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]])
```

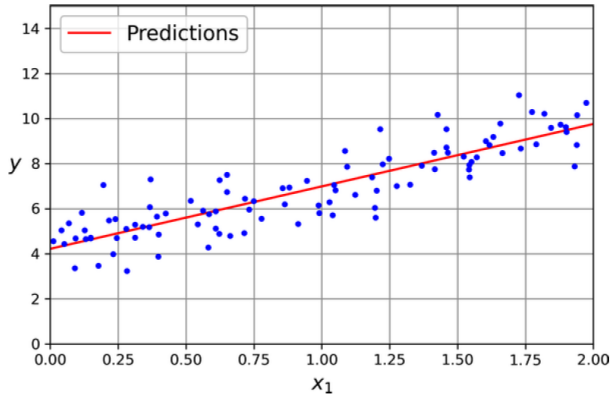


Figure 4-2. Linear regression model predictions

- 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.

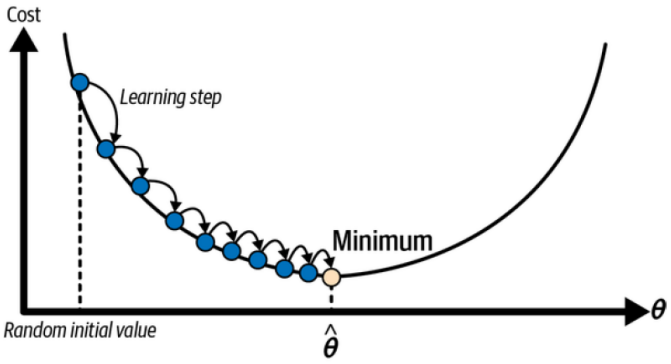


Figure 4-3. In this depiction of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the cost approaches the minimum

- 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

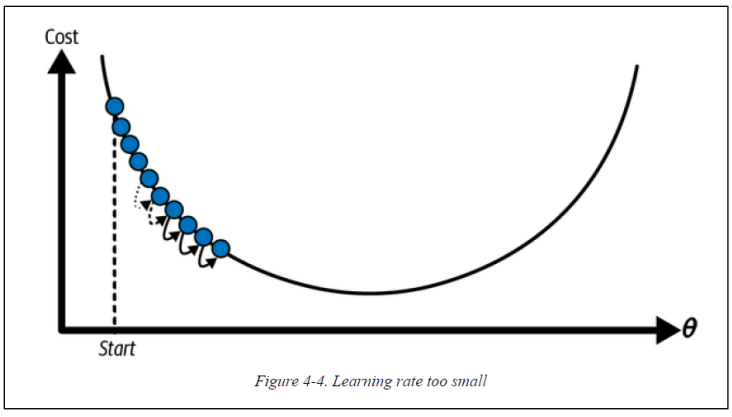


Figure 4-4. Learning rate too small

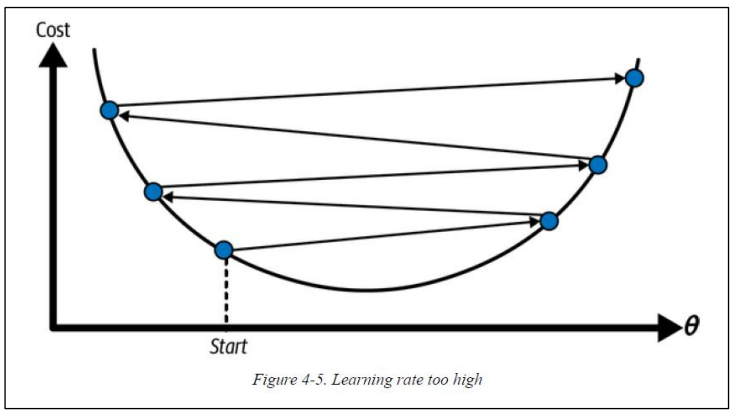


Figure 4-5. Learning rate too high

- 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**.

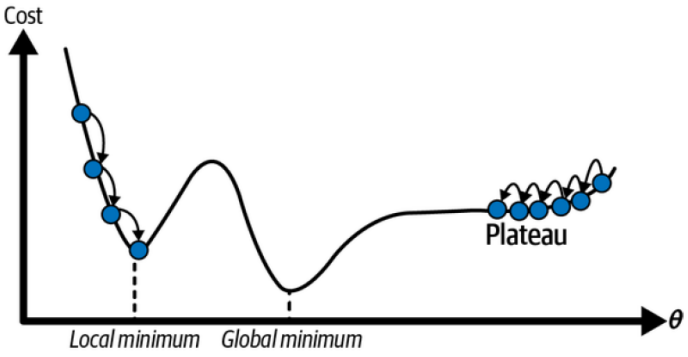


Figure 4-6. Gradient descent pitfalls

- 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.

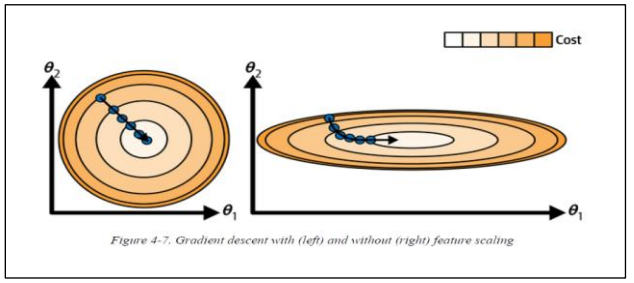


Figure 4-7. Gradient descent with (left) and without (right) feature scaling

