_2\left(\mathbf{x}_p\right) w_2\right)\right) = 0$. Replacing $\max(\cdot)$ with a softmax $(\cdot)$ function and summing over p (as first described in Section 4.1.2) we may tune the weights by minimizing the softmax cost over the transformed data as

$$\underset{b,\mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \log\left(1 + e^{y_p\left(1 + f_1(\mathbf{x}_p) w_1 + f_2(\mathbf{x}_p) w_2\right)}\right). \tag{4.64}$$

This can be minimized precisely as shown in Section 4.1.2, i.e., by using gradient descent or Newton's method. Shown in the top middle and top right panels of Fig. 4.21 are the corresponding learned boundary given by

$$1 + f_1(\mathbf{x})\, w_1^\star + f_2(\mathbf{x})\, w_2^\star = 0, \tag{4.65}$$

whose weights were tuned by minimizing (4.64) via gradient descent, forming an ellipse in the original feature space and a line in the transformed feature space.

Finally note, as displayed in the bottom panels of Fig. 4.21, that the data generating function, that is a function determined by our chosen features as one which generates the given dataset, is not a "step function" in the original feature space because the boundary between the upper and lower sections is nonlinear. However, it is in fact a step function in the transformed feature space since the boundary there is linear. Since every point above[19] or below the learned linear boundary is declared to be of class $+1$ or -1 respectively, the estimated data generating function is given by simply taking the sign of the boundary as

$$y(\mathbf{x}) = \text{sign}\left(1 + f_1(\mathbf{x})\, w_1^\star + f_2(\mathbf{x})\, w_2^\star\right). \tag{4.66}$$

4.5.1 General conclusions

As shown in the previous example, a general characteristic of well-designed feature transformations is that they produce good nonlinear separation in the original feature

[19] Note that the linear separator in this case has negative slope, and we refer to the half-space to its left as the area "above" the separator.

space while simultaneously producing good linear separation in the transformed feature space.[20]

> Properly designed features for linear classification provide good *nonlinear* separation in the original feature space and, simultaneously, good *linear* separation in the transformed feature space.

For any given dataset of arbitrary input dimension, N, if we determine a set of feature transformations $f_1, \ldots, f_M$ so that the boundary given by

$$b + \sum_{m=1}^{M} f_m(\mathbf{x})\, w_m = 0 \tag{4.67}$$

provides proper separation in the original space, it simultaneously splits the data equally well as a hyperplane in the transformed feature space whose M coordinate axes are given by $f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})$. This is in complete analogy to the case of regression where, as we saw in Section 3.2, proper features produce a nonlinear fit in the original feature space and a corresponding linear fit in the transformed feature space. The corresponding estimated data generating function in general is then given by

$$y(\mathbf{x}) = \text{sign}\left(b + \sum_{m=1}^{M} f_m(\mathbf{x})\, w_m\right), \tag{4.68}$$

which produces the sort of generalized step function we saw in the previous example.

Rarely, however, can we design perfect features using our knowledge of a dataset. In many applications data is too high dimensional to visualize or to perfectly understand through some scientific framework. Even in the instance where the data can be visualized, as with the example dataset shown in Fig. 4.22, determining a precise functional form for each feature transformation by visual inspection can be extremely difficult. Later in Chapter 6 we describe a set of tools for the automatic design, or *learning*, of feature transformations directly from the data which can ameliorate this problem.

4.6 Histogram features for real data types

Unlike the synthetic dataset described in Example 4.7, more often than not real instances of classification data cannot be visualized due to the high dimensionality. Because of this, knowledge can rarely be used to define features algebraically for real data, i.e., by proposing a specific functional form for a set of feature transformations (as was

[20] Technically speaking there is one subtle yet important caveat to the use of the word "good" in this statement, in that we do not want to "overfit" the data (an issue we discuss at length in Chapter 6). However, for now this issue will not concern us.

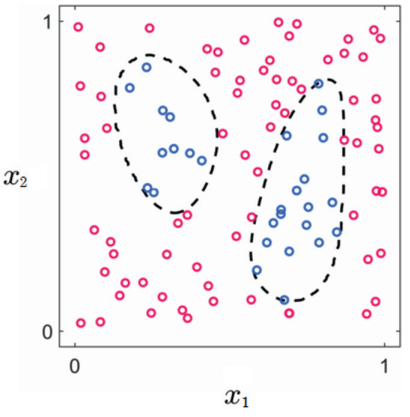

Fig. 4.22 A toy classification dataset where determining proper feature transformations by visual inspection is challenging. The two ovoid boundaries in dashed black are the boundaries between the top and bottom "steps" (i.e., the regions taking on values 1 and -1 respectively) of the data generating function.

done with the toy dataset in Example 4.7). Instead, due to our weaker level of understanding of real data, feature transformations often consist of discrete processing steps which aim at ensuring that instances within a single class are "similar" while those from different classes are "dissimilar." These processing steps are still feature transformations $f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})$ of the input data, but again they are not so easily expressed algebraically.

> Feature transformations for real data often consist of discrete processing steps which aim at ensuring that instances within a single class are "similar" while those from different classes are "dissimilar." While they are not always easily expressed as closed form mathematical equations, these processing steps are, when taken as a whole, still feature transformations $f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})$ of the input data. Thus, properly designed instances will (as discussed in Section 4.5.1) produce good nonlinear separation in the original space and equally good linear separation in the transformed feature space.

In this section we briefly overview methods of knowledge-driven feature design for naturally high dimensional text, image, and audio data types, all of which are based on the same core concept for representing data: the *histogram*. A histogram is just a simple way of summarizing/representing the contents of an array of numbers as a vector showing how many times each number appears in the array. Although each of the aforementioned data types differs substantially in nature, we will see how the notion of a histogram-based feature makes sense in each context. While histogram features are not guaranteed to produce perfect separation, their simplicity and all around solid performance make them quite popular in practice.

Lastly, note that the discussion in this section is only aimed at giving the reader a high level, intuitive understanding of how common knowledge-driven feature design methods work. The interested reader is encouraged to consult specialized texts (referenced throughout this section) on each subject for further study.

4.6.1 Histogram features for text data

Many popular uses of classification, including spam detection and sentiment analysis (see Examples 4.8 and 4.9), are based on text data (e.g., online articles, emails, social-media updates, etc.). However with text data, the initial input (i.e., the document itself) requires a significant amount of preprocessing and transformation prior to further feature design and classification. The most basic yet widely used feature of a document for regression/classification tasks is called a Bag of Words (BoW) histogram or feature vector. Here we introduce the BoW histogram and discuss its strengths, weaknesses, and common extensions.

