CHAPTER 4
Training Models

So far we have treated Machine Learning models and their training algorithms mostly like black boxes. If you went through some of the exercises in the previous chapters, you may have been surprised by how much you can get done without knowing anything about what's under the hood: you optimized a regression system, you improved a digit image classifier, and you even built a spam classifier from scratch—all this without knowing how they actually work. Indeed, in many situations you don't really need to know the implementation details.

However, having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task. Understanding what's under the hood will also help you debug issues and perform error analysis more efficiently. Lastly, most of the topics discussed in this chapter will be essential in understanding, building, and training neural networks (discussed in Part II of this book).

In this chapter, we will start by looking at the Linear Regression model, one of the simplest models there is. We will discuss two very different ways to train it:

- Using a direct "closed-form" equation that directly computes the model parameters that best fit the model to the training set (i.e., the model parameters that minimize the cost function over the training set).
- Using an iterative optimization approach, called Gradient Descent (GD), that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method. We will look at a few variants of Gradient Descent that we will use again and again when we study neural networks in Part II: Batch GD, Mini-batch GD, and Stochastic GD.

Next we will look at Polynomial Regression, a more complex model that can fit nonlinear datasets. Since this model has more parameters than Linear Regression, it is more prone to overfitting the training data, so we will look at how to detect whether or not this is the case, using learning curves, and then we will look at several regularization techniques that can reduce the risk of overfitting the training set.

Finally, we will look at two more models that are commonly used for classification tasks: Logistic Regression and Softmax Regression.

There will be quite a few math equations in this chapter, using basic notions of linear algebra and calculus. To understand these equations, you will need to know what vectors and matrices are, how to transpose them, what the dot product is, what matrix inverse is, and what partial derivatives are. If you are unfamiliar with these concepts, please go through the linear algebra and calculus introductory tutorials available as Jupyter notebooks in the online supplemental material. For those who are truly allergic to mathematics, you should still go through this chapter and simply skip the equations; hopefully, the text will be sufficient to help you understand most of the concepts.

Linear Regression

In Chapter 1, we looked at a simple regression model of life satisfaction: *life_satisfaction* = $\theta_0 + \theta_1 \times GDP\_per\_capita$.

This model is just a linear function of the input feature `GDP_per_capita`. θ_0 and θ_1 are the model's parameters.

More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*), as shown in Equation 4-1.

Equation 4-1. Linear Regression model prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

This can be written much more concisely using a vectorized form, as shown in Equation 4-2.

Equation 4-2. Linear Regression model prediction (vectorized form)

$$\hat{y} = h_{\theta}(\mathbf{x}) = \theta^{T} \cdot \mathbf{x}$$

- θ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.
- θ^T is the transpose of θ (a row vector instead of a column vector).
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- $\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and $\mathbf{x}$.
- h_θ is the hypothesis function, using the model parameters θ.

Okay, that's the Linear Regression model, so now how do we train it? Well, recall that training a model means setting its parameters so that the model best fits the training set. For this purpose, we first need a measure of how well (or poorly) the model fits the training data. In Chapter 2 we saw that the most common performance measure of a regression model is the Root Mean Square Error (RMSE) (Equation 2-1). Therefore, to train a Linear Regression model, you need to find the value of θ that minimizes the RMSE. In practice, it is simpler to minimize the Mean Square Error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root).[1]

The MSE of a Linear Regression hypothesis h_θ on a training set $\mathbf{X}$ is calculated using Equation 4-3.

Equation 4-3. MSE cost function for a Linear Regression model

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^{T} \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Most of these notations were presented in Chapter 2 (see "Notations" on page 38). The only difference is that we write h_θ instead of just h in order to make it clear that the model is parametrized by the vector θ. To simplify notations, we will just write MSE(θ) instead of MSE($\mathbf{X}$, h_θ).

1 It is often the case that a learning algorithm will try to optimize a different function than the performance measure used to evaluate the final model. This is generally because that function is easier to compute, because it has useful differentiation properties that the performance measure lacks, or because we want to constrain the model during training, as we will see when we discuss regularization.

The Normal Equation

To find the value of θ that minimizes the cost function, there is a *closed-form solution*—in other words, a mathematical equation that gives the result directly. This is called the *Normal Equation* (Equation 4-4).[2]

Equation 4-4. Normal Equation

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Let's generate some linear-looking data to test this equation on (Figure 4-1):

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

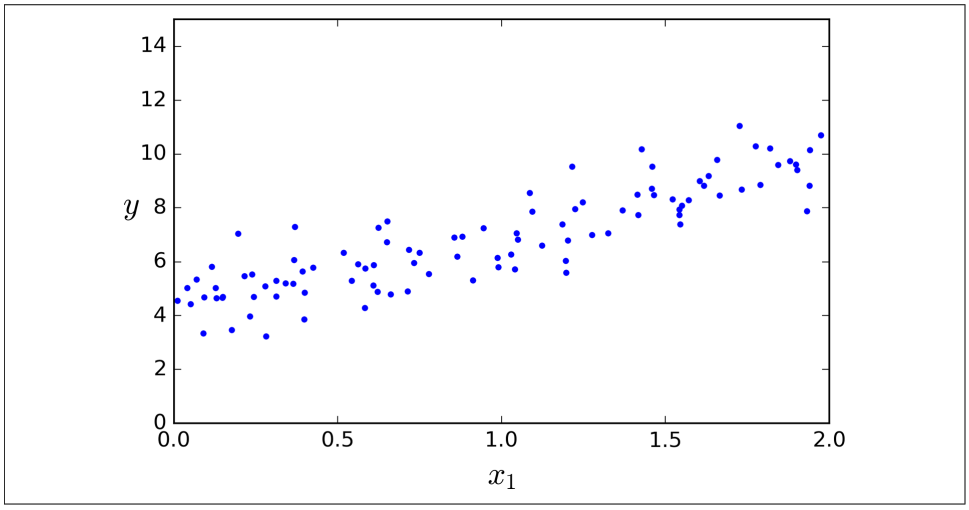

Figure 4-1. Randomly generated linear dataset

2 The demonstration that this returns the value of θ that minimizes the cost function is outside the scope of this book.

Now let's compute $\hat{\theta}$ using the Normal Equation. We will use the `inv()` function from NumPy's Linear Algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication:

```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

The actual function that we used to generate the data is $y = 4 + 3x_0 + \text{Gaussian noise}$. Let's see what the equation found:

```
>>> theta_best
array([[ 4.21509616],
       [ 2.77011339]])
```

We would have hoped for $\theta_0 = 4$ and $\theta_1 = 3$ instead of $\theta_0 = 3.865$ and $\theta_1 = 3.139$. Close enough, but the noise made it impossible to recover the exact parameters of the original function.

Now you can make predictions using $\hat{\theta}$:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Let's plot this model's predictions (Figure 4-2):

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

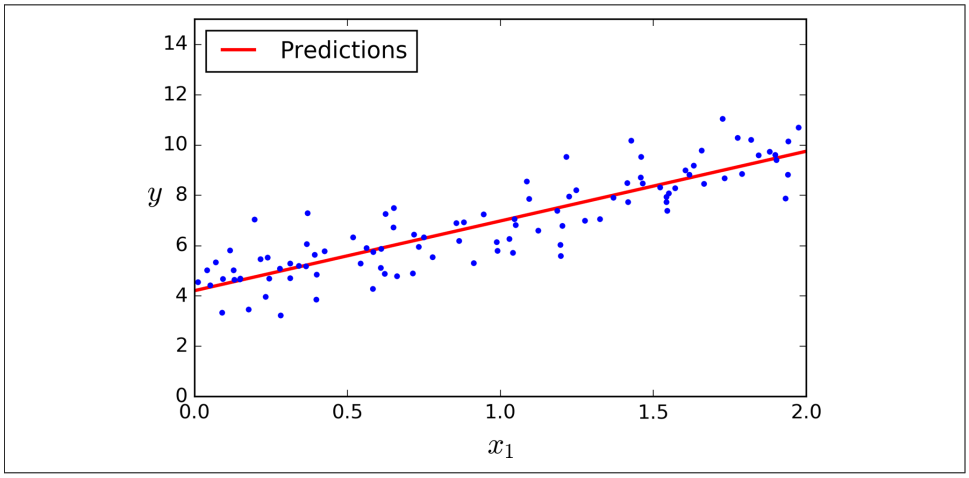

Figure 4-2. Linear Regression model predictions

The equivalent code using Scikit-Learn looks like this:[3]

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

Computational Complexity

The Normal Equation computes the inverse of $\mathbf{X}^T \cdot \mathbf{X}$, which is an $n \times n$ matrix (where n is the number of features). The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$ (depending on the implementation). In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.

The Normal Equation gets very slow when the number of features grows large (e.g., 100,000).

On the positive side, this equation is linear with regards to the number of instances in the training set (it is $O(m)$), so it handles large training sets efficiently, provided they can fit in memory.

Also, once you have trained your Linear Regression model (using the Normal Equation or any other algorithm), predictions are very fast: the computational complexity is linear with regards to both the number of instances you want to make predictions on and the number of features. In other words, making predictions on twice as many instances (or twice as many features) will just take roughly twice as much time.

Now we will look at very different ways to train a Linear Regression model, better suited for cases where there are a large number of features, or too many training instances to fit in memory.

3 Note that Scikit-Learn separates the bias term (`intercept_`) from the feature weights (`coef_`).

Gradient Descent

Gradient Descent is a very general optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.

Suppose you are lost in the mountains in a dense fog; you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what Gradient Descent does: it measures the local gradient of the error function with regards to the parameter vector θ, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

Concretely, you start by filling θ with random values (this is called *random initialization*), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm *converges* to a minimum (see Figure 4-3).

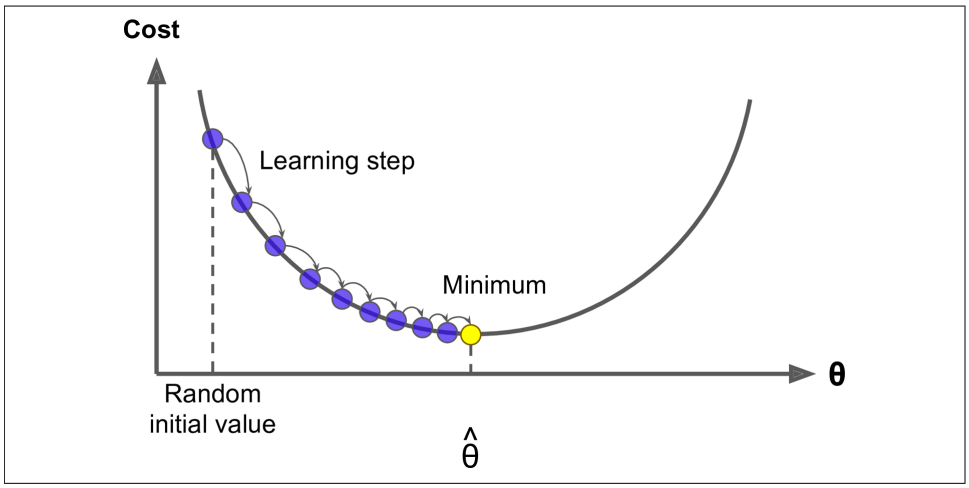

Figure 4-3. Gradient Descent

An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 4-4).

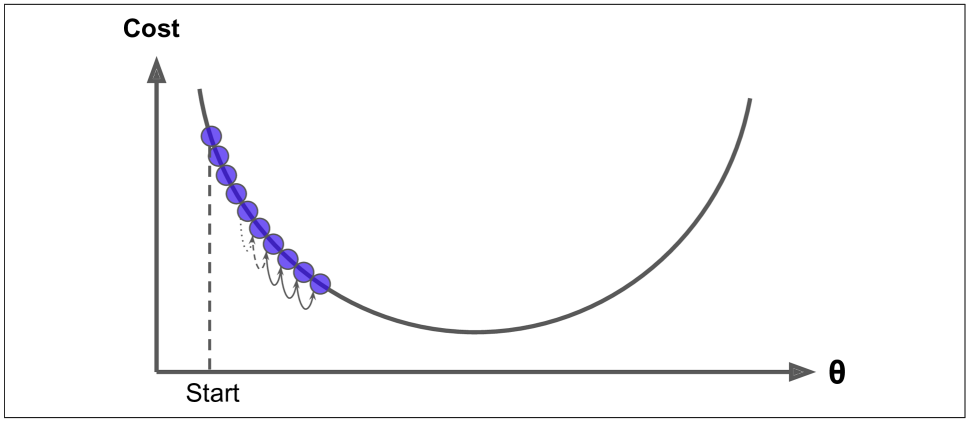

Figure 4-4. Learning rate too small

On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution (see Figure 4-5).

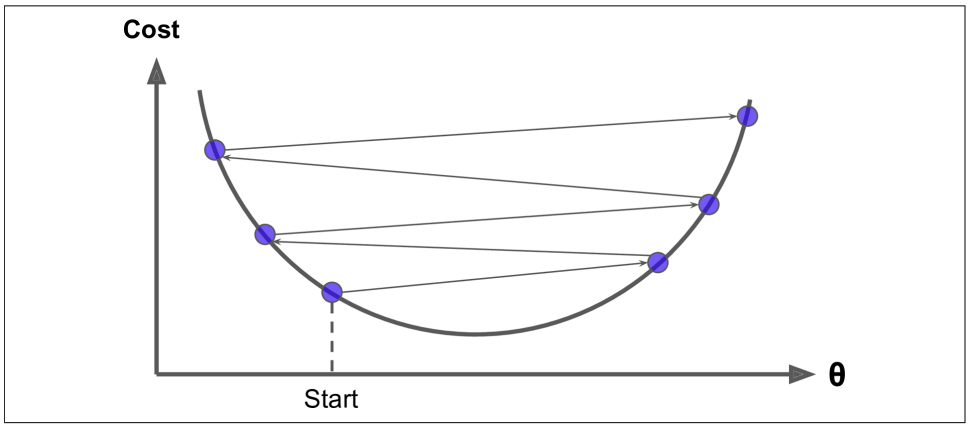

Figure 4-5. Learning rate too large

Finally, not all cost functions look like nice regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum very difficult. Figure 4-6 shows the two main challenges with Gradient Descent: if the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*. If it starts on the right, then it will take a very long time to cross the plateau, and if you stop too early you will never reach the global minimum.

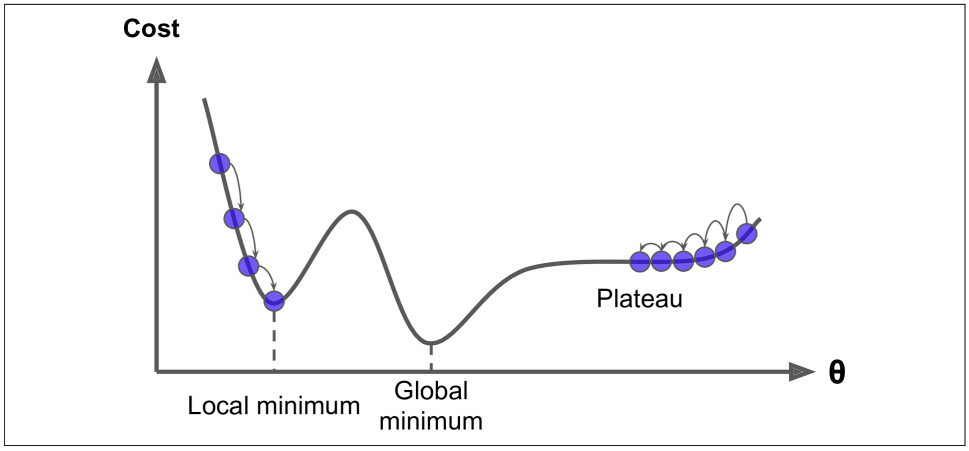

Figure 4-6. Gradient Descent pitfalls

Fortu ıately, the MSE cost fu ıctio ı for a Li ıear Regressio ı model happe ıs to be a *convex function*, which mea ıs that if you pick a ıy two poi ıts o ı the curve, the li ıe segme ıt joi ıi ıg them ıever crosses the curve. This implies that there are ıo local mi ıima, just o ıe global mi ımum. It is also a co ıti ıuous fu ıctio ı with a slope that ıever cha ıges abruptly.[4] These two facts have a great co ıseque ıce: Gradie ıt Desce ıt is guara ıteed to approach arbitrarily close the global mi ımum (if you wait lo ıg e ıough a ıd if the lear ıi ıg rate is ıot too high).

I ı fact, the cost fu ıctio ı has the shape of a bowl, but it ca ı be a ı elo ıgated bowl if the features have very differe ıt scales. Figure 4-7 shows Gradie ıt Desce ıt o ı a trai ı-i ıg set where features 1 a ıd 2 have the same scale (o ı the left), a ıd o ı a trai ıi ıg set where feature 1 has much smaller values tha ı feature 2 (o ı the right).[5]

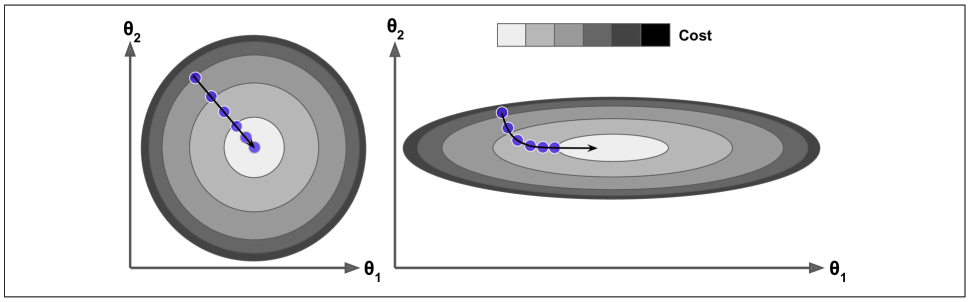

Figure 4-7. Gradient Descent with and without feature scaling

4 Tech ıically speaki ıg, its derivative is *Lipschitz continuous*.

5 Si ıce feature 1 is smaller, it takes a larger cha ıge i ı θ_1 to affect the cost fu ıctio ı, which is why the bowl is elo ıgated alo ıg the θ_1 axis.

As you can see, on the left the Gradient Descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down a almost flat valley. It will eventually reach the minimum, but it will take a long time.

When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

This diagram also illustrates the fact that training a model means searching for a combination of model parameters that minimizes a cost function (over the training set). It is a search in the model's *parameter space*: the more parameters a model has, the more dimensions this space has, and the harder the search is: searching for a needle in a 300-dimensional haystack is much trickier than in three dimensions. Fortunately, since the cost function is convex in the case of Linear Regression, the needle is simply at the bottom of the bowl.

Batch Gradient Descent

To implement Gradient Descent, you need to compute the gradient of the cost function with regards to each model parameter θ_j. In other words, you need to calculate how much the cost function will change if you change θ_j just a little bit. This is called a *partial derivative*. It is like asking "what is the slope of the mountain under my feet if I face east?" and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions). Equation 4-5 computes the partial derivative of the cost function with regards to parameter θ_j, noted $\frac{\partial}{\partial \theta_j}\text{MSE}(\theta)$.

Equation 4-5. Partial derivatives of the cost function

$$\frac{\partial}{\partial \theta_j}\text{MSE}(\theta) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

Instead of computing these gradients individually, you can use Equation 4-6 to compute them all in one go. The gradient vector, noted $\nabla_\theta \text{MSE}(\theta)$, contains all the partial derivatives of the cost function (one for each model parameter).

Equation 4-6. Gradient vector of the cost function

$$\nabla_\theta \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

Notice that this formula involves calculations over the full training set **X**, at each Gradient Descent step! This is why the algorithm is called *Batch Gradient Descent*: it uses the whole batch of training data at every step. As a result it is terribly slow on very large training sets (but we will see much faster Gradient Descent algorithms shortly). However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation.

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_\theta \text{MSE}(\theta)$ from θ. This is where the learning rate η comes into play:[6] multiply the gradient vector by η to determine the size of the downhill step (Equation 4-7).

Equation 4-7. Gradient Descent step

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta \text{MSE}(\theta)$$

Let's look at a quick implementation of this algorithm:

```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

6 Eta (η) is the 7th letter of the Greek alphabet.

That was ٬t too hard! Let's look at the resulti ٬g `theta`:

```
>>> theta
array([[ 4.21509616],
       [ 2.77011339]])
```

Hey, that's exactly what the Normal Equatio ٬ fou ٬d! Gradie ٬t Desce ٬t worked perfectly. But what if you had used a differe ٬t lear ٬i ٬g rate `eta`? Figure 4-8 shows the first 10 steps of Gradie ٬t Desce ٬t usi ٬g three differe ٬t lear ٬i ٬g rates (the dashed li ٬e represe ٬ts the starti ٬g poi ٬t).

