Training Models

- Having a good understanding of how machine learning models 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.
- Most of the topics discussed in this chapter will be essential in understanding, building, and training neural networks
- We will discuss two very different ways to train **Linear Regression** model:
  - Using a "**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.
- Discuss a few variants of **gradient descent** : **batch GD**, **minibatch GD,** and **stochastic GD**.
- Discuss **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.
- Discuss two more models that are commonly used for **classification** tasks: **logistic regression** and **softmax regression**.

Linear Regression

- 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)

Equation 4-1. Linear regression model prediction

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

- In this equation:
  - $\hat{\mathbf{y}}$ is the predicted value.
  - **n** is the number of features.
  - $\mathbf{X_i}$ is the $\mathbf{i^{th}}$ feature value.
  - $\boldsymbol{\theta_j}$ is the $\mathbf{j^{th}}$ model parameter, including the bias term $\boldsymbol{\theta_0}$ and the feature weights $\boldsymbol{\theta_1}$, $\boldsymbol{\theta_2}$, ⋯, $\boldsymbol{\theta_n}$.
- **Linear Regression** written in a vectorized form

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

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

In this equation:

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

- how do we train a Linear Regression model?
  - 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.

- we saw that the most common performance measure of a **regression model** is the **root mean square error**
- 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 positive function also minimizes its square root).

- **Learning algorithms** will often **optimize** a different **loss function** during training than the performance measure used to evaluate the final model.
- This is generally because the **function** is easier to **optimize** and/or because it has extra terms needed during training only (e.g., for regularization).
- A good **performance metric** is as close as possible to the final business objective.
- A good **training loss** is easy to **optimize** and strongly correlated with the metric.
- **classifiers** are often trained using a **cost function** such as the **log loss** but evaluated using **precision/recall**. The log loss is easy to minimize, and doing so will usually improve **precision/recall**.

- The **MSE** of a **linear regression hypothesis h_0** on a **training set X** is calculated using:

Equation 4-3. MSE cost function for a linear regression model

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

The Normal Equation

- To find the value of **θ** that minimizes the **MSE**, there exists a **closed-form** solution
- A mathematical equation that gives the result directly. This is called the **Normal equation**

Equation 4-4. Normal equation

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

In this equation:

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

- Computing the **Normal Equation** using 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:

```
from sklearn.preprocessing import add_dummy_feature

X_b = add_dummy_feature(X)  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

- Making Predictions using **θ** and plotting it :

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

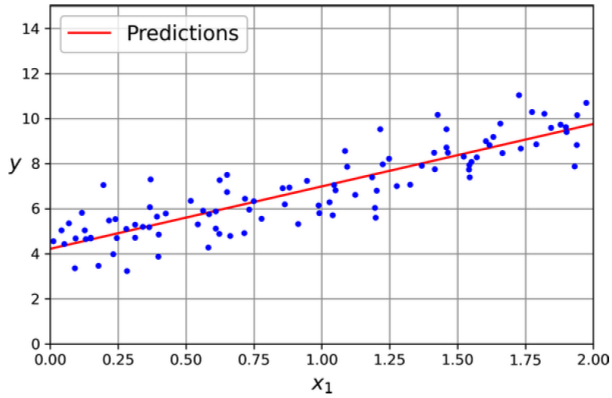


Figure 4-2. Linear regression model predictions

- Performing **linear regression** using Scikit-Learn:

```
>>> 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]])
```

- Scikit-Learn separates the **bias** term (**intercept_)** from the feature weights **(coef_).**
- The **LinearRegression** class is based on the **scipy.linalg.lstsq()** function (the name stands for "**least squares**"):

```
>>> 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 $\boldsymbol{\theta = X^{+}y}$, where $\mathbf{X^{+}}$ is the **pseudoinverse** of **X** You can use **np.linalg.pinv()** to compute the **pseudoinverse** directly:

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

- The **pseudoinverse** is computed using a **standard matrix factorization** technique called **singular value decomposition** (**SVD**) that can decompose the training set matrix **X** into the matrix multiplication of three matrices $\mathbf{U\ \Sigma\ V^{T}}$.
- This approach is more efficient than computing the **Normal equation**, plus it handles edge cases nicely: the **Normal equation** may not work if the matrix $\mathbf{X^{T}\ X}$ is not invertible , such as if **m < n** or if some features are redundant, but the **pseudoinverse** is always defined.

Computational Complexity

- The computational complexity of **Normal Equation** is typically about $\boldsymbol{O(n^{2.4})}$ to $\boldsymbol{O(n^{3})}$**,** 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.

- 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 regard 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 take roughly twice as much time.

Gradient Descent

- we will discuss 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**!
- In practice, you start by filling **θ** with **random** values (this is called **random initialization**). 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.

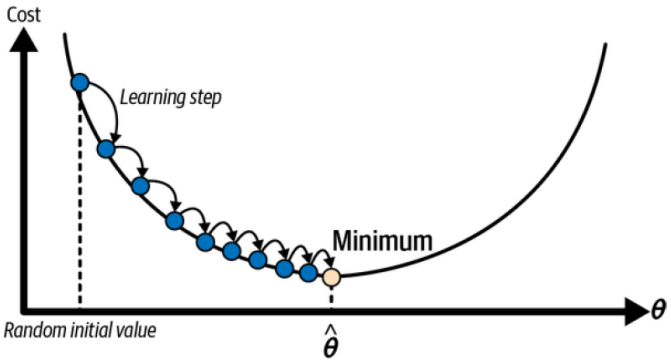


Figure 4-3. In this depiction of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the cost approaches the minimum

- An important parameter in **gradient descen**t 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

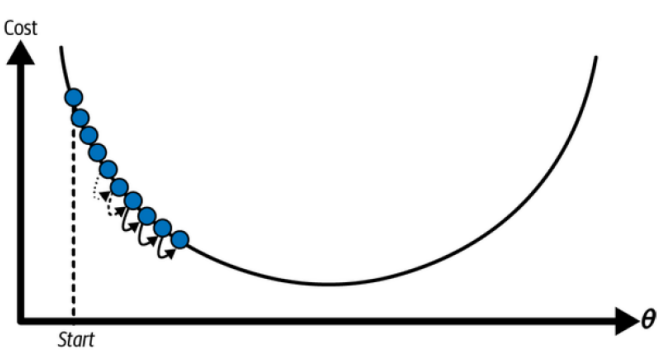


Figure 4-4. Learning rate too small

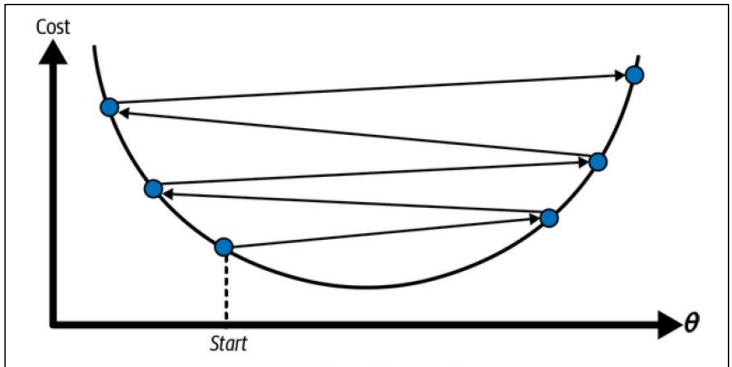


Figure 4-5. Learning rate too high

- Additionally, not all **cost functions** look like nice, **regular bowls**. There may be **holes**, **ridges**, **plateaus**, and all sorts of irregular terrain, making **convergence** to the minimum 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**.