A BoW feature vector of a document is a simple histogram count of the different words it contains with respect to a single corpus or collection of documents (each count of an individual word is a feature, and taken together gives a feature vector), minus those nondistinctive words that do not characterize the document. To illustrate this idea let us build a BoW representation for the following corpus of two documents each containing a single sentence:

$$\begin{array}{l} 1)\ \text{dogs are the best.} \\ 2)\ \text{cats are the worst.} \end{array} \tag{4.69}$$

To make the BoW representation of these documents we begin by *parsing* them, creating representative vectors (histograms) $\mathbf{x}_1$ and $\mathbf{x}_2$ which contain the number of times each word appears in each document. For the two documents in (4.69) these vectors take the form

$$\mathbf{x}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}\begin{pmatrix} \text{best} \\ \text{cat} \\ \text{dog} \\ \text{worst} \end{pmatrix} \quad \mathbf{x}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}\begin{pmatrix} \text{best} \\ \text{cat} \\ \text{dog} \\ \text{worst} \end{pmatrix}. \tag{4.70}$$

Note that uninformative words such as "are" and "the", typically referred to as *stop words*, are not included in the representation. Further note that we count the singular "dog" and "cat" in place of their plural which appeared in the actual documents in (4.69). This preprocessing step is commonly called *stemming*, where related words with a common stem or root are reduced to and then represented by their common root. For instance, the words "learn," "learning," "learned," and "learner," in the final BoW feature vector are represented by and counted as "learn." Additionally, each BoW vector is normalized to have unit length.

Given that the BoW vector contains only non-negative entries and has unit length, the correlation between two BoW vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ always ranges between $0 \leq \mathbf{x}_1^T\mathbf{x}_2 \leq 1$. When the correlation is zero (i.e., the vectors are perpendicular), as with the two vectors

in (4.70), the two vectors are considered maximally different and will therefore (hopefully) belong to different classes. In the instances shown in (4.70) the fact that $\mathbf{x}_1^T \mathbf{x}_2 = 0$ makes sense: the two documents are completely different, containing entirely different words and polar opposite sentiment. On the other hand, the higher the correlation between two vectors the more similar the documents are purported to be, with highly correlated documents (hopefully) belonging to the same class. For example, the BoW vector of the document "I love dogs" would have positive correlation with $\mathbf{x}_1$, the document in (4.70) about dogs.

However, because the BoW vector is such a simple representation of a document, completely ignoring word order, punctuation, etc., it can only provide a gross summary of a document's contents and is thus not always distinguishing. For example, the two documents "dogs are better than cats" and "cats are better than dogs" would be considered the same document using BoW representation, even though they imply completely opposite relations. Nonetheless, the gross summary provided by BoW can be distinctive enough for many applications. Additionally, while more complex representations of documents (capturing word order, parts of speech, etc.,) may be employed they can often be unwieldy (see e.g., [54]).

Example 4.8 Sentiment analysis

Determining the aggregated feelings of a large base of customers, using text-based content like product reviews, tweets, and comments, is commonly referred to as *sentiment analysis* (as first discussed in Example 1.5). Classification models are often used to perform sentiment analysis, learning to identify consumer data of either positive or negative feelings.

For example, Fig. 4.23 shows BoW vector representations for two brief reviews of a controversial comedy movie, one with a positive opinion and the other with a negative one. The BoW vectors are rotated sideways in this figure so that the horizontal axis contains the common words between the two sentences (after stop word removal

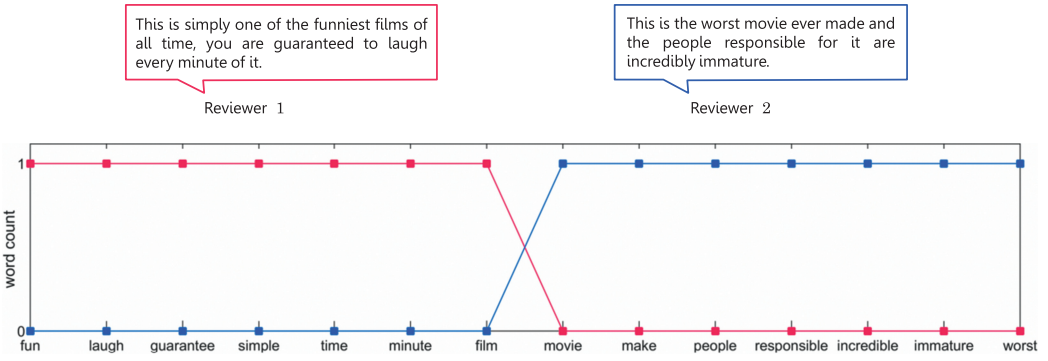

Fig. 4.23 BoW representation of two movie review excerpts, with words (after the removal of stop words and stemming) shared between the two reviews listed along the horizontal axis. The vastly different opinion of each review is reflected very well by the BoW histograms, which have zero correlation.

and stemming), and the vertical axis represents the count for each word (before normalization). The polar opposite sentiment of these two reviews is perfectly represented in their BoW representations, which as one can see are orthogonal (i.e., they have zero correlation).

Example 4.9 Spam detection

Spam detection is a standard text-based two class classification problem. Implemented in most email systems, spam detection automatically identifies unwanted messages (e.g., advertisements), referred to as spam, as distinct from the emails users want to see. Once trained, a spam detector can remove unwanted messages without user input, greatly improving a user's email experience. In many spam detectors the BoW feature vectors are formed with respect to a specific list of spam words (or phrases) including "free," "guarantee," "bargain," "act now," "all natural," etc., that are frequently seen in spam emails. Additionally features like the frequency of certain characters like ! and * are appended to the BoW feature, as are other spam-targeted features like the total number of capital letters in the email and the length of longest uninterrupted sequence of capital letters, as these features can further distinguish the two classes.

In Fig. 4.24 we show classification results on a spam email dataset consisting of BoW, character frequencies, and other spam-focused features (including those mentioned

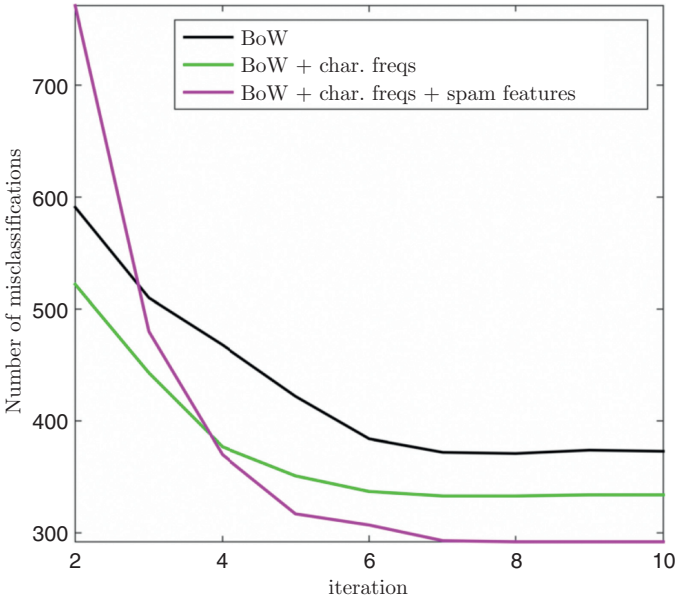

Fig. 4.24 Results of applying the softmax cost (using Newton's method) to distinguish spam from real email using BoW and additional features. The number of misclassifications per iteration of Newton's method is shown in the case of BoW features (in black), BoW and character frequencies (in green), and BoW, character frequencies, as well as spam-focused features (in magenta). In each case adding more distinguishing features (on top of the BoW vector) improves classification. Data in this figure is taken from [47].

previously) taken from 1813 spam and 2788 real email messages for a total of $P = 4601$ data points (this data is taken from [47]). Employing the softmax cost to learn the separator, the figure shows the number of misclassifications per iteration of Newton's method (using the counting cost in (4.25) at each iteration). More specifically these classification results are shown for the same dataset using only BoW features (in black), BoW and character frequencies (in green), and the BoW/character frequencies as well as spam-targeted features (in magenta) (see Exercise 4.20 for further details). Unsurprisingly the addition of character frequencies improves the classification, with the best performance occurring when the spam-focused features are used as well.

