the steps below, show that the expected in-sample error of linear regression with respect to $\mathcal{D}$ is given by

$$\mathbb{E}_{\mathcal{D}}[E_{\text{in}}(\mathbf{w}_{\text{lin}})] = \sigma^2\left(1 - \frac{d+1}{N}\right).$$

(a) Show that the in-sample estimate of $\mathbf{y}$ is given by $\hat{\mathbf{y}} = \mathrm{X}\mathbf{w}^* + \mathrm{H}\boldsymbol{\epsilon}$.

(b) Show that the in-sample error vector $\hat{\mathbf{y}} - \mathbf{y}$ can be expressed by a matrix times $\boldsymbol{\epsilon}$. What is the matrix?

(c) Express $E_{\text{in}}(\mathbf{w}_{\text{lin}})$ in terms of $\boldsymbol{\epsilon}$ using (b), and simplify the expression using Exercise 3.3(c).

(d) Prove that $\mathbb{E}_{\mathcal{D}}[E_{\text{in}}(\mathbf{w}_{\text{lin}})] = \sigma^2\left(1 - \frac{d+1}{N}\right)$ using (c) and the independence of $\epsilon_1, \cdots, \epsilon_N$. *[Hint: The sum of the diagonal elements of a matrix (the trace) will play a role. See Exercise 3.3(d).]*

For the expected out-of-sample error, we take a special case which is easy to analyze. Consider a test data set $\mathcal{D}_{\text{test}} = \{(\mathbf{x}_1, y_1'), \ldots, (\mathbf{x}_N, y_N')\}$, which shares the same input vectors $\mathbf{x}_n$ with $\mathcal{D}$ but with a different realization of the noise terms. Denote the noise in y_n' as ϵ_n' and let $\boldsymbol{\epsilon}' = [\epsilon_1', \epsilon_2', \ldots, \epsilon_N']^{\mathrm{T}}$. Define $E_{\text{test}}(\mathbf{w}_{\text{lin}})$ to be the average squared error on $\mathcal{D}_{\text{test}}$.

(e) Prove that $\mathbb{E}_{\mathcal{D},\boldsymbol{\epsilon}'}[E_{\text{test}}(\mathbf{w}_{\text{lin}})] = \sigma^2\left(1 + \frac{d+1}{N}\right)$.

The special test error E_{test} is a very restricted case of the general out-of-sample error. Some detailed analysis shows that similar results can be obtained for the general case, as shown in Problem 3.11.

Figure 3.4 illustrates the learning curve of linear regression under the assumptions of Exercise 3.4. The best possible linear fit has expected error σ^2. The expected in-sample error is smaller, equal to $\sigma^2(1 - \frac{d+1}{N})$ for $N \geq d+1$. The learned linear fit has eaten into the in-sample noise as much as it could with the $d+1$ degrees of freedom that it has at its disposal. This occurs because the fitting cannot distinguish the noise from the 'signal.' On the other hand, the expected out-of-sample error is $\sigma^2(1 + \frac{d+1}{N})$, which is more than the unavoidable error of σ^2. The additional error reflects the drift in $\mathbf{w}_{\text{lin}}$ due to fitting the in-sample noise.

3.3 Logistic Regression

The core of the linear model is the 'signal' $s = \mathbf{w}^{\mathrm{T}}\mathbf{x}$ that combines the input variables linearly. We have seen two models based on this signal, and we are now going to introduce a third. In linear regression, the signal itself is taken as the output, which is appropriate if you are trying to predict a real response that could be unbounded. In linear classification, the signal is thresholded at zero to produce a ± 1 output, appropriate for binary decisions. A third possibility, which has wide application in practice, is to output a *probability*,

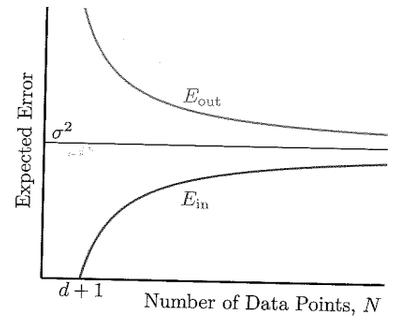


Figure 3.4: The learning curve for linear regression.

a value between 0 and 1. Our new model is called *logistic* regression. It has similarities to both previous models, as the output is real (like regression) but bounded (like classification).