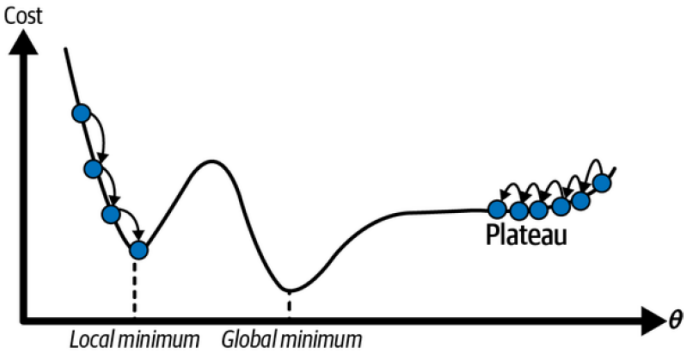


Figure 4-6. Gradient descent pitfalls

- 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 is never below 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 closely the **global minimum** (if you wait long enough and if the learning rate is not too high).
- While the **cost function** has the shape of a bowl, 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).☐
- 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.

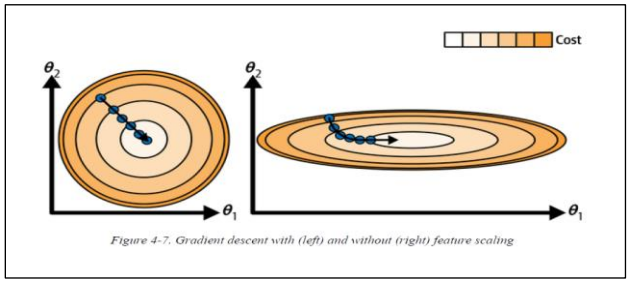


Figure 4-7. Gradient descent with (left) and without (right) feature scaling

- 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.

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**.
- Equation 4-5 computes the partial derivative of the **MSE** with regard to parameter θ_j , noted $\partial\, \mathrm{MSE}(\theta) / \partial\theta_j$.

Equation 4-5. Partial derivatives of the cost function

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

- Equation 4-6 to compute all **partial derivatives** in one go. The **gradient** vector, noted ∇_θ **MSE(θ),** contains all the **partial derivatives** of the **cost function** (one for each model parameter).

Equation 4-6. Gradient vector of the cost function

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

- **batch gradient descent** a formula that involves calculations over the full **training set X**, at each **gradient descent** step, it uses the whole batch of training data at every step.
- As a result, it is terribly slow on very large training sets; However, **gradient descent** scales well with the number of features.

- Once you have the **gradient vector**, which points uphill, just go in the opposite direction to go downhill. This means subtracting **∇_{θ} MSE(θ)**from **θ**. This is where the **learning rate η** comes into play:☐ multiply the **gradient vector** by **η** to determine the size of the downhill step .

Equation 4-7. Gradient descent step

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

- **Batch Gradient Descent implementation:**

```
eta = 0.1  # learning rate
n_epochs = 1000
m = len(X_b)  # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1)  # randomly initialized model parameters

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

- Each iteration over the training set is called an ***epoch***

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

- What happens if you use a different **learning rate** (**eta**)?
  - On the **left**, the **learning rate** is too **low**: the algorithm will eventually reach the solution, but it will take a long time.
  - In the **middle**, the **learning rate** looks pretty **good**: in just a few **epochs**, it has already converged to the solution.
  - On the **right**, the **learning rate** is too **high**: the algorithm **diverges**, jumping all over the place and actually getting further and further away from the solution at every step.

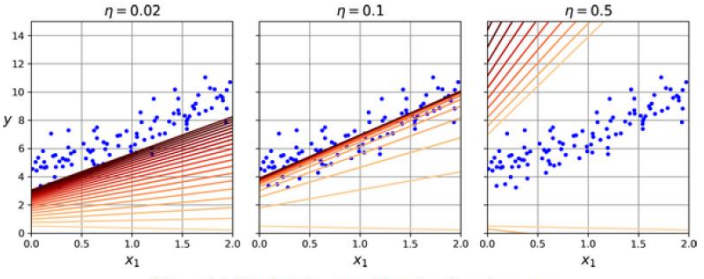


Figure 4-8. Gradient descent with various learning rates

  - To find a good **learning rate**, you can use **grid search**. However, you may want to limit the number of **epochs** so that grid search can eliminate models that take too long to converge.

- You may wonder how to set the number of **epochs**. If it is too low, you will still be far away from the optimal solution when the algorithm stops; but if it is too high, you will waste time while the model parameters do not change anymore.
- A simple solution is to set a very large number of **epochs** but to **interrupt** the algorithm when the **gradient vecto**r becomes **tiny**—that is, when its norm becomes smaller than a tiny number **ϵ** (called the **tolerance**)—because this happens when gradient descent has (almost) reached the **minimum**.

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.
- **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 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 (**stochastic GD** can be implemented as an **out-of-core algorithm**).
- 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. Once the algorithm stops, the final parameter values will be good, but not optimal.

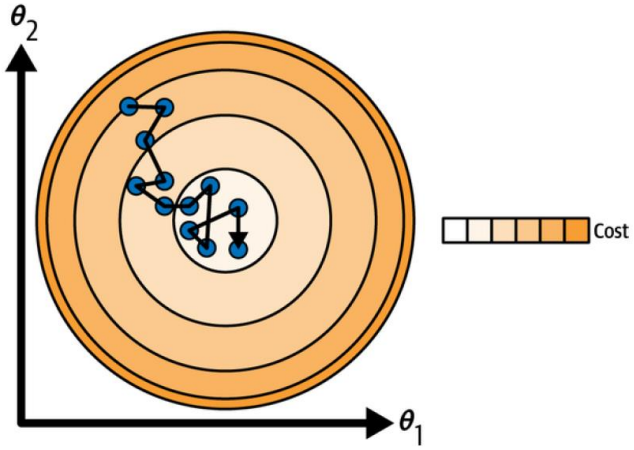


Figure 4-9. With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent

- 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** .
- **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**. If the **learning rate** is reduced too quickly, you may get stuck in a **local minimum**, or even end up frozen halfway to the minimum. If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.
- This code implements **stochastic gradient descent** using a simple **learning schedule**:

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

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

np.random.seed(42)
theta = np.random.randn(2, 1)  # random initialization

for epoch in range(n_epochs):
    for iteration 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 @ (xi @ theta - yi)  # for SGD, do not divide by m
        eta = learning_schedule(epoch * m + iteration)
        theta = theta - eta * gradients
```

- By convention we iterate by rounds of **m** iterations; each round is called an **epoch**.
- While the **batch gradient descent** code iterated **1,000** times through the whole training set, this code goes through the training set only **50 times** and reaches a pretty good solution:

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

- **FIG 4-10** shows the first **20** steps of training (notice how irregular the steps are). Note that since **instances** are picked **randomly**, some **instances** may be picked several times per **epoch**, while others may not be picked at all.