4.6.2 Histogram features for image data

To perform classification tasks on image data, like object detection (see Example 1.4), the raw input features are pixel values of an image itself. The pixel values of an 8-bit grayscale image are each just a single integer in the range of 0 (black) to 255 (white), as illustrated in Fig. 4.25. In other words, a grayscale image is just a matrix of integers ranging from 0 to 255. A color image is then just a set of three such grayscale matrices: one for each of the red, blue, and green channels.

Pixel values themselves are typically not discriminative enough to be useful for classification tasks. We illustrate why this is the case using a simple example in Fig. 4.26. Consider the three simple images of shapes shown in the left column of this figure. The first two are similar triangles while the third shape is a square, and we would like an ideal set of features to reflect the similarity of the first two images as well as their distinctness from the last image. However, due to the difference in their relative size, position in the image, and the contrast of the image itself (the image with the smaller triangle is darker toned overall), if we were to use raw pixel values to compare the images (by taking the difference between each image pair[21]) we would find that the square and larger triangle

Fig. 4.25 An 8-bit grayscale image consists of pixels, each taking a value between 0 (black) and 255 (white). To visualize individual pixels, a small 8×8 block from the original image is enlarged on the right.

[21] This is to say that if we denote by $\mathbf{X}_i$ the ith image then we would find that $\|\mathbf{X}_1 - \mathbf{X}_3\|_F < \|\mathbf{X}_1 - \mathbf{X}_2\|_F$.

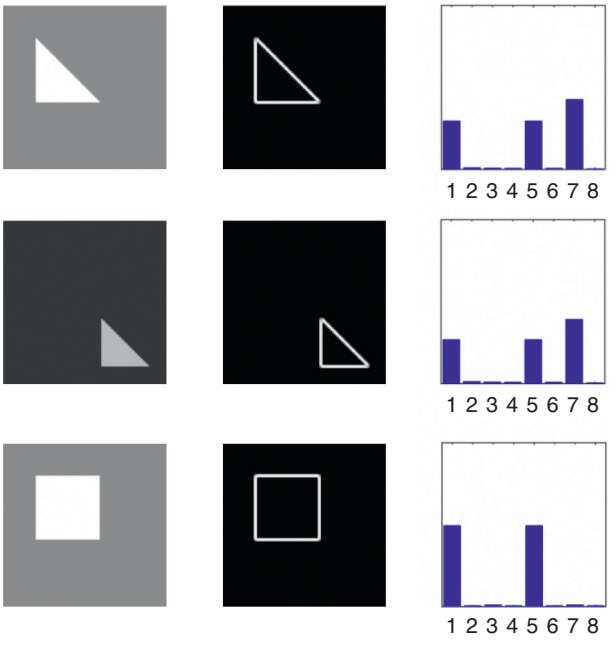

Fig. 4.26 (left column) Three images of simple shapes. While the triangles in the top two images are visually similar, this similarity is not reflected by comparing their raw pixel values. (middle column) Edge-detected versions of the original images, here using eight edge orientations, retain the distinguishing structural content while significantly reducing the amount of information in each image. (right column) By taking normalized histograms of the edge content we have a feature representation that captures the similarity of the two triangles quite well while distinguishing both from the square.

in the top image are more similar than the two triangles themselves. This is because the pixel values of the first and third image, due to their identical contrast and location of the triangle/square, are indeed more similar than those of the two triangle images.

In the middle and right columns of Fig. 4.26 we illustrate a two step procedure that generates the sort of discriminating feature transformation we are after. In the first part we shift perspective from the pixels themselves to the edge content at each pixel. As first detailed in Example 1.8, by taking edges instead of pixel values we significantly reduce the amount of information we must deal with in an image without destroying its identifying structures. In the middle column of the figure we show corresponding edge-detected images, in particular highlighting eight equally (angularly) spaced edge orientations, starting from 0 degrees (horizontal edges) with seven additional orientations at increments of 22.5 degrees, including 45 degrees (capturing the diagonal edges of the triangles) and 90 degrees (vertical edges). Clearly the edges retain distinguishing characteristics from each original image, while significantly reducing the amount of total information in each case.

We then make a normalized histogram of each image's edge content. That is, we make a vector consisting of the amount of each edge orientation found in the image

and normalize the resulting vector to have unit length. This is completely analogous to the bag of words feature representation described for text data previously and is often referred to as the bag of visual words or bag of features method [1, 2], with the counting of edge orientations being the analog of counting "words" in the case of text data. Here we also have a normalized histogram which represents an image grossly while ignoring the location and ordering of its information. However, as shown in the right panel of the figure, unlike raw pixel values these histogram feature vectors capture characteristic information about each image, with the top two triangle images having very similar histograms and both differing significantly from that of the third image of the square.

Example 4.10 Object detection

Generalizations of the previously described edge histogram concept are widely used as feature transformations for visual object detection. As detailed in Example 1.4, the task of object detection is a popular classification problem where objects of interest (e.g., faces) are located in an example image. While the basic principles which led to the consideration of an edge histogram still hold, example images for such a task are significantly more complicated than the simple geometric shapes shown in Fig. 4.26. In particular, preserving local information at smaller scales of an image is considerably more important. Thus a natural way to extend the edge histogram feature is to compute it not over the entire image, but by breaking the image into relatively small patches and computing an edge histogram of each patch, then concatenating the results. In Fig. 4.27 we show a diagram of a common variation of this technique often used in

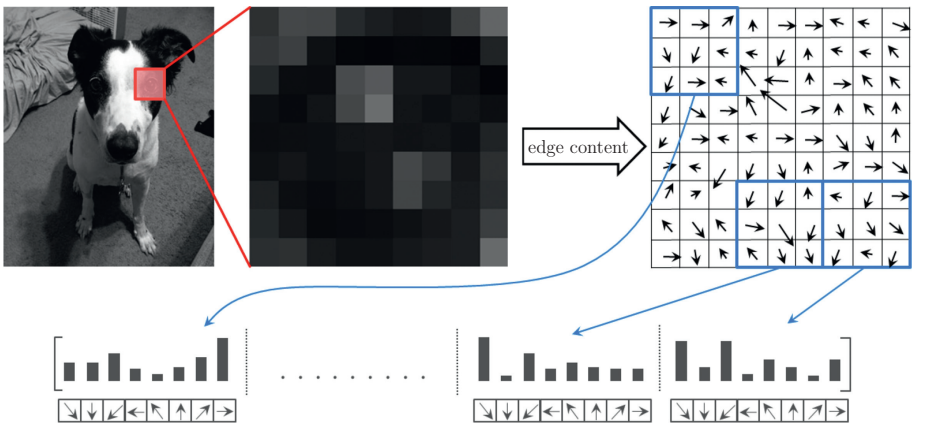

Fig. 4.27 A representation of the sort of generalized edge histogram feature transformation commonly used for object detection. An input image is broken down into small (here 9×9) blocks, and an edge histogram is computed on each of the smaller non-overlapping (here 3×3) patches that make up the block. The resulting histograms are then concatenated and normalized jointly, producing a feature vector for the entire block. Concatenating such block features by scanning the block window over the entire image gives the final feature vector.

