

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

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

- $\hat{\theta}$ 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)
```

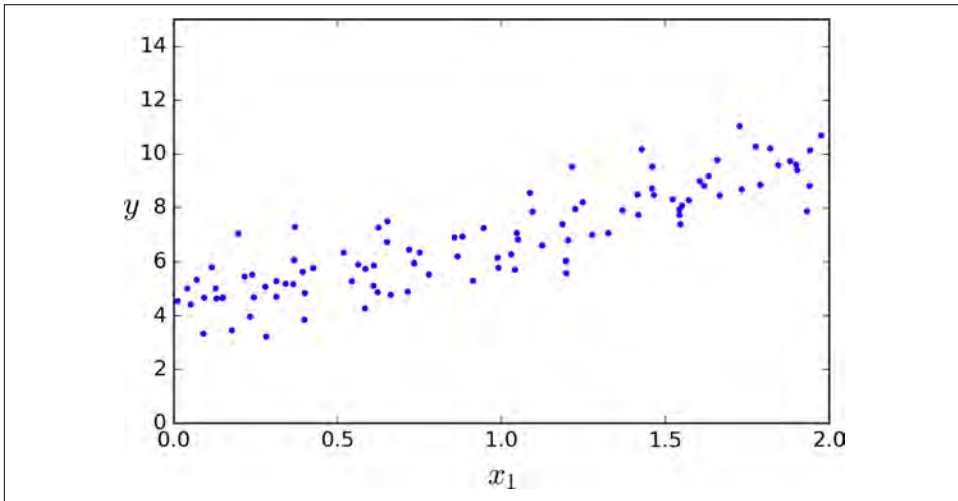


Figure 4-1. Randomly generated linear dataset

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()
```

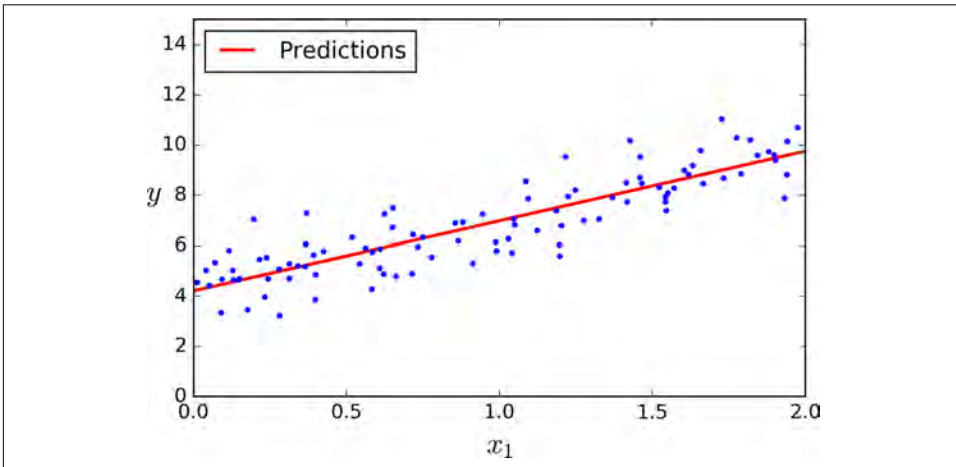


Figure 4-2. Linear Regression model predictions

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_`).