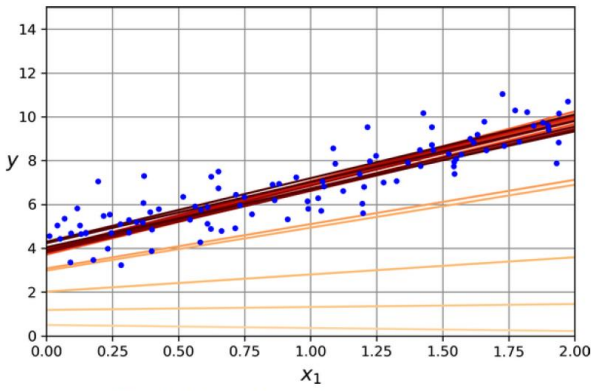


Figure 4-10. The first 20 steps of stochastic gradient descent

- 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,
- 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).

- To perform **linear regression** using **stochastic GD** with Scikit-Learn, you can use the **SGDRegressor** class, which defaults to optimizing the **MSE cost function**.
- The following code runs for maximum **1,000 epochs** (**max_iter**) or until the loss drops by less than **10 (tol)** during **100** epochs (**n_iter_no_change**). It starts with a learning rate of **0.01 (eta0)**, using the **default learning schedule** (different from the one we used). Lastly, it does not use any **regularization** (**penalty**=**None**):

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None, eta0=0.01,
                       n_iter_no_change=100, random_state=42)
sgd_reg.fit(X, y.ravel())  # y.ravel() because fit() expects 1D targets
```

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.21278812]), array([2.77270267]))
```

- **TIP:** All Scikit-Learn estimators can be trained using the **fit()** method, but some estimators also have a **partial_fit()** method that you can call to run a single round of training on one or more instances (it ignores hyperparameters like **max_iter** or **tol**).
- Repeatedly calling **partial_fit()** will gradually train the model. This is useful when you need more control over the training process.
- Other models have a **warm_start hyperparameter** instead (and some have both): if you set **warm_start=True**, calling the **fit()** method on a trained model will not reset the model; it will just continue training where it left off, respecting hyperparameters like **max_iter** and **tol**.
- Note that **fit()** resets the iteration counter used by the **learning schedule**, while partial_fit() does not.

Mini-batch Gradient Descent

- **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**),
- **Minibatch GD** computes the **gradients** on small random sets of instances called **minibatches**.
- 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 **stochastic GD**, especially with fairly large **mini-batches**.
- As a result, **mini-batch GD** will end up walking around a bit closer to the **minimum** than **stochastic GD**—but it may be harder for it to escape from **local minima**
- **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**.

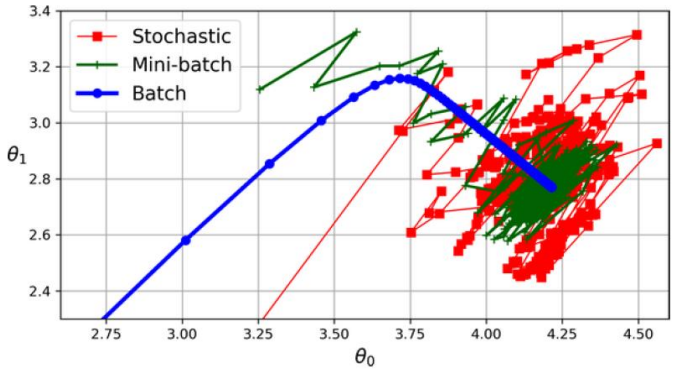


Figure 4-11. Gradient descent paths in parameter space

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?
  - you can also 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**.
- Generate some **nonlinear** based on a simple quadratic equation that's an equation of the form $\mathbf{y = ax^2 + bx + c}$—plus some noise:

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```

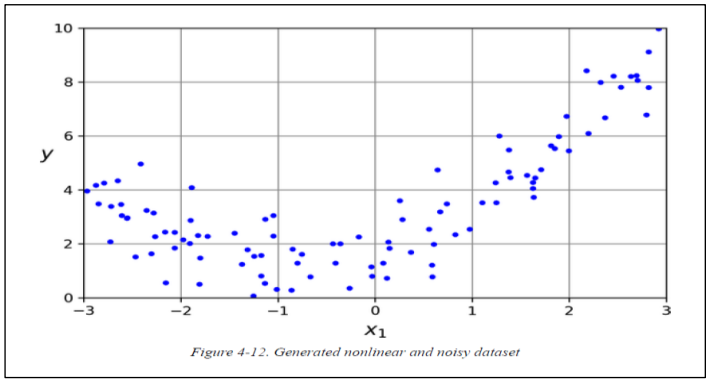


Figure 4-12. Generated nonlinear and noisy dataset

- Clearly, a **straight line** will never **fit** this data properly. So, we will use Scikit-Learn's **PolynomialFeatures** class to transform our training data, adding the square (second-degree 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 we 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]]))
```

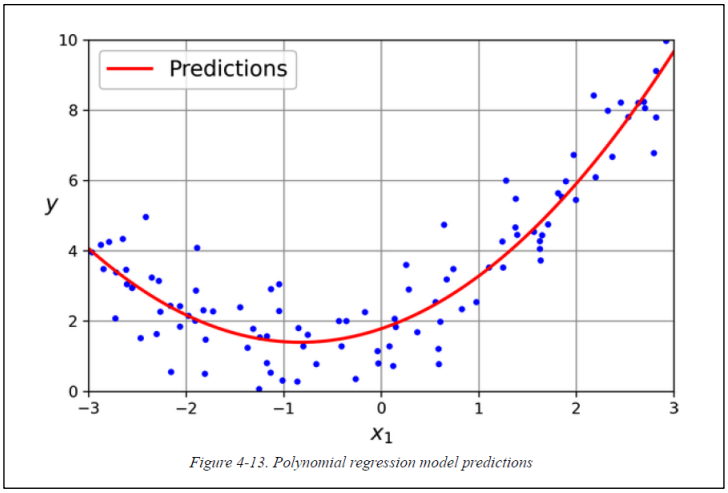


Figure 4-13. Polynomial regression model predictions

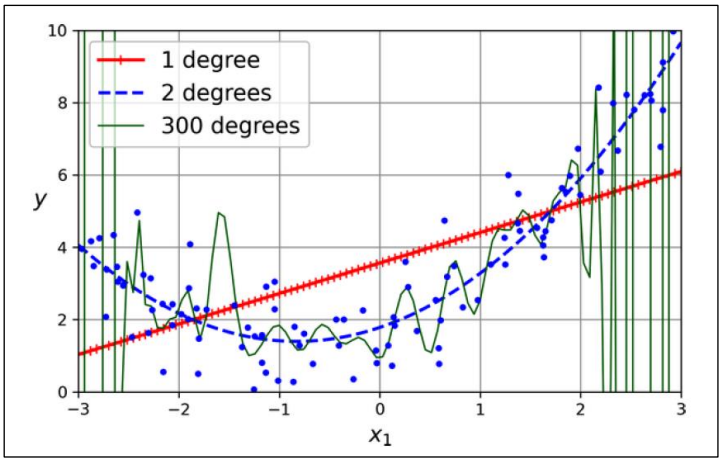
- the model estimates $\mathbf{y = 0.56x_1^2 + 0.93x_1 + 1.78}$ when in fact the original function was $\mathbf{y = 0.5x_1^2 + 1.0x_1 + 2.0 +}$ **Gaussian noise.**
- Note that when there are multiple features, **polynomial regression** is capable of finding relationships between features, which is something a plain linear regression model cannot do.
- This is made possible because **PolynomialFeatures** also adds all combinations of features up to the given degree.

- 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^2 **b**, and **ab**^{2} .

- **WARNING: 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, 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.
- 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**, which makes sense because 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?



- One way is to use **cross-validation** to get an **estimate** of a **model's generalization** performance.
- Another way to tell is to look at the **learning curves**, which are plots of the model's **training error** and **validation error** as a **function** of the training iteration: just **evaluate** the model at **regular intervals** during **training** on both the **training set** and the **validation set** and **plot** the results.
- If the model cannot be trained **incrementally** (i.e., if it does not support **partial_fit()** or **warm_start**), then you must train it several times on gradually larger subsets of the training set.
- Scikit-Learn has a useful **learning_curve()** function to help with this: it trains and evaluates the model using **cross-validation**. By default, it retrains the model on growing subsets of the training set, but if the model supports **incremental learning** you can set **exploit_incremental_learning=True** when calling **learning_curve()** and it will train the model **incrementally** instead.
- The function returns the **training set** sizes at which it **evaluated** the model, and the **training** and **validation scores** it measured for each size and for each **cross-validation** fold.

```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, valid_scores = learning_curve(
    LinearRegression(), X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
train_errors = -train_scores.mean(axis=1)
valid_errors = -valid_scores.mean(axis=1)

plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="train")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="valid")
[...] # beautify the figure: add labels, axis, grid, and legend
plt.show()
```

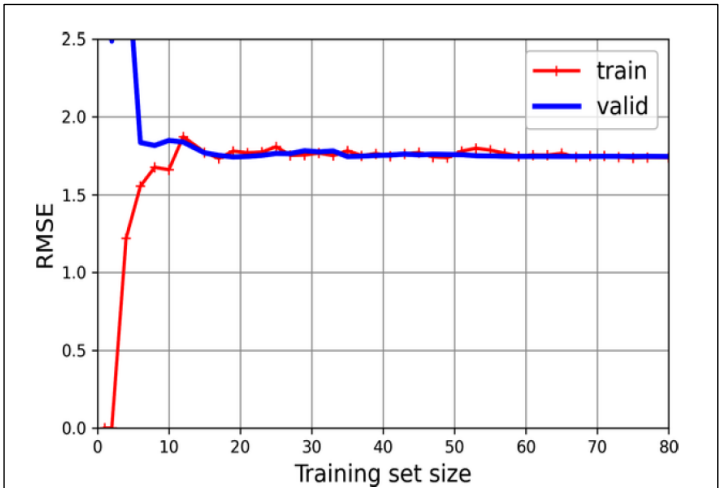


Figure 4-15. Learning curves

- This model is **underfitting**. To see why, first let's look at the **training error**. 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 se**t doesn't make the average error much better or worse.
- Now let's look at the **validation error**. 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 large. 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 cannot do a good job of modeling the data, so the error ends up at a **plateau**, very close to the other curve.
- These **learning curves** are typical of a model that's **underfitting**. Both curves have reached a **plateau**; they are close and fairly high.

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

- Create **learning curves** of a **10th-degree polynomial** model on the same data:

```
from sklearn.pipeline import make_pipeline

polynomial_regression = make_pipeline(
    PolynomialFeatures(degree=10, include_bias=False),
    LinearRegression())