Example 3.2 (Prediction of heart attacks)**.** Suppose we want to predict the occurrence of heart attacks based on a person's cholesterol level, blood pressure, age, weight, and other factors. Obviously, we cannot predict a heart attack with any certainty, but we may be able to predict how likely it is to occur given these factors. Therefore, an output that varies continuously between 0 and 1 would be a more suitable model than a binary decision. The closer y is to 1, the more likely that the person will have a heart attack. □

3.3.1 Predicting a Probability

Linear classification uses a hard threshold on the signal $s = \mathbf{w}^{\mathrm{T}}\mathbf{x}$,

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}),$$

while linear regression uses no threshold at all,

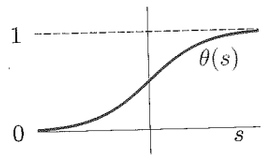
$$h(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}.$$

In our new model, we need something in between these two cases that smoothly restricts the output to the probability range $[0, 1]$. One choice that accomplishes this goal is the logistic regression model,

$$h(\mathbf{x}) = \theta(\mathbf{w}^{\mathrm{T}}\mathbf{x}),$$

where θ is the so-called *logistic* function $\theta(s) = \frac{e^s}{1+e^s}$ whose output is between 0 and 1.

The output can be interpreted as a probability for a binary event (heart attack or no heart attack, digit '1' versus digit '5', etc.). Linear classification also deals with a binary event, but the difference is that the 'classification' in logistic regression is allowed to be uncertain, with intermediate values between 0 and 1 reflecting this uncertainty. The logistic function θ is referred to as a *soft threshold*, in contrast to the hard threshold in classification. It is also called a *sigmoid* because its shape looks like a flattened out 's'.



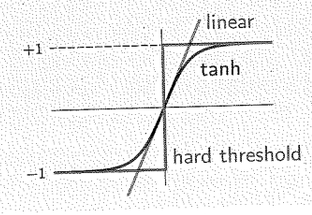
Exercise 3.5

Another popular soft threshold is the hyperbolic tangent

$$\tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}.$$

(a) How is tanh related to the logistic function θ? *[Hint: shift and scale]*

(b) Show that $\tanh(s)$ converges to a hard threshold for large $|s|$, and converges to no threshold for small $|s|$. *[Hint: Formalize the figure below.]*



The specific formula of $\theta(s)$ will allow us to define an error measure for learning that has analytical and computational advantages, as we will see shortly. Let us first look at the target that logistic regression is trying to learn. The target is a probability, say of a patient being at risk for heart attack, that depends on the input $\mathbf{x}$ (the characteristics of the patient). Formally, we are trying to learn the target function

$$f(\mathbf{x}) = \mathbb{P}[y = +1 \mid \mathbf{x}].$$

The data does not give us the value of f explicitly. Rather, it gives us samples generated by this probability, e.g., patients who had heart attacks and patients who didn't. Therefore, the data is in fact generated by a noisy target $P(y \mid \mathbf{x})$,

$$P(y \mid \mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{for } y = +1; \\ 1 - f(\mathbf{x}) & \text{for } y = -1. \end{cases} \tag{3.7}$$

To learn from such data, we need to define a proper error measure that gauges how close a given hypothesis h is to f in terms of these noisy ± 1 examples.

Error measure. The standard error measure $\mathsf{e}(h(\mathbf{x}), y)$ used in logistic regression is based on the notion of *likelihood*; how 'likely' is it that we would get this output y from the input $\mathbf{x}$ if the target distribution $P(y \mid \mathbf{x})$ was indeed captured by our hypothesis $h(\mathbf{x})$? Based on (3.7), that likelihood would be

$$P(y \mid \mathbf{x}) = \begin{cases} h(\mathbf{x}) & \text{for } y = +1; \\ 1 - h(\mathbf{x}) & \text{for } y = -1. \end{cases}$$

We substitute for $h(\mathbf{x})$ by its value $\theta(\mathbf{w}^{\mathrm{T}}\mathbf{x})$, and use the fact that $1 - \theta(s) = \theta(-s)$ (easy to verify) to get

$$P(y \mid \mathbf{x}) = \theta(y\, \mathbf{w}^{\mathrm{T}}\mathbf{x}). \tag{3.8}$$

One of our reasons for choosing the mathematical form $\theta(s) = e^s/(1 + e^s)$ is that it leads to this simple expression for $P(y \mid \mathbf{x})$.