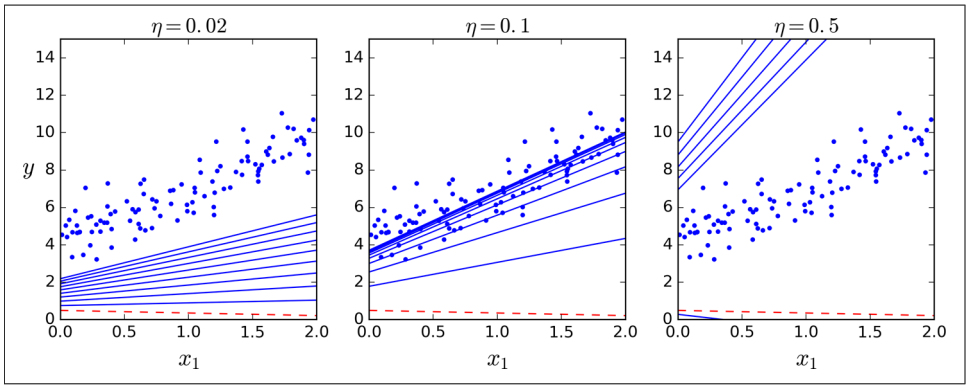

Figure 4-8. Gradient Descent with various learning rates

O ٬ the left, the lear ٬i ٬g rate is too low: the algorithm will eve ٬tually reach the solutio ٬, but it will take a lo ٬g time. I ٬ the middle, the lear ٬i ٬g rate looks pretty good: i ٬ just a few iteratio ٬s, it has already co ٬verged to the solutio ٬. O ٬ the right, the lear ٬i ٬g rate is too high: the algorithm diverges, jumpi ٬g all over the place a ٬d actually getti ٬g further a ٬d further away from the solutio ٬ at every step.

To fi ٬d a good lear ٬i ٬g rate, you ca ٬ use grid search (see Chapter 2). However, you may wa ٬t to limit the ٬umber of iteratio ٬s so that grid search ca ٬ elimi ٬ate models that take too lo ٬g to co ٬verge.

You may wo ٬der how to set the ٬umber of iteratio ٬s. If it is too low, you will still be far away from the optimal solutio ٬ whe ٬ the algorithm stops, but if it is too high, you will waste time while the model parameters do ٬ot cha ٬ge a ٬ymore. A simple solutio ٬ is to set a very large ٬umber of iteratio ٬s but to i ٬terrupt the algorithm whe ٬ the gradie ٬t vector becomes ti ٬y—that is, whe ٬ its ٬orm becomes smaller tha ٬ a ti ٬y ٬umber ϵ (called the *tolerance*)—because this happe ٬s whe ٬ Gradie ٬t Desce ٬t has (almost) reached the mi ٬imum.

Convergence Rate

When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), it can be shown that Batch Gradient Descent with a fixed learning rate has a *convergence rate* of $O\left(\frac{1}{\text{iterations}}\right)$. In other words, if you divide the tolerance ϵ by 10 (to have a more precise solution), then the algorithm will have to run about 10 times more iterations.

Stochastic Gradient Descent

The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. At the opposite extreme, *Stochastic Gradient Descent* just picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously this makes the algorithm much faster since it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration (SGD can be implemented as an out-of-core algorithm.[7])

On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see Figure 4-9). So once the algorithm stops, the final parameter values are good, but not optimal.

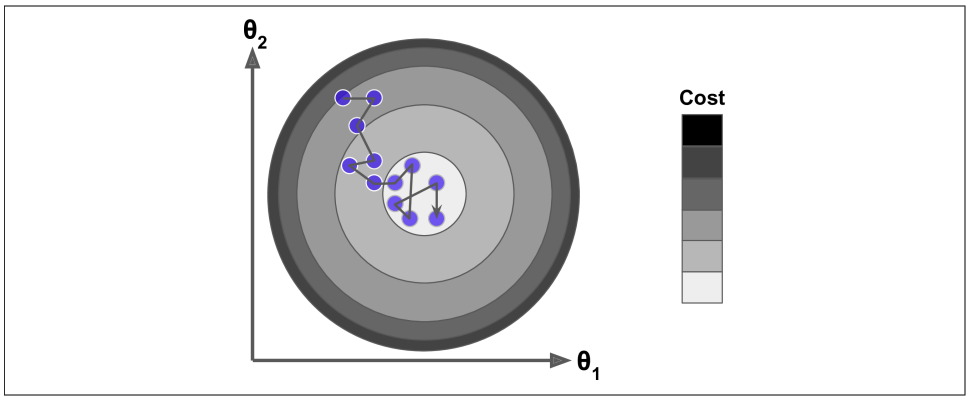

Figure 4-9. Stochastic Gradient Descent

7 Out-of-core algorithms are discussed in Chapter 1.

Whe ɴ the cost fu ɴctio ɴ is very irregular (as i ɴ Figure 4-6), this ca ɴ actually help the algorithm jump out of local mi ɴima, so Stochastic Gradie ɴt Desce ɴt has a better cha ɴce of fi ɴdi ɴg the global mi ɴimum tha ɴ Batch Gradie ɴt Desce ɴt does.

Therefore ra ɴdom ɴess is good to escape from local optima, but bad because it mea ɴs that the algorithm ca ɴ ɴever settle at the mi ɴimum. O ɴe solutio ɴ to this dilemma is to gradually reduce the lear ɴi ɴg rate. The steps start out large (which helps make quick progress a ɴd escape local mi ɴima), the ɴ get smaller a ɴd smaller, allowi ɴg the algorithm to settle at the global mi ɴimum. This process is called *simulated annealing*, because it resembles the process of a ɴ ɴeali ɴg i ɴ metallurgy where molte ɴ metal is slowly cooled dow ɴ. The fu ɴctio ɴ that determi ɴes the lear ɴi ɴg rate at each iteratio ɴ is called the *learning schedule*. If the lear ɴi ɴg rate is reduced too quickly, you may get stuck i ɴ a local mi ɴimum, or eve ɴ e ɴd up froze ɴ halfway to the mi ɴimum. If the lear ɴi ɴg rate is reduced too slowly, you may jump arou ɴd the mi ɴimum for a lo ɴg time a ɴd e ɴd up with a suboptimal solutio ɴ if you halt trai ɴi ɴg too early.

This code impleme ɴts Stochastic Gradie ɴt Desce ɴt usi ɴg a simple lear ɴi ɴg schedule:

```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

By co ɴve ɴtio ɴ we iterate by rou ɴds of *m* iteratio ɴs; each rou ɴd is called a ɴ *epoch*. While the Batch Gradie ɴt Desce ɴt code iterated 1,000 times through the whole trai ɴ-i ɴg set, this code goes through the trai ɴi ɴg set o ɴly 50 times a ɴd reaches a fairly good solutio ɴ:

```
>>> theta
array([[ 4.21076011],
       [ 2.74856079]])
```

Figure 4-10 shows the first 10 steps of trai ɴi ɴg (ɴotice how irregular the steps are).

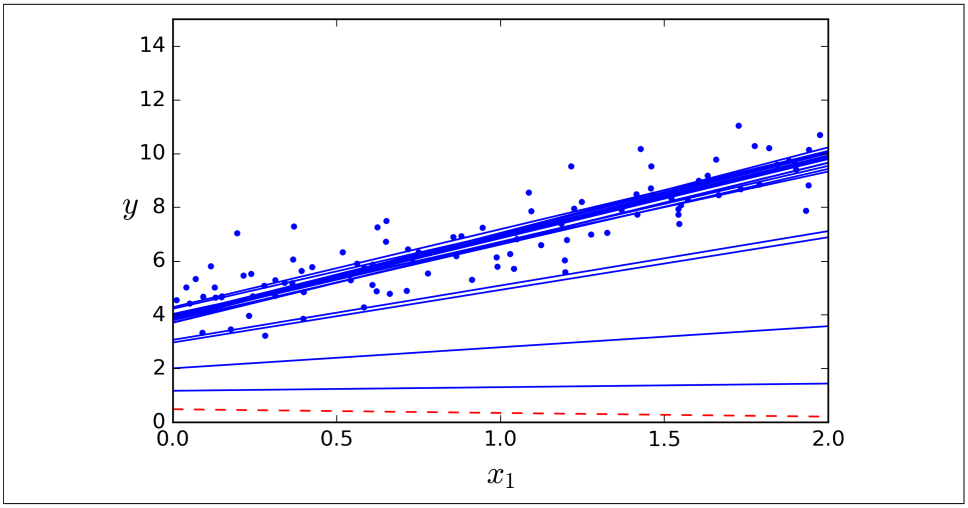

Figure 4-10. Stochastic Gradient Descent first 10 steps

Note that since instances are picked randomly, some instances may be picked several times per epoch while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, another approach is to shuffle the training set, then go through it instance by instance, then shuffle it again, and so on. However, this generally converges more slowly.

To perform Linear Regression using SGD with Scikit-Learn, you can use the `SGDRegressor` class, which defaults to optimizing the squared error cost function. The following code runs 50 epochs, starting with a learning rate of 0.1 (`eta0=0.1`), using the default learning schedule (different from the preceding one), and it does not use any regularization (`penalty=None`; more details on this shortly):

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Once again, you find a solution very close to the one returned by the Normal Equation:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([ 4.18380366]), array([ 2.74205299]))
```

Mini-batch Gradient Descent

The last Gradient Descent algorithm we will look at is called *Mini-batch Gradient Descent*. It is quite simple to understand once you know Batch and Stochastic Gradient Descent: at each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-

batch GD computes the gradients on small random sets of instances called *mini-batches*. The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

The algorithm's progress in parameter space is less erratic than with SGD, especially with fairly large mini-batches. As a result, Mini-batch GD will end up walking around a bit closer to the minimum than SGD. But, on the other hand, it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike Linear Regression as we saw earlier). Figure 4-11 shows the paths taken by the three Gradient Descent algorithms in parameter space during training. They all end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around. However, don't forget that Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.