train_sizes, train_scores, valid_scores = learning_curve(
    polynomial_regression, X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
[...] # same as earlier
```

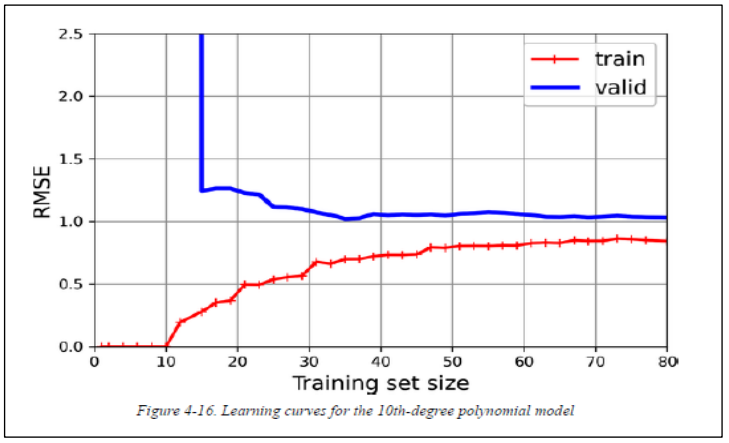


Figure 4-16. Learning curves for the 10th-degree polynomial model

- 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 before.
  - 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.

- **TIP:** 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 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 **highdegree 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

- 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. 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 regression**, which implement three different ways to constrain the weights.

Ridge Regression

$$\frac{\alpha}{m}\sum_{i=1}^{n}\theta_i^2$$

- **Ridge regression** (also called **Tikhonov regularization**) is a regularized version of **linear regression**: a **regularization term** is added to the **MSE**(**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 use the **unregularized MSE** (or the **RMSE**) to evaluate the model's performance.
- The hyperparameter **α** controls how much you want to regularize the model. If **α = 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**.

Equation 4-8. Ridge regression cost function

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

- Note that the **bias term θ_0** is not **regularized** (the sum starts at **i = 1**, not **0**).
- If we define **w** as the **vector** of **feature weights** (θ_1 to θ_n), then the **regularization term** is equal to $\alpha(\|\mathbf{w}\|_2)^2/m$, where $\|\mathbf{w}\|_2$ represents the **ℓ_2 norm** of the **weight** vector.□
- For **batch gradient descent**, just add **2αw / m** to the part of the **MSE** gradient vector that corresponds to the feature weights, without adding anything to the gradient of the bias term (see Equation 4-6).

- **WARNING:** 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** shows several **ridge models** that were trained on some very **noisy linear** data using different **α** values.
- On the **left**, plain ridge models are used, leading to **linear** predictions.
- On the **right**, the data is first expanded using **PolynomialFeatures(degree=10),** then it is **scaled** using a **StandardScaler**, and finally, the **ridge** models are applied to the resulting features.
- This is **polynomial regression** with **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**.

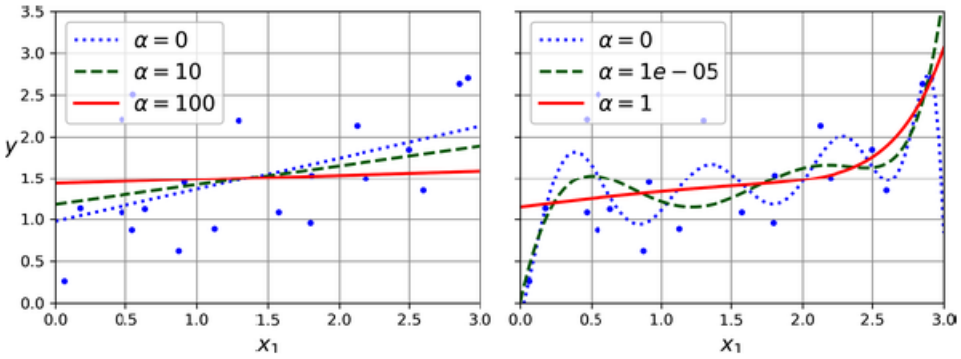


Figure 4-17. Linear (left) and a polynomial (right) models, both with various levels of ridge regularization

- As with **linear regression**, we can perform **ridge regression** either by computing a **closed-form** equation or by performing **gradient descent**. The pros and cons are the same.
- Equation 4-9 shows the **closed-form solution**, where **A** is the **(n + 1) × (n + 1) identity matrix**,☐ except with a **0** in the top-left cell, corresponding to the **bias term**.

Equation 4-9. Ridge regression closed-form solution

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

- Performing **ridge regression** with Scikit-Learn using a **closed-form** solution (a variant of Equation 4-9 that uses a matrix factorization technique by André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=0.1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55325833]])
```

- And using **stochastic gradient descent**:

```
>>> sgd_reg = SGDRegressor(penalty="l2", alpha=0.1 / m, tol=None,
...                        max_iter=1000, eta0=0.01, random_state=42)
...
>>> sgd_reg.fit(X, y.ravel())  # y.ravel() because fit() expects 1D targets
>>> sgd_reg.predict([[1.5]])
array([1.55302613])
```

- 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 **MSE cost function** equal to **alpha** times the **square** of the **ℓ2** norm of the weight vector.
- This is just like **ridge regression**, except there's no division by **m** in this case; that's why we passed **alpha=0.1 / m**, to get the same result as **Ridge(alpha=0.1).**

- **TIP**: The **RidgeCV class** also performs **ridge regression**, but it automatically **tunes hyperparameters** using **cross-validation**. It's roughly equivalent to using **GridSearchCV**, but it's optimized for **ridge regression** and runs much faster. Several other estimators (mostly linear) also have efficient CV variants, such as **LassoCV** and **ElasticNetCV**.

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 the square of the **ℓ_2 norm**

- (see Equation 4-10). Notice that the **ℓ_1 norm** is multiplied by **2α,** whereas the **ℓ_2 norm** was multiplied by **α / m** in **ridge regression**. These factors were chosen to ensure that the optimal **α** value is independent from the training set size: different norms lead to different factors.

Equation 4-10. Lasso regression cost function

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

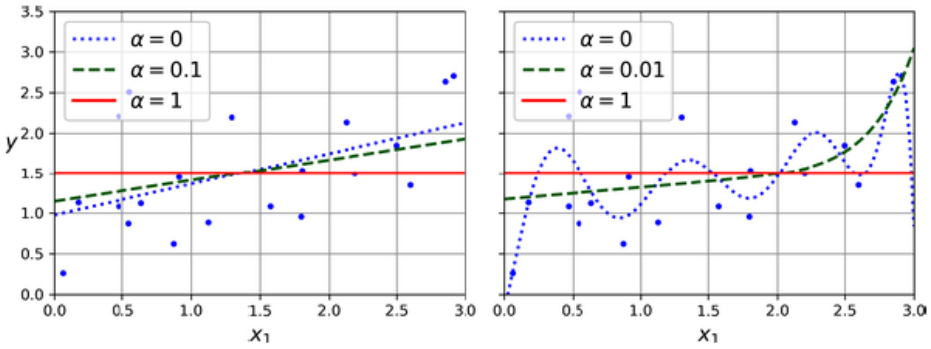


Figure 4-18. Linear (left) and polynomial (right) models, both using various levels of lasso regularization

- 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 (with α = 0.01) looks roughly cubic: 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 with few **nonzero** feature weights.
- **Figure 4-19**: the **axes** represent two model parameters, and the background contours represent different **loss functions**.
- In the **top-left plot**, the contours represent the **ℓ_1 loss (|θ_1| + |θ_2|)**, which drops linearly as you get closer to any axis.
- For example, if you initialize the model parameters to **θ_1 = 2** and **θ_2 = 0.5**, running **gradient descent** will **decrement** both parameters equally (as represented by the dashed yellow line); therefore, **θ_2** will reach **0** first .(After that, gradient descent will roll down the gutter until it reaches **θ1 = 0** (with a bit of bouncing around, since the gradients of **ℓ_1** never get close to 0: they are either **–1** or **1** for each parameter).

