- 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)

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

- $\boldsymbol{\theta}$ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ is the dot product of the vectors $\boldsymbol{\theta}$ and $\mathbf{x}$, which is of course equal to $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$.
- $h_{\boldsymbol{\theta}}$ is the hypothesis function, using the model parameters $\boldsymbol{\theta}$.

In Machine Learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column. If $\boldsymbol{\theta}$ and $\mathbf{x}$ are column vectors, then the prediction is: $\hat{y} = \boldsymbol{\theta}^{\mathbf{T}}\mathbf{x}$, where $\boldsymbol{\theta}^{\mathbf{T}}$ is the *transpose* of $\boldsymbol{\theta}$ (a row vector instead of a column vector) and $\boldsymbol{\theta}^{\mathbf{T}}\mathbf{x}$ is the matrix multiplication of $\boldsymbol{\theta}^{\mathbf{T}}$ and $\mathbf{x}$. It is of course the same prediction, except it is now represented as a single cell matrix rather than a scalar value. In this book we will use this notation to avoid switching between dot products and matrix multiplications.

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 $\boldsymbol{\theta}$ 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 $h_{\boldsymbol{\theta}}$ on a training set $\mathbf{X}$ is calculated using Equation 4-3.

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

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

Most of these notations were presented in Chapter 2 (see "Notations" on page 43). The only difference is that we write $h_{\boldsymbol{\theta}}$ instead of just h in order to make it clear that the model is parametrized by the vector $\boldsymbol{\theta}$. To simplify notations, we will just write MSE($\boldsymbol{\theta}$) instead of MSE($\mathbf{X}$, $h_{\boldsymbol{\theta}}$).

The Normal Equation

To find the value of $\boldsymbol{\theta}$ 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

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

- $\widehat{\boldsymbol{\theta}}$ is the value of $\boldsymbol{\theta}$ 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)
```

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.

2 The demonstration that this returns the value of $\boldsymbol{\theta}$ that minimizes the cost function is outside the scope of this book.

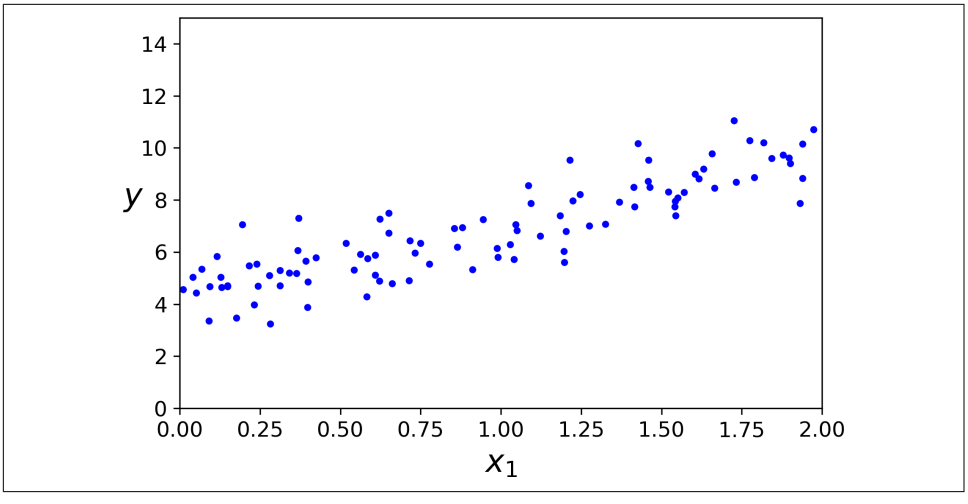

Figure 4-1. Randomly generated linear dataset

Now let's compute $\widehat{\boldsymbol{\theta}}$ 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 $y = 4 + 3x_1 + \text{Gaussian noise}$. 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 $\theta_0 = 4.215$ and $\theta_1 = 2.770$. Close enough, but the noise made it impossible to recover the exact parameters of the original function.

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