Fig. 4.28 Example images taken from a large face detection dataset of (left panel) 3000 facial and (right panel) 7000 non-facial images (see text for further details). The facial images shown in this figure are taken from [2].

practice where we normalize neighboring histograms jointly in larger blocks (for further details see e.g., [3, 29, 65] and [4, 5] for extensions of this approach). Interestingly this sort of feature transformation can in fact be written out algebraically as a set of quadratic transformations of the input image [25].

To give a sense of just how much histogram-based features improve our ability to detect visual objects we now show the results of a simple experiment on a large face detection dataset. This data consists of 3000 cropped 28×28 (or dimension $N = 784$) images of faces (taken from [2]) and 7000 equal sized non-face images (taken from various images not containing faces), a sample of which is shown in Fig. 4.28.

We then compare the classification accuracy of the softmax classifier on this large training set of data using a) raw pixels and b) a popular histogram-based feature known as the histogram of oriented gradients (HoG) [29]. HoG features were extracted using the Vlfeat software library [67], providing a corresponding feature vector of each image in the dataset (of length $N = 496$). In Fig. 4.29 we show the resulting number of misclassifications per iteration of Newton's method applied to the raw pixel (black) and HoG feature (magenta) versions of data. While the raw images are not linearly separable, with over 300 misclassifications upon convergence of Newton's method, the HoG feature version of the data is perfectly separable by a hyperplane and presents zero misclassifications upon convergence.

4.6.3 Histogram features for audio data

Like images, raw audio signals are not discriminative enough to be used for audio-based classification tasks (e.g., speech recognition) and once again properly designed histogram-based features are used. In the case of an audio signal it is the histogram of its frequencies, otherwise known as its *spectrum,* that provides a robust summary of its contents. As illustrated in Fig. 4.30, the spectrum of an audio signal counts up

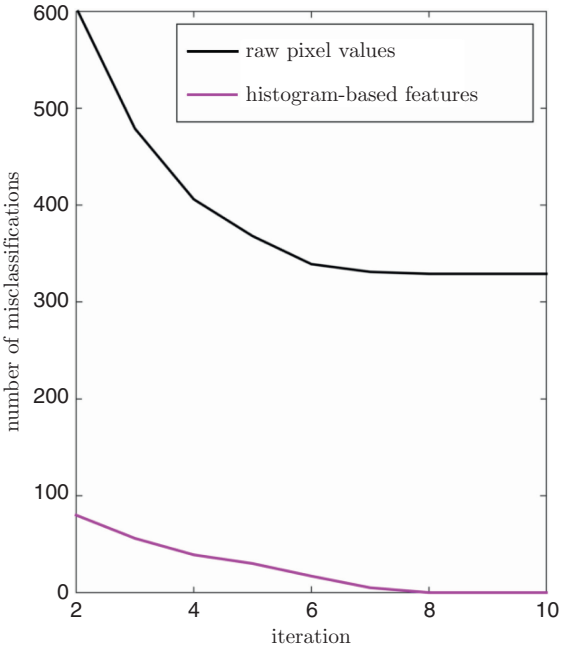

Fig. 4.29 An experiment comparing the classification efficacy of raw pixel versus histogram-based features for a large training set of face detection data (see text for further details). Employing the softmax classifier, the number of misclassifications per iteration of Newton's method is shown for both raw pixel data (in black) and histogram-based features (in magenta). While the raw data itself has overlapping classes, with a large number of misclassifications upon convergence of Newton's method, the histogram-based feature representation of the data is perfectly linearly separable with zero misclassifications upon convergence.

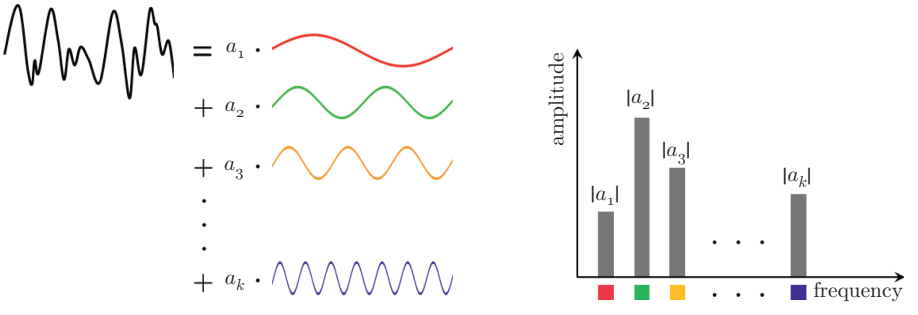

Fig. 4.30 A representation of an audio signal and its representation as a frequency histogram or spectrum. (left panel) A figurative audio signal can be decomposed as a linear combination of simple sinusoids with varying frequencies (or oscillations). (right panel) The frequency histogram then contains the strength of each sinusoid in the representation of the audio signal.

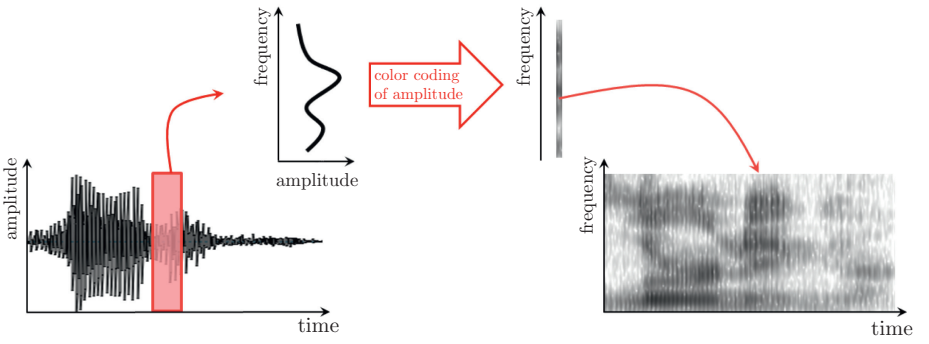

Fig. 4.31 A representation of histogram-based features for audio data. The original speech signal (shown on the left) is broken up into small (overlapping) windows whose frequency histograms are computed and stacked vertically to produce a spectrogram (shown on the right). Classification tasks like speech recognition are then performed using this feature representation, or a further refinement of it (see text for further details).

(in histogram fashion) the strength of each level of its frequency or oscillation. This is done by decomposing the speech signal over a basis of sine waves of ever increasing frequency, with the weights on each sinusoid representing the amount of that frequency in the original signal. Each oscillation level is analogous to an edge direction in the case of an image, or an individual word in the case of a BoW text feature.

