CHAPTER 3
Classification

In Chapter 1 I mentioned that the most common supervised learning tasks are regression (predicting values) and classification (predicting classes). In Chapter 2 we explored a regression task, predicting housing values, using various algorithms such as linear regression, decision trees, and random forests (which will be explained in further detail in later chapters). Now we will turn our attention to classification systems.

MNIST

In this chapter we will be using the MNIST dataset, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents. This set has been studied so much that it is often called the "hello world" of machine learning: whenever people come up with a new classification algorithm they are curious to see how it will perform on MNIST, and anyone who learns machine learning tackles this dataset sooner or later.

Scikit-Learn provides many helper functions to download popular datasets. MNIST is one of them. The following code fetches the MNIST dataset from OpenML.org:[1]

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
```

1 By default Scikit-Learn caches downloaded datasets in a directory called *scikit_learn_data* in your home directory.

The `sklearn.datasets` package contains mostly three types of functions: `fetch_*` functions such as `fetch_openml()` to download real-life datasets, `load_*` functions to load small toy datasets bundled with Scikit-Learn (so they don't need to be downloaded over the internet), and `make_*` functions to generate fake datasets, useful for tests. Generated datasets are usually returned as an `(X, y)` tuple containing the input data and the targets, both as NumPy arrays. Other datasets are returned as `sklearn.utils.Bunch` objects, which are dictionaries whose entries can also be accessed as attributes. They generally contain the following entries:

`"DESCR"`
: A description of the dataset

`"data"`
: The input data, usually as a 2D NumPy array

`"target"`
: The labels, usually as a 1D NumPy array

The `fetch_openml()` function is a bit unusual since by default it returns the inputs as a Pandas DataFrame and the labels as a Pandas Series (unless the dataset is sparse). But the MNIST dataset contains images, and DataFrames aren't ideal for that, so it's preferable to set `as_frame=False` to get the data as NumPy arrays instead. Let's look at these arrays:

```
>>> X, y = mnist.data, mnist.target
>>> X
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
>>> X.shape
(70000, 784)
>>> y
array(['5', '0', '4', ..., '4', '5', '6'], dtype=object)
>>> y.shape
(70000,)
```

There are 70,000 images, and each image has 784 features. This is because each image is 28 × 28 pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black). Let's take a peek at one digit from the dataset (Figure 3-1). All we need to do is grab an instance's feature vector, reshape it to a 28 × 28 array, and display it using Matplotlib's `imshow()` function. We use `cmap="binary"` to get a grayscale color map where 0 is white and 255 is black:

```
import matplotlib.pyplot as plt

def plot_digit(image_data):
    image = image_data.reshape(28, 28)
    plt.imshow(image, cmap="binary")
    plt.axis("off")

some_digit = X[0]
plot_digit(some_digit)
plt.show()
```

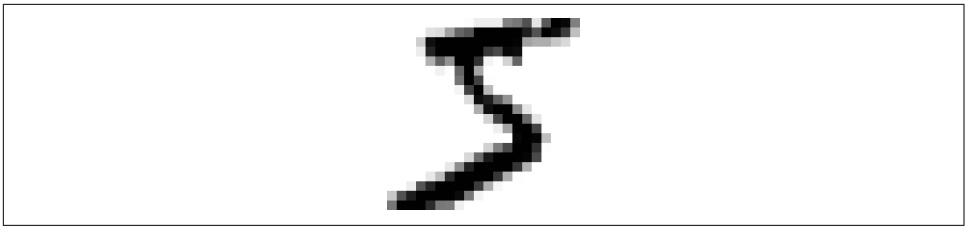

Figure 3-1. Example of an MNIST image

This looks like a 5, and indeed that's what the label tells us:

```
>>> y[0]
'5'
```

To give you a feel for the complexity of the classification task, Figure 3-2 shows a few more images from the MNIST dataset. There's quite a large variety of digit shapes. That said, the images are clean, well centered, not too rotated, and the digits all have roughly the same size: this dataset will not require much preprocessing (real-world datasets aren't usually that friendly).

But wait! You should always create a test set and set it aside before inspecting the data closely. The MNIST dataset returned by `fetch_openml()` is actually already split into a training set (the first 60,000 images) and a test set (the last 10,000 images):[2]

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

The training set is already shuffled for us, which is good because this guarantees that all cross-validation folds will be similar (we don't want one fold to be missing some digits). Moreover, some learning algorithms are sensitive to the order of the training instances, and they perform poorly if they get many similar instances in a row. Shuffling the dataset ensures that this won't happen.[3]

2 Datasets returned by `fetch_openml()` are not always shuffled or split.

3 Shuffling may be a bad idea in some contexts—for example, if you are working on time series data (such as stock market prices or weather conditions). We will explore this in Chapter 13.

Figure 3-2. Digits from the MNIST dataset

Training a Binary Classifier

Let's simplify the problem for now and only try to identify one digit—for example, the number 5. This "5-detector" will be an example of a *binary classifier*, capable of distinguishing between just two classes, 5 and non-5. First we'll create the target vectors for this classification task:

```
y_train_5 = (y_train == '5')  # True for all 5s, False for all other digits
y_test_5 = (y_test == '5')
```

Now let's pick a classifier and train it. A good place to start is with a *stochastic gradient descent* (SGD, or stochastic GD) classifier, using Scikit-Learn's `SGDClassifier` class. This classifier is capable of handling very large datasets efficiently. This is in part because SGD deals with training instances independently, one at a time, which also makes SGD well suited for online learning, as you will see later. Let's create an `SGDClassifier` and train it on the whole training set:

```
from sklearn.linear_model import SGDClassifier
```

```
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

Now we can use it to detect images of the number 5:

```
>>> sgd_clf.predict([some_digit])
array([ True])
```

The classifier guesses that this image represents a 5 (`True`). Looks like it guessed right in this particular case! Now, let's evaluate this model's performance.