```
>>> 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()
```

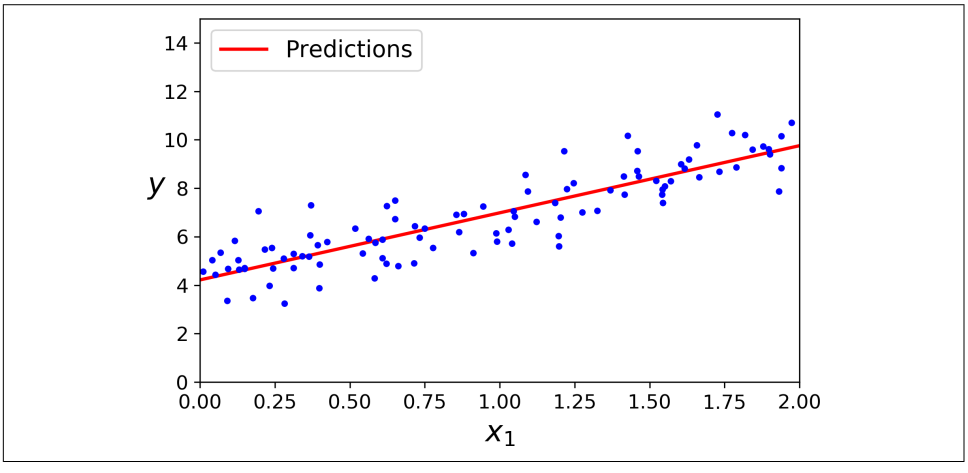

Figure 4-2. Linear Regression model predictions

Performing linear regression using Scikit-Learn is quite simple:[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]])
```

The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name stands for "least squares"), which you could call directly:

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

This function computes $\widehat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$, where $\mathbf{X}^{+}$ is the *pseudoinverse* of $\mathbf{X}$ (specifically the Moore-Penrose inverse). You can use `np.linalg.pinv()` to compute the pseudoinverse directly:

```
>>> np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])
```

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

The pseudoinverse itself is computed using a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices $\mathbf{U}\ \mathbf{\Sigma}\ \mathbf{V}^T$ (see `numpy.linalg.svd()`). The pseudoinverse is computed as $\mathbf{X}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^T$. To compute the matrix $\mathbf{\Sigma}^+$, the algorithm takes $\mathbf{\Sigma}$ and sets to zero all values smaller than a tiny threshold value, then it replaces all the non-zero values with their inverse, and finally it transposes the resulting matrix. This approach is more efficient than computing the Normal Equation, plus it handles edge cases nicely: indeed, the Normal Equation may not work if the matrix $\mathbf{X}^T\mathbf{X}$ is not invertible (i.e., singular), such as if $m < n$ or if some features are redundant, but the pseudoinverse is always defined.

Computational Complexity

The Normal Equation computes the inverse of $\mathbf{X}^T\ \mathbf{X}$, which is an $(n + 1) \times (n + 1)$ 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 SVD approach used by Scikit-Learn's `LinearRegression` class is about $O(n^2)$. If you double the number of features, you multiply the computation time by roughly 4.

Both the Normal Equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000). On the positive side, both are linear with regards to the number of instances in the training set (they are $O(m)$), so they handle 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.

Gradient Descent

Gradient Descent is a very generic 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 $\boldsymbol{\theta}$, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

Concretely, you start by filling $\boldsymbol{\theta}$ 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).