Example 4.11 Speech recognition

In Example 4.10 we discussed how edge histograms computed on overlapping blocks of an image provide a useful feature transformation for object detection since they preserve characteristic local information. Likewise computing frequency histograms over overlapping windows of an audio signal (forming a "spectrogram" as illustrated in Fig. 4.31) produces a feature vector that preserves important local information as well, and is a common feature transformation used for speech recognition. Further processing of the windowed histograms, in order to e.g., emphasize the frequencies of sound best recognized by the human ear, are also commonly performed in practical implementations of this sort of feature transformation [40, 67].

4.7 Summary

In Section 4.1 we first described the fundamental cost function associated with linear two class classification: the perceptron. We then saw how to derive two convex and differentiable relatives of the basic perceptron, the softmax and squared margin perceptron cost functions. These two costs are often used in practice and, given their close resemblance, typically perform very similarly. We then saw in Sections 4.2 and 4.3 how these two cost functions can be derived classically, as logistic regression and

soft-margin support vector machines respectively, with the logistic regression "surface-fitting perspective" of the softmax perceptron being of particular value as it provides a second way of thinking about classification. Next, in Section 4.4 we discussed two approaches to multiclass classification, the multiclass softmax and one-versus-all (OvA) classifiers. Like the two commonly used two class cost functions, these two methods perform similarly in practice as well.

We then discussed in Section 4.5 how the design of proper feature transformations corresponds geometrically with finding features that produce a good nonlinear separator in the original feature space and, simultaneously, a good linear separator in the transformed feature space. In the final section we described common histogram-based features for text, image, and audio data types and how understanding of each guides both their representation as well as practical feature design for common classification problems.

4.8 Exercises

Section 4.1 exercises

Exercises 4.1 The perceptron cost is convex

In this exercise you will show that the original perceptron cost given in Equation (4.5) is convex using two steps.

a) Use the zeroth order definition of convexity (described in Appendix D) to show that $\max\left(0,\, -y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)$ is convex in both parameters $(b, \mathbf{w})$.

b) Use the zeroth order definition of convexity to show that if both $g(t)$ and $h(t)$ are convex, then so too is $g(t) + h(t)$. Use this to conclude that the perceptron cost is indeed convex.

Exercises 4.2 The softmax/logistic regression cost is convex

Show that the softmax/logistic regression cost function given in Equation (4.9) is convex by verifying that it satisfies the second order definition of convexity. *Hint: the Hessian, already given in Equation (4.13), is a weighted outer product matrix like the one described in Exercise 2.10.*

Exercises 4.3 Code up gradient descent for the softmax cost/logistic regression on a toy dataset

In this exercise you will code up gradient descent to minimize the softmax cost function on a toy dataset, reproducing the left panel of Fig. 4.3 in Example 4.1.

a) Verify the gradient of the softmax cost shown in Equation (4.12).

b) (optional) 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 in matrix-vector form as

$$\nabla g\left(\widetilde{\mathbf{w}}\right) = \widetilde{\mathbf{X}}\mathbf{r}, \tag{4.71}$$

where $\widetilde{\mathbf{X}}$ is the $(N+1) \times P$ matrix formed by stacking the P vectors $\widetilde{\mathbf{x}}_p$ column-wise, and where $\mathbf{r}$ is a $P \times 1$ vector based on the form of the gradient shown in Equation (4.12). Verify that this can be done and determine $\mathbf{r}$.

c) Code up gradient descent to minimize the softmax cost, reproducing the left panel of Fig. 4.3. This figure is generated via the wrapper *softmax_grad_demo_hw* using the dataset *imbalanced_2class.csv*. You must complete a short gradient descent function located within the wrapper which takes the form

$$\widetilde{\mathbf{w}} = \text{softmax\_grad}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \widetilde{\mathbf{w}}^0, \text{ alpha}\right). \tag{4.72}$$

Here $\widetilde{\mathbf{w}}$ is the optimal weights learned via gradient descent, $\widetilde{\mathbf{X}}$ is the input data matrix, $\mathbf{y}$ the output values, and $\widetilde{\mathbf{w}}^0$ the initial point.

Almost all of this function has already been constructed for you. For example, the step length is given and fixed for all iterations, etc., and you must 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.

Exercises 4.4 Code up Newton's method to learn a softmax/logistic regression classifier on a toy dataset

In this exercise you will code up Newton's method to minimize the softmax/logistic regression cost function on a toy dataset, producing a plot similar to the right panel of Fig. 4.3 in Example 4.1.

a) Verify that the Hessian of the softmax given in (4.13) is correct.

b) (optional) The gradient and Hessian can be written more efficiently for programming languages like Python and MATLAB/OCTAVE that have especially good implementations of matrix/vector operations by writing them more compactly. In particular the gradient can be written compactly as discussed in part b) of Exercise 4.3, and likewise the Hessian can be written more compactly as

$$\nabla^2 g\left(\widetilde{\mathbf{w}}\right) = \widetilde{\mathbf{X}}\text{diag}\left(\mathbf{r}\right)\widetilde{\mathbf{X}}^T, \tag{4.73}$$

where $\widetilde{\mathbf{X}}$ is the $(N+1) \times P$ matrix formed by stacking P data vectors $\widetilde{\mathbf{x}}_p$ column-wise, and where $\mathbf{r}$ is a $P \times 1$ vector based on the form of the Hessian shown in Equation (4.13). Verify that this can be done and determine $\mathbf{r}$. Note that for large datasets you do not want to explicitly form the matrix diag $(\mathbf{r})$, but compute $\widetilde{\mathbf{X}}$diag $(\mathbf{r})$ by broadcasting the multiplication of each entry of $\mathbf{r}$ across the columns of $\widetilde{\mathbf{X}}$.

c) Using the wrapper *softmax_Newton_demo_hw* code up Newton's method to minimize the softmax cost with the dataset *overlapping_2class.csv*. You must complete a short Newton's method function located within the wrapper,

$$\widetilde{\mathbf{w}} = \text{softmax\_newton}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \widetilde{\mathbf{w}}^0\right). \tag{4.74}$$

Here $\widetilde{\mathbf{w}}$ is the optimal weights learned via Newton's method, $\widetilde{\mathbf{X}}$ is the input data matrix, $\mathbf{y}$ the output values, and $\widetilde{\mathbf{w}}^0$ the initial point.

Almost all of this function has already been constructed for you and all you must do is enter the form of the Newton step of the associated cost function. All of the additional code necessary to generate the associated plot is already provided in the wrapper.

Exercises 4.5 The softmax cost and diverging weights with linearly separable data

Suppose that a two class dataset of P points is linearly separable, and that the pair of finite-valued parameters $(b, \mathbf{w})$ defines a separating hyperplane for the data.

a) Show while multiplying these weights by a positive constant $C > 1$, that as $(C \cdot b,\ C \cdot \mathbf{w})$ does not alter the equation of the separating hyperplane, the scaled parameters reduce the value of the softmax cost as $g\,(C \cdot b, C \cdot \mathbf{w}) < g\,(b, \mathbf{w})$ where g is the softmax cost in (4.9). *Hint: remember from (4.3) that if the point $\boldsymbol{x}_p$ is classified correctly then* $-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right) < 0$.

b) Using part a) describe how, in minimizing the softmax cost over a linearly separable dataset, it is possible for the parameters to grow infinitely large. Why do you think this is a problem, practically speaking?