Performance Measures

Evaluating a classifier is often significantly trickier than evaluating a regressor, so we will spend a large part of this chapter on this topic. There are many performance measures available, so grab another coffee and get ready to learn a bunch of new concepts and acronyms!

Measuring Accuracy Using Cross-Validation

A good way to evaluate a model is to use cross-validation, just as you did in Chapter 2. Let's use the `cross_val_score()` function to evaluate our `SGDClassifier` model, using *k*-fold cross-validation with three folds. Remember that *k*-fold cross-validation means splitting the training set into *k* folds (in this case, three), then training the model *k* times, holding out a different fold each time for evaluation (see Chapter 2):

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.95035, 0.96035, 0.9604 ])
```

Wow! Above 95% accuracy (ratio of correct predictions) on all cross-validation folds? This looks amazing, doesn't it? Well, before you get too excited, let's look at a dummy classifier that just classifies every single image in the most frequent class, which in this case is the negative class (i.e., *non*-5):

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train)))  # prints False: no 5s detected
```

Can you guess this model's accuracy? Let's find out:

```
>>> cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.90965, 0.90965, 0.90965])
```

That's right, it has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is *not* a 5, you will be right about 90% of the time. Beats Nostradamus.

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with *skewed datasets* (i.e., when some classes are much more frequent than others). A much better way to evaluate the performance of a classifier is to look at the *confusion matrix* (CM).

Implementing Cross-Validation

Occasionally you will need more control over the cross-validation process than what Scikit-Learn provides off the shelf. In these cases, you can implement cross-validation yourself. The following code does roughly the same thing as Scikit-Learn's `cross_val_score()` function, and it prints the same result:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3)  # add shuffle=True if the dataset is
                                       # not already shuffled
for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))  # prints 0.95035, 0.96035, and 0.9604
```

The `StratifiedKFold` class performs stratified sampling (as explained in Chapter 2) to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds, and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

Confusion Matrices

The general idea of a confusion matrix is to count the number of times instances of class A are classified as class B, for all A/B pairs. For example, to know the number of times the classifier confused images of 8s with 0s, you would look at row #8, column #0 of the confusion matrix.

To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets. You could make predictions on the test set, but it's best to keep that untouched for now (remember that you want to use the test set only at the very end of your project, once you have a classifier that you are ready to launch). Instead, you can use the `cross_val_predict()` function:

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Just like the `cross_val_score()` function, `cross_val_predict()` performs *k*-fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold. This means that you get a clean prediction for each instance in the training set (by "clean" I mean "out-of-sample": the model makes predictions on data that it never saw during training).

Now you are ready to get the confusion matrix using the `confusion_matrix()` function. Just pass it the target classes (`y_train_5`) and the predicted classes (`y_train_pred`):

```
>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_train_5, y_train_pred)
>>> cm
array([[53892,   687],
       [ 1891,  3530]])
```

Each row in a confusion matrix represents an *actual class*, while each column represents a *predicted class*. The first row of this matrix considers non-5 images (the *negative class*): 53,892 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 687 were wrongly classified as 5s (*false positives*, also called *type I errors*). The second row considers the images of 5s (the *positive class*): 1,891 were wrongly classified as non-5s (*false negatives*, also called *type II errors*), while the remaining 3,530 were correctly classified as 5s (*true positives*). A perfect classifier would only have true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal (top left to bottom right):

```
>>> y_train_perfect_predictions = y_train_5  # pretend we reached perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [    0,  5421]])
```

The confusion matrix gives you a lot of information, but sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called the *precision* of the classifier (Equation 3-1).

Equation 3-1. Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

TP is the number of true positives, and *FP* is the number of false positives.

Now consider a model that only makes positive predictions when it's extremely confident. Let's push this to the extreme and suppose that it always makes negative

predictions, except for a single positive prediction on the instance it's most confident about. If this one prediction is correct, then the classifier has 100% precision (precision = 1/1 = 100%). Obviously, such a classifier would not be very useful, since it would ignore all but one positive instance. For this reason, precision is typically used along with another metric named *recall*, also called *sensitivity* or the *true positive rate* (TPR): this is the ratio of positive instances that are correctly detected by the classifier (Equation 3-2).

Equation 3-2. Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

FN is, of course, the number of false negatives.

If you are confused about the confusion matrix, Figure 3-3 may help.

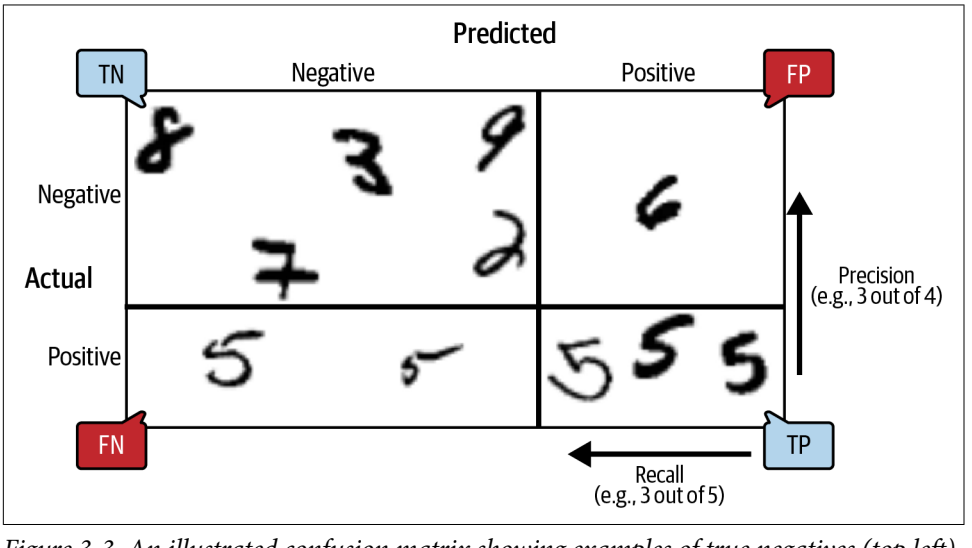

Figure 3-3. An illustrated confusion matrix showing examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)

Precision and Recall

Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred)  # == 3530 / (687 + 3530)
np.float64(0.8370879772350012)
>>> recall_score(y_train_5, y_train_pred)  # == 3530 / (1891 + 3530)
np.float64(0.6511713705958311)
```