Since the data points $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ are independently generated, the probability of getting all the y_n's in the data set from the corresponding $\mathbf{x}_n$'s would be the product

$$\prod_{n=1}^{N} P(y_n \mid \mathbf{x}_n).$$

The method of *maximum likelihood* selects the hypothesis h which maximizes this probability.[3] We can equivalently minimize a more convenient quantity,

$$-\frac{1}{N} \ln \left(\prod_{n=1}^{N} P(y_n \mid \mathbf{x}_n) \right) = \frac{1}{N} \sum_{n=1}^{N} \ln \left(\frac{1}{P(y_n \mid \mathbf{x}_n)} \right),$$

since '$-\frac{1}{N} \ln(\cdot)$' is a monotonically decreasing function. Substituting with Equation (3.8), we would be minimizing

$$\frac{1}{N} \sum_{n=1}^{N} \ln \left(\frac{1}{\theta(y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n)} \right)$$

with respect to the weight vector $\mathbf{w}$. The fact that we are *minimizing* this quantity allows us to treat it as an 'error measure.' Substituting the functional form for $\theta(y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n)$ produces the in-sample error measure for logistic regression,

$$E_{\mathrm{in}}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \ln \left(1 + e^{-y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n} \right). \tag{3.9}$$

The implied pointwise error measure is $\mathsf{e}(h(\mathbf{x}_n), y_n) = \ln(1 + e^{-y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n})$. Notice that this error measure is small when $y_n \mathbf{w}^{\mathrm{T}}\mathbf{x}_n$ is large and *positive*, which would imply that $\mathrm{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) = y_n$. Therefore, as our intuition would expect, the error measure encourages $\mathbf{w}$ to 'classify' each $\mathbf{x}_n$ correctly.

[3] Although the method of maximum likelihood is intuitively plausible, its rigorous justification as an inference tool continues to be discussed in the statistics community.

Exercise 3.6 [Cross-entropy error measure]

(a) More generally, if we are learning from ± 1 data to predict a noisy target $P(y \mid \mathbf{x})$ with candidate hypothesis h, show that the maximum likelihood method reduces to the task of finding h that minimizes

$$E_{\text{in}}(\mathbf{w}) = \sum_{n=1}^{N} [\![y_n = +1]\!] \ln \frac{1}{h(\mathbf{x}_n)} + [\![y_n = -1]\!] \ln \frac{1}{1 - h(\mathbf{x}_n)}.$$

(b) For the case $h(\mathbf{x}) = \theta(\mathbf{w}^{\mathsf{T}}\mathbf{x})$, argue that minimizing the in-sample error in part (a) is equivalent to minimizing the one in (3.9).

For two probability distributions $\{p, 1-p\}$ and $\{q, 1-q\}$ with binary outcomes, the cross-entropy (from information theory) is

$$p \log \frac{1}{q} + (1-p) \log \frac{1}{1-q}.$$

The in-sample error in part (a) corresponds to a cross-entropy error measure on the data point $(\mathbf{x}_n, y_n)$, with $p = [\![y_n = +1]\!]$ and $q = h(\mathbf{x}_n)$.

For linear classification, we saw that minimizing E_{in} for the perceptron is a combinatorial optimization problem; to solve it, we introduced a number of algorithms such as the perceptron learning algorithm and the pocket algorithm. For linear regression, we saw that training can be done using the analytic pseudo-inverse algorithm for minimizing E_{in} by setting $\nabla E_{\text{in}}(\mathbf{w}) = \mathbf{0}$. These algorithms were developed based on the specific form of linear classification or linear regression, so none of them would apply to logistic regression.

To train logistic regression, we will take an approach similar to linear regression in that we will try to set $\nabla E_{\text{in}}(\mathbf{w}) = \mathbf{0}$. Unfortunately, unlike the case of linear regression, the mathematical form of the gradient of E_{in} for logistic regression is not easy to manipulate, so an analytic solution is not feasible.

Exercise 3.7

For logistic regression, show that

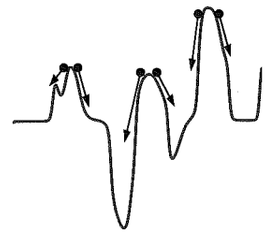
$$\begin{aligned} \nabla E_{\text{in}}(\mathbf{w}) &= -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\mathsf{T}} \mathbf{x}_n}} \\ &= \frac{1}{N} \sum_{n=1}^{N} -y_n \mathbf{x}_n \theta(-y_n \mathbf{w}^{\mathsf{T}} \mathbf{x}_n). \end{aligned}$$

Argue that a 'misclassified' example contributes more to the gradient than a correctly classified one.

