




AI in Biomedical Data

Dr. M.B. Khodabakhshi

Amir Hossein Fouladi

Alireza Javadi

 github.com/mbkhodabakhshi/AI_in_BiomedicalData



یادگیری ماشین در زیست پزشکی

Chapter 4. Training Models

دکتر محمدباقر خدابخشی

mb.khodabakhshi@gmail.com

So far, we have treated Machine Learning models and their training algorithms mostly like **black boxes**, without knowing how they actually work

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.

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**

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):

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

- \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 $\theta_1, \theta_2, \dots, \theta_n$).

This can be written much more concisely using a vectorized form:

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

In this equation:

- $\boldsymbol{\theta}$ 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 $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$.
- h_{θ} is the hypothesis function, using the model parameters $\boldsymbol{\theta}$.

how do we train it?

Training a Linear regression model

In Chapter 2 we saw that the most common performance measure of a regression model is the *Root Mean Square Error (RMSE)*. Therefore, to train a Linear Regression model, we need to find the value of θ that minimizes the RMSE.

In practice, it is simpler to minimize the mean squared error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root).

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

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

If θ and \mathbf{x} are column vectors, then the prediction is $\hat{y} = \theta^T \mathbf{x}$, where θ^T is the transpose of θ (a row vector instead of a column vector) and $\theta^T \mathbf{x}$ is the matrix multiplication of θ^T and \mathbf{x} .

To simplify notations, we will just write $\text{MSE}(\theta)$ instead of $\text{MSE}(\mathbf{X}, h_{\theta})$.

Least Squares Optimization Derivation

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**

$$MSE(\theta) = (X\theta - y)^2$$

$$\frac{d}{d\theta} MSE(\theta) = 2X^T(X\theta - y) = 0$$

$$X^T X \cdot \theta = X^T \cdot y$$

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

Normal Equation

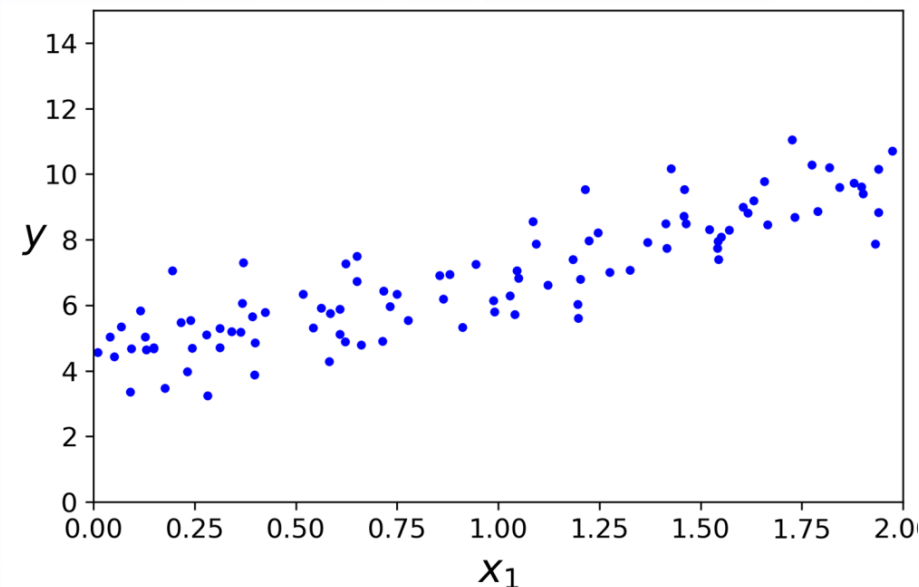
- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- y is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Let's generate some linear-looking data to test this equation on

```
import numpy as np
```

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

Now let's compute θ 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:

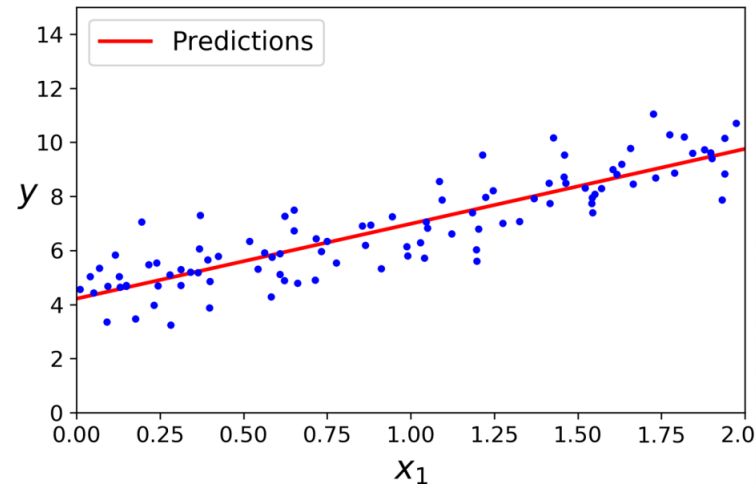


```
X_b = np.c_[np.ones((100, 1)), X] # add  $x_0 = 1$  to each instance  
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

The 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. Let's plot this model's predictions.



We can also compute $\hat{\theta} = X^+y$, where X^+ is the pseudoinverse of 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]])
```

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 nonzero 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

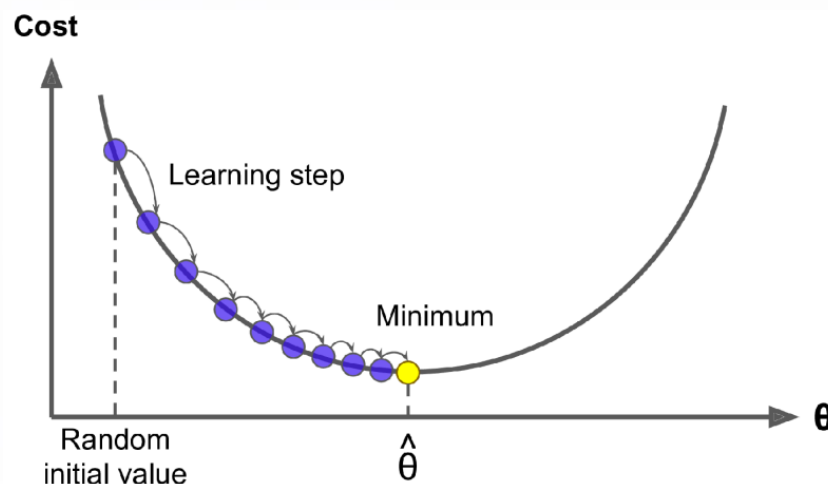
- ❑ The Normal Equation computes the inverse of $X^T X$, which is an $(n + 1) \times (n + 1)$ matrix (where n is the number of features).
- ❑ The computational complexity of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$, depending on the implementation.
- ❑ In other words, if you 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 regard 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.