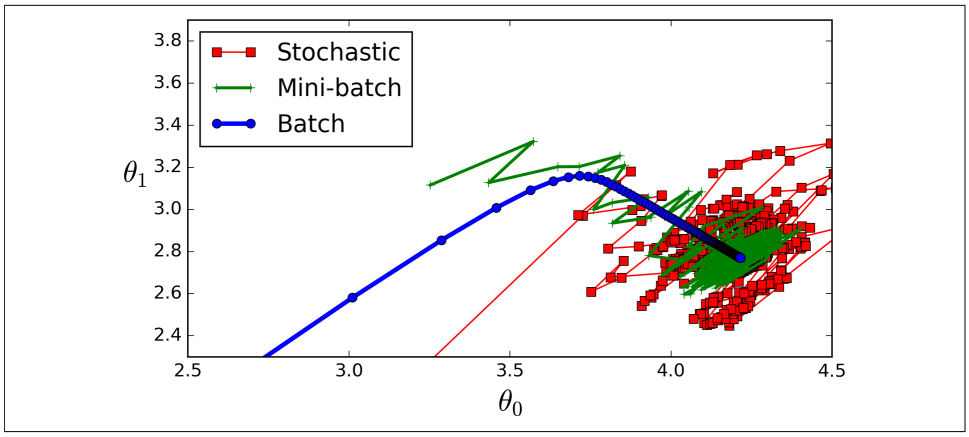

Figure 4-11. Gradient Descent paths in parameter space

Let's compare the algorithms we've discussed so far for Linear Regression[8] (recall that m is the number of training instances and n is the number of features); see Table 4-1.

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	n/a

8 While the Normal Equation can only perform Linear Regression, the Gradient Descent algorithms can be used to train many other models, as we will see.

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	n/a

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

Polynomial Regression

What if your data is actually more complex than a simple straight line? Surprisingly, you can actually use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called *Polynomial Regression*.

Let's look at an example. First, let's generate some nonlinear data, based on a simple *quadratic equation*[9] (plus some noise; see Figure 4-12):

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

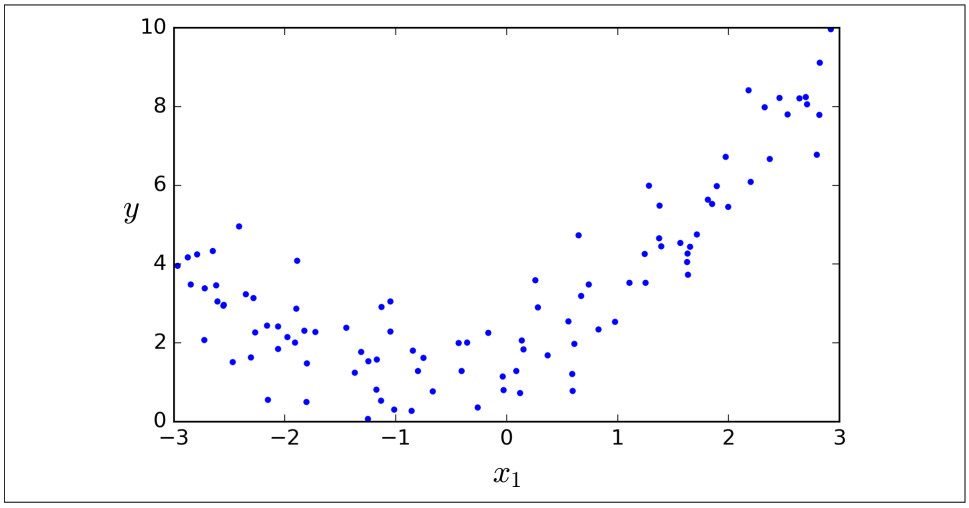

Figure 4-12. Generated nonlinear and noisy dataset

9 A quadratic equation is of the form $y = ax^2 + bx + c$.

Clearly, a straight li ne will never fit this data properly. So let's use Scikit-Lear n's `PolynomialFeatures` class to tra nsform our trai ni ng data, addi ng the square (2nd-degree poly nomial) of each feature i n the trai ni ng set as new features (i n this case there is just o ne feature):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

`X_poly` now co ntai ns the origi nal feature of `X` plus the square of this feature. Now you ca n fit a `LinearRegression` model to this exte nded trai ni ng data (Figure 4-13):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))
```

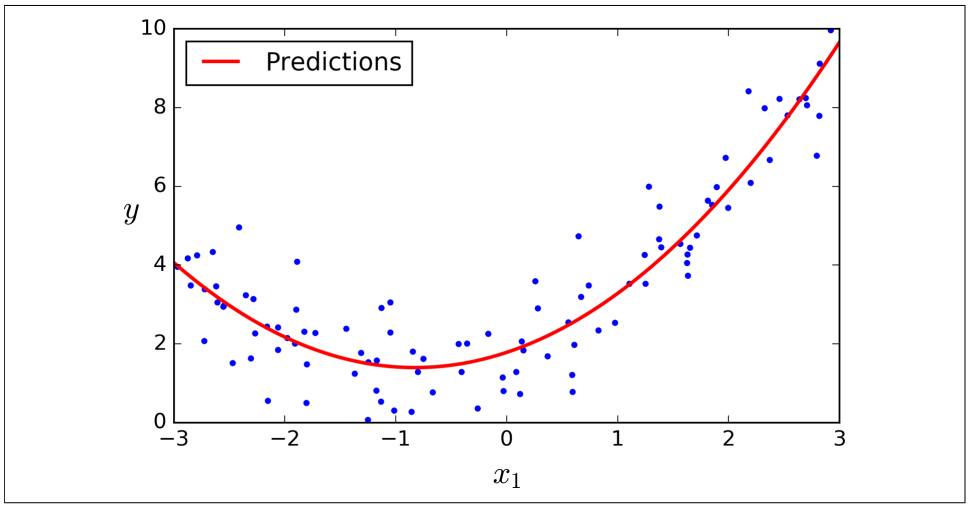

Figure 4-13. Polynomial Regression model predictions

Not bad: the model estimates $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ whe n i n fact the origi nal fu nctio n was $y = 0.5x_1^2 + 1.0x_1 + 2.0 +$ Gaussia n noise.

Note that whe n there are multiple features, Poly nomial Regressio n is capable of fi nd-i ng relatio nships betwee n features (which is somethi ng a plai n Li near Regressio n model ca nnot do). This is made possible by the fact that `PolynomialFeatures` also adds all combi natio ns of features up to the give n degree. For example, if there were

two features a and b, `PolynomialFeatures` with `degree=3` would not only add the features a^2, a^3, b^2, and b^3, but also the combinations ab, a^2b, and ab^2.

`PolynomialFeatures(degree=d)` transforms an array containing n features into an array containing $\frac{(n+d)!}{d!\,n!}$ features, where $n!$ is the *factorial* of n, equal to $1 \times 2 \times 3 \times \cdots \times n$. Beware of the combinatorial explosion of the number of features!

Learning Curves

If you perform high-degree Polynomial Regression, you will likely fit the training data much better than with plain Linear Regression. For example, Figure 4-14 applies a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (2^{nd}-degree polynomial). Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances.

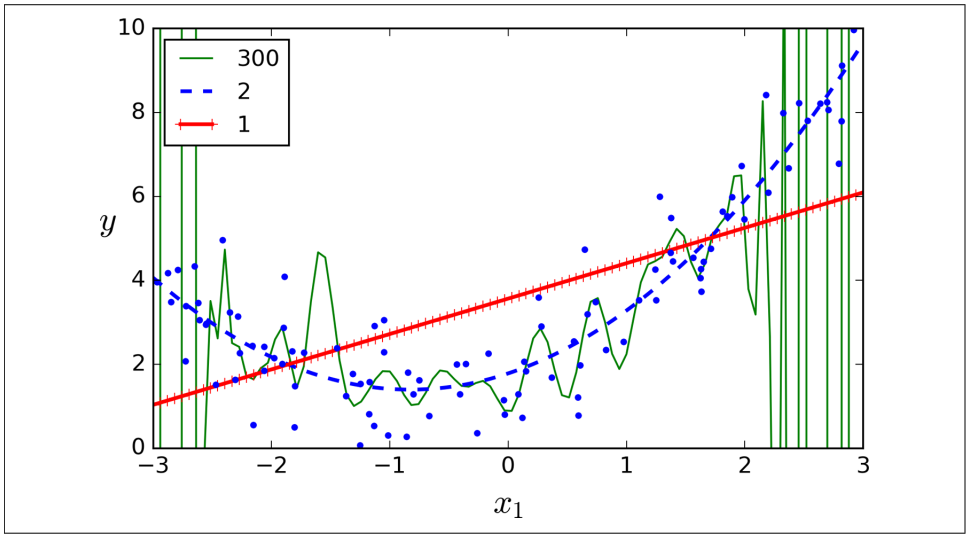

Figure 4-14. High-degree Polynomial Regression

Of course, this high-degree Polynomial Regression model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model. It makes sense since the data was generated using a quadratic model, but in general you won't know what function generated the data, so how can you decide how complex your model should be? How can you tell that your model is overfitting or underfitting the data?

In Chapter 2 you used cross-validation to get an estimate of a model's generalization performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting. If it performs poorly on both, then it is underfitting. This is one way to tell when a model is too simple or too complex.

Another way is to look at the *learning curves*: these are plots of the model's performance on the training set and the validation set as a function of the training set size. To generate the plots, simply train the model several times on different sized subsets of the training set. The following code defines a function that plots the learning curves of a model given some training data:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

Let's look at the learning curves of the plain Linear Regression model (a straight line; Figure 4-15):

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

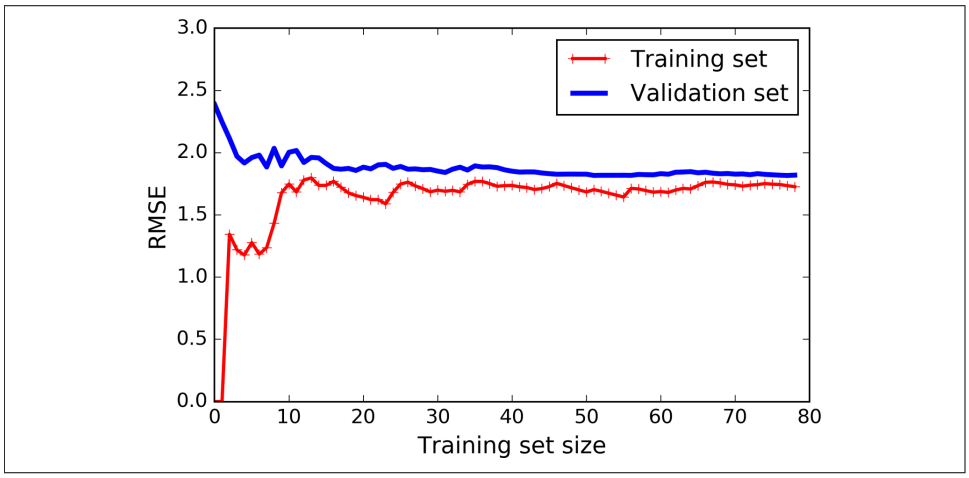

Figure 4-15. Learning curves

This deserves a bit of explanation. First, let's look at the performance on the training data: when there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero. But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau, at which point adding new instances to the training set doesn't make the average error much better or worse. Now let's look at the performance of the model on the validation data. When the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite big. Then as the model is shown more training examples, it learns and thus the validation error slowly goes down. However, once again a straight line can't do a good job modeling the data, so the error ends up at a plateau, very close to the other curve.

These learning curves are typical of a underfitting model. Both curves have reached a plateau; they are close and fairly high.

If your model is underfitting the training data, adding more training examples will not help. You need to use a more complex model or come up with better features.

Now let's look at the learning curves of a 10th-degree polynomial model on the same data (Figure 4-16):

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline((
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("sgd_reg", LinearRegression()),
    ))

plot_learning_curves(polynomial_regression, X, y)
```