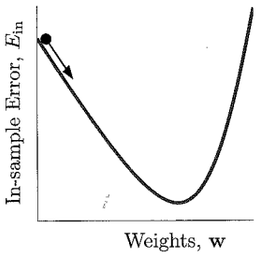
Instead of analytically setting the gradient to zero, we will *iteratively* set it to zero. To do so, we will introduce a new algorithm, *gradient descent*. Gradient

descent is a very general algorithm that can be used to train many other learning models with smooth error measures. For logistic regression, gradient descent has particularly nice properties.

3.3.2 Gradient Descent



Gradient descent is a general technique for minimizing a twice-differentiable function, such as $E_{\text{in}}(\mathbf{w})$ in logistic regression. A useful physical analogy of gradient descent is a ball rolling down a hilly surface. If the ball is placed on a hill, it will roll down, coming to rest at the bottom of a valley. The same basic idea underlies gradient descent. $E_{\text{in}}(\mathbf{w})$ is a 'surface' in a high-dimensional space. At step 0, we start somewhere on this surface, at $\mathbf{w}(0)$, and try to roll down this surface, thereby decreasing E_{in}. One thing which you immediately notice from the physical analogy is that the ball will not necessarily come to rest in the lowest valley of the entire surface. Depending on where you start the ball rolling, you will end up at the bottom of one of the valleys – a *local minimum*. In general, the same applies to gradient descent. Depending on your starting weights, the path of descent will take you to a local minimum in the error surface.



A particular advantage for logistic regression with the cross-entropy error is that the picture looks much nicer. There is only one valley! So, it does not matter where you start your ball rolling, it will always roll down to the same (unique) *global minimum*. This is a consequence of the fact that $E_{\text{in}}(\mathbf{w})$ is a *convex* function of $\mathbf{w}$, a mathematical property that implies a single 'valley' as shown to the right. This means that gradient descent will not be trapped in local minima when minimizing such convex error measures.[4]

Let's now determine how to 'roll' down the E_{in}-surface. We would like to take a step in the direction of steepest descent, to gain the biggest bang for our buck. Suppose that we take a small step of size η in the direction of a unit vector $\hat{\mathbf{v}}$. The new weights are $\mathbf{w}(0) + \eta\hat{\mathbf{v}}$. Since η is small, using the Taylor expansion to first order, we compute the change in E_{in} as

$$\begin{aligned} \Delta E_{\text{in}} &= E_{\text{in}}(\mathbf{w}(0) + \eta\hat{\mathbf{v}}) - E_{\text{in}}(\mathbf{w}(0)) \\ &= \eta \nabla E_{\text{in}}(\mathbf{w}(0))^{\mathsf{T}} \hat{\mathbf{v}} + O(\eta^2) \\ &\geq -\eta \|\nabla E_{\text{in}}(\mathbf{w}(0))\|, \end{aligned}$$

[4] In fact, the squared in-sample error in linear regression is also convex, which is why the analytic solution found by the pseudo-inverse is guaranteed to have optimal in-sample error.

where we have ignored the small term $O(\eta^2)$. Since $\hat{\mathbf{v}}$ is a unit vector, equality holds if and only if

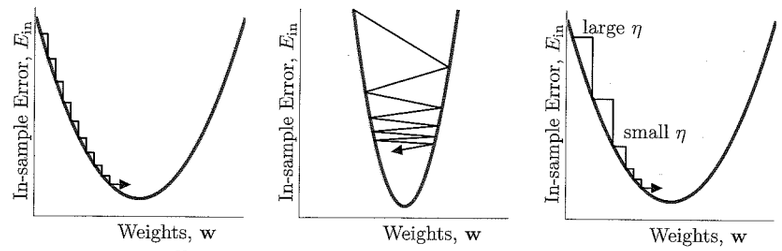
$$\hat{\mathbf{v}} = -\frac{\nabla E_{\text{in}}(\mathbf{w}(0))}{\|\nabla E_{\text{in}}(\mathbf{w}(0))\|}. \tag{3.10}$$

This direction, specified by $\hat{\mathbf{v}}$, leads to the largest decrease in E_{in} for a given step size η.

Exercise 3.8

The claim that $\hat{\mathbf{v}}$ is the direction which gives largest decrease in E_{in} only holds for small η. Why?