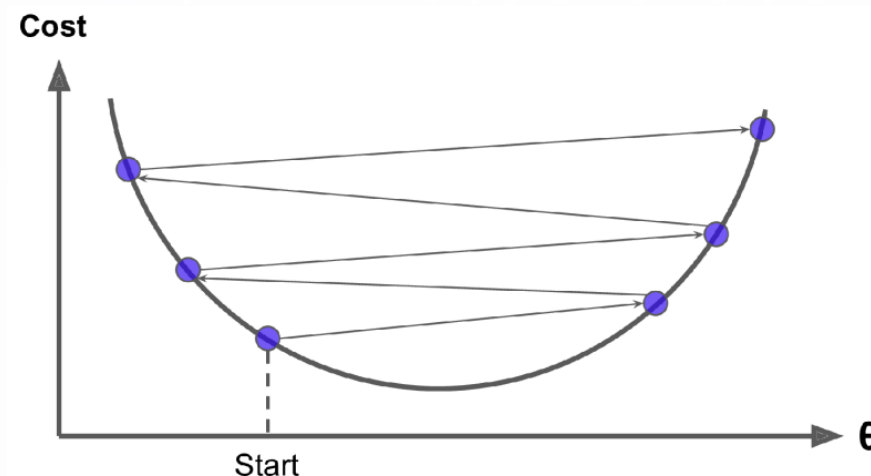
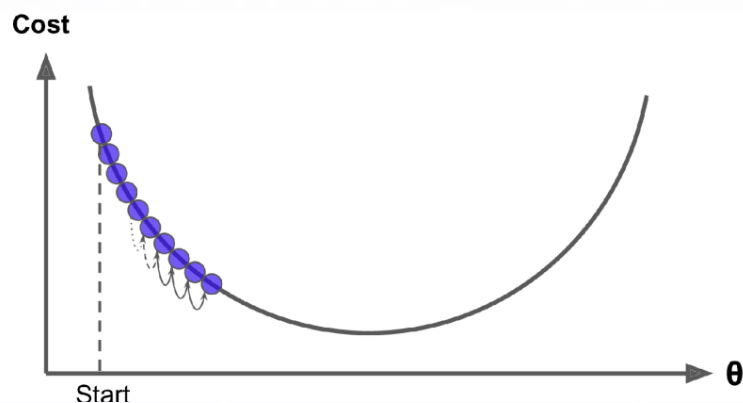
Gradient Descent

- ❖ Now we will look at a very different way to train a Linear Regression model, which is **better suited for cases where there are a large number of features or too many training instances to fit in memory.**
- ❖ *Gradient Descent* is a **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.**
- ❖ It measures **the local gradient of the error function with regard to the parameter vector θ** , and **it goes in the direction of descending gradient.** Once the gradient is zero, you have reached a minimum!



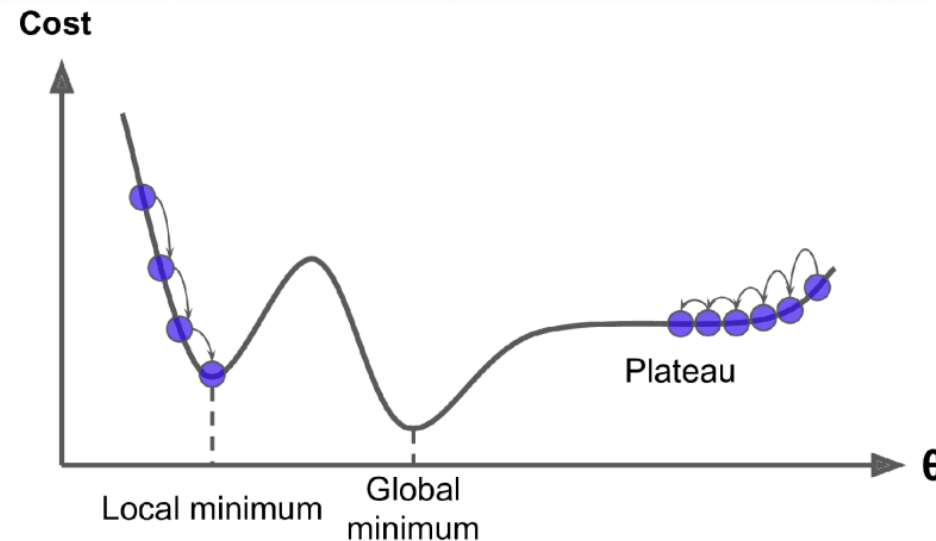
An important parameter in Gradient Descent is **the size of the steps**, determined by the *learning rate* hyperparameter. **If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time.**

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



There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult.

- ✓ 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.



- ❑ 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.
- ❑ 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).

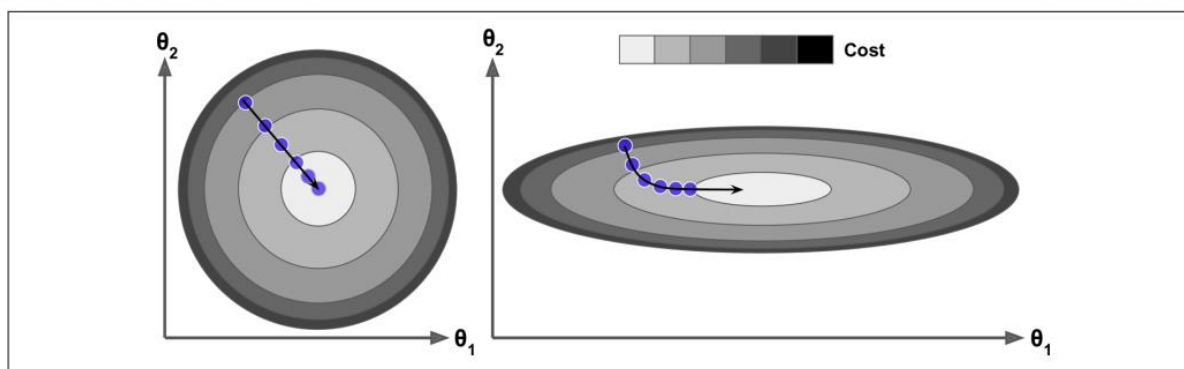


Figure 4-7. Gradient Descent with (left) and without (right) feature scaling

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.

Batch Gradient Descent

To implement Gradient Descent, you need to compute the gradient of the cost function with regard 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*.

$$\frac{\partial}{\partial \theta_i} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^m \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$
$$\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill.

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

```
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
```