Now our 5-detector does not look as shiny as it did when we looked at its accuracy. When it claims an image represents a 5, it is correct only 83.7% of the time. Moreover, it only detects 65.1% of the 5s.

It is often convenient to combine precision and recall into a single metric called the *F_1 score*, especially when you need a single metric to compare two classifiers. The F_1 score is the *harmonic mean* of precision and recall (Equation 3-3). Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F_1 score if both recall and precision are high.

Equation 3-3. F_1 score

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

To compute the F_1 score, simply call the `f1_score()` function:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7325171197343846
```

The F_1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall. For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters in surveillance images: it is probably fine if your classifier only has 30% precision as long as it has 99% recall. Sure, the security guards will get a few false alerts, but almost all shoplifters will get caught. Similarly, medical diagnosis usually requires a high recall to avoid missing anything important. False positives can be ruled out by follow-up medical tests.

Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the *precision/recall trade-off*.

The Precision/Recall Trade-Off

To understand this trade-off, let's look at how the `SGDClassifier` makes its classification decisions. For each instance, it computes a score based on a *decision function*. If that score is greater than a threshold, it assigns the instance to the positive class; otherwise it assigns it to the negative class. Figure 3-4 shows a few digits positioned

from the lowest score on the left to the highest score on the right. Suppose the *decision threshold* is positioned at the central arrow (between the two 5s): you will find 4 true positives (actual 5s) on the right of that threshold, and 1 false positive (actually a 6). Therefore, with that threshold, the precision is 80% (4 out of 5). But out of 6 actual 5s, the classifier only detects 4, so the recall is 67% (4 out of 6). If you raise the threshold (move it to the arrow on the right), the false positive (the 6) becomes a true negative, thereby increasing the precision (up to 100% in this case), but one true positive becomes a false negative, decreasing recall down to 50%. Conversely, lowering the threshold increases recall and reduces precision.

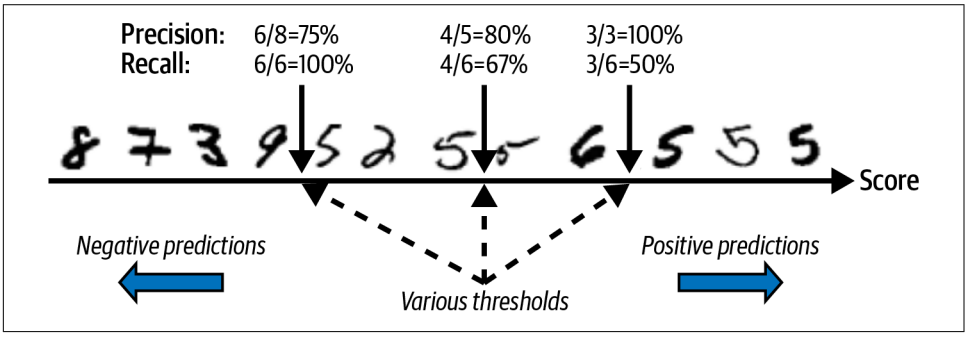

Figure 3-4. The precision/recall trade-off: images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

Instead of calling the classifier's `predict()` method, you can call its `decision_function()` method, which returns a score for each instance. You can then use any threshold you want to make predictions based on those scores:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2164.22030239])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([ True])
```

The `SGDClassifier` uses a threshold equal to 0, so the preceding code returns the same result as the `predict()` method (i.e., `True`). Let's raise the threshold:

```
>>> threshold = 3000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

This confirms that raising the threshold decreases recall. The image actually represents a 5, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 3,000.

How do you decide which threshold to use? One option is to use the `cross_val_predict()` function to get the scores of all instances in the training set, but this time specify that you want to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

With these scores, use the `precision_recall_curve()` function to compute precision and recall for all possible thresholds (the function adds a last precision of 1 and a last recall of 0, corresponding to an infinite threshold):

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Finally, use Matplotlib to plot precision and recall as functions of the threshold value (Figure 3-5). Let's show the threshold of 3,000 we selected:

```
plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.vlines(threshold, 0, 1.0, "k", "dotted", label="threshold")
[...]  # beautify the figure: add grid, legend, axis, labels, and circles
plt.show()
```

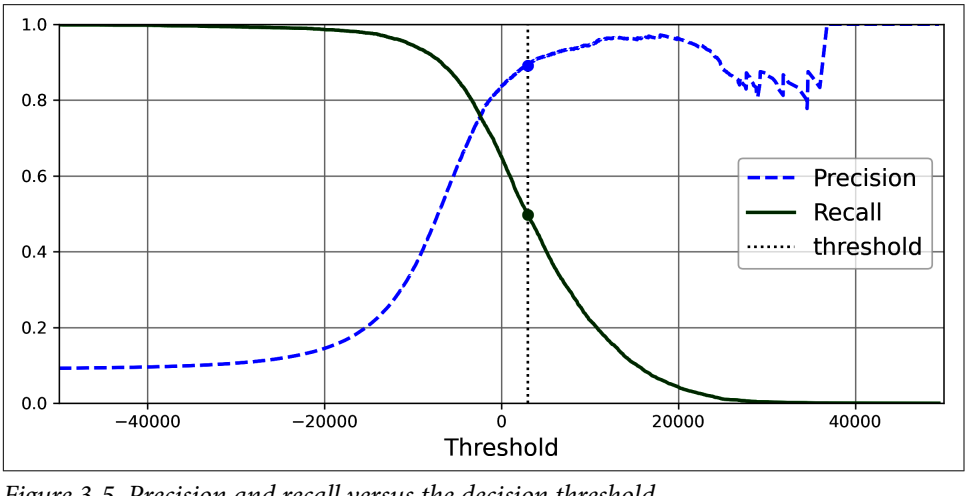

Figure 3-5. Precision and recall versus the decision threshold

You may wonder why the precision curve is bumpier than the recall curve in Figure 3-5. The reason is that precision may sometimes go down when you raise the threshold (although in general it will go up). To understand why, look back at Figure 3-4 and notice what happens when you start from the central threshold and move it just one digit to the right: precision goes from 4/5 (80%) down to 3/4 (75%). On the other hand, recall can only go down when the threshold is increased, which explains why its curve looks smooth.

At this threshold value, precision is near 90% and recall is around 50%. Another way to select a good precision/recall trade-off is to plot precision directly against recall, as shown in Figure 3-6 (the same threshold is shown):

```
plt.plot(recalls, precisions, linewidth=2, label="Precision/Recall curve")
[...]  # beautify the figure: add labels, grid, legend, arrow, and text
plt.show()
```

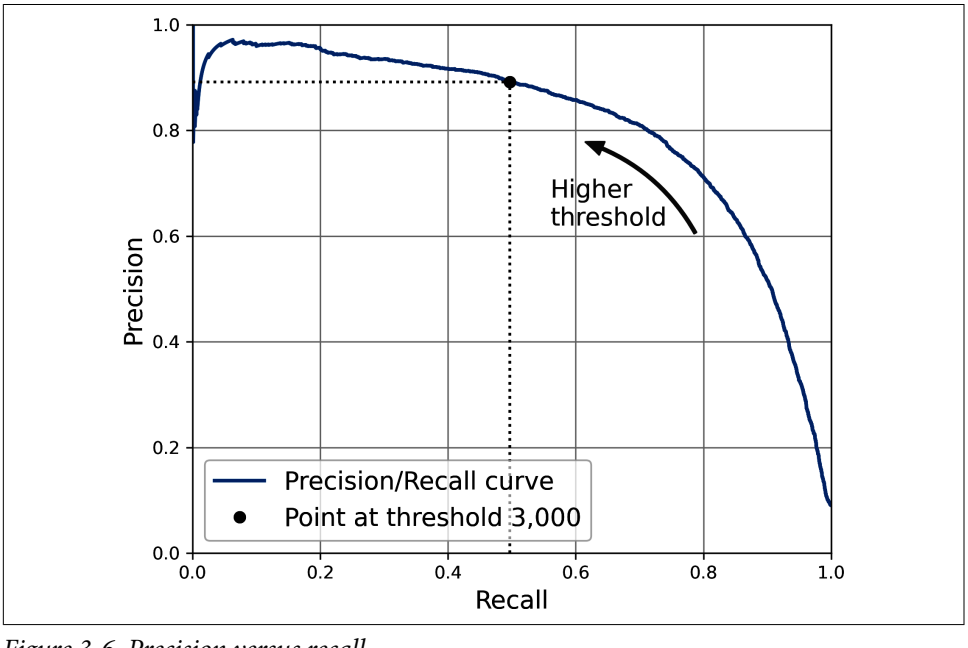

Figure 3-6. Precision versus recall

You can see that precision really starts to fall sharply at around 80% recall. You will probably want to select a precision/recall trade-off just before that drop—for example, at around 60% recall. But of course, the choice depends on your project.

Suppose you decide to aim for 90% precision. You could use the first plot to find the threshold you need to use, but that's not very precise. Alternatively, you can search for the lowest threshold that gives you at least 90% precision. For this, you can use the

NumPy array's `argmax()` method. This returns the first index of the maximum value, which in this case means the first `True` value:

```
>>> idx_for_90_precision = (precisions >= 0.90).argmax()
>>> threshold_for_90_precision = thresholds[idx_for_90_precision]
>>> threshold_for_90_precision
np.float64(3370.0194991439557)
```

To make predictions (on the training set for now), instead of calling the classifier's `predict()` method, you can run this code:

```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)
```

Let's check these predictions' precision and recall:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000345901072293
>>> recall_at_90_precision = recall_score(y_train_5, y_train_pred_90)
>>> recall_at_90_precision
0.4799852425751706
```

Great, you have a 90% precision classifier! As you can see, it is fairly easy to create a classifier with virtually any precision you want: just set a high enough threshold, and you're done. But wait, not so fast: a high-precision classifier is not very useful if its recall is too low! For many applications, 48% recall wouldn't be great at all.