There are several simple ways to prevent this problem: one is to add a stopping condition that halts gradient descent/Newton's method if the parameters $(b,\ \mathbf{w})$ become larger than a preset maximum value. A second option is to add an ℓ_2 regularizer (see Section 3.3.2) to the softmax cost with a small penalty parameter λ, since adding the regularizer $\lambda \left\|\mathbf{w}\right\|_2^2$ will stop $\mathbf{w}$ from growing too large (since otherwise the value of the regularized softmax cost will grow to infinity).

Exercises 4.6 The margin cost function is convex

In this exercise you will show that the margin and squared margin cost functions are convex using two steps.

a) Use the zeroth order definition of convexity (described in Appendix D) to show that $\max\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)$ is convex in both parameters $(b, \mathbf{w})$. Do the same for the squared margin $\max^2\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right)$.

b) Use the zeroth order definition of convexity to show that if both $g\,(t)$ and $h\,(t)$ are convex, then so too is $g\,(t) + h\,(t)$. Use this to conclude that the margin and squared margin perceptron costs are indeed convex.

Exercises 4.7 Code up gradient descent to learn a squared margin classifier

In this exercise you will code up gradient descent for minimizing the squared margin cost function discussed in Section 4.1.4.

a) Verify that the gradient of the squared margin cost is given as in Equation (4.21).

b) (optional) 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 in matrix-vector form as

$$\nabla g\left(\widetilde{\mathbf{w}}\right) = -2\widetilde{\mathbf{X}}\text{diag}\left(\mathbf{y}\right)\textbf{max}\left(\mathbf{0}_{P\times 1},\,\mathbf{1}_{P\times 1} - \text{diag}\left(\mathbf{y}\right)\widetilde{\mathbf{X}}^{T}\widetilde{\mathbf{w}}\right), \tag{4.75}$$

where **max** is the maximum function applied entrywise, $\widetilde{\mathbf{X}}$ is the $(N+1)\times P$ matrix formed by stacking P data vectors $\widetilde{\mathbf{x}}_p$ column-wise. Verify that this can be done. (Note that for large datasets you do not want to explicitly form the matrix diag $(\mathbf{y})$, but compute $\widetilde{\mathbf{X}}$diag $(\mathbf{y})$ by broadcasting the multiplication of each entry of $\mathbf{y}$ across the columns of $\widetilde{\mathbf{X}}$.)

c) Code up gradient descent to minimize the squared margin cost, reproducing the left panel of Fig. 4.3. This figure is generated via the wrapper *squared_margin_grad_demo_hw* using the dataset *imbalanced_2class.csv*. You must complete a short gradient descent function located within the wrapper which takes the form

$$\widetilde{\mathbf{w}} = \text{squared\_margin\_grad}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \widetilde{\mathbf{w}}^{0},\,\text{alpha}\right). \tag{4.76}$$

Here $\widetilde{\mathbf{w}}$ is the optimal weights learned via gradient descent, $\widetilde{\mathbf{X}}$ is the input data matrix, $\mathbf{y}$ the output values, and $\widetilde{\mathbf{w}}^{0}$ the initial point.

Almost all of this function has already been constructed for you. For example, the step length is given and fixed for all iterations, etc., and you must 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.

Exercises 4.8 Code up Newton's method to learn a squared margin classifier

In this exercise you will code up Newton's method to minimize the squared margin cost function on a toy dataset, producing a plot similar to the right panel of Fig. 4.5 in Example 4.2.

a) Code up Newton's method to minimize the squared margin cost. You may use the wrapper *squared_margin_Newton_demo_hw* with the dataset *overlapping_2class.csv*. You must complete a short Newton's method function located within the wrapper, which takes the form

$$\widetilde{\mathbf{w}} = \text{squared\_margin\_newton}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \widetilde{\mathbf{w}}^{0}\right). \tag{4.77}$$

Here $\widetilde{\mathbf{w}}$ is the optimal weights learned via Newton's method, $\widetilde{\mathbf{X}}$ is the input data matrix, $\mathbf{y}$ the output values, and $\widetilde{\mathbf{w}}^{0}$ the initial point.

Almost all of this function has already been constructed for you and all you must do is enter the form of the Newton step. All of the additional code necessary to generate the associated plot is already provided in the wrapper.

Exercises 4.9 Perform classification on the breast cancer dataset

Compare the efficacy of the softmax and squared margin costs in distinguishing healthy from cancerous tissue using the entire breast cancer dataset as training data, located in

breast_cancer_dataset.csv, first discussed in Example 4.3. This dataset consists of $P = 699$ data points, with each data point having nine medically valuable features (i.e., $N = 9$) which you may read about by reviewing the readme file *breast_cancer_readme.txt*. Note that for simplicity we have removed the sixth feature from the original version of this data, taken from [47], due to its absence from many of the data points.

To compare the two cost functions create a plot like the one shown in Fig. 4.8, which compares the number of misclassifications per iteration of Newton's method as applied to minimize each cost function over the data (note: depending on your initialization it could take between 10–20 iterations to achieve the results shown in this figure). As mentioned in footnote 6, you need to be careful not to overflow the exponential function used with the softmax cost here. In particular make sure to choose a small initial point for your Newton's method algorithm with the softmax cost.

Exercises 4.10 Perform classification on histogram-based features for face detection

Compare the efficacy of the softmax and squared margin costs in distinguishing face from non-face images using the histogram-based feature face detection training dataset, located in *feat_face_data.csv*, first discussed in Example 4.3 and later in Example 4.10. This set of training data consists of $P = 10\,000$ feature data points from 3000 face images (taken from [2]) and 7000 non-face images like those shown in Fig. 4.28. Here each data point is a histogram-based feature vector of length $N = 496$ taken from a corresponding 28×28 grayscale image.

To compare the two cost functions create a plot like the one shown in Fig. 4.8 which compares the number of misclassifications per iteration of Newton's method as applied to minimize each cost function over the data. However, in this case use gradient descent to minimize both cost functions. You may determine a fixed step size for each cost function by trial and error, or by simply using the "conservatively optimal" fixed step lengths shown in Table 8.1 (which are guaranteed to cause gradient descent to converge to a minimum in each instance).

As mentioned in footnote 6, you need to be careful here not to overflow the exponential function used with the softmax cost, in particular make sure to choose a small initial point. In calculating the value of the softmax cost at each iteration you may find it useful to include a conditional statement that deals with the possibility of e^s overflowing for large values of s, which will cause $\log(1 + e^s)$ to be returned as ∞ (as the computer will see it as $\log(1 + \infty)$), by simply returning s since for large values $s \approx \log(1 + e^s)$.