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

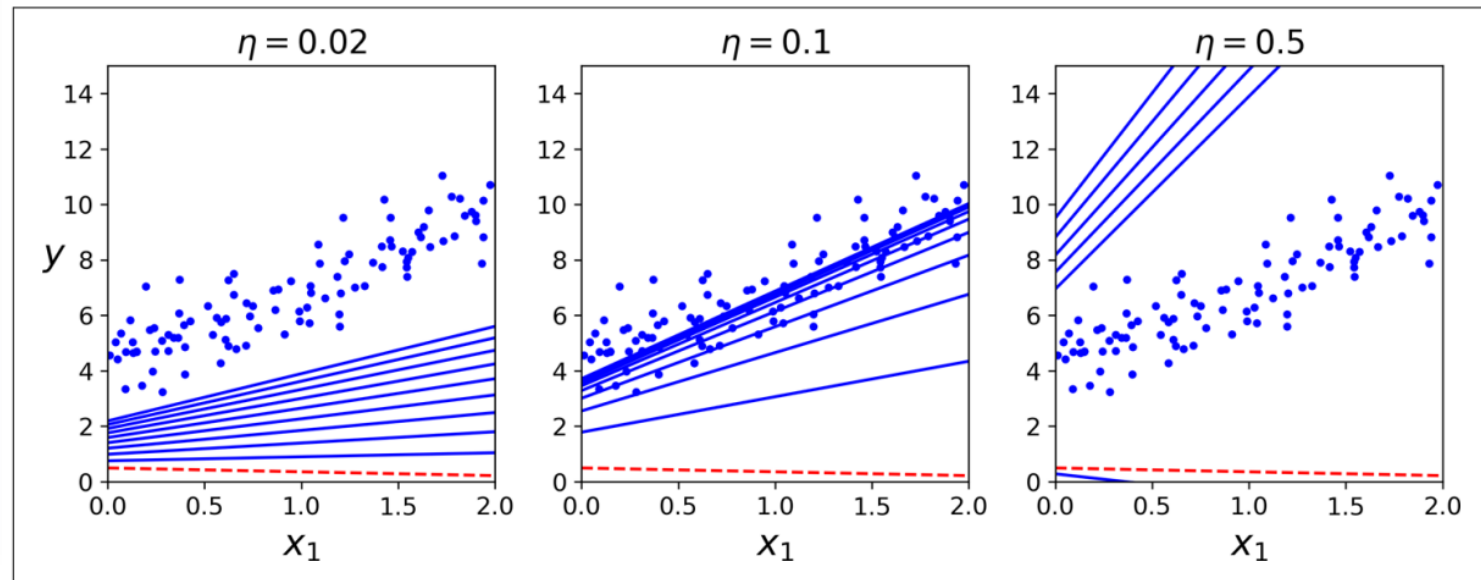
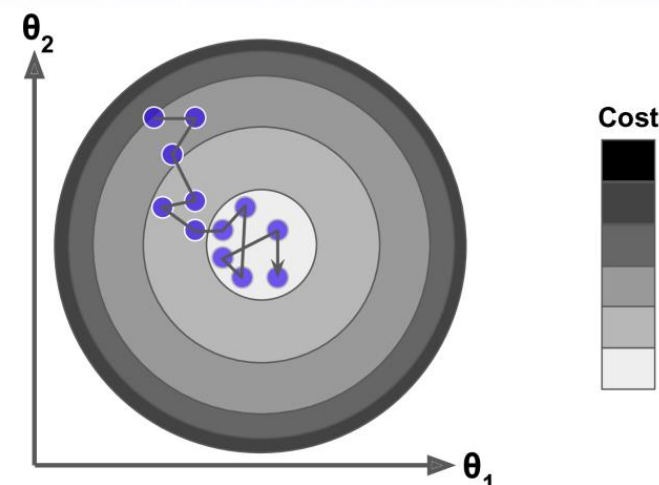


Figure 4-8. Gradient Descent with various learning rates

Stochastic Gradient Descent

- ❖ The 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.
- ❖ *Stochastic Gradient Descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance.
- ❖ Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to main manipulate at every iteration.
- ❖ Due to its stochastic (i.e., random) nature, instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average.

When the cost function is very irregular, this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.



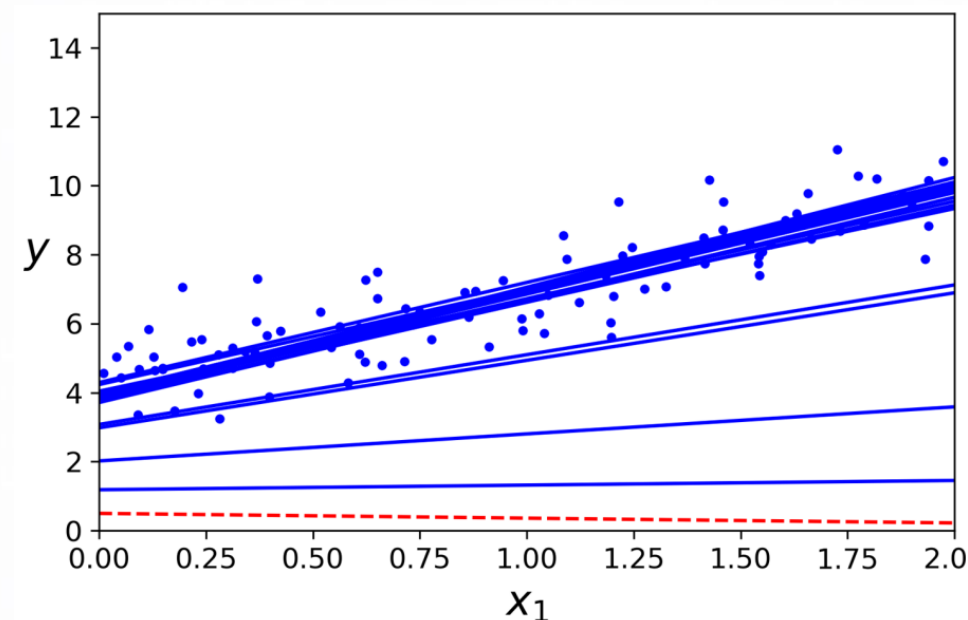
- ❖ Therefore, randomness is good to escape from local optima, but bad because **it means that the algorithm can never settle at the minimum.**
- ❖ One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then **get smaller and smaller, allowing the algorithm to settle at the global minimum.**
- ❖ The function that determines the learning rate at each iteration is called the *learning 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
```



The first 20 steps of Stochastic Gradient Descent

When using Stochastic Gradient Descent, **the training instances must be independent and identically distributed (IID)** to ensure that the parameters get pulled toward the global optimum, on average.

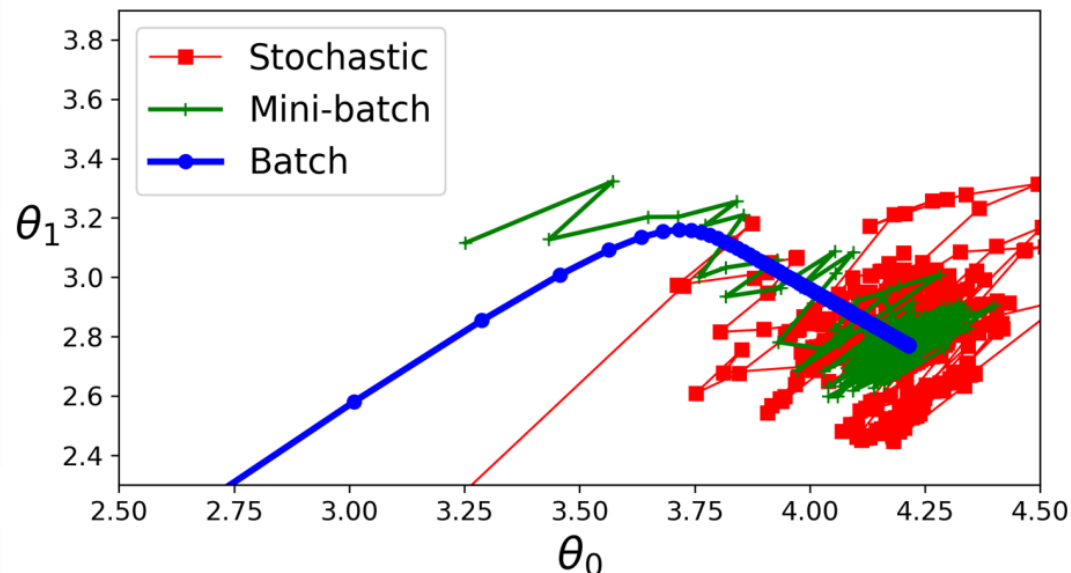
- ✓ A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch).
- ✓ If you do not shuffle the instances—for example, if the instances are sorted by label—then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum.