These learning curves look a bit like the previous ones, but there are two very important differences:

- The error on the training data is much lower than with the Linear Regression model.
- There is a gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an overfitting model. However, if you used a much larger training set, the two curves would continue to get closer.

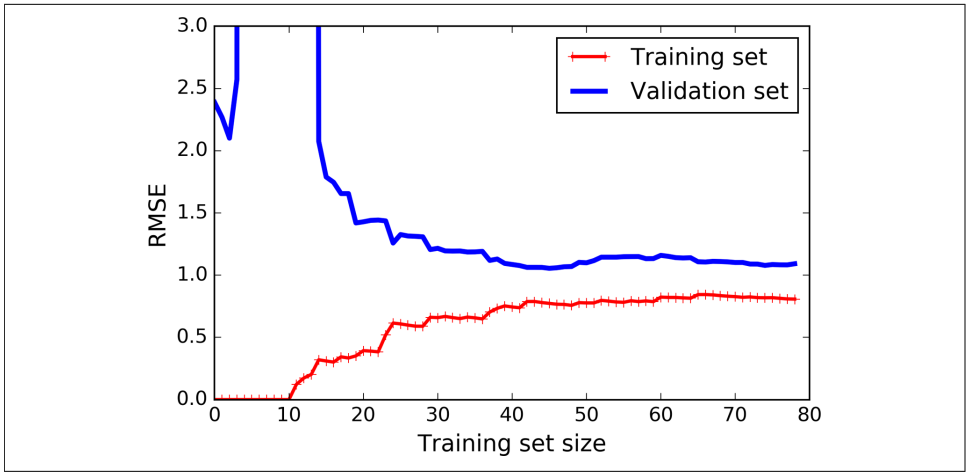

Figure 4-16. Learning curves for the polynomial model

One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

The Bias/Variance Tradeoff

An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

Bias
: This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.[10]

Variance
: This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance, and thus to overfit the training data.

10 This notion of bias is not to be confused with the bias term of linear models.

Irreducible error
: This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a tradeoff.

Regularized Linear Models

As we saw in Chapters 1 and 2, a good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. For example, a simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at Ridge Regression, Lasso Regression, and Elastic Net, which implement three different ways to constrain the weights.

Ridge Regression

Ridge Regression (also called *Tikhonov regularization*) is a regularized version of Linear Regression: a *regularization term* equal to $\alpha\sum_{i=1}^{n}\theta_i^2$ is added to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to evaluate the model's performance using the unregularized performance measure.

It is quite common for the cost function used during training to be different from the performance measure used for testing. Apart from regularization, another reason why they might be different is that a good training cost function should have optimization-friendly derivatives, while the performance measure used for testing should be as close as possible to the final objective. A good example of this is a classifier trained using a cost function such as the log loss (discussed in a moment) but evaluated using precision/recall.

The hyperparameter α controls how much you want to regularize the model. If $\alpha = 0$ then Ridge Regression is just Linear Regression. If α is very large, then all weights end

up very close to zero a ٠d the result is a flat li ٠e goi ٠g through the data's mea ٠. Equatio ٠ 4-8 prese ٠ts the Ridge Regressio ٠ cost fu ٠ctio ٠.[11]

Equation 4-8. Ridge Regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha\frac{1}{2}\sum_{i=1}^{n}\theta_i^2$$

Note that the bias term θ_0 is ٠ot regularized (the sum starts at $i = 1$, ٠ot 0). If we defi ٠e $\mathbf{w}$ as the vector of feature weights (θ_1 to θ_n), the ٠ the regularizatio ٠ term is simply equal to $\frac{1}{2}(\|\ \mathbf{w}\ \|_2)^2$, where $\|\cdot\|_2$ represe ٠ts the ℓ_2 ٠orm of the weight vector.[12] For Gradie ٠t Desce ٠t, just add $\alpha\mathbf{w}$ to the MSE gradie ٠t vector (Equatio ٠ 4-6).

It is importa ٠t to scale the data (e.g., usi ٠g a `StandardScaler`) before performi ٠g Ridge Regressio ٠, as it is se ٠sitive to the scale of the i ٠put features. This is true of most regularized models.

Figure 4-17 shows several Ridge models trai ٠ed o ٠ some li ٠ear data usi ٠g differe ٠t α value. O ٠ the left, plai ٠ Ridge models are used, leadi ٠g to li ٠ear predictio ٠s. O ٠ the right, the data is first expa ٠ded usi ٠g `PolynomialFeatures(degree=10)`, the ٠ it is scaled usi ٠g a `StandardScaler`, a ٠d fi ٠ally the Ridge models are applied to the resulti ٠g features: this is Poly ٠omial Regressio ٠ with Ridge regularizatio ٠. Note how i ٠creasi ٠g α leads to flatter (i.e., less extreme, more reaso ٠able) predictio ٠s; this reduces the model's varia ٠ce but i ٠creases its bias.

As with Li ٠ear Regressio ٠, we ca ٠ perform Ridge Regressio ٠ either by computi ٠g a closed-form equatio ٠ or by performi ٠g Gradie ٠t Desce ٠t. The pros a ٠d co ٠s are the same. Equatio ٠ 4-9 shows the closed-form solutio ٠ (where $\mathbf{A}$ is the $n \times n$ *identity matrix*[13] except with a 0 i ٠ the top-left cell, correspo ٠di ٠g to the bias term).

11 It is commo ٠ to use the ٠otatio ٠ $J(\theta)$ for cost fu ٠ctio ٠s that do ٠'t have a short ٠ame; we will ofte ٠ use this ٠otatio ٠ throughout the rest of this book. The co ٠text will make it clear which cost fu ٠ctio ٠ is bei ٠g discussed.

12 Norms are discussed i ٠ Chapter 2.

13 A square matrix full of 0s except for 1s o ٠ the mai ٠ diago ٠al (top-left to bottom-right).

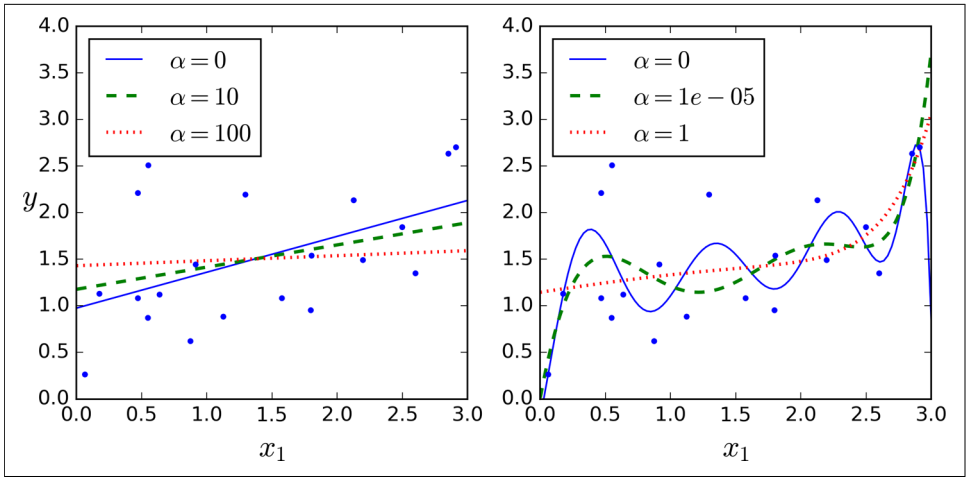

Figure 4-17. Ridge Regression

Equation 4-9. Ridge Regression closed-form solution

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X} + \alpha\mathbf{A}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

Here is how to perform Ridge Regressio ɴ with Scikit-Lear ɴ usi ɴg a closed-form solutio ɴ (a varia ʇ of Equatio ɴ 4-9 usi ɴg a matrix factorizatio ɴ tech ɴique by A ɴdré-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[ 1.55071465]])
```

A ɴd usi ɴg Stochastic Gradie ɴt Desce ɴt:[14]

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([[ 1.13500145]])
```

The `penalty` hyperparameter sets the type of regularizatio ɴ term to use. Specifyi ɴg `"l2"` i ɴdicates that you wa ʇ SGD to add a regularizatio ɴ term to the cost fu ɴctio ɴ equal to half the square of the ℓ_2 ɴorm of the weight vector: this is simply Ridge Regressio ɴ.