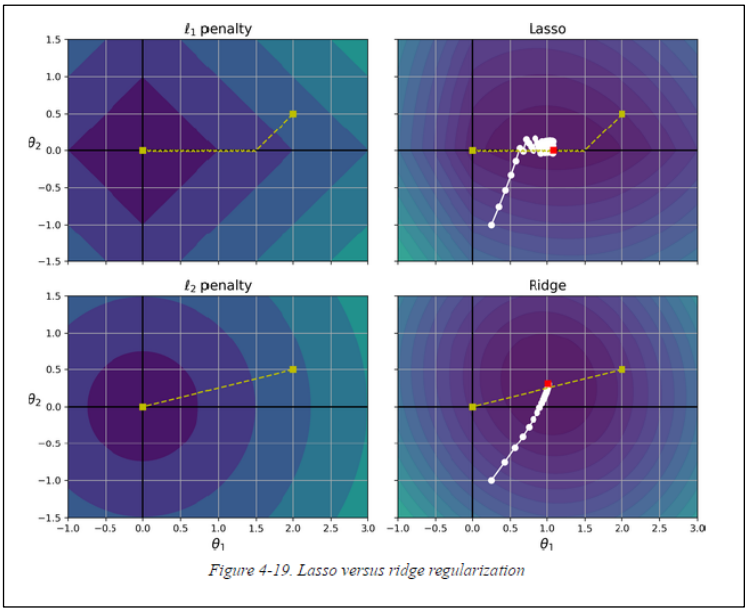


Figure 4-19. Lasso versus ridge regularization

- In the **top-right plot**, the contours represent **lasso regression's cost function** (i.e., an **MSE** cost function plus an **ℓ_1 loss**). The small white circles show the path that **gradient descent** takes to optimize some model parameters that were initialized around **θ_1 = 0.25** and **θ_2 = –1**: notice once again how the path quickly reaches **θ = 0,** then rolls down the gutter and ends up bouncing around the **global optimum** (represented by the red square). If we increased **α**, the **global optimum** would move **left** along the dashed yellow line, while if we decreased **α**, the **global optimum** would move **right** (in this example, the optimal parameters for the **unregularized MSE** are **θ_1 = 2** and **θ_2 = 0.5**)
- The two bottom plots show the same thing but with an **ℓ_2 penalty** instead.
- In the **bottom-left plot,** you can see that the **ℓ_2 loss** decreases as we get closer to the origin, so gradient descent just takes a straight path toward that point.

- In the **bottom-right plot**, the contours represent **ridge regression's cost function** (i.e., an **MSE cost function** plus an **ℓ_2 loss**). As you can see, the **gradients** get smaller as the parameters approach the **global optimum**, so **gradient descen**t naturally slows down. This limits the bouncing around, which helps **ridge converge faster** than **lasso regression**.
- Also note that the **optimal parameters** (represented by the **red square**) get closer and closer to the **origin** when you **increase α,** but they never get eliminated entirely.

- **TIP**: To keep **gradient descent** from bouncing around the **optimum** at the end when using **lasso regression**, you need to gradually **reduce** the **learning rate** during training. It will still bounce around the **optimum**, but the steps will get smaller and smaller, so it will converge

- The **lasso cost function** is not **differentiable** at **$\theta_i = 0$ (for i = 1, 2, ⋯, n),** but **gradient descent** still works if you use a **subgradient vector g**□ instead when any **θ = 0**.
- a **subgradient vector** equation you can use for **gradient descent** with the **lasso cost function**.

Equation 4-11. Lasso regression subgradient vector

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \operatorname{MSE}(\boldsymbol{\theta}) + 2\alpha \begin{pmatrix} \operatorname{sign}(\theta_1) \\ \operatorname{sign}(\theta_2) \\ \vdots \\ \operatorname{sign}(\theta_n) \end{pmatrix} \quad \text{where } \operatorname{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

- Implementing **Lasso Regression** using **Lasso class** in Scikit-Learn:

```
>>> 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])
```

- Also, you could instead use **SGDRegressor(penalty="l1", alpha=0.1**).

Elastic Net

- **Elastic net regression** is a **middle ground** between **ridge Regression** and **lasso Regression**.
- The **regularization term** is a weighted sum 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**

Equation 4-12. Elastic net cost function

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

- when should you use **elastic net regression**, or **ridge**, **lasso**, or **plain linear regression** (i.e., without any **regularization**)?
- 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, also it is a little better when most variables are useful.
- 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**.
- 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**.

- Implementing **Elastic Net using** Scikit-Learn's **ElasticNet class** (**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 different way to **regularize** iterative learning algorithms such as **gradient descent** is to **stop training** as soon as the **validation error** reaches a **minimum**. This is called **early stopping**.
- Figure 4-20 shows a complex model (in this case, a **highdegree polynomial regression model**) being trained with **batch gradient descent** on the quadratic dataset we used earlier. As the **epochs** go by, the algorithm learns, and its **prediction error (RMSE)** on the **training set** goes down, along with its **prediction error** on the **validation set**.
- After a while, though, the **validation error** stops **decreasing** and starts to go back up. This indicates that the model has started to **overfit** the training data. With **early stopping** you just stop training as soon as the **validation error** reaches the minimum

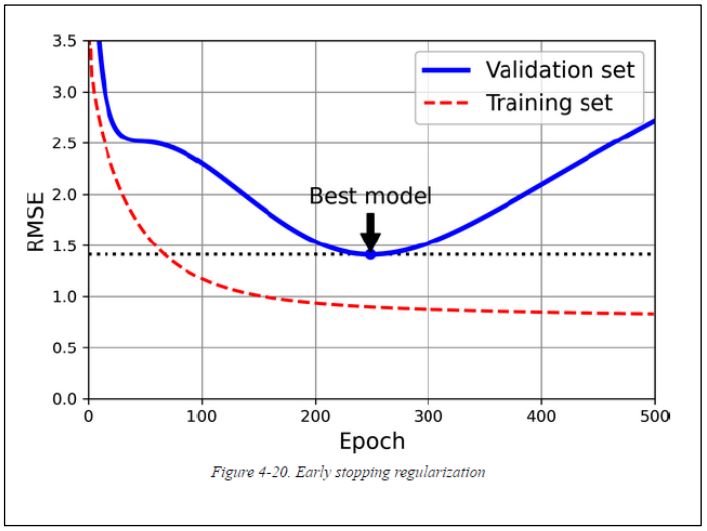


Figure 4-20. Early stopping regularization

- **TIP**: 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.

- **Early stopping** implementation:
- This code first adds the **polynomial features** and **scales** all the input features, both for the **training set** and for the **validation set**.
- Then it creates an **SGDRegressor** model with no **regularization** and a small **learning rate**.
- In the training loop, it calls **partial_fit()** instead of **fit(),** to perform **incremental learning**. At each epoch, it measures the **RMSE** on the **validation set**.
- If it is lower than the lowest **RMSE** seen so far, it saves a copy of the model in the **best_model variable**.
- This implementation does not actually stop training, but it lets you **revert** to the **best model** after training.
- The model is copied using **copy.deepcopy()**, because it copies both the model's **hyperparameters** and the **learned parameters**. In contrast, **sklearn.base.clone()** only copies the **model's hyperparameters**.

```
from copy import deepcopy
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

X_train, y_train, X_valid, y_valid = [...]  # split the quadratic dataset

preprocessing = make_pipeline(PolynomialFeatures(degree=90, include_bias=False),
                              StandardScaler())
X_train_prep = preprocessing.fit_transform(X_train)
X_valid_prep = preprocessing.transform(X_valid)
sgd_reg = SGDRegressor(penalty=None, eta0=0.002, random_state=42)
n_epochs = 500
best_valid_rmse = float('inf')

for epoch in range(n_epochs):
    sgd_reg.partial_fit(X_train_prep, y_train)
    y_valid_predict = sgd_reg.predict(X_valid_prep)
    val_error = mean_squared_error(y_valid, y_valid_predict, squared=False)
    if val_error < best_valid_rmse:
        best_valid_rmse = val_error
        best_model = deepcopy(sgd_reg)
```

Logistic Regression

- Some **regression** algorithms can be used for **classification** (and vice versa).
- **Logistic regression** (also called **logit regression**) is commonly used to estimate the probability that an instance belongs to a particular class.
- It estimates the probability of an instance given a **threshold**(typically **50%**), predicts the instance belongs to **positive** class(labeled "**1**") or **negative** class(labeled "**0**"), which makes it a **binary classifier**.