```
from sklearn.linear_model import SGDRegressor  
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)  
sgd_reg.fit(X, y.ravel())
```

```
>>> sgd_reg.intercept_, sgd_reg.coef_  
(array([4.24365286]), array([2.8250878]))
```


Mini-batch 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.*
- 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.



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.

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	N/A
SVD	Fast	No	Slow	0	No	LinearRegression
Batch GD	Slow	No	Fast	2	Yes	SGDRegressor
Stochastic GD	Fast	Yes	Fast	≥ 2	Yes	SGDRegressor
Mini-batch GD	Fast	Yes	Fast	≥ 2	Yes	SGDRegressor



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

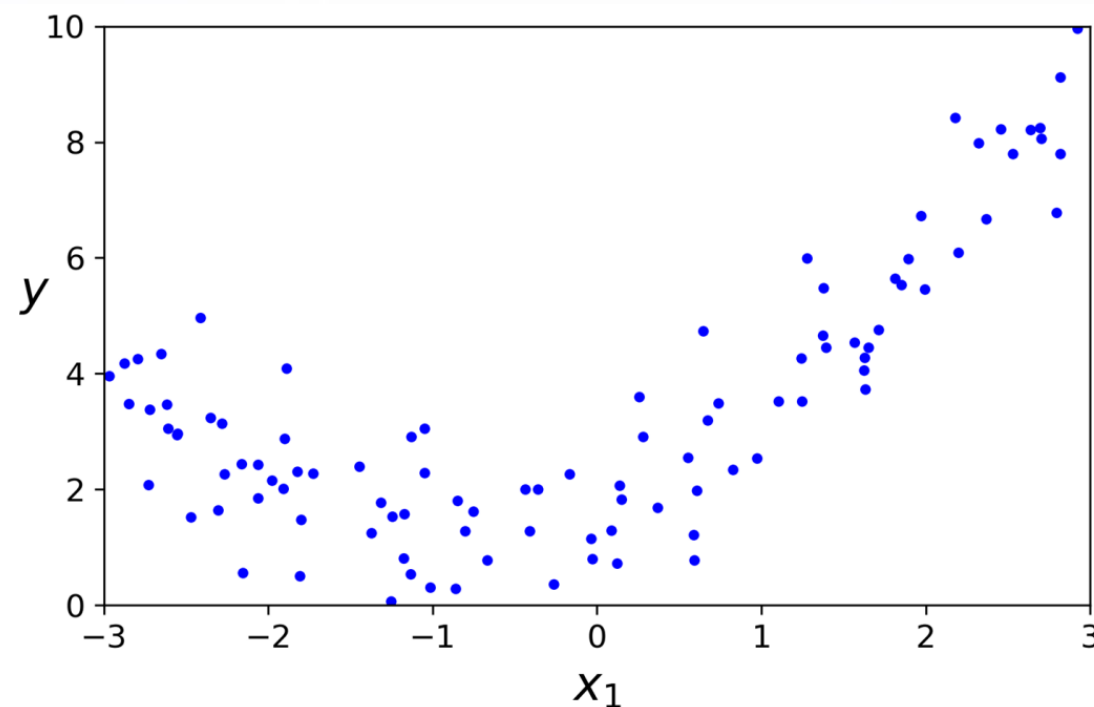
What if your data is more complex than a straight line?

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 generate some nonlinear data, based on a simple quadratic equation, plus some noise:

$$y = ax^2 + bx + c.$$

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

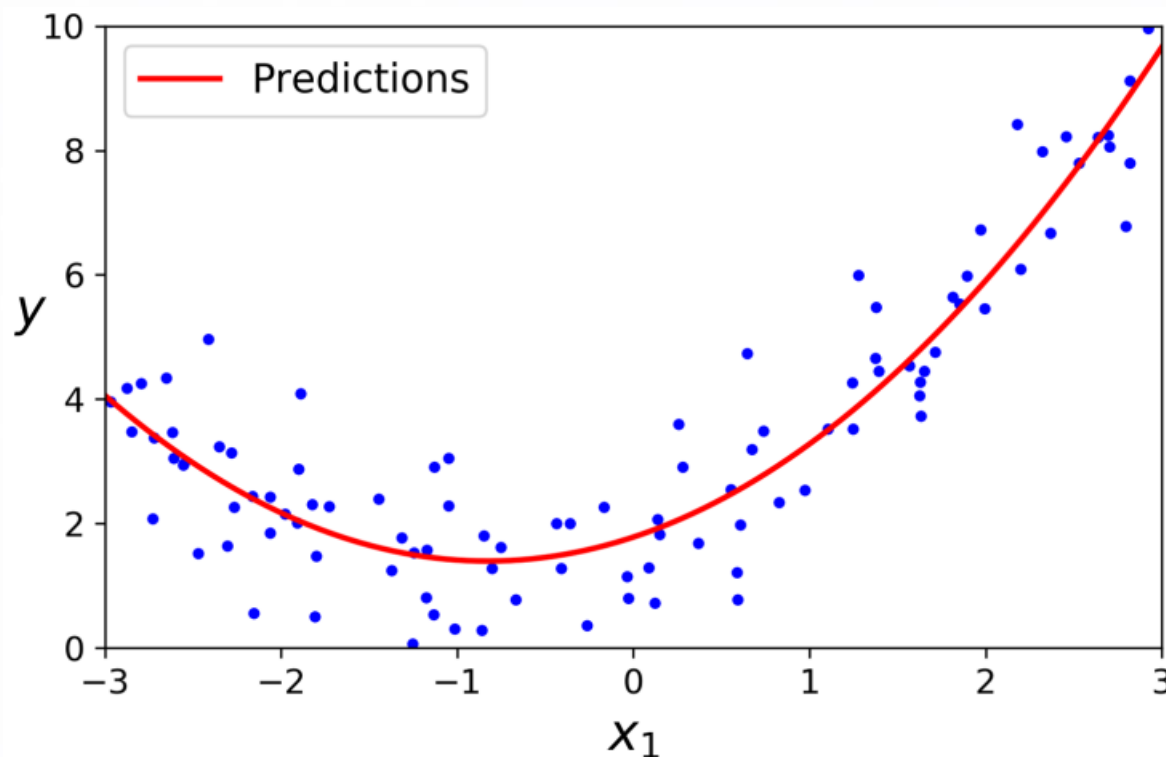


Let's use Scikit-Learn's *PolynomialFeatures* class to transform our training data, adding the square (seconddegree polynomial) of each feature in the training set as a new feature (in this case there is just one feature):

```
>>> 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 contains the original feature of *X* plus the square of this feature. Now you can fit a *LinearRegression* model to this extended training data

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



Not bad: the model estimates $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ when in fact the original function was $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$.

Note that when there are multiple features, Polynomial Regression is capable of finding relationships between features (which is something a plain Linear Regression model cannot do). This is made possible by the fact that `PolynomialFeatures` also adds all combinations of features up to the given 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 $(n + d)! / d!n!$ features, where $n!$ is the *factorial* of n , equal to $1 \times 2 \times 3 \times \dots \times n$. Beware of the combinatorial explosion of the number of features!