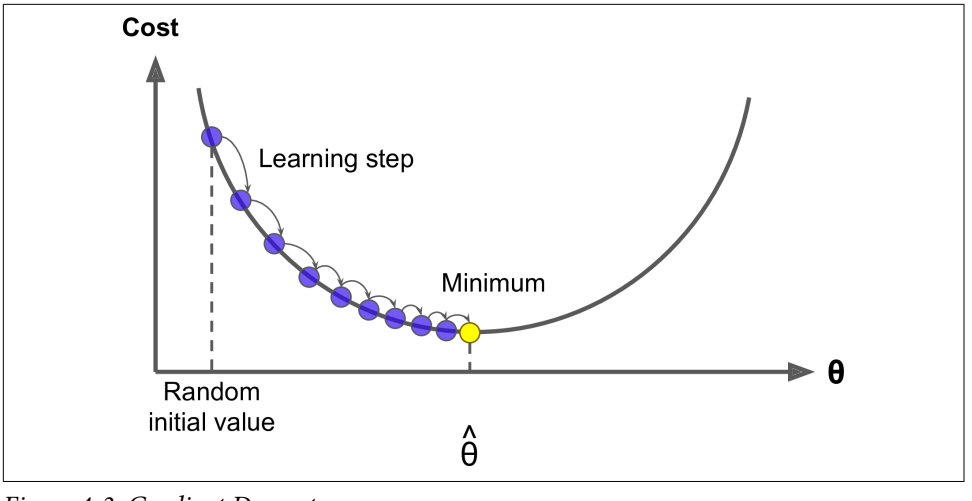

Figure 4-3. Gradient Descent

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).

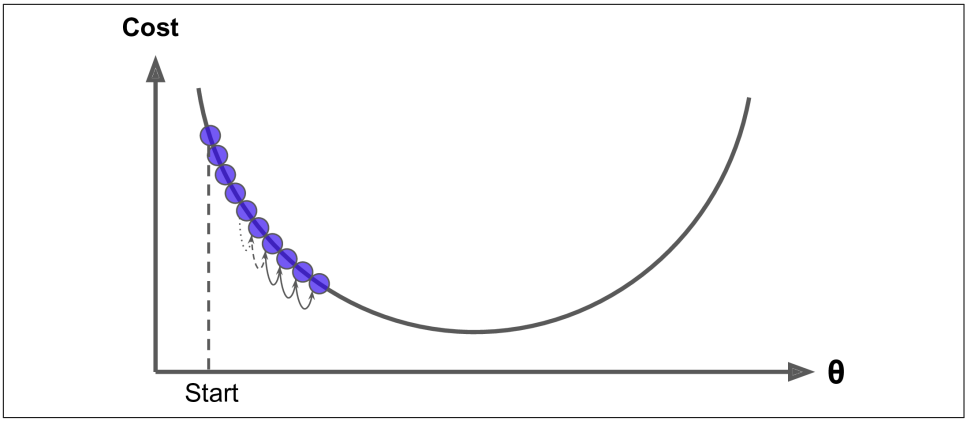

Figure 4-4. Learning rate too small

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).

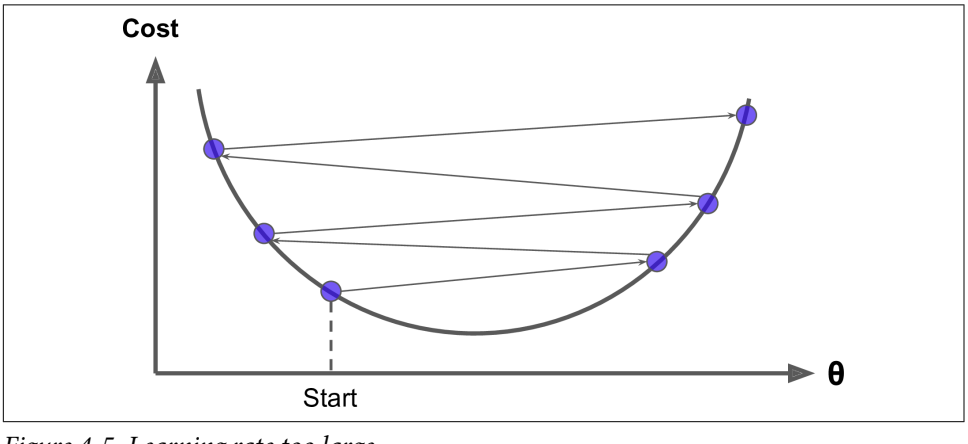

Figure 4-5. Learning rate too large

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.