Estimating Probabilities

- How does **logistic regression** 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.

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

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

  - The **logistic**—noted **σ(·)**—is a **sigmoid function** (i.e., **S-shaped**) that outputs a number between **0** and **1**.

Equation 4-14. Logistic function

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

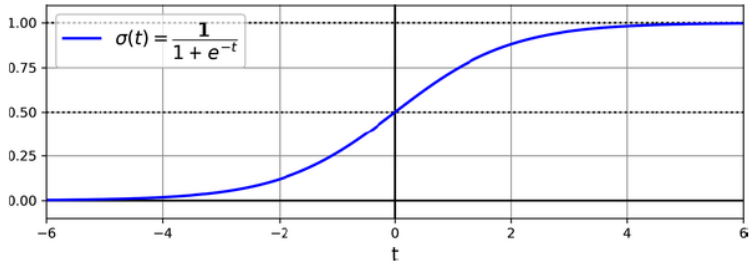


Figure 4-21. Logistic function

  - Once the **logistic regression** model has estimated the **probability *p*** = **$h_{\theta}(x)$** that an **instance x** belongs to the positive class, it can make its prediction **ŷ** easily

Equation 4-15. Logistic regression model prediction using a 50% threshold probability

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

  - Notice that **σ(t) < 0.5** when **t < 0**, and **σ(t) ≥ 0.5** when **t ≥ 0**, so a **logistic regression** model using the default threshold of **50%** probability predicts **1** if **θ^T x** is positive and **0** if it is negative.
  - **NOTE:** The score **t** is often called the **logit**. The name comes from the fact that the **logit** function, defined as **logit(p)** = **log(p / (1 – p))**, is the **inverse** of the **logistic function**. Indeed, if you compute the **logit** of the estimated probability p, you will find that the result is **t**. The **logit** is also called the **log-odds**, since it is the log of the **ratio** between the estimated probability for the positive class and the estimated probability for the negative class.

Training and Cost Function

- how **Logistic Regression** is 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** for a single training instance **x**.

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

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

- 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** instance, and it will also be large if the model estimates a probability close to **1** for a **negative** instance. On the other hand, **–log(t)** is close to **0** when **t** is close to **1**, so the cost will be close to **0** if the estimated probability is close to **0** for a negative instance or close to **1** for a positive instance, which is precisely what we want.
- 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**.

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

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

- **WARNING**: The **log loss** can be shown mathematically (using Bayesian inference) that **minimizing** this **loss** will result in the model with the **maximum likelihood** of being optimal, assuming that the instances follow a Gaussian distribution around the mean of their class. When you use the **log loss**, this is the implicit assumption you are making. The more wrong this assumption is, the more biased the model will be. Similarly, when we used the MSE to train linear regression models, we were implicitly assuming that the data was purely linear, plus some Gaussian noise. So, if the data is not linear (e.g., if it's quadratic) or if the noise is not Gaussian (e.g., if outliers are not exponentially rare), then the model will be biased.

- There is no known **closed-form equation** to compute the value of **θ** that **minimizes** this **cost function**. But 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 **jth** model parameter **θ_j** :

Equation 4-18. Logistic cost function partial derivatives

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

- For each instance it computes the **prediction error** and multiplies it by the **jth** 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 mini-batch at a time.

Decision boundaries

- Building a **classifier** to detect the **Iris virginic**a type based only on the **petal width** feature.
  - The first step is to load the data and take a quick peek:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> list(iris)
['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
 'filename', 'data_module']
>>> iris.data.head(3)
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
>>> iris.target.head(3)  # note that the instances are not shuffled
0    0
1    0
2    0
Name: target, dtype: int64
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

- Next we'll split the data and train a **logistic regression** model on the **training set**:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = iris.data[["petal width (cm)"]].values
y = iris.target_names[iris.target] == 'virginica'


X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

- model's estimated **probabilities** for flowers with **petal widths** varying from **0 cm** to **3 cm**:

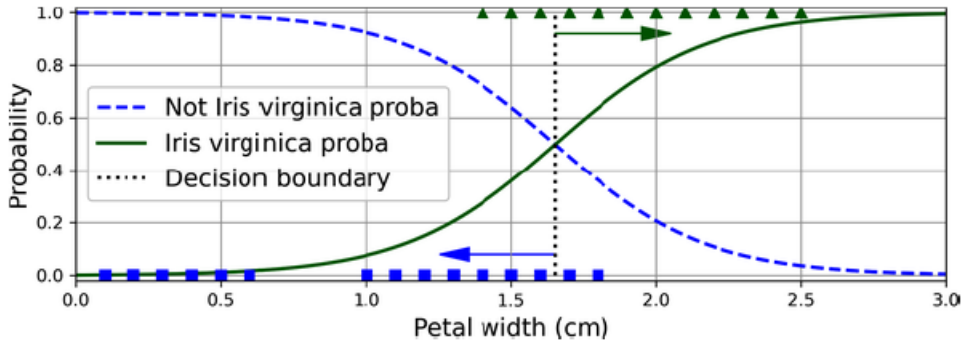


Figure 4-23. Estimated probabilities and decision boundary

- The **petal width** of **Iris virginica** flowers (represented as 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 for 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 greater than **1.6** cm the **classifier** will predict that the flower is an **Iris virginica**, and otherwise it will predict that it is not (even if it is not very confident):

```
>>> decision_boundary
1.6516516516516517
>>> log_reg.predict([[1.7], [1.5]])
array([ True, False])
```

- Figure **4-24** shows the same dataset, but this time displaying **two features**: **petal width** and **length**. Once trained, the **logistic regression** classifier can, based on these two features, estimate the **probability** that a new flower is an **Iris virginica**.
- 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**.☐ 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 over **90%** chance of being **Iris virginica**, according to the model.