Learning Curves

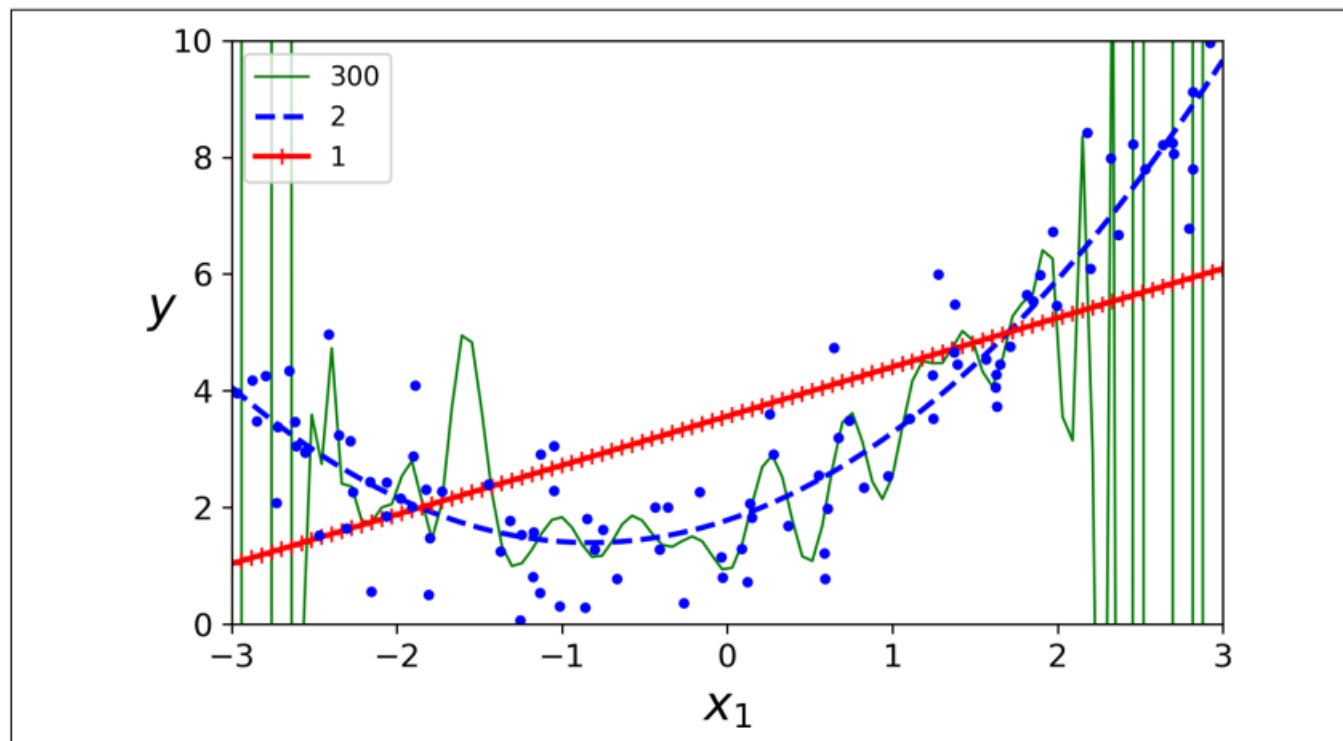


Figure 4-14. High-degree Polynomial Regression

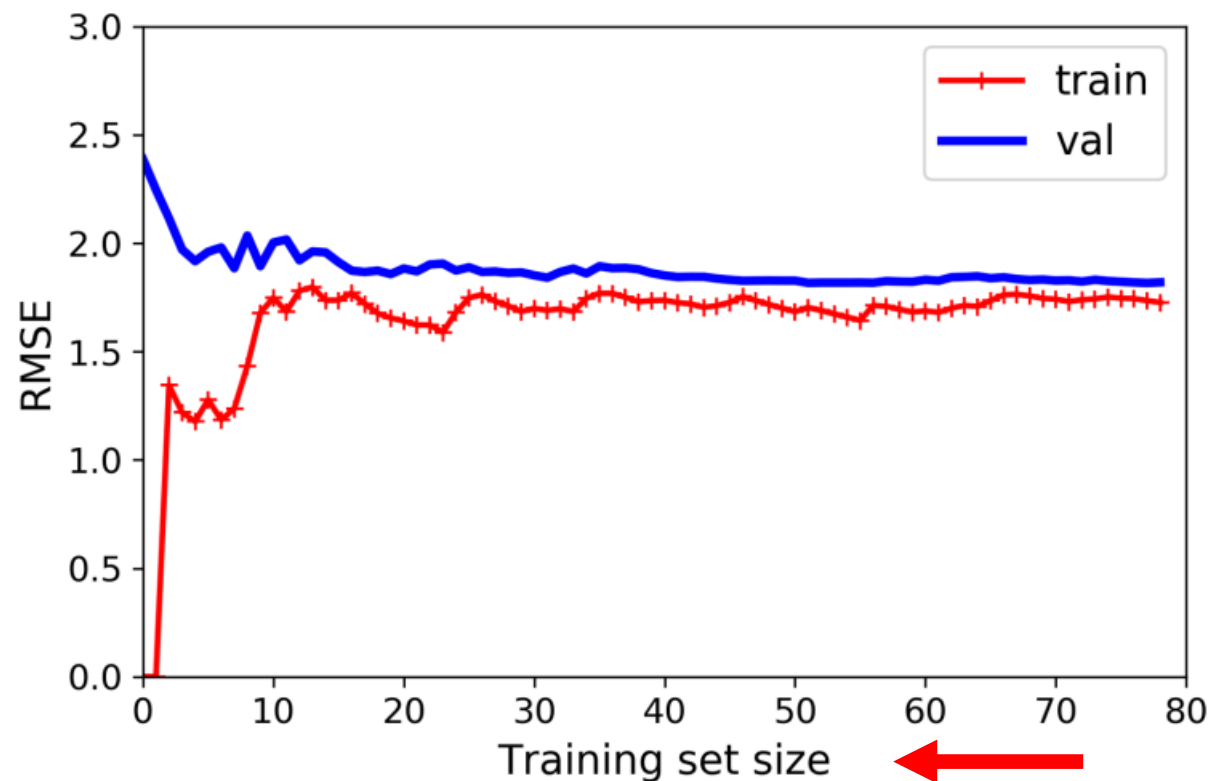
In general, you won't know what function generated the data, so how can you decide how complex your model should be? How can you say that your model is overfitting or underfitting the data