There is nothing to prevent us from continuing to take steps of size η, re-evaluating the direction $\hat{\mathbf{v}}_t$ at each iteration $t = 0, 1, 2, \ldots$. How large a step should one take at each iteration? This is a good question, and to gain some insight, let's look at the following examples.



η too small | η too large | variable η – just right

A fixed step size (if it is too small) is inefficient when you are far from the local minimum. On the other hand, too large a step size when you are close to the minimum leads to bouncing around, possibly even increasing E_{in}. Ideally, we would like to take large steps when far from the minimum to get in the right ballpark quickly, and then small (more careful) steps when close to the minimum. A simple heuristic can accomplish this: far from the minimum, the norm of the gradient is typically large, and close to the minimum, it is small. Thus, we could set $\eta_t = \eta\|\nabla E_{\text{in}}\|$ to obtain the desired behavior for the variable step size; choosing the step size proportional to the norm of the gradient will also conveniently cancel the term normalizing the unit vector $\hat{\mathbf{v}}$ in Equation (3.10), leading to the *fixed learning rate gradient descent* algorithm for minimizing E_{in} (with redefined η):

Fixed learning rate gradient descent:

1: Initialize the weights at time step $t = 0$ to $\mathbf{w}(0)$.
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Compute the gradient $\mathbf{g}_t = \nabla E_{\text{in}}(\mathbf{w}(t))$.
4: Set the direction to move, $\mathbf{v}_t = -\mathbf{g}_t$.
5: Update the weights: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta\mathbf{v}_t$.
6: Iterate to the next step until it is time to stop.
7: Return the final weights.

In the algorithm, $\mathbf{v}_t$ is a direction that is no longer restricted to unit length. The parameter η (the *learning rate*) has to be specified. A typically good choice for η is around 0.1 (a purely practical observation). To use gradient descent, one must compute the gradient. This can be done explicitly for logistic regression (see Exercise 3.7).

Example 3.3. Gradient descent is a general algorithm for minimizing twice-differentiable functions. We can apply it to the logistic regression in-sample error to return weights that approximately minimize

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1 + e^{-y_n \mathbf{w}^{\mathsf{T}}\mathbf{x}_n}\right).$$

Logistic regression algorithm:

1: Initialize the weights at time step $t = 0$ to $\mathbf{w}(0)$.
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Compute the gradient

$$\mathbf{g}_t = -\frac{1}{N}\sum_{n=1}^{N} \frac{y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\mathsf{T}}(t)\mathbf{x}_n}}.$$

4: Set the direction to move, $\mathbf{v}_t = -\mathbf{g}_t$.
5: Update the weights: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta\mathbf{v}_t$.
6: Iterate to the next step until it is time to stop.
7: Return the final weights $\mathbf{w}$.

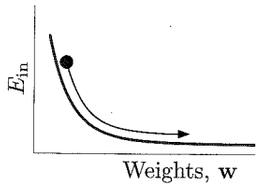
□

Initialization and termination. We have two more loose ends to tie: the first is how to choose $\mathbf{w}(0)$, the initial weights, and the second is how to set the criterion for "... until it is time to stop" in step 6 of the gradient descent algorithm. In some cases, such as logistic regression, initializing the weights $\mathbf{w}(0)$ as zeros works well. However, in general, it is safer to initialize the weights randomly, so as to avoid getting stuck on a perfectly symmetric hilltop. Choosing each weight independently from a Normal distribution with zero mean and small variance usually works well in practice.

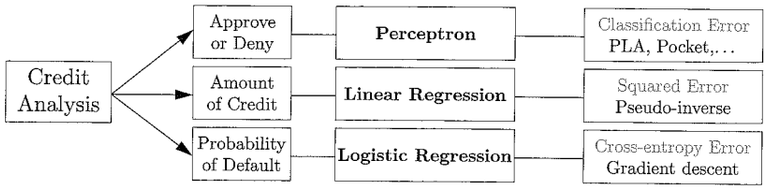
That takes care of initialization, so we now move on to termination. How do we decide when to stop? Termination is a non-trivial topic in optimization. One simple approach, as we encountered in the pocket algorithm, is to set an upper bound on the number of iterations, where the upper bound is typically in the thousands, depending on the amount of training time we have. The problem with this approach is that there is no guarantee on the quality of the final weights.