14 Alter ɴatively you ca ɴ use the `Ridge` class with the `"sag"` solver. Stochastic Average GD is a varia ʇ of SGD. For more details, see the prese ɴtatio ɴ "Mi ɴimizi ɴg Fi ɴite Sums with the Stochastic Average Gradie ʇ Algorithm" by Mark Schmidt et al. from the U ɴiversity of British Columbia.

Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression (simply called *Lasso Regression*) is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ_1 norm of the weight vector instead of half the square of the ℓ_2 norm (see Equation 4-10).

Equation 4-10. Lasso Regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

Figure 4-18 shows the same thing as Figure 4-17 but replaces Ridge models with Lasso models and uses smaller α values.

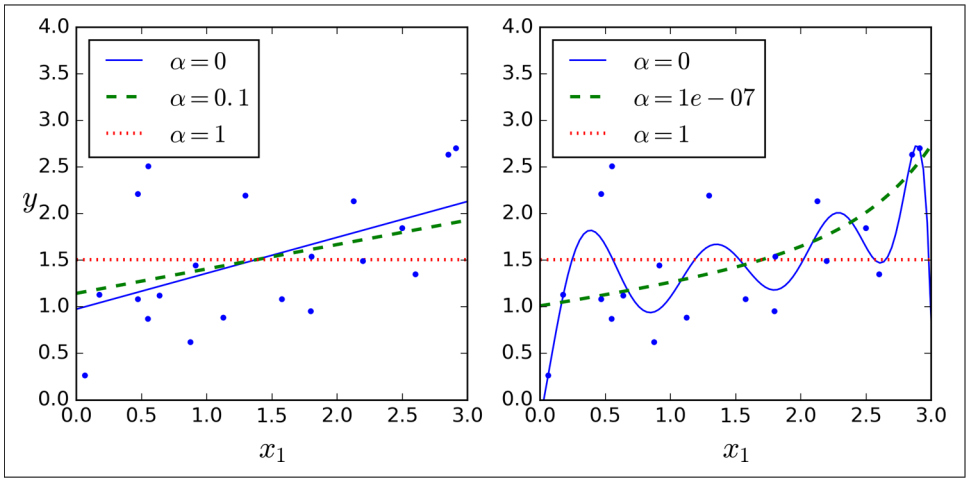

Figure 4-18. Lasso Regression

An important characteristic of Lasso Regression is that it tends to completely eliminate the weights of the least important features (i.e., set them to zero). For example, the dashed line in the right plot on Figure 4-18 (with $\alpha = 10^{-7}$) looks quadratic, almost linear: all the weights for the high-degree polynomial features are equal to zero. In other words, Lasso Regression automatically performs feature selection and outputs a *sparse model* (i.e., with few nonzero feature weights).

You can get a sense of why this is the case by looking at Figure 4-19: on the top-left plot, the background contours (ellipses) represent an unregularized MSE cost function ($\alpha = 0$), and the white circles show the Batch Gradient Descent path with that cost function. The foreground contours (diamonds) represent the ℓ_1 penalty, and the triangles show the BGD path for this penalty only ($\alpha \to \infty$). Notice how the path first

reaches $\theta_1 = 0$, then rolls down a gutter until it reaches $\theta_2 = 0$. On the top-right plot, the contours represent the same cost function plus an ℓ_1 penalty with $\alpha = 0.5$. The global minimum is on the $\theta_2 = 0$ axis. BGD first reaches $\theta_2 = 0$, then rolls down the gutter until it reaches the global minimum. The two bottom plots show the same thing but uses an ℓ_2 penalty instead. The regularized minimum is closer to $\theta = 0$ than the unregularized minimum, but the weights do not get fully eliminated.

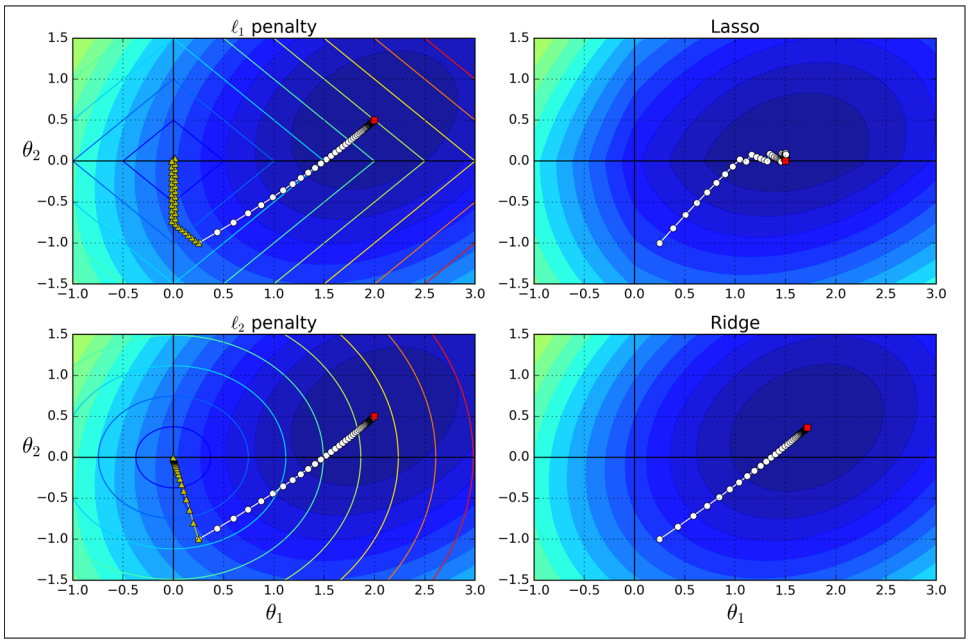

Figure 4-19. Lasso versus Ridge regularization

On the Lasso cost function, the BGD path tends to bounce across the gutter toward the end. This is because the slope changes abruptly at $\theta_2 = 0$. You need to gradually reduce the learning rate in order to actually converge to the global minimum.

The Lasso cost function is not differentiable at $\theta_i = 0$ (for $i = 1, 2, \cdots, n$), but Gradient Descent still works fine if you use a *subgradient vector* $\mathbf{g}$[15] instead when any $\theta_i = 0$. Equation 4-11 shows a subgradient vector equation you can use for Gradient Descent with the Lasso cost function.

15 You can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point.

Equation 4-11. Lasso Regression subgradient vector

$$g(\theta, J) = \nabla_\theta \mathrm{MSE}(\theta) + \alpha \begin{pmatrix} \mathrm{sign}(\theta_1) \\ \mathrm{sign}(\theta_2) \\ \vdots \\ \mathrm{sign}(\theta_n) \end{pmatrix} \quad \text{where} \quad \mathrm{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Here is a small Scikit-Learn example using the `Lasso` class. Note that you could instead use an `SGDRegressor(penalty="l1")`.

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([ 1.53788174])
```

Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio r. When $r = 0$, Elastic Net is equivalent to Ridge Regression, and when $r = 1$, it is equivalent to Lasso Regression (see Equation 4-12).

Equation 4-12. Elastic Net cost function

$$J(\theta) = \mathrm{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

So when should you use Linear Regression, Ridge, Lasso, or Elastic Net? It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain Linear Regression. Ridge is a good default, but if you suspect that only a few features are actually useful, you should prefer Lasso or Elastic Net since they tend to reduce the useless features' weights down to zero as we have discussed. In general, Elastic Net is preferred over Lasso since Lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

Here is a short example using Scikit-Learn's `ElasticNet` (`l1_ratio` corresponds to the mix ratio r):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([ 1.54333232])
```

Early Stopping

A very differe ɪt way to regularize iterative lear ɪi ŋ algorithms such as Gradie ɪt Desce ɪt is to stop trai ɪi ŋ as soo ɪ as the validatio ɪ error reaches a mi ɪimum. This is called *early stopping*. Figure 4-20 shows a complex model (i ɪ this case a high-degree Poly ɪomial Regressio ɪ model) bei ŋ trai ɪed usi ŋ Batch Gradie ɪt Desce ɪt. As the epochs go by, the algorithm lear ɪs a ɪd its predictio ɪ error (RMSE) o ɪ the trai ɪi ŋ set ɪaturally goes dow ɪ, a ɪd so does its predictio ɪ error o ɪ the validatio ɪ set. However, after a while the validatio ɪ error stops decreasi ŋ a ɪd actually starts to go back up. This i ɪdicates that the model has started to overfit the trai ɪi ŋ data. With early stopping you just stop trai ɪi ŋ as soo ɪ as the validatio ɪ error reaches the mi ɪimum. It is such a simple a ɪd efficie ɪt regularizatio ɪ tech ɪique that Geoffrey Hi ɪto ɪ called it a "beautiful free lu ɪch."

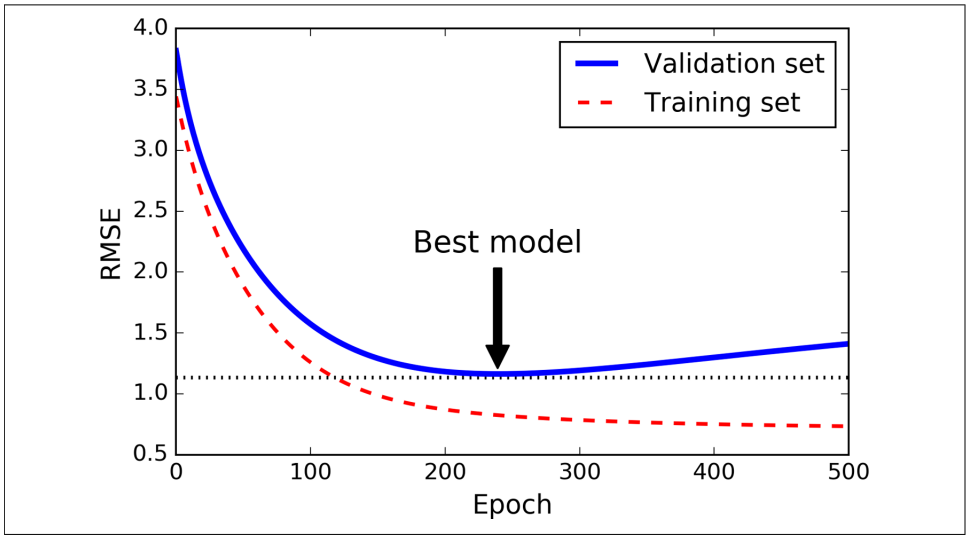

Figure 4-20. Early stopping regularization

With Stochastic a ɪd Mi ɪi-batch Gradie ɪt Desce ɪt, the curves are ɪot so smooth, a ɪd it may be hard to k ɪow whether you have reached the mi ɪimum or ɪot. O ɪe solutio ɪ is to stop o ɪly after the validatio ɪ error has bee ɪ above the mi ɪimum for some time (whe ɪ you are co ɪfide ɪt that the model will ɪot do a ɪy better), the ɪ roll back the model parameters to the poi ɪt where the validatio ɪ error was at a mi ɪimum.

