

- 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 non-linear 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, multiply them, and inverse them, 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`.  $\theta_0$  and  $\theta_1$  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*

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

- $\hat{y}$  is the predicted value.

- $n$  is the number of features.
- $x_i$  is the  $i^{\text{th}}$  feature value.
- $\theta_j$  is the  $j^{\text{th}}$  model parameter (including the bias term  $\theta_0$  and the feature weights  $\theta_1, \theta_2, \dots, \theta_n$ ).

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)*

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

- $\boldsymbol{\theta}$  is the model's *parameter vector*, containing the bias term  $\theta_0$  and the feature weights  $\theta_1$  to  $\theta_n$ .
- $\mathbf{x}$  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  $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$ .
- $h_{\theta}$  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:  $\hat{y} = \boldsymbol{\theta}^T \mathbf{x}$ , where  $\boldsymbol{\theta}^T$  is the *transpose* of  $\boldsymbol{\theta}$  (a row vector instead of a column vector) and  $\boldsymbol{\theta}^T \mathbf{x}$  is the matrix multiplication of  $\boldsymbol{\theta}^T$  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).<sup>1</sup>

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

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

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

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

## The Normal Equation

To find the value of  $\theta$  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).<sup>2</sup>

*Equation 4-4. Normal Equation*

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

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

---

<sup>1</sup> 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.

<sup>2</sup> The demonstration that this returns the value of  $\theta$  that minimizes the cost function is outside the scope of this book.

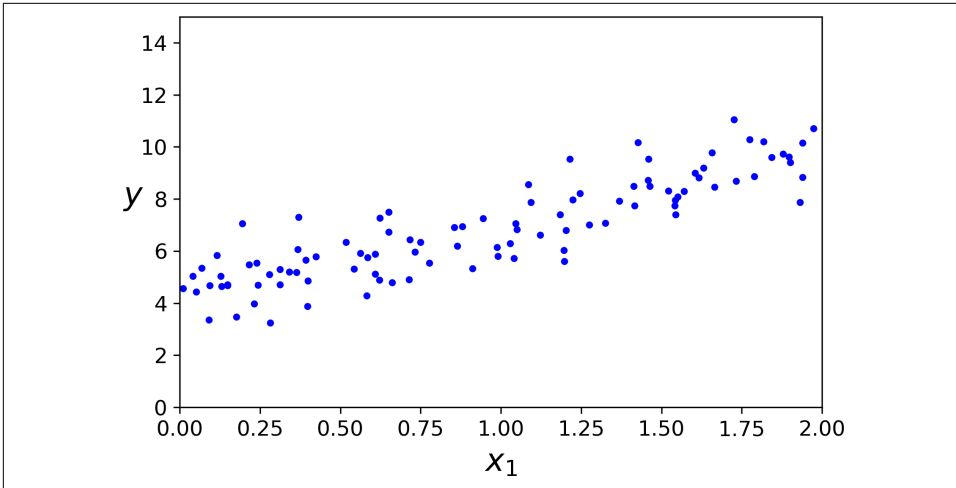


Figure 4-1. Randomly generated linear dataset

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  $y = 4 + 3x_1 + \text{Gaussian noise}$ . 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  $\hat{\theta}$ :

```
>>> 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()
```

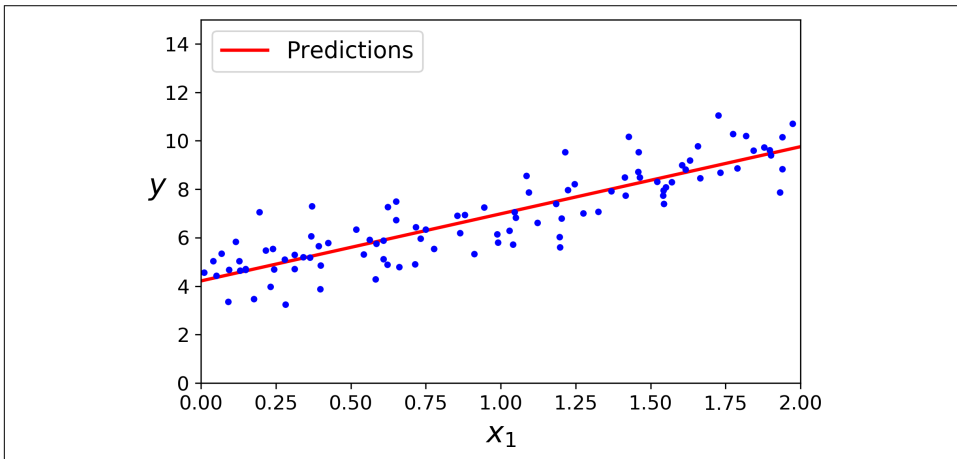


Figure 4-2. Linear Regression model predictions

Performing linear regression using Scikit-Learn is quite simple:<sup>3</sup>

```
>>> 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  $\hat{\theta} = \mathbf{X}^+ \mathbf{y}$ , 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]])
```

---

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