- 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 ∂ **MSE(θ) /** $\partial\theta_j$.

Equation 4-5. Partial derivatives of the cost function

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

- 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.

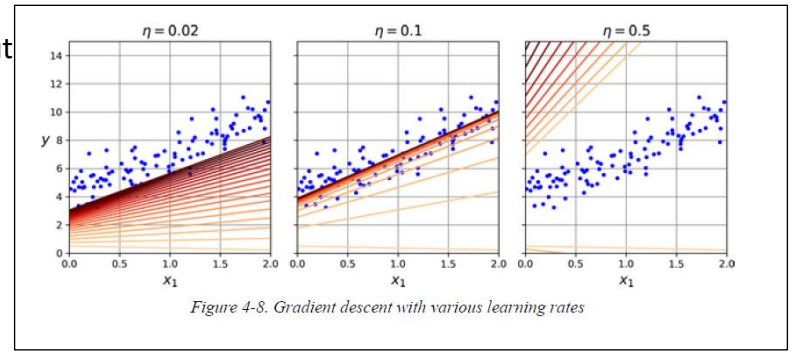


Figure 4-8. Gradient descent with various learning rates

  - 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.

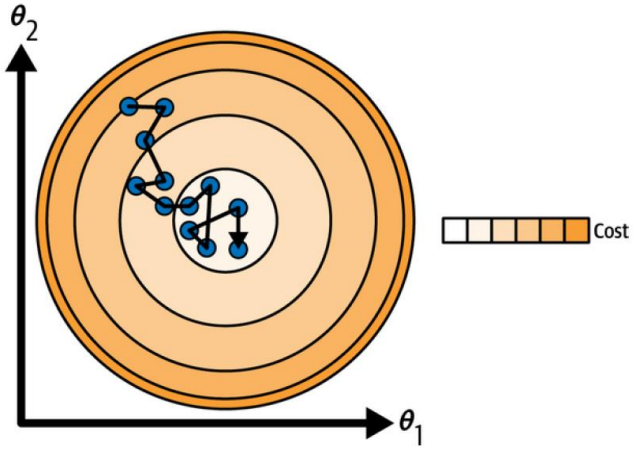


Figure 4-9. With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent

- 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.

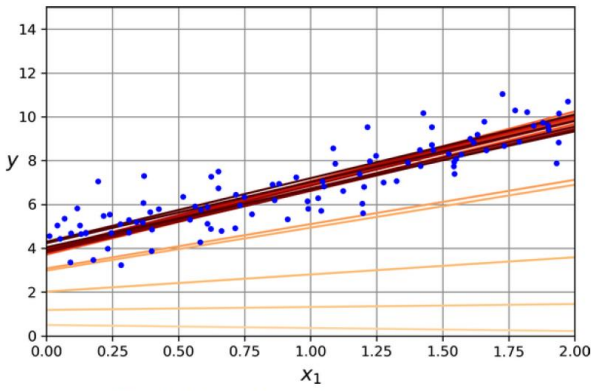


Figure 4-10. The first 20 steps of stochastic gradient descent

- 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**.

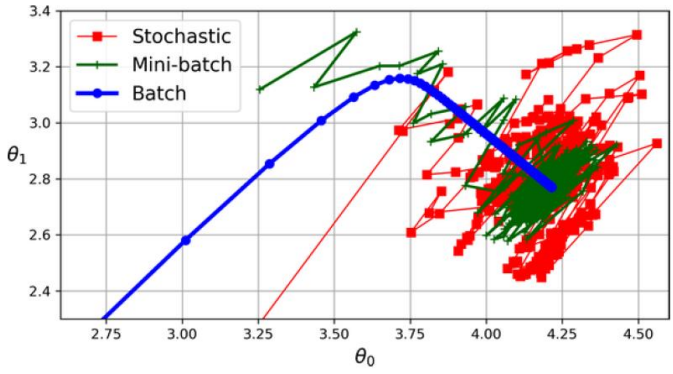


Figure 4-11. Gradient descent paths in parameter space

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)
```