Section 4.2 exercises

Exercises 4.11 Alternative form of logistic regression

In this section we saw how the desire for having the following approximation for the pth data point $(\mathbf{x}_p, y_p)$:

$$\tanh\left(y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)\right) \approx 1, \tag{4.78}$$

led us to forming the softmax perceptron cost function $h_1(b, \mathbf{w}) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)}\right)$.

a) Following a similar set of steps, show that Equation (4.78) can be used to arrive at the related cost function given by $h_2(b, \mathbf{w}) = \sum_{p=1}^{P} e^{-y_p\left(b + \mathbf{x}_p^T\mathbf{w}\right)}$.

b) Code up gradient descent to minimize both cost functions using the two-dimensional dataset shown in Fig. 4.32 (located in the data file *exp_vs_log_data.csv*). After performing gradient descent on each, the final separators provided by h_1 and h_2 are shown in black and magenta respectively.

Using the wrapper *exp_vs_log_demo_hw* you must complete two short gradient descent functions corresponding to h_1 and h_2 respectively:

$$\widetilde{\mathbf{w}} = \text{grad\_descent\_soft\_cost}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \widetilde{\mathbf{w}}^0, \text{alpha}\right) \tag{4.79}$$

and

$$\widetilde{\mathbf{w}} = \text{grad\_descent\_exp\_cost}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \widetilde{\mathbf{w}}^0, \text{alpha}\right). \tag{4.80}$$

Here $\widetilde{\mathbf{w}}$ is the optimal weights, $\widetilde{\mathbf{X}}$ is the input data matrix, $\mathbf{y}$ the output values, and $\widetilde{\mathbf{w}}^0$ the initial point.

Almost all of this function has already been constructed for you. For example, the step length is fixed for all iterations, etc., and you must only enter the gradient of each associated cost function. All of the additional code necessary to generate the associated plot is already provided in the wrapper.

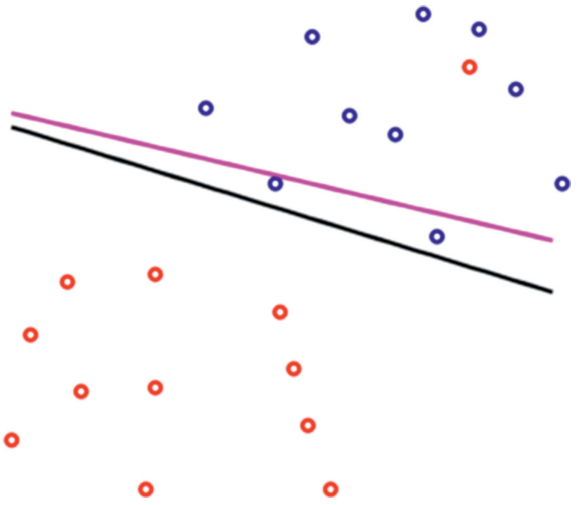

Fig. 4.32 A two-dimensional dataset used for Exercise 4.11. See text for details.

c) Compare the two separating hyperplanes found in the previous part of this exercise. Which cost function does a better job at separating the two classes of data? Why? *Hint: note the error contribution of the outlier to each cost function.*

Exercises 4.12 Probabilistic perspective of logistic regression

In the previous chapter (in Section 3.3.1) we first introduced logistic regression in the context of its original application: modeling population growth. We then followed this geometric perspective to re-derive the softmax cost function in the instance of classification.

In the classification setting, logistic regression may also be derived from a probabilistic perspective.[22] Doing so one comes to the following cost function for logistic regression:

$$h\left(b, \mathbf{w}\right) = -\sum_{p=1}^{P} \bar{y}_p \log\, \sigma\left(b + \mathbf{x}_p^T \mathbf{w}\right) + \left(1 - \bar{y}_p\right) \log\left(1 - \sigma\left(b + \mathbf{x}_p^T \mathbf{w}\right)\right), \tag{4.81}$$

where the modified labels $\bar{y}_p$ are defined as

$$\bar{y}_p = \begin{cases} 0 & \text{if } y_p = -1 \\ 1 & \text{if } y_p = +1. \end{cases} \tag{4.82}$$

Show that the cost function $h\left(b, \mathbf{w}\right)$, also referred to as the *cross-entropy* cost for logistic regression, is equivalent to the softmax cost function $g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{x}_p^T \mathbf{w}\right)}\right)$.

Hint: this can be done in cases, i.e., suppose $y_p = +1$, show that the corresponding summand of the softmax cost becomes that of the cross-entropy cost when substituting $\bar{y}_p = 1$.

Section 4.3 exercises

Exercises 4.13 Code up gradient descent for the soft-margin SVM cost

Extend Exercise 4.7 by using the wrapper and dataset discussed there to test the performance of the soft-margin SVM classifier using the squared margin perceptron as the base cost function, i.e., an ℓ_2 regularized form of the squared margin cost function. How does the gradient change due to the addition of the regularizer? Input those changes into the gradient descent function described in that exercise and run the wrapper for values of $\lambda \in \left[10^{-2}, 10^{-1}, 1, 10\right]$. Describe the consequences of choosing each in terms of the final classification accuracy.

[22] Using labels $\bar{y}_p \in \{0, 1\}$ this is done by assuming a sigmoidal conditional probability for the point $\mathbf{x}_p$ to have label $\bar{y}_p = 1$ as $p\left(\mathbf{x}_p\right) = \sigma\left(b + \mathbf{x}_p^T \mathbf{w}\right) = \frac{1}{1 + e^{-\left(b + \mathbf{x}_p^T \mathbf{w}\right)}}$. The cross-entropy cost in (4.81) is then found by maximizing the so-called log likelihood function associated to this choice of model (see e.g., [52] for further details).

Section 4.4 exercises

Exercises 4.14 One-versus-all classification

In this exercise you will reproduce the result of performing one-versus-all classification on the $C = 4$ dataset shown in Fig. 4.19.

a) Use the Newton's method subfunction produced in (4.83) to complete the one-versus-all wrapper *one_versus_all_demo_hw* to classify the $C = 4$ class dataset *four_class_data.csv* shown in Fig. 4.19. With your Newton's method module you must complete a short subfunction in this wrapper called

$$\widetilde{\mathbf{W}} = \text{learn\_separators}\left(\widetilde{\mathbf{X}}, \mathbf{y}\right), \tag{4.83}$$

that enacts the OvA framework, outputting learned weights for all C separators (i.e., this should call your Newton's method module C times, once for each individual two class classifier). Here $\widetilde{\mathbf{W}} = \begin{bmatrix} \widetilde{\mathbf{w}}_1 & \widetilde{\mathbf{w}}_2 & \cdots & \widetilde{\mathbf{w}}_C \end{bmatrix}$ is an $(N+1) \times C$ matrix of weights, where $\widetilde{\mathbf{w}}_c$ is the compact weight/bias vector associated with the cth classifier, $\widetilde{\mathbf{X}}$ is the input data matrix, $\mathbf{y}$ the associated labels. All of the additional code necessary to generate the associated plot is already provided in the wrapper.