- ❑ In Chapter 2 you used cross-validation to get an estimate of a model's generalization performance.
- ❑ Another way to tell 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 (or the training iteration). The following code defines a function that, **given some training data**, plots the learning curves of a model:

```
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[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

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



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

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 Training Models model is shown more training examples, **it learns, and thus the validation error slowly goes down**.

These learning curves are typical of a model that's underfitting.

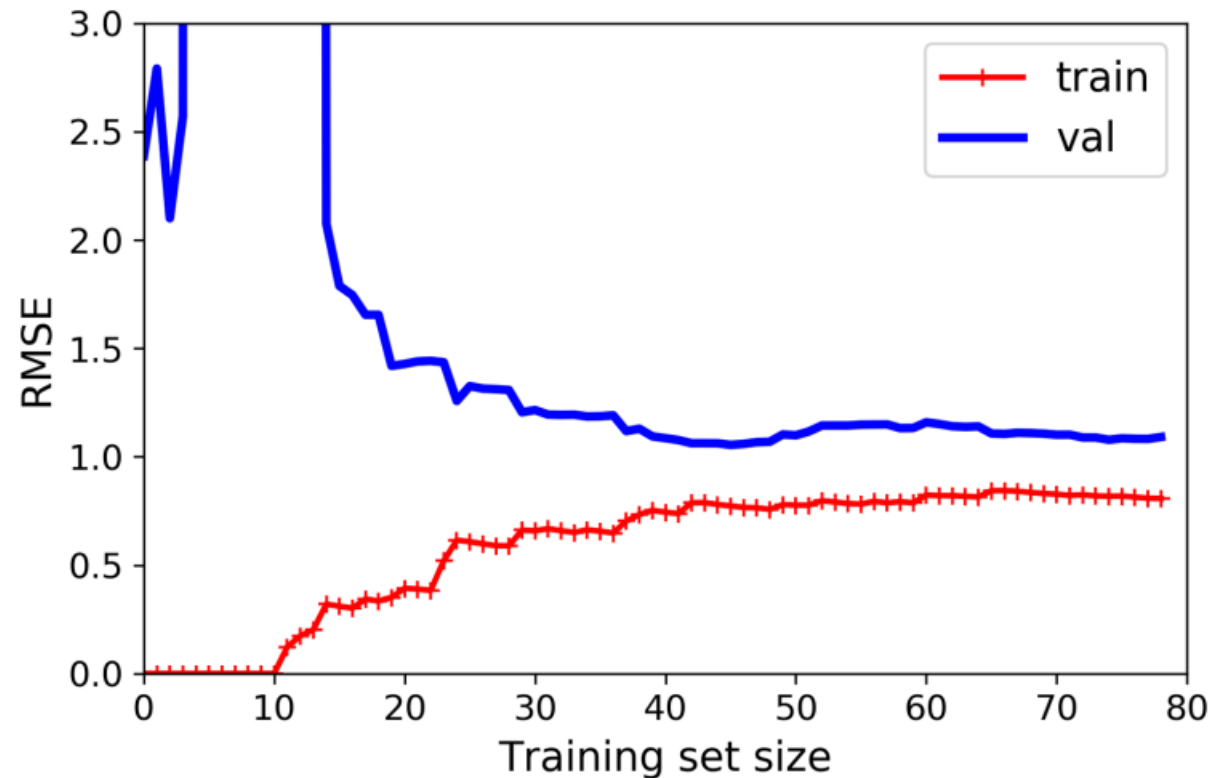
Both curves have reached a **plateau**; they are close and fairly high. If your model is underfitting the training data, adding more training examples will not help. **You need to use a more complex model or come up with better features.**

Learning curves of a 10th-degree polynomial model

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("lin_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. If you used a much larger training set, however, the two curves would continue to get closer

The Bias/Variance Trade-off

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.

✓ 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 overfit the training data.

✓ 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 trade-off.

Regularized Linear Models

As we saw in Chapters 1 and 2, a good way to reduce overfitting is to regularize the model. 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,
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.

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 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.

For example, classifiers are often 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 and the result is a flat line going through the data's mean

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

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define \mathbf{w} as the vector of feature weights (θ_1 to θ_n), then the regularization term is equal to $\frac{1}{2}(\|\mathbf{w}\|_2)^2$, where $\|\mathbf{w}\|_2$ represents the ℓ_2 norm of the weight vector.¹⁰ For Gradient Descent, just add $\alpha \mathbf{w}$ to the MSE gradient vector (**Equation 4-6**).

It is important to scale the data (e.g., using a *StandardScaler*) before performing Ridge Regression, **as it is sensitive to the scale of the input features**. This is true of most regularized models.

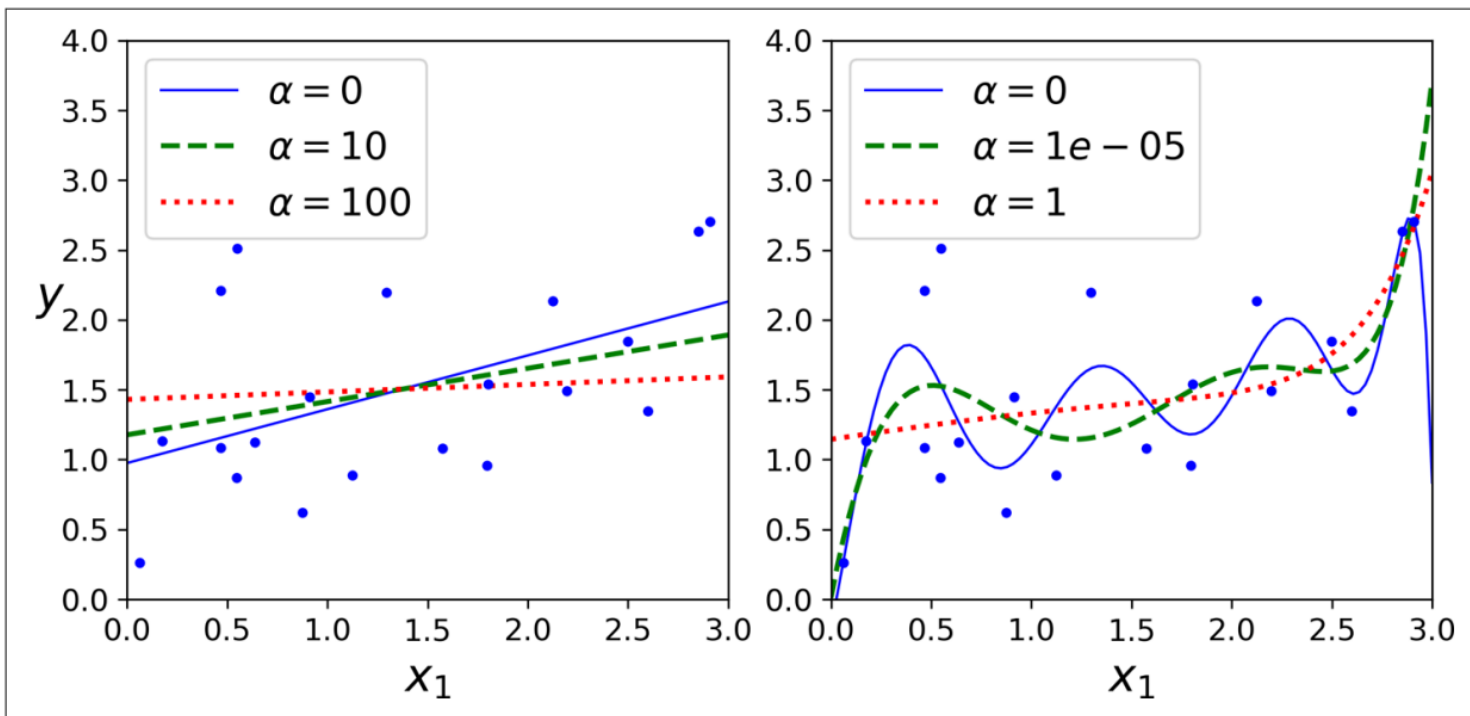


Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

Note how increasing α leads to flatter (i.e., less extreme, more reasonable) predictions, thus reducing the model's variance but increasing its bias.

As with Linear Regression, we can perform Ridge Regression either by computing a *closed-form* equation or by performing Gradient Descent.

The closed-form solution, where A is the $(n + 1) \times (n + 1)$ identity matrix, except with a 0 in the top-left cell, corresponding to the bias term.

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

using Stochastic Gradient Descent:

```
>>> sgd_reg = SGDRegressor(penalty="l2")  
>>> sgd_reg.fit(X, y.ravel())  
>>> sgd_reg.predict([[1.5]])  
array([1.47012588])
```

The penalty hyperparameter sets the type of regularization term to use.

Specifying "l2" indicates that you want SGD to add a regularization term to the cost function equal to half the square of the ℓ_2 norm of the weight vector: this is simply Ridge Regression.

Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression (usually 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

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

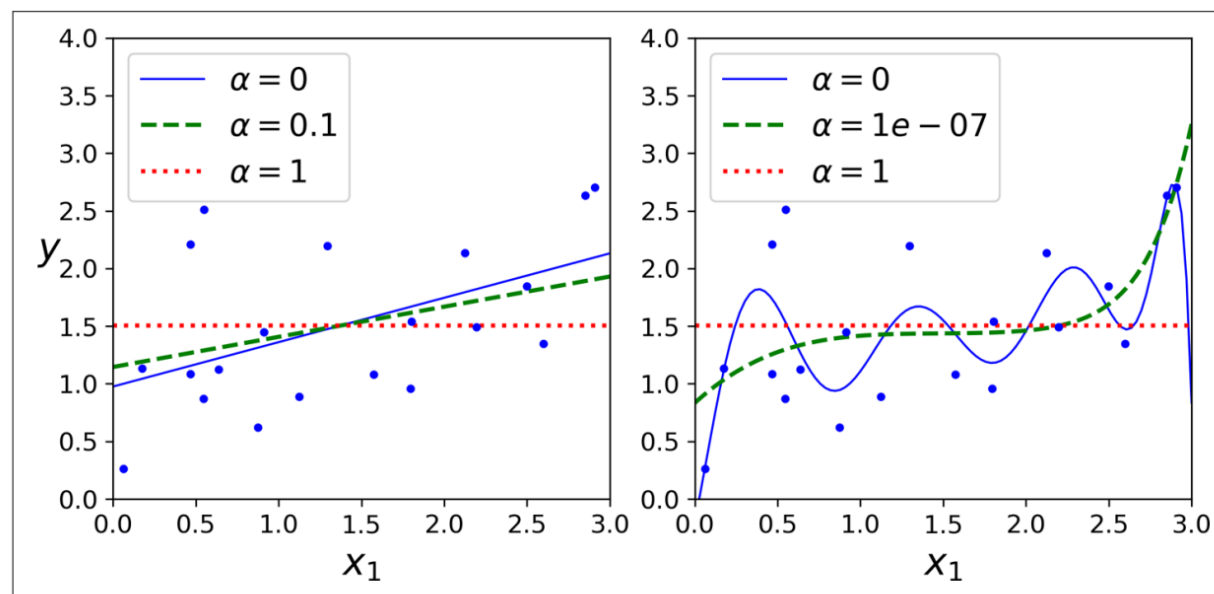


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

- An important characteristic of Lasso Regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero).
- For example, the dashed line in the righthand plot in 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).

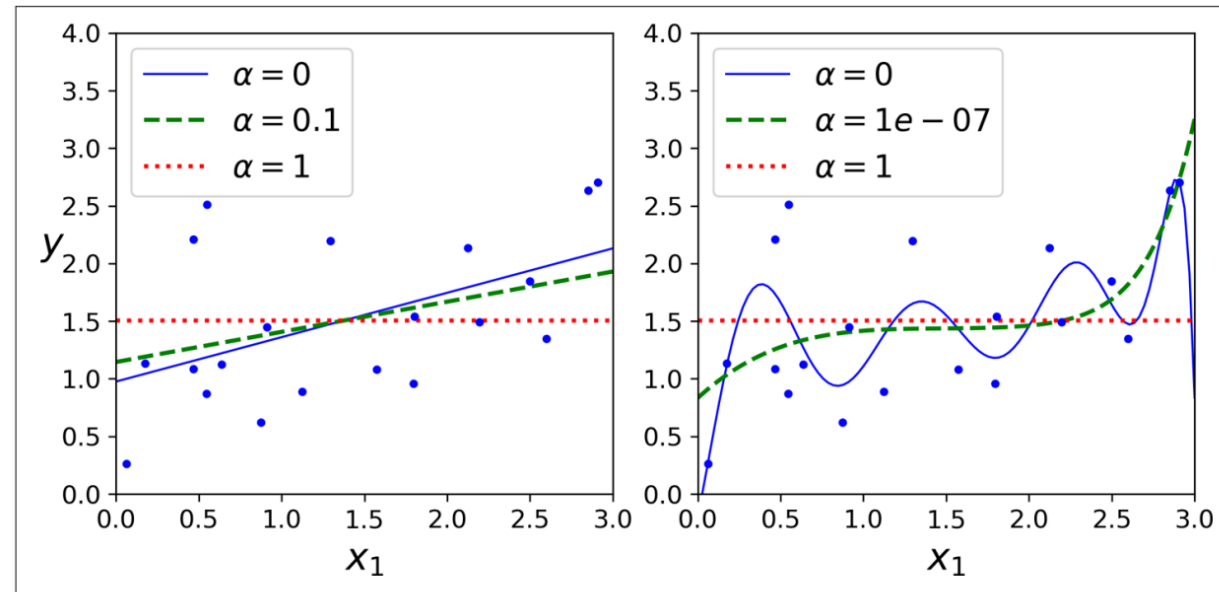


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

The Lasso cost function is not differentiable at $\theta_i = 0$ (for $i = 1, 2, \dots, n$), but Gradient Descent still works fine if you use a subgradient vector g instead when any $\theta_i = 0$.

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

Here is a small Scikit-Learn example using the Lasso class:

```
>>> 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])
```

Note that you could instead use `SGDRegressor(penalty="l1")`.

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

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

- So when should you use plain Linear Regression (i.e., without any regularization), 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 useful, you should prefer Lasso or Elastic Net because they tend to reduce the useless features' weights down to zero, as we have discussed.
- In general, Elastic Net is preferred over Lasso because 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 that uses 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

It is such a simple and efficient regularization technique that **Geoffrey Hinton** called it a “beautiful free lunch.”

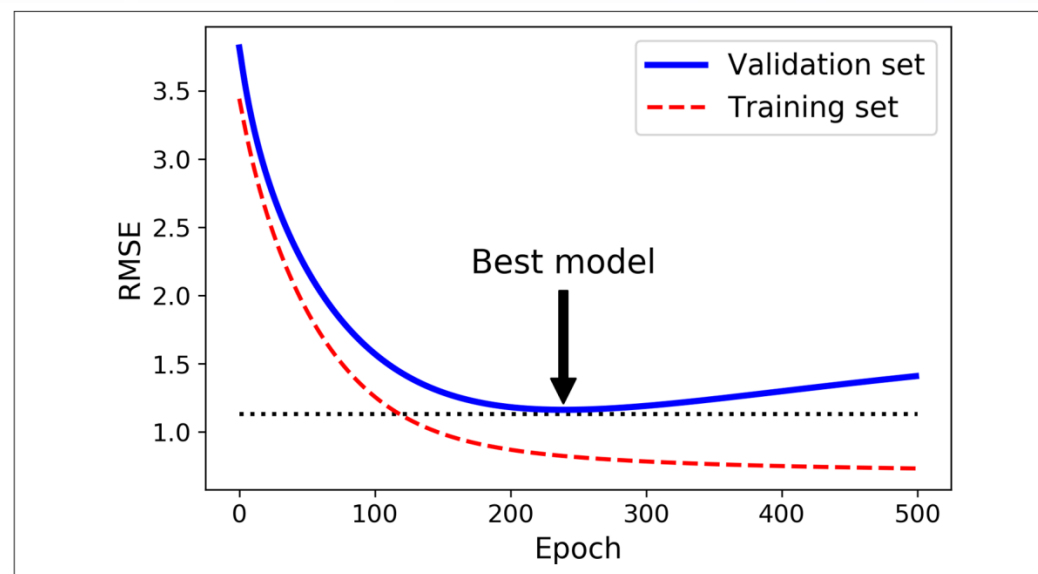


Figure 4-20. Early stopping regularization

- With Stochastic and Mini-batch Gradient Descent, **the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not.**
- One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum

Logistic Regression

Some regression algorithms can be used for classification.

Logistic Regression (also called Logit Regression) is commonly used to estimate the probability that an instance belongs to a particular class.

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”), and otherwise it predicts that it does not (i.e., it belongs to the negative class, labeled “0”).

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.

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

The logistic—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

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

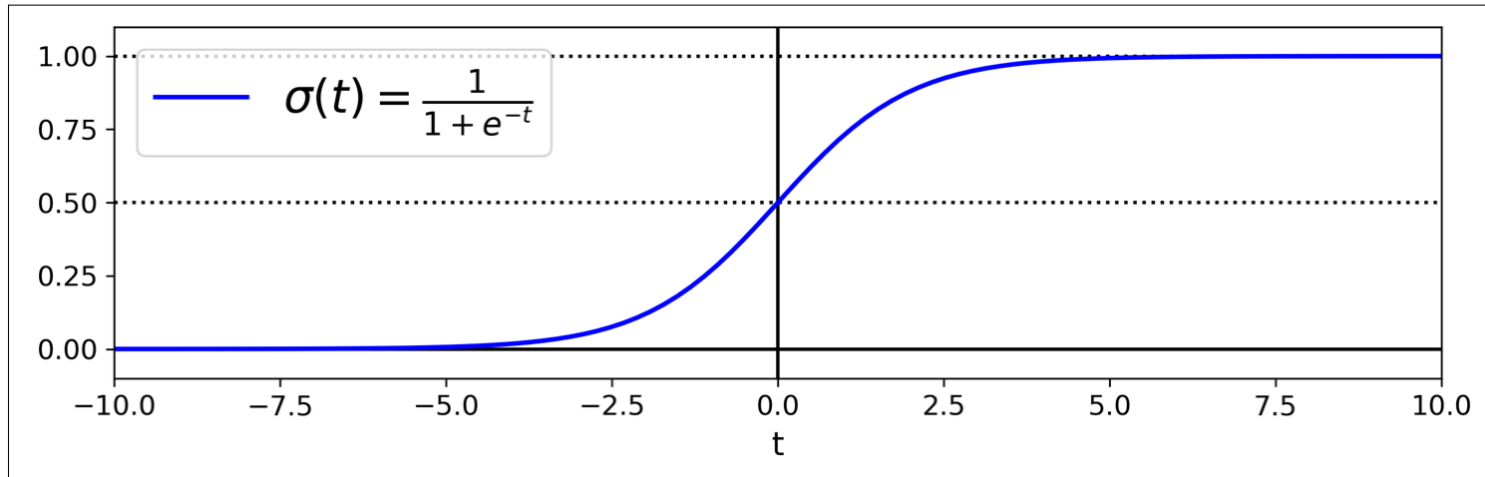


Figure 4-21. Logistic function

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 $\mathbf{x}^\top \boldsymbol{\theta}$ is positive and 0 if it is negative.

Training and Cost Function

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

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



The cost function over the whole training set is the average cost over all training instances. It can be written in a single expression called the *log loss*, shown in Equation 4-17.

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

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

- ❖ The bad news is that **there is no known closed-form equation to compute the value of θ that minimizes this cost function.**
- ❖ The good news is that **this cost function is convex**, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough).
- ❖ The partial derivatives of the cost function with regard to the j 'th model parameter θ_j are given by:

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

For each instance it computes the prediction error and multiplies it by the j 'th feature value, and then it computes the average over all training instances

Once you have the gradient vector containing all the partial derivatives, you can use it in the Batch Gradient Descent algorithm.

For Stochastic GD you would take one instance at a time, and for Mini-batch GD you would use a minibatch at a time

Softmax Regression

The Logistic Regression model **can be generalized to support multiple classes directly**, without having to train and combine multiple binary classifiers (as discussed in Chapter 3). This is called *Softmax Regression*, or Multinomial Logistic Regression.

In Softmax, **each class has its own dedicated parameter vector $\theta^{(k)}$** . All these vectors are typically stored as **rows in a parameter matrix Θ** .

The idea is simple: when given an instance \mathbf{x} , the Softmax Regression model first computes a score $s_k(\mathbf{x})$ for each class k , then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores. The equation to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equation for Linear Regression prediction (see **Equation 4-19**).

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

Once you have computed the score of every class for the instance \mathbf{x} , you can estimate the probability \hat{p}_k that the instance belongs to class k by running the scores through the softmax function (Equation 4-20). The function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials). The scores are generally called logits or log-odds (although they are actually unnormalized log-odds).

Equation 4-20. Softmax function

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

- 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.

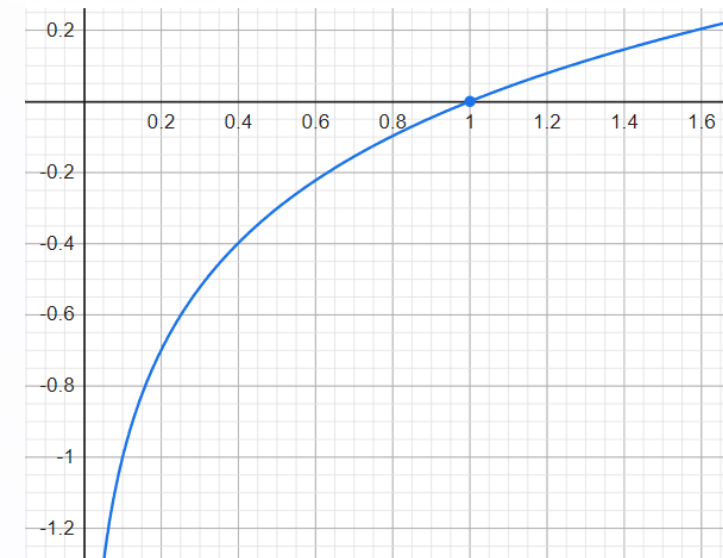
Equation 4-21. Softmax Regression classifier prediction

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

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 matches the target classes.

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

- $y_k^{(i)}$ is the target probability that the i^{th} instance belongs to class k . In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.



The gradient vector of this cost function with regard to $\theta^{(k)}$ is given by **Equation 4-23**.

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

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

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.