If someone says, "Let's reach 99% precision", you should ask, "At what recall?"

Since Scikit-Learn 1.5, there are two new classes you can use to more easily adjust the decision threshold:

- The `FixedThresholdClassifier` class lets you wrap a binary classifier and set the desired threshold manually. If the underlying classifier has a `predict_proba()` method, then the threshold should be a value between 0 and 1 (the default is 0.5). Otherwise, it should be a decision score, comparable to the output of the model's `decision_function()` (the default is 0).
- The `TunedThresholdClassifierCV` class uses *k*-fold cross-validation to automatically find the optimal threshold for a given metric. By default, it tries to find the threshold that maximizes the model's *balanced accuracy*: that's the average of each class's recall. However, you can select another metric to optimize for (see the documentation for the full list of options).

The ROC Curve

The *receiver operating characteristic* (ROC) curve is another common tool used with binary classifiers. It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate* (FPR). The FPR (also called the *fall-out*) is the ratio of negative instances that are incorrectly classified as positive. It is equal to 1 – the *true negative rate* (TNR), which is the ratio of negative instances that are correctly classified as negative. The TNR is also called *specificity*. Hence, the ROC curve plots *sensitivity* (recall) versus 1 – *specificity*.

To plot the ROC curve, you first use the `roc_curve()` function to compute the TPR and FPR for various threshold values:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

Then you can plot the FPR against the TPR using Matplotlib. The following code produces the plot in Figure 3-7. To find the point that corresponds to 90% precision, we need to look for the index of the desired threshold. Since thresholds are listed in decreasing order in this case, we use `<=` instead of `>=` on the first line:

```
idx_for_threshold_at_90 = (thresholds <= threshold_for_90_precision).argmax()
tpr_90, fpr_90 = tpr[idx_for_threshold_at_90], fpr[idx_for_threshold_at_90]

plt.plot(fpr, tpr, linewidth=2, label="ROC curve")
plt.plot([0, 1], [0, 1], 'k:', label="Random classifier's ROC curve")
plt.plot([fpr_90], [tpr_90], "ko", label="Threshold for 90% precision")
[...]  # beautify the figure: add labels, grid, legend, arrow, and text
plt.show()
```

Once again there is a trade-off: the higher the recall (TPR), the more false positives (FPR) the classifier produces. The dotted line represents the ROC curve of a purely random classifier; a good classifier stays as far away from that line as possible (toward the top-left corner).

One way to compare classifiers is to measure the *area under the curve* (AUC). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5. Scikit-Learn provides a function to estimate the ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
np.float64(0.9604938554008616)
```

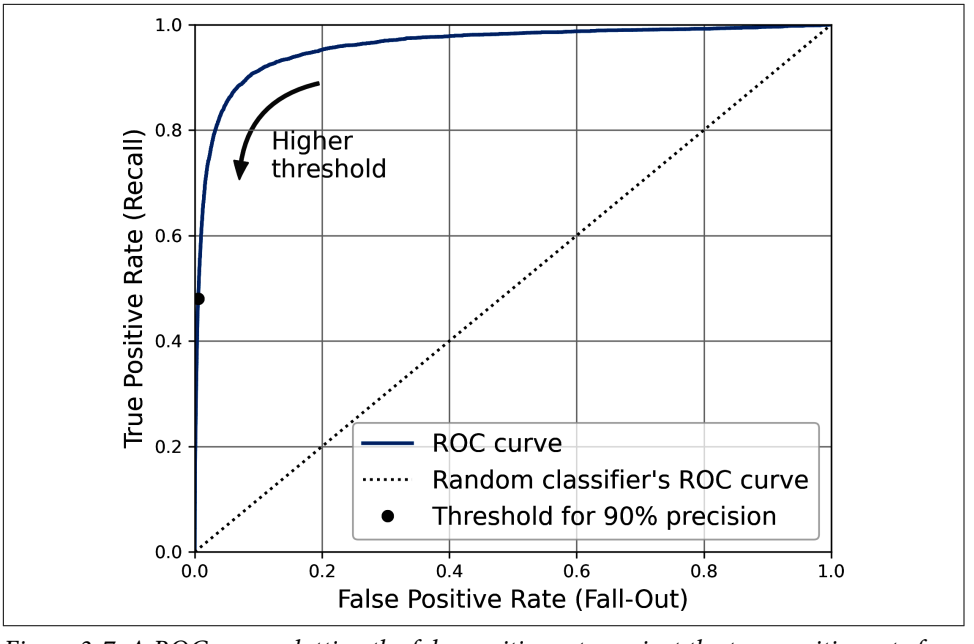

Figure 3-7. A ROC curve plotting the false positive rate against the true positive rate for all possible thresholds; the black circle highlights the chosen ratio (at 90% precision and 48% recall)

Since the ROC curve is so similar to the precision/recall (PR) curve, you may wonder how to decide which one to use. As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives. Otherwise, use the ROC curve. For example, looking at the previous ROC curve (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s). In contrast, the PR curve makes it clear that the classifier has room for improvement: the curve could really be closer to the top-right corner (see Figure 3-6 again).

Let's now create a `RandomForestClassifier`, whose PR curve and F_1 score we can compare to those of the `SGDClassifier`:

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
```