Another plausible approach is based on the gradient being zero at any minimum. A natural termination criterion would be to stop once $\|\mathbf{g}_t\|$ drops below a certain threshold. Eventually this must happen, but we do not know when it will happen. For logistic regression, a combination of the two conditions (setting a large upper bound for the number of iterations, and a small lower bound for the size of the gradient) usually works well in practice.

There is a problem with relying solely on the size of the gradient to stop, which is that you might stop prematurely as illustrated on the right. When the iteration reaches a relatively flat region (which is more common than you might suspect), the algorithm will prematurely stop when we may want to continue. So one solution is to require that termination occurs only if the error change is small and the error itself is small. Ultimately a combination of termination criteria (a maximum number of iterations, marginal error improvement, coupled with small value for the error itself) works reasonably well.



Example 3.4. By way of summarizing linear models, we revisit our old friend the credit example. If the goal is to decide whether to approve or deny, then we are in the realm of classification; if you want to assign an amount of credit line, then linear regression is appropriate; if you want to predict the probability that someone will default, use logistic regression.



The three linear models have their respective goals, error measures, and algorithms. Nonetheless, they not only share similar sets of linear hypotheses, but are in fact related in other ways. We would like to point out one important relationship: Both logistic regression and linear regression can be used in linear classification. Here is how.

Logistic regression produces a final hypothesis $g(\mathbf{x})$ which is our estimate of $\mathbb{P}[y = +1 \mid \mathbf{x}]$. Such an estimate can easily be used for classification by

setting a threshold on $g(\mathbf{x})$; a natural threshold is $\frac{1}{2}$, which corresponds to classifying $+1$ if $+1$ is more likely. This choice for threshold corresponds to using the logistic regression weights as weights in the perceptron for classification. Not only can logistic regression weights be used for classification in this way, but they can also be used as a way to train the perceptron model. The perceptron learning problem (3.2) is a very hard combinatorial optimization problem. The convexity of E_{in} in logistic regression makes the optimization problem much easier to solve. Since the logistic function is a soft version of a hard threshold, the logistic regression weights should be good weights for classification using the perceptron.

A similar relationship exists between classification and linear regression. Linear regression can be used with any real-valued target function, which includes real values that are ± 1. If $\mathbf{w}_{\text{lin}}^{\mathrm{T}}\mathbf{x}$ is fit to ± 1 values, $\text{sign}(\mathbf{w}_{\text{lin}}^{\mathrm{T}}\mathbf{x})$ will likely agree with these values and make good classification predictions. In other words, the linear regression weights $\mathbf{w}_{\text{lin}}$, which are easily computed using the pseudo-inverse, are also an approximate solution for the perceptron model. The weights can be directly used for classification, or used as an initial condition for the pocket algorithm to give it a head start. □

Exercise 3.9

Consider pointwise error measures $\mathsf{e}_{\text{class}}(s, y) = [\![y \neq \text{sign}(s)]\!]$, $\mathsf{e}_{\text{sq}}(s, y) = (y - s)^2$, and $\mathsf{e}_{\text{log}}(s, y) = \ln(1 + \exp(-ys))$, where the signal $s = \mathbf{w}^{\mathrm{T}}\mathbf{x}$.

(a) For $y = +1$, plot $\mathsf{e}_{\text{class}}$, e_{sq} and $\frac{1}{\ln 2}\mathsf{e}_{\text{log}}$ versus s, on the same plot.

(b) Show that $\mathsf{e}_{\text{class}}(s, y) \leq \mathsf{e}_{\text{sq}}(s, y)$, and hence that the classification error is upper bounded by the squared error.

(c) Show that $\mathsf{e}_{\text{class}}(s, y) \leq \frac{1}{\ln 2}\mathsf{e}_{\text{log}}(s, y)$, and, as in part (b), get an upper bound (up to a constant factor) using the logistic regression error.

These bounds indicate that minimizing the squared or logistic regression error should also decrease the classification error, which justifies using the weights returned by linear or logistic regression as approximations for classification.