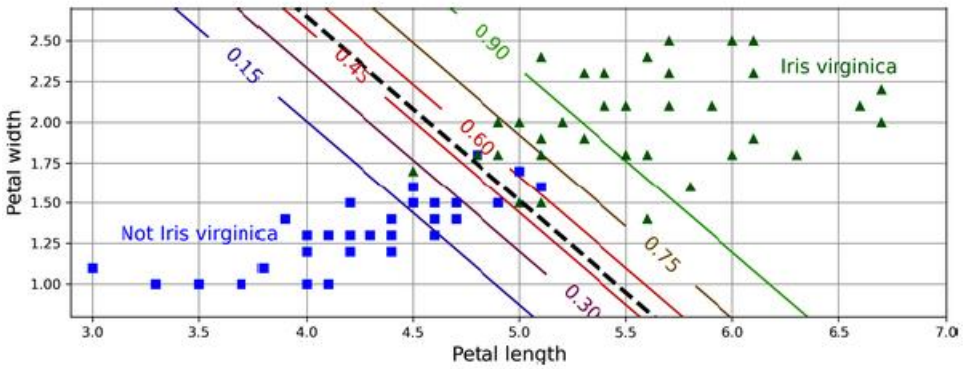


Figure 4-24. Linear decision boundary

- **NOTE:** The **hyperparameter** controlling the **regularization** strength of a Scikit-Learn **LogisticRegression** model is not **alpha** (as in other linear models), but its **inverse**: **C**.
- The higher the value of **C**, the less the model is **regularized**.

- Just like the other linear models, **logistic regression** models can be **regularized** using ℓ_1 or ℓ_2 penalties. Scikit-Learn actually adds an ℓ_2 penalty by default.

Softmax Regression

- The **logistic regression** model can be generalized to support **multiple classes** directly, without having to train and combine **multiple binary classifiers**. This is called **softmax regression**, or **multinomial logistic regression**.
- The idea is when given an instance **x**, the **softmax regression** model first computes a score $S_k(x)$ 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 $\mathbf{S_k(x)}$ is just like the equation for **linear regression** prediction.

Equation 4-19. Softmax score for class k

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

- Note that each class has its own dedicated **parameter vector** $\boldsymbol{\theta}^{(k)}$.
- All these vectors are typically stored as rows in a **parameter matrix Θ**.
- Once you have computed the score of every class for the **instance x**, you can estimate the **probability** $\hat{p}_k$ that the instance belongs to class **k** by running the scores through the **softmax function.**
- 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

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

- In this equation:
  - **K** is the number of **classes**.
  - **s(x)** is a **vector** containing the **scores** of each **class** for instance **x**.
  - **σ(s(x))** is the estimated **probability** that the instance **x** belongs to class **k,** given the scores of each class for that instance.
- Just like the **logistic regression classifier**, by default the **softmax regression classifier** predicts the class with the highest estimated **probability** (which is simply the class with the highest score).

Equation 4-21. Softmax regression classifier prediction

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

- 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 σ(s(x))** .

- **TIP:** 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 species of plants. You cannot use it to recognize multiple people in one picture.

- 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** 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.

Equation 4-22. Cross entropy cost function

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

- In this equation, $\mathbf{y}^{(i)}_k$ 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.
- Notice that when there are just two classes (**K = 2**), this **cost function** is equivalent to the **logistic regression cost function** (**log loss**).

Cross Entropy

- The **gradient vector** of this **cost function** with regard to $\boldsymbol{\theta}^{(k)}$:

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

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

- 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**.
- Create a **softmax regression** to **classify** the **iris plan**ts into all **three classes**. Scikit-Learn's **LogisticRegression classifier** uses **softmax regression** automatically when you train it on more than two classes (assuming you use **solver="lbfgs"**, which is the default). It also applies **ℓ_2 regularization** by default, which you can control using the **hyperparameter C:**

```
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = iris["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)


softmax_reg = LogisticRegression(C=30, random_state=42)
softmax_reg.fit(X_train, y_train)


>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]]).round(2)
array([[0.  , 0.04, 0.96]])
```

- **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 **0.30** represents the **30% 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%**.

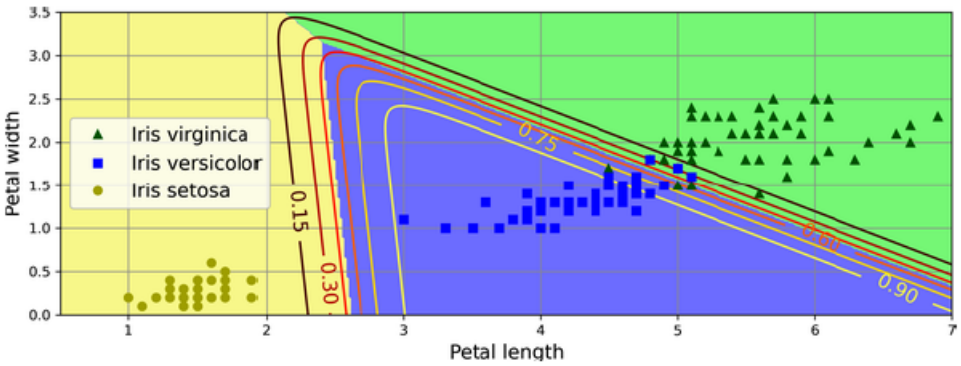


Figure 4-25. Softmax regression decision boundaries

Exercises

- Which linear regression training algorithm can you use if you have a training set with millions of features?

If you have millions of features the naive implementation of linear regression using the normal equations would be too difficult computationally to perform. One could instead use a gradient decent technique (like stochastic gradient descent or mini-batch gradient decent) which requires only inner products and avoids the matrix inversion required in the normal equations.

- Suppose the features in your training set have very different scales. Which algorithms might suffer from this, and how? What can you do about it?

If the features in your training set have very different scales, the cost function will have the shape of elongated bowl, So GD algorithms will take a long time to converge, you should scale the data before training the model

Normal equation or SVD approach will work just fine without scaling.

Regularized models may converge to suboptimal solutions if the features are not scaled, since regularization penalize large weights features with smaller values will tend to be ignored compared to features with larger values

- Do all gradient descent algorithms lead to the same model, provided you let them run long enough?

If the optimization problem is convex and assuming the learning rate is not too high, then all GD algorithms will approach the global minimum and end up producing a fairly similar models.

However, unless you gradually reduce the learning rate, stochastic GD and mini-batch GD will never truly converge; instead, they will keep jumping back and forth around the global optimum

- Do all gradient descent algorithms lead to the same model, provided you let them run long enough?

If the validation error is increasing it means that the learning rate is too high and the algorithm is diverging, if the training error is also going up, then this is clearly the problem, and you should reduce the learning rate.

If the training error is not going up, then your model is overfitting the training set and you should stop training

- Is it a good idea to stop mini-batch gradient descent immediately when the validation error goes up?

Due to their random nature, if you immediately stop training when the validation error goes up, you may stop much too early, before the optimum is reached.

You can save the model at regular intervals, then when it is not improved for a long time, you can revert to the best model

- 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?

This is likely because your model is overfitting the training set

You can try fix this by reducing the polynomial degree, you can try to regularize the model or increase the size of the training set

- Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two logistic regression classifiers or one softmax regression classifier?

Since all of these are not exclusive classes(all four combinations are possible) you should train two logistic regression classifiers