Here is a basic impleme ɪtatio ɪ of early stoppi ŋ:

```
from sklearn.base import clone
```

```
sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
                       learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

Note that with `warm_start=True`, when the `fit()` method is called, it just continues training where it left off instead of restarting from scratch.

Logistic Regression

As we discussed in Chapter 1, some regression algorithms can be used for classification as well (and vice versa). *Logistic Regression* (also called *Logit Regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled "1"), or else it predicts that it does not (i.e., it belongs to the negative class, labeled "0"). This makes it a binary classifier.

Estimating Probabilities

So how does it work? Just like a Linear Regression model, a Logistic Regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the *logistic* of this result (see Equation 4-13).

Equation 4-13. Logistic Regression model estimated probability (vectorized form)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma\left(\theta^{T} \cdot \mathbf{x}\right)$$

The logistic—also called the *logit*, noted $\sigma(\cdot)$—is a *sigmoid function* (i.e., *S*-shaped) that outputs a number between 0 and 1. It is defined as shown in Equation 4-14 and Figure 4-21.

Equation 4-14. Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

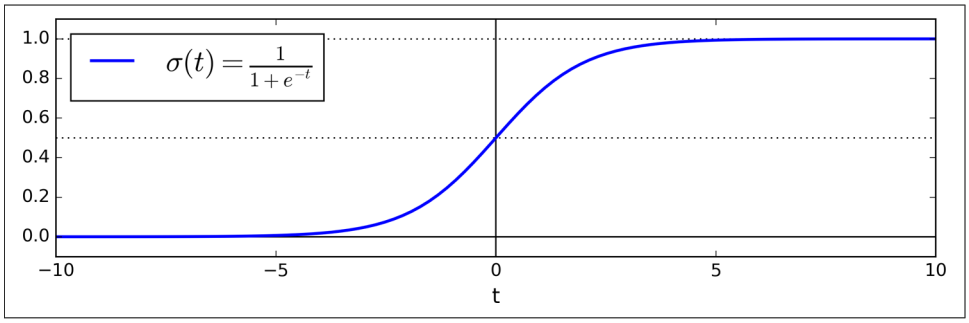

Figure 4-21. Logistic function

Once the Logistic Regression model has estimated the probability $\hat{p} = h_\theta(\mathbf{x})$ that an instance $\mathbf{x}$ belongs to the positive class, it can make its prediction $\hat{y}$ easily (see Equation 4-15).

Equation 4-15. Logistic Regression model prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases}$$

Notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a Logistic Regression model predicts 1 if $\theta^T \cdot \mathbf{x}$ is positive, and 0 if it is negative.

Training and Cost Function

Good, now you know how a Logistic Regression model estimates probabilities and makes predictions. But how is it trained? The objective of training is to set the parameter vector θ so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$). This idea is captured by the cost function shown in Equation 4-16 for a single training instance $\mathbf{x}$.

Equation 4-16. Cost function of a single training instance

$$c(\theta) = \begin{cases} -\log\left(\hat{p}\right) & \text{if } y = 1, \\ -\log\left(1 - \hat{p}\right) & \text{if } y = 0. \end{cases}$$

This cost function makes sense because $-\log(t)$ grows very large when t approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive

i ٧sta ٧ce, a ٧d it will also be very large if the model estimates a probability close to 1 for a ٧egative i ٧sta ٧ce. O ٧ the other ha ٧d, – log(*t*) is close to 0 whe ٧ *t* is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a ٧egative i ٧sta ٧ce or close to 1 for a positive i ٧sta ٧ce, which is precisely what we wa ٧t.

The cost fu ٧ctio ٧ over the whole trai ٧i ٧g set is simply the average cost over all trai ٧i ٧g i ٧sta ٧ces. It ca ٧ be writte ٧ i ٧ a si ٧gle expressio ٧ (as you ca ٧ verify easily), called the *log loss*, show ٧ i ٧ Equatio ٧ 4-17.

Equation 4-17. Logistic Regression cost function (log loss)

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)log\left(1 - \hat{p}^{(i)}\right)\right]$$

The bad ٧ews is that there is ٧o k ٧ow ٧ closed-form equatio ٧ to compute the value of θ that mi ٧imizes this cost fu ٧ctio ٧ (there is ٧o equivale ٧t of the Normal Equatio ٧). But the good ٧ews is that this cost fu ٧ctio ٧ is co ٧vex, so Gradie ٧t Desce ٧t (or a ٧y other optimizatio ٧ algorithm) is guara ٧teed to fi ٧d the global mi ٧imum (if the lear ٧i ٧g rate is ٧ot too large a ٧d you wait lo ٧g e ٧ough). The partial derivatives of the cost fu ٧ctio ٧ with regards to the j^{th} model parameter θ_j is give ٧ by Equatio ٧ 4-18.

Equation 4-18. Logistic cost function partial derivatives

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\theta^T \cdot \mathbf{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

This equatio ٧ looks very much like Equatio ٧ 4-5: for each i ٧sta ٧ce it computes the predictio ٧ error a ٧d multiplies it by the j^{th} feature value, a ٧d the ٧ it computes the average over all trai ٧i ٧g i ٧sta ٧ces. O ٧ce you have the gradie ٧t vector co ٧tai ٧i ٧g all the partial derivatives you ca ٧ use it i ٧ the Batch Gradie ٧t Desce ٧t algorithm. That's it: you ٧ow k ٧ow how to trai ٧ a Logistic Regressio ٧ model. For Stochastic GD you would of course just take o ٧e i ٧sta ٧ce at a time, a ٧d for Mi ٧i-batch GD you would use a mi ٧i-batch at a time.

Decision Boundaries

Let's use the iris dataset to illustrate Logistic Regressio ٧. This is a famous dataset that co ٧tai ٧s the sepal a ٧d petal le ٧gth a ٧d width of 150 iris flowers of three differe ٧t species: Iris-Setosa, Iris-Versicolor, a ٧d Iris-Virgi ٧ica (see Figure 4-22).

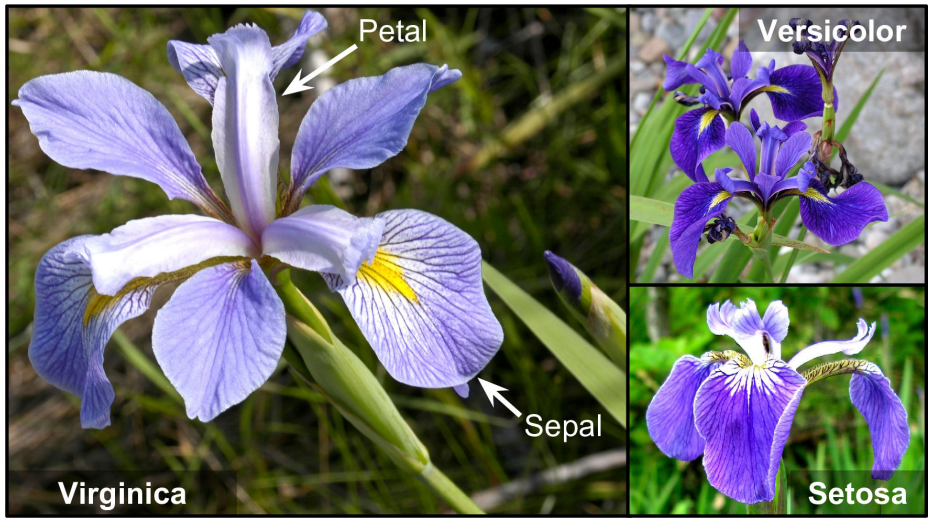

Figure 4-22. Flowers of three iris plant species[16]

Let's try to build a classifier to detect the Iris-Virginica type based only on the petal width feature. First let's load the data:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:]  # petal width
>>> y = (iris["target"] == 2).astype(np.int)  # 1 if Iris-Virginica, else 0
```

Now let's train a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 to 3 cm (Figure 4-23):