Exercises 4.15 Code up gradient descent for the multiclass softmax classifier

In this exercise you will code up gradient descent to minimize the multiclass softmax cost function on a toy dataset, reproducing the result shown in Fig. 4.20.

a) Confirm that the gradient of the multiclass softmax perceptron is given by Equation (4.57) for each class $c = 1, \ldots, C$.

b) Code up gradient descent to minimize the multiclass softmax perceptron, reproducing the result shown for the $C = 4$ class dataset shown in Fig. 4.20. This figure is generated via the wrapper *softmax_multiclass_grad_hw* and you must complete a short gradient descent function located within which takes the form

$$\widetilde{\mathbf{W}} = \text{softmax\_multiclass\_grad}\left(\widetilde{\mathbf{X}}, \mathbf{y}, \widetilde{\mathbf{W}}^0, \text{ alpha}\right). \tag{4.84}$$

Here $\widetilde{\mathbf{W}} = \begin{bmatrix} \widetilde{\mathbf{w}}_1 & \widetilde{\mathbf{w}}_2 & \cdots & \widetilde{\mathbf{w}}_C \end{bmatrix}$ is an $(N+1) \times C$ matrix of weights, where $\widetilde{\mathbf{w}}_c$ is the compact bias/weight vector associated with the cth classifier, $\widetilde{\mathbf{X}}$ is the input data matrix, $\mathbf{y}$ the associated labels, and $\widetilde{\mathbf{W}}^0$ the initialization for the weights. Almost all of this function has already been constructed for you. For example, the step length is fixed for all iterations, etc., and you must 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.

Exercises 4.16 Handwritten digit recognition

In this exercise you will perform $C = 10$ multiclass classification for handwritten digit recognition, as described in Example 4.4 , employing the OvA multiclass classification

framework. Employ the softmax cost with gradient descent or Newton's method to solve each of the two-class subproblems.

a) Train your classifier on the training set located in *MNIST_training_data.csv*, that contains $P = 60\,000$ examples of handwritten digits $0 - 9$ (all examples are vectorized grayscale images of size 28×28 pixels). Report the accuracy of your trained model on this training set.

b) Using the weights learned from part a) report the accuracy of your model on a new test dataset of handwritten digits located in *MNIST_testing_data.csv*. This contains $P = 10\,000$ new examples of handwritten digits that were not used in the training of your model.

Exercises 4.17 Show the multiclass softmax reduces to two-class softmax when $C = 2$

Show that the multiclass softmax cost function given in (4.54) reduces to the two class softmax cost in (4.9) when $C = 2$.

Exercises 4.18 Calculating the Hessian of the multiclass softmax cost

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

Section 4.5 exercises

Exercises 4.19 Learn a quadratic separator

Shown in the left panel of Fig. 4.33 are $P = 150$ data points which, by visual inspection, can be seen to be separable not by a line but by some quadratic boundary. In other words, points from each class all lie either above or below a quadratic of the form $f(x_1, x_2) = b + x_1^2 w_1 + x_2 w_2 = 0$ in the original feature space, i.e.,

$$\begin{aligned} b + x_{1,p}^2 w_1 + x_{2,p} w_2 &> 0 \text{ if } y_p = 1 \\ b + x_{1,p}^2 w_1 + x_{2,p} w_2 &< 0 \text{ if } y_p = -1. \end{aligned} \tag{4.85}$$

As illustrated in the right panel of the figure, this quadratic boundary is simultaneously a linear boundary in the *feature space* defined by the quadratic *feature transformation* or *mapping* of $(x_1, x_2) \longrightarrow \left(x_1^2, x_2\right)$.

Using any cost function and the dataset *quadratic_classification.csv*, reproduce the result shown in the figure by learning the proper parameters b and $\mathbf{w}$ for the quadratic boundary, and by plotting the data and its associated separator in both the original and transformed feature spaces.

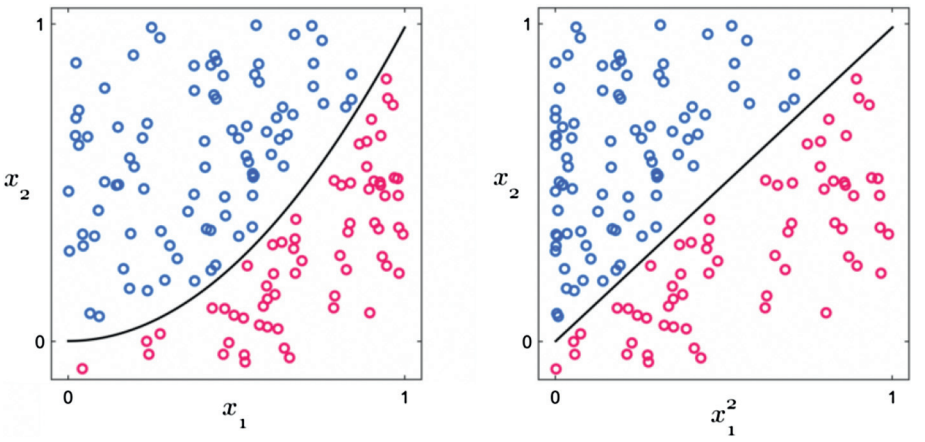

Fig. 4.33 Data separable via a quadratic boundary. (left panel) A quadratic boundary given as $b + x_1^2 w_1 + x_2 w_2 = 0$ can perfectly separate the two classes. (right panel) Finding the weights associated to this quadratic boundary in the original feature space is equivalent to finding a line to separate the data in the transformed feature space where the input has undergone a quadratic feature transformation $(x_1, x_2) \longrightarrow \left(x_1^2, x_2\right)$.

Section 4.6 exercises

Exercises 4.20 Perform spam detection using BoW and spam-specific features

Compare the efficacy of using various combinations of features to perform spam detection on a real dataset of emails, as described in Example 4.9. Your job is to reproduce as well as possible the final result (i.e., the final number of misclassifications) shown in Fig. 4.24, using only the squared margin cost and gradient descent (instead of the softmax cost and Newton's method as shown there). You may determine a fixed step size by trial and error or by using the "conservatively optimal" fixed step length shown in Table 8.1 (which is guaranteed to cause gradient descent to converge to a minimum).

Use the entire dataset, taken from [47] and consisting of features taken from 1813 spam and 2788 real email messages (for a total of $P = 4601$ data points), as your training data. The features for each data point include: 48 BoW features, six character frequency features, and three spam-targeted features (further details on these features can be found by reviewing the readme file *spambase_data_readme.txt*). This dataset may be found in *spambase_data.csv*. Note that you may find it useful to rescale the final two spam-targeted features by taking their natural log, as they are considerably larger than the other features.

Exercises 4.21 Comparing pixels and histogram-based features for face detection

In this exercise you will reproduce as well as possible the result shown in Fig. 4.29, using a cost function and descent algorithm of your choosing, which compares the classification efficacy of raw pixel features versus a set of standard histogram-based features on a

large training set of face detection data (described in Example 4.10 and Exercise 4.10). Note that it may take between 10–20 Newton steps to achieve around the same number of misclassifications as shown in this figure depending on your initialization. The raw pixel features are located in *raw_face_data.csv* and the histogram-based features may be found in *feat_face_data.csv*.