Stochastic gradient descent. The version of gradient descent we have described so far is known as *batch* gradient descent – the gradient is computed for the error on the whole data set before a weight update is done. A sequential version of gradient descent known as *stochastic gradient descent* (SGD) turns out to be very efficient in practice. Instead of considering the full batch gradient on all N training data points, we consider a stochastic version of the gradient. First, pick a training data point $(\mathbf{x}_n, y_n)$ uniformly at random (hence the name 'stochastic'), and consider only the error on that data point

(in the case of logistic regression),

$$\mathsf{e}_n(\mathbf{w}) = \ln\left(1 + e^{-y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n}\right).$$

The gradient of this single data point's error is used for the weight update in exactly the same way that the gradient was used in batch gradient descent. The gradient needed for the weight update of SGD is (see Exercise 3.7)

$$\nabla \mathsf{e}_n(\mathbf{w}) = \frac{-y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n}},$$

and the weight update is $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \mathsf{e}_n(\mathbf{w})$. Insight into why SGD works can be gained by looking at the expected value of the change in the weight (the expectation is with respect to the random point that is selected). Since n is picked uniformly at random from $\{1, \ldots, N\}$, the expected weight change is

$$-\eta \cdot \frac{1}{N} \sum_{n=1}^{N} \nabla \mathsf{e}_n(\mathbf{w}).$$

This is exactly the same as the deterministic weight change from the batch gradient descent weight update. That is, 'on average' the minimization proceeds in the right direction, but is a bit wiggly. In the long run, these random fluctuations cancel out. The computational cost is cheaper by a factor of N, though, since we compute the gradient for only one point per iteration, rather than for all N points as we do in batch gradient descent.

Notice that SGD is similar to PLA in that it decreases the error with respect to one data point at a time. Minimizing the error on one data point may interfere with the error on the rest of the data points that are not considered at that iteration. However, also similar to PLA, the interference cancels out on average as we have just argued.

Exercise 3.10

(a) Define an error for a single data point $(\mathbf{x}_n, y_n)$ to be

$$\mathsf{e}_n(\mathbf{w}) = \max(0, -y_n \mathbf{w}^{\mathrm{T}} \mathbf{x}_n).$$

Argue that PLA can be viewed as SGD on e_n with learning rate $\eta = 1$.

(b) For logistic regression with a very large $\mathbf{w}$, argue that minimizing E_{in} using SGD is similar to PLA. This is another indication that the logistic regression weights can be used as a good approximation for classification.

SGD is successful in practice, often beating the batch version and other more sophisticated algorithms. In fact, SGD was an important part of the algorithm that won the million-dollar Netflix competition, discussed in Section 1.1. It scales well to large data sets, and is naturally suited to online learning, where

a stream of data present themselves to the learning algorithm sequentially. The randomness introduced by processing one data point at a time can be a plus, helping the algorithm to avoid flat regions and local minima in the case of a complicated error surface. However, it is challenging to choose a suitable termination criterion for SGD. A good stopping criterion should consider the total error on all the data, which can be computationally demanding to evaluate at each iteration.

3.4 Nonlinear Transformation

All formulas for the linear model have used the sum

$$\mathbf{w}^{\mathrm{T}} \mathbf{x} = \sum_{i=0}^{d} w_i x_i \tag{3.11}$$

as the main quantity in computing the hypothesis output. This quantity is linear, not only in the x_i's but also in the w_i's. A closer inspection of the corresponding learning algorithms shows that *the linearity in w_i's* is the key property for deriving these algorithms; the x_i's are just constants as far as the algorithm is concerned. This observation opens the possibility for allowing nonlinear versions of x_i's while still remaining in the analytic realm of linear models, because the form of Equation (3.11) remains linear in the w_i parameters.

Consider the credit limit problem for instance. It makes sense that the 'years in residence' field would affect a person's credit since it is correlated with stability. However, it is less plausible that the credit limit would grow *linearly* with the number of years in residence. More plausibly, there is a threshold (say 1 year) below which the credit limit is affected negatively and another threshold (say 5 years) above which the credit limit is affected positively. If x_i is the input variable that measures years in residence, then two nonlinear 'features' derived from it, namely $[\![x_i < 1]\!]$ and $[\![x_i > 5]\!]$, would allow a linear formula to reflect the credit limit better.

We have already seen the use of features in the classification of handwritten digits, where intensity and symmetry features were derived from input pixels. Nonlinear transforms can be further applied to those features, as we will see shortly, creating more elaborate features and improving the performance. The scope of linear methods expands significantly when we represent the input by a set of appropriate features.

3.4.1 The $\mathcal{Z}$ Space

Consider the situation in Figure 3.1(b) where a linear classifier can't fit the data. By transforming the inputs x_1, x_2 in a nonlinear fashion, we will be able to separate the data with more complicated boundaries while still using the