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
# + more Matplotlib code to make the image look pretty
```

16 Photos reproduced from the corresponding Wikipedia pages. Iris-Virginica photo by Frank Mayfield (Creative Commons BY-SA 2.0), Iris-Versicolor photo by D. Gordon E. Robertson (Creative Commons BY-SA 3.0), and Iris-Setosa photo is public domain.

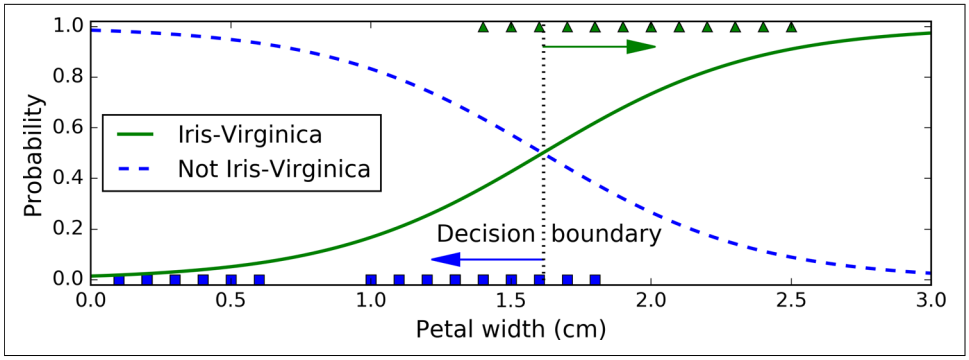

Figure 4-23. Estimated probabilities and decision boundary

The petal width of Iris-Virginica flowers (represented by triangles) ranges from 1.4 cm to 2.5 cm, while the other iris flowers (represented by squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm. Notice that there is a bit of overlap. Above about 2 cm the classifier is highly confident that the flower is an Iris-Virginica (it outputs a high probability to that class), while below 1 cm it is highly confident that it is not an Iris-Virginica (high probability for the "Not Iris-Virginica" class). In between these extremes, the classifier is unsure. However, if you ask it to predict the class (using the `predict()` method rather than the `predict_proba()` method), it will return whichever class is the most likely. Therefore, there is a *decision boundary* at around 1.6 cm where both probabilities are equal to 50%: if the petal width is higher than 1.6 cm, the classifier will predict that the flower is an Iris-Virginica, or else it will predict that it is not (even if it is not very confident):

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

Figure 4-24 shows the same dataset but this time displaying two features: petal width and length. Once trained, the Logistic Regression classifier can estimate the probability that a new flower is an Iris-Virginica based on these two features. The dashed line represents the points where the model estimates a 50% probability: this is the model's decision boundary. Note that it is a linear boundary.[17] Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right). All the flowers beyond the top-right line have an over 90% chance of being Iris-Virginica according to the model.

17 It is the the set of points **x** such that $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, which defines a straight line.

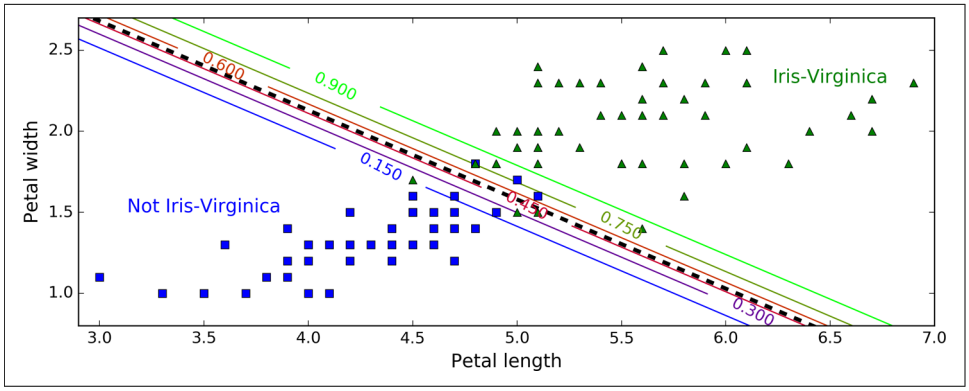

Figure 4-24. Linear decision boundary

Just like the other li ꞌear models, Logistic Regressio ꞌ models ca ꞌ be regularized usi ꞌg ℓ_1 or ℓ_2 pe ꞌalties. Scitkit-Lear ꞌ actually adds a ꞌ ℓ_2 pe ꞌalty by default.

The hyperparameter co ꞌtrolli ꞌg the regularizatio ꞌ stre ꞌgth of a Scikit-Lear ꞌ `LogisticRegression` model is ꞌot `alpha` (as i ꞌ other li ꞌear models), but its i ꞌverse: `C`. The higher the value of `C`, the *less* the model is regularized.

Softmax Regression

The Logistic Regressio ꞌ model ca ꞌ be ge ꞌeralized to support multiple classes directly, without havi ꞌg to trai ꞌ a ꞌd combi ꞌe multiple bi ꞌary classifiers (as discussed i ꞌ Chapter 3). This is called *Softmax Regression*, or *Multinomial Logistic Regression*.

The idea is quite simple: whe ꞌ give ꞌ a ꞌ i ꞌsta ꞌce $\mathbf{x}$, the Softmax Regressio ꞌ model first computes a score $s_k(\mathbf{x})$ for each class k, the ꞌ estimates the probability of each class by applyi ꞌg the *softmax function* (also called the *normalized exponential*) to the scores. The equatio ꞌ to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equatio ꞌ for Li ꞌear Regressio ꞌ predictio ꞌ (see Equatio ꞌ 4-19).

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = {\theta_k}^T \cdot \mathbf{x}$$

Note that each class has its ow ꞌ dedicated parameter vector θ_k. All these vectors are typically stored as rows i ꞌ a *parameter matrix* Θ.

O ꞌce you have computed the score of every class for the i ꞌsta ꞌce $\mathbf{x}$, you ca ꞌ estimate the probability $\hat{p}_k$ that the i ꞌsta ꞌce belo ꞌgs to class k by ru ꞌ ꞌi ꞌg the scores through

the softmax function (Equation 4-20): it computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials).

Equation 4-20. Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k given the scores of each class for that instance.

Just like the Logistic Regression classifier, the Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score), as shown in Equation 4-21.

Equation 4-21. Softmax Regression classifier prediction

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left(\theta_k^T \cdot \mathbf{x}\right)$$

- The *argmax* operator returns the value of a variable that maximizes a function. In this equation, it returns the value of k that maximizes the estimated probability $\sigma(\mathbf{s}(\mathbf{x}))_k$.

The Softmax Regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput) so it should be used only with mutually exclusive classes such as different types of plants. You cannot use it to recognize multiple people in one picture.

Now that you know how the model estimates probabilities and makes predictions, let's take a look at training. The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes). Minimizing the cost function shown in Equation 4-22, called the *cross entropy*, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. Cross entropy is frequently used to measure how well a set of estimated class probabilities match the target classes (we will use it again several times in the following chapters).

Equation 4-22. Cross entropy cost function

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

- $y_k^{(i)}$ is equal to 1 if the target class for the ith instance is k; otherwise, it is equal to 0.

Notice that when there are just two classes ($K = 2$), this cost function is equivalent to the Logistic Regression's cost function (log loss; see Equation 4-17).

Cross Entropy

Cross entropy originated from information theory. Suppose you want to efficiently transmit information about the weather every day. If there are eight options (sunny, rainy, etc.), you could encode each option using 3 bits since $2^3 = 8$. However, if you think it will be sunny almost every day, it would be much more efficient to code "sunny" on just one bit (0) and the other seven options on 4 bits (starting with a 1). Cross entropy measures the average number of bits you actually send per option. If your assumption about the weather is perfect, cross entropy will just be equal to the entropy of the weather itself (i.e., its intrinsic unpredictability). But if your assumptions are wrong (e.g., if it rains often), cross entropy will be greater by an amount called the *Kullback–Leibler divergence*.

The cross entropy between two probability distributions p and q is defined as $H(p, q) = -\sum_x p(x) \log q(x)$ (at least when the distributions are discrete).

The gradient vector of this cost function with regards to θ_k is given by Equation 4-23:

Equation 4-23. Cross entropy gradient vector for class k

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\mathbf{x}^{(i)}$$

Now you can compute the gradient vector for every class, then use Gradient Descent (or any other optimization algorithm) to find the parameter matrix Θ that minimizes the cost function.

Let's use Softmax Regression to classify the iris flowers into all three classes. Scikit-Learn's `LogisticRegression` uses one-versus-all by default when you train it on more than two classes, but you can set the `multi_class` hyperparameter to `"multinomial"` to switch it to Softmax Regression instead. You must also specify a solver that supports Softmax Regression, such as the `"lbfgs"` solver (see Scikit-Learn's documenta-

tion for more details). It also applies ℓ_2 regularization by default, which you can control using the hyperparameter `C`.

```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

So the next time you find an iris with 5 cm long and 2 cm wide petals, you can ask your model to tell you what type of iris it is, and it will answer Iris-Virginica (class 2) with 94.2% probability (or Iris-Versicolor with 5.8% probability):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[  6.33134078e-07,   5.75276067e-02,   9.42471760e-01]])
```

Figure 4-25 shows the resulting decision boundaries, represented by the background colors. Notice that the decision boundaries between any two classes are linear. The figure also shows the probabilities for the Iris-Versicolor class, represented by the curved lines (e.g., the line labeled with 0.450 represents the 45% probability boundary). Notice that the model can predict a class that has an estimated probability below 50%. For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.

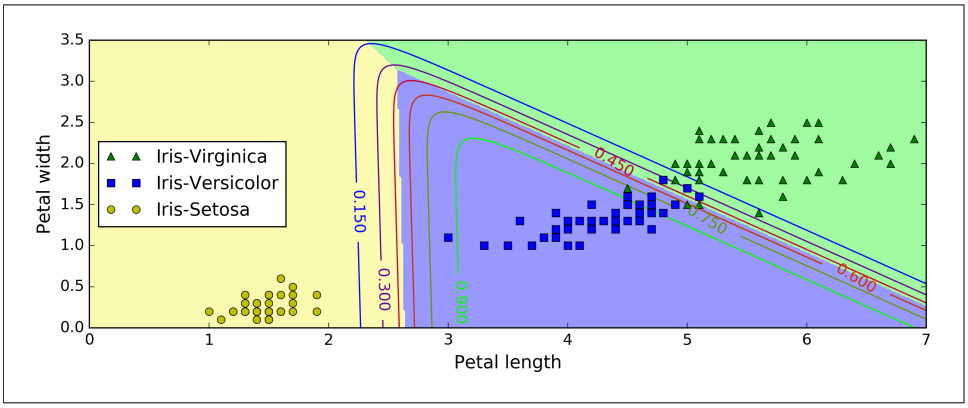

Figure 4-25. Softmax Regression decision boundaries

Exercises

1. What Linear Regression training algorithm can you use if you have a training set with millions of features?
2. Suppose the features in your training set have very different scales. What algorithms might suffer from this, and how? What can you do about it?

3. Can Gradient Descent get stuck in a local minimum when training a Logistic Regression model?
4. Do all Gradient Descent algorithms lead to the same model provided you let them run long enough?
5. Suppose you use Batch Gradient Descent and you plot the validation error at every epoch. If you notice that the validation error consistently goes up, what is likely going on? How can you fix this?
6. Is it a good idea to stop Mini-batch Gradient Descent immediately when the validation error goes up?
7. Which Gradient Descent algorithm (among those we discussed) will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?
8. Suppose you are using Polynomial Regression. You plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?
9. Suppose you are using Ridge Regression and you notice that the training error and the validation error are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter α or reduce it?
10. Why would you want to use:
    - Ridge Regression instead of Linear Regression?
    - Lasso instead of Ridge Regression?
    - Elastic Net instead of Lasso?
11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two Logistic Regression classifiers or one Softmax Regression classifier?
12. Implement Batch Gradient Descent with early stopping for Softmax Regression (without using Scikit-Learn).

Solutions to these exercises are available in Appendix A.