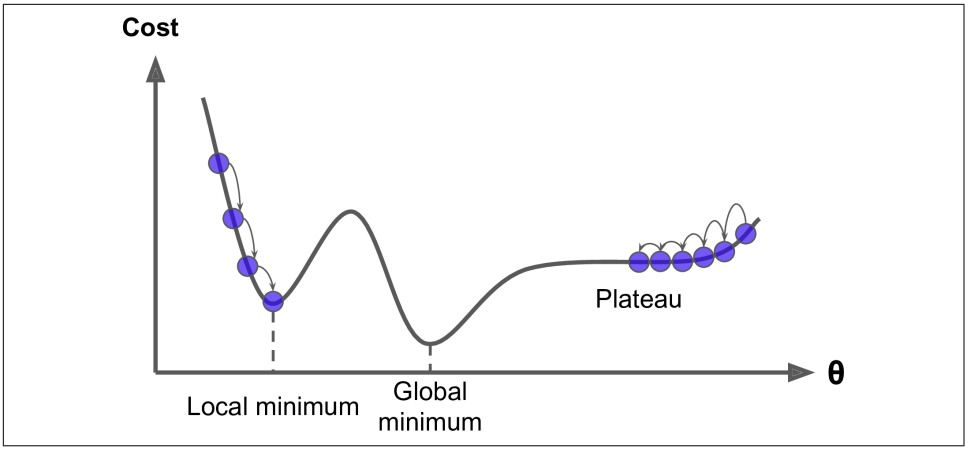

Figure 4-6. Gradient Descent pitfalls

Fortunately, the MSE cost function for a Linear Regression model happens to be a *convex function*, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve. This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly.[4] These two facts have a great consequence: Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure 4-7 shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).[5]

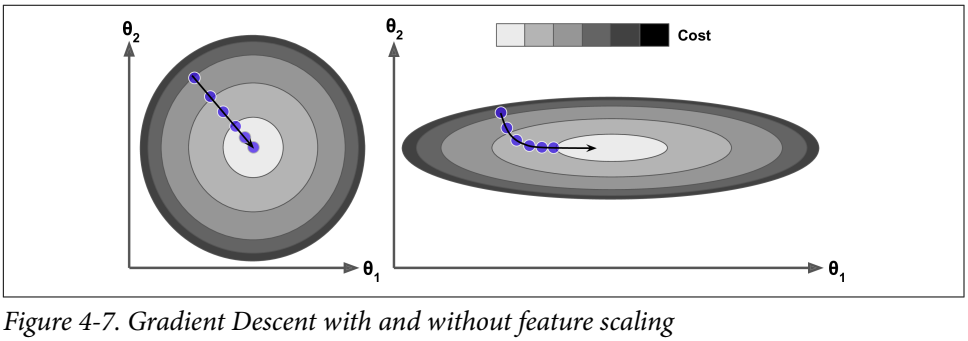

Figure 4-7. Gradient Descent with and without feature scaling

4 Technically speaking, its derivative is *Lipschitz continuous*.

5 Since feature 1 is smaller, it takes a larger change in θ_1 to affect the cost function, which is why the bowl is elongated along 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 an 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 $\frac{\partial}{\partial \theta_j}\text{MSE}(\boldsymbol{\theta})$.

Equation 4-5. Partial derivatives of the cost function

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

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

Equation 4-6. Gradient vector of the cost function

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

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 (actually, *Full Gradient Descent* would probably be a better name). 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 or SVD decomposition.

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_{\boldsymbol{\theta}}\mathrm{MSE}(\boldsymbol{\theta})$ from $\boldsymbol{\theta}$. 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

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

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.