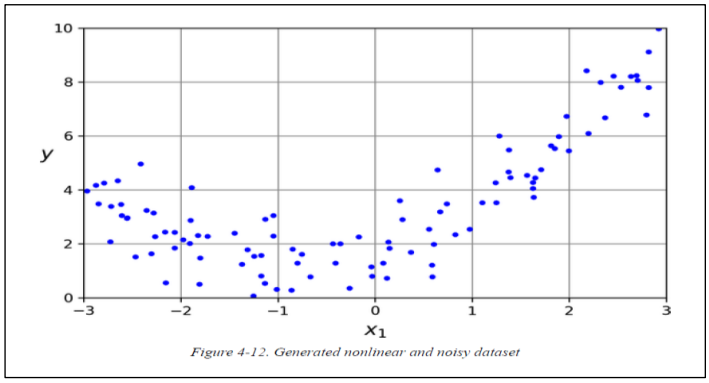


Figure 4-12. Generated nonlinear and noisy dataset

- 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]]))
```

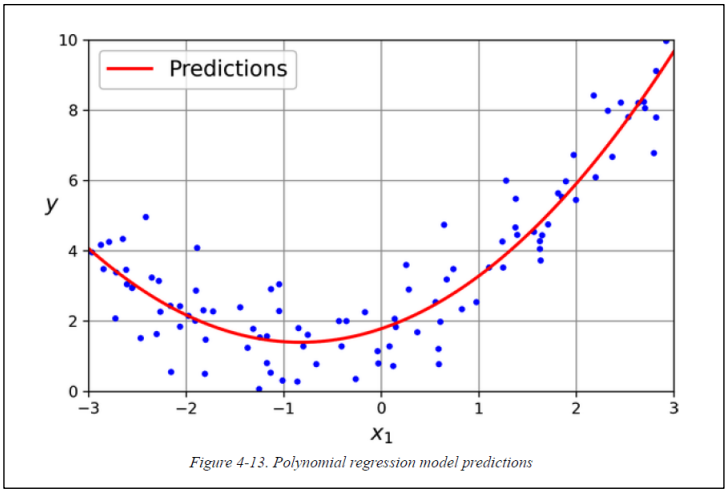


Figure 4-13. Polynomial regression model predictions

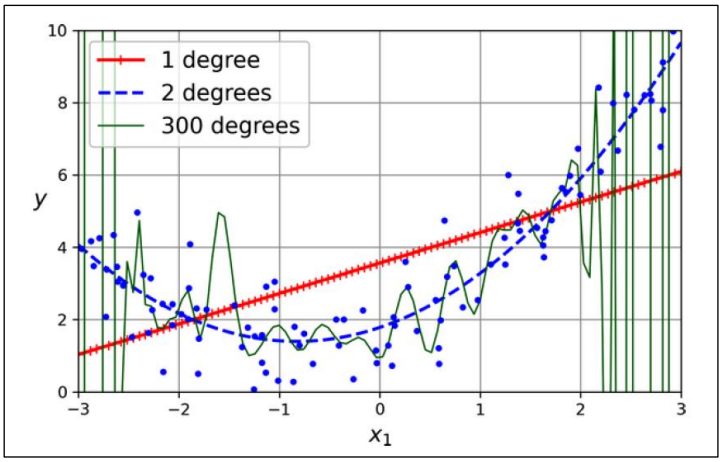
- the model estimates $\mathbf{y = 0.56x_1^2 + 0.93x_1 + 1.78}$ when in fact the original function was $\mathbf{y = 0.5x_1^2 + 1.0x_1 + 2.0 +}$ **Gaussian noise.**
- Note that when there are multiple features, **polynomial regression** is capable of finding relationships between features, which is something 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()
```

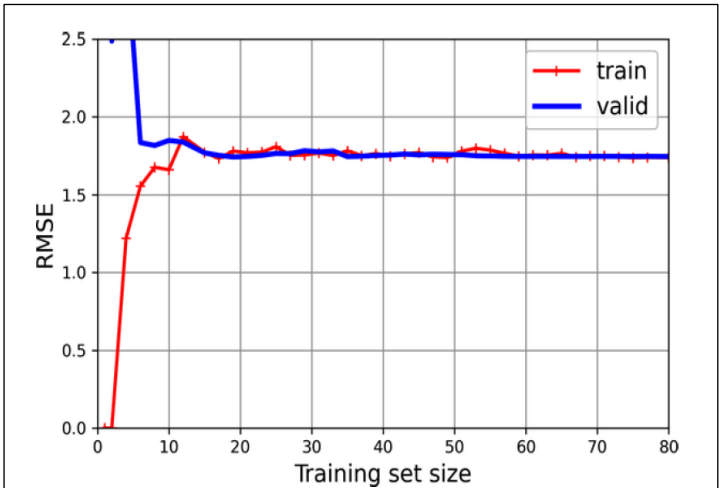


Figure 4-15. Learning curves

The Bias/Variance Trade-off

-

Regularized Linear Models

-

Ridge Regression

-

Lasso Regression

-

Elastic Net

-

Early Stopping

-

Logistic Regression

-

Estimating Probabilities

-

Training and Cost Function

-

Decision boundaries

-

Softmax Regression

-

Cross Entropy

-

Exercises

-