The `precision_recall_curve()` function expects labels and scores for each instance, so we need to train the random forest classifier and make it assign a

score to each instance. But the `RandomForestClassifier` class does not have a `decision_function()` method, due to the way it works (we will cover this in Chapter 6). Luckily, it has a `predict_proba()` method that returns estimated class probabilities for each instance, and we can just use the probability of the positive class as a score, so `precision_recall_curve()` will work.[4] We can call the `cross_val_predict()` function to train the `RandomForestClassifier` using cross-validation and make it predict class probabilities for every image as follows:

```
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

Let's look at the estimated class probabilities for the first two images in the training set:

```
>>> y_probas_forest[:2]
array([[0.11, 0.89],
       [0.99, 0.01]])
```

The model predicts that the first image is positive with 89% probability, and it predicts that the second image is negative with 99% probability. Since each image is either positive or negative, the estimated probabilities in each row add up to 100%.

These are *estimated* probabilities, not actual probabilities. For example, if you look at all the images that the model classified as positive with an estimated probability between 50% and 60%, roughly 94% of them are actually positive. So, the model's estimated probabilities were much too low in this case—but models can be overconfident as well. The `CalibratedClassifierCV` class from the `sklearn.calibration` package can calibrate the estimated probabilities using cross-validation, making them much closer to actual probabilities (see the notebook for a code example). This is important in some scenarios, such as medical diagnosis, financial risk assessment, or fraud detection.

The second column contains the estimated probabilities for the positive class, so let's pass them to the `precision_recall_curve()` function:

```
y_scores_forest = y_probas_forest[:, 1]
precisions_forest, recalls_forest, thresholds_forest = precision_recall_curve(
    y_train_5, y_scores_forest)
```

Now we're ready to plot the PR curve. It is useful to plot the first PR curve as well to see how they compare (Figure 3-8):

4 Scikit-Learn classifiers always have either a `decision_function()` method or a `predict_proba()` method, or sometimes both.

```
plt.plot(recalls_forest, precisions_forest, "b-", linewidth=2,
         label="Random Forest")
plt.plot(recalls, precisions, "--", linewidth=2, label="SGD")
[...]  # beautify the figure: add labels, grid, and legend
plt.show()
```

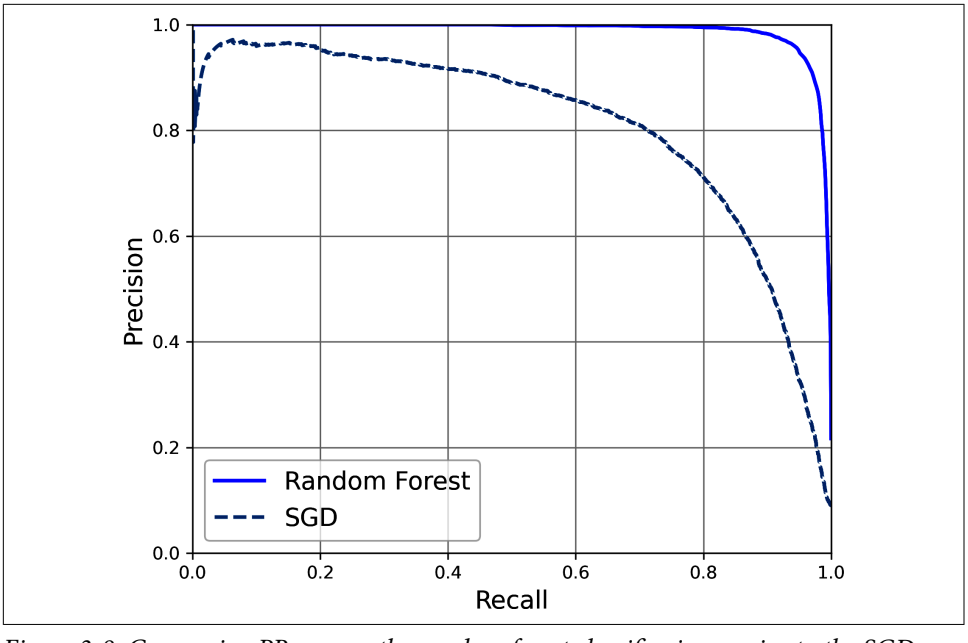

Figure 3-8. Comparing PR curves: the random forest classifier is superior to the SGD classifier because its PR curve is much closer to the top-right corner, and it has a greater AUC

As you can see in Figure 3-8, the `RandomForestClassifier`'s PR curve looks much better than the `SGDClassifier`'s: it comes much closer to the top-right corner. Its F_1 score and ROC AUC score are also significantly better:

```
>>> y_train_pred_forest = y_probas_forest[:, 1] >= 0.5  # positive proba ≥ 50%
>>> f1_score(y_train_5, y_train_pred_forest)
0.9274509803921569
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

Try measuring the precision and recall scores: you should find about 99.0% precision and 87.3% recall. Not too bad!

You now know how to train binary classifiers, choose the appropriate metric for your task, evaluate your classifiers using cross-validation, select the precision/recall trade-off that fits your needs, and use several metrics and curves to compare various models. You're ready to try to detect more than just the 5s.

Multiclass Classification

Whereas binary classifiers distinguish between two classes, *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.

Some Scikit-Learn classifiers (e.g., `LogisticRegression`, `RandomForestClassifier`, and `GaussianNB`) are capable of handling multiple classes natively. Others are strictly binary classifiers (e.g., `SGDClassifier` and `SVC`). However, there are various strategies that you can use to perform multiclass classification with multiple binary classifiers.

One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on). Then when you want to classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score. This is called the *one-versus-the-rest* (OvR) strategy, or sometimes *one-versus-all* (OvA).

Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. This is called the *one-versus-one* (OvO) strategy. If there are N classes, you need to train $N \times (N - 1) / 2$ classifiers. For the MNIST problem, this means training 45 binary classifiers! When you want to classify an image, you have to run the image through all 45 classifiers and see which class wins the most duels. The main advantage of OvO is that each classifier only needs to be trained on the part of the training set containing the two classes that it must distinguish.

Some algorithms (such as support vector machine classifiers) scale poorly with the size of the training set. For these algorithms OvO is preferred because it is faster to train many classifiers on small training sets than to train few classifiers on large training sets. For most binary classification algorithms, however, OvR is preferred.

Scikit-Learn detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvR or OvO, depending on the algorithm. Let's try this with a support vector machine classifier using the `sklearn.svm.SVC` class (see the online chapter on SVMs at *https://homl.info*). We'll only train on the first 2,000 images, or else it will take a very long time:

```
from sklearn.svm import SVC

svm_clf = SVC(random_state=42)
svm_clf.fit(X_train[:2000], y_train[:2000])  # y_train, not y_train_5
```

That was easy! We trained the `SVC` using the original target classes from 0 to 9 (`y_train`), instead of the 5-versus-the-rest target classes (`y_train_5`). Since there are 10 classes (i.e., more than 2), Scikit-Learn used the OvO strategy and trained 45 binary classifiers. Now let's make a prediction on an image:

```
>>> svm_clf.predict([some_digit])
array(['5'], dtype=object)
```

That's correct! This code actually made 45 predictions—one per pair of classes—and it selected the class that won the most duels.[5] If you call the `decision_function()` method, you will see that it returns 10 scores per instance: one per class. Each class gets a score equal to the number of won duels plus or minus a small tweak (max ±0.33) to break ties, based on the classifier scores:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores.round(2)
array([[ 3.79,  0.73,  6.06,  8.3 , -0.29,  9.3 ,  1.75,  2.77,  7.21,
         4.82]])
```

The highest score is 9.3, and it's indeed the one corresponding to class 5:

```
>>> class_id = some_digit_scores.argmax()
>>> class_id
np.int64(5)
```

When a classifier is trained, it stores the list of target classes in its `classes_` attribute, ordered by value. In the case of MNIST, the index of each class in the `classes_` array conveniently matches the class itself (e.g., the class at index 5 happens to be class `'5'`), but in general you won't be so lucky; you will need to look up the class label like this:

```
>>> svm_clf.classes_
array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object)
>>> svm_clf.classes_[class_id]
'5'
```

If you want to force Scikit-Learn to use one-versus-one or one-versus-the-rest, you can use the `OneVsOneClassifier` or `OneVsRestClassifier` classes. Simply create an instance and pass a classifier to its constructor (it doesn't even have to be a binary classifier). For example, this code creates a multiclass classifier using the OvR strategy, based on an `SVC`:

```
from sklearn.multiclass import OneVsRestClassifier

ovr_clf = OneVsRestClassifier(SVC(random_state=42))
ovr_clf.fit(X_train[:2000], y_train[:2000])
```

Let's make a prediction, and check the number of trained classifiers:

```
>>> ovr_clf.predict([some_digit])
array(['5'], dtype='<U1')
>>> len(ovr_clf.estimators_)
10
```

5 In case of a tie, the first class is selected, unless you set the `break_ties` hyperparameters to `True`, in which case ties are broken using the output of the `decision_function()`.

Training an `SGDClassifier` on a multiclass dataset and using it to make predictions is just as easy:

```
>>> sgd_clf = SGDClassifier(random_state=42)
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array(['3'], dtype='<U1')
```

Oops, that's incorrect. Prediction errors do happen! This time Scikit-Learn used the OvR strategy under the hood: since there are 10 classes, it trained 10 binary classifiers. The `decision_function()` method now returns one value per class. Let's look at the scores that the SGD classifier assigned to each class:

```
>>> sgd_clf.decision_function([some_digit]).round()
array([[-31893., -34420.,  -9531.,   1824., -22320.,  -1386., -26189.,
        -16148.,  -4604., -12051.]])
```

You can see that the classifier is not very confident about its prediction: almost all scores are very negative, while class 3 has a score of +1,824, and class 5 is not too far behind at –1,386. Of course, you'll want to evaluate this classifier on more than one image. Since there are roughly the same number of images in each class, the accuracy metric is fine. As usual, you can use the `cross_val_score()` function to evaluate the model:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.87365, 0.85835, 0.8689 ])
```

It gets over 85.8% on all test folds. If you used a random classifier, you would get 10% accuracy, so this is not such a bad score, but you can still do much better. Simply scaling the inputs (as discussed in Chapter 2) increases accuracy above 89.1%:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype("float64"))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([0.8983, 0.891 , 0.9018])
```

Error Analysis

If this were a real project, you would now follow the steps in your machine learning project checklist (see *https://homl.info/checklist*). You'd explore data preparation options, try out multiple models, shortlist the best ones, fine-tune their hyperparameters using `GridSearchCV`, and automate as much as possible. Here, we will assume that you have found a promising model and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.

First, look at the confusion matrix. For this, you first need to make predictions using the `cross_val_predict()` function; then you can pass the labels and predictions to the `confusion_matrix()` function, just like you did earlier. However, since there are now 10 classes instead of 2, the confusion matrix will contain quite a lot of numbers, and it may be hard to read.

A colored diagram of the confusion matrix is much easier to analyze. To plot such a diagram, use the `ConfusionMatrixDisplay.from_predictions()` function like this:

```
from sklearn.metrics import ConfusionMatrixDisplay

y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)
plt.show()
```

This produces the left diagram in Figure 3-9. This confusion matrix looks pretty good: most images are on the main diagonal, which means that they were classified correctly. Notice that the cell on the diagonal in row #5 and column #5 looks slightly darker than the other digits. This could be because the model made more errors on 5s, or because there are fewer 5s in the dataset than the other digits. That's why it's important to normalize the confusion matrix by dividing each value by the total number of images in the corresponding (true) class (i.e., divide by the row's sum). This can be done simply by setting `normalize="true"`. We can also specify the `values_format=".0%"` argument to show percentages with no decimals. The following code produces the diagram on the right in Figure 3-9:

```
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                        normalize="true", values_format=".0%")
plt.show()
```

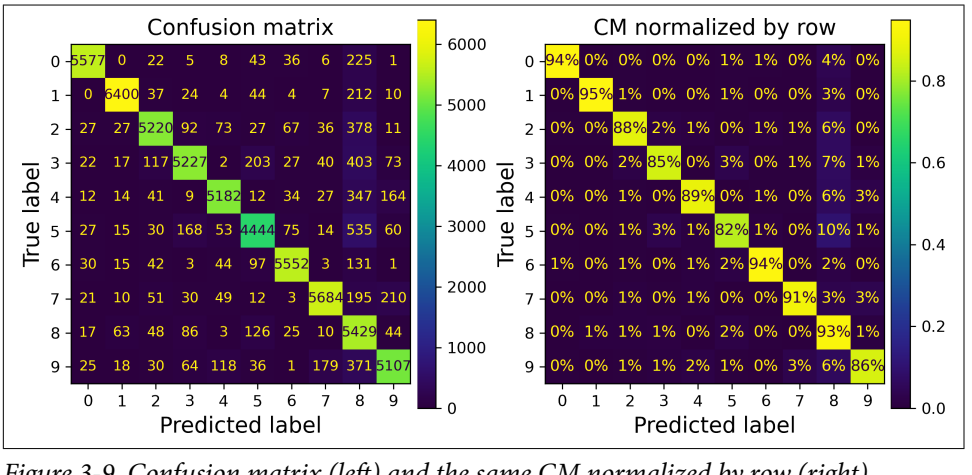

Figure 3-9. Confusion matrix (left) and the same CM normalized by row (right)

Now we can easily see that only 82% of the images of 5s were classified correctly. The most common error the model made with images of 5s was to misclassify them as 8s: this happened for 10% of all 5s. But only 2% of 8s got misclassified as 5s; confusion matrices are generally not symmetrical! If you look carefully, you will notice that many digits have been misclassified as 8s, but this is not immediately obvious from this diagram. If you want to make the errors stand out more, you can try putting zero weight on the correct predictions. The following code does just that and produces the diagram on the left in Figure 3-10:

```
sample_weight = (y_train_pred != y_train)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                        sample_weight=sample_weight,
                                        normalize="true", values_format=".0%")
plt.show()
```

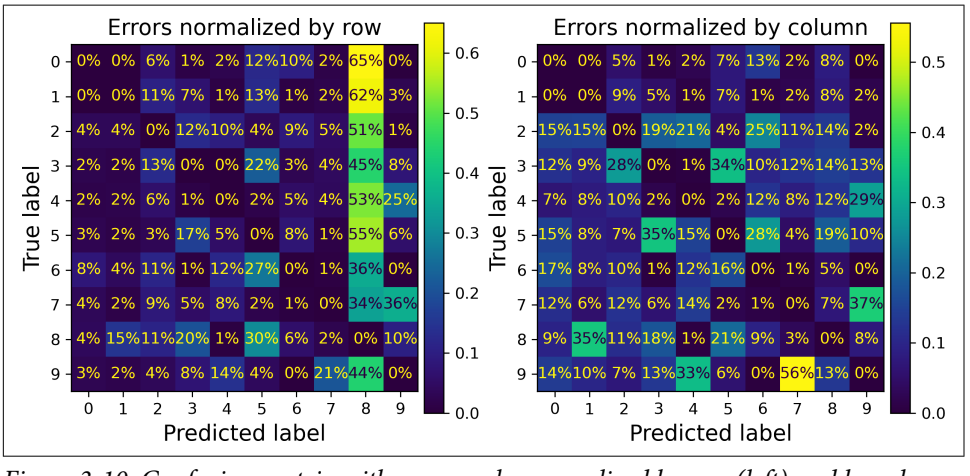

Figure 3-10. Confusion matrix with errors only, normalized by row (left) and by column (right)

Now you can see much more clearly the kinds of errors the classifier makes. The column for class 8 is now really bright, which confirms that many images got misclassified as 8s. In fact this is the most common misclassification for almost all classes. But be careful how you interpret the percentages in this diagram: remember that we've excluded the correct predictions. For example, the 36% in row #7, column #9 in the left grid does *not* mean that 36% of all images of 7s were misclassified as 9s. It means that 36% of the *errors* the model made on images of 7s were misclassifications as 9s. In reality, only 3% of images of 7s were misclassified as 9s, as you can see in the diagram on the right in Figure 3-9.

It is also possible to normalize the confusion matrix by column rather than by row: if you set `normalize="pred"`, you get the diagram on the right in Figure 3-10. For example, you can see that 56% of misclassified 7s are actually 9s.

Analyzing the confusion matrix often gives you insights into ways to improve your classifier. Looking at these plots, it seems that your efforts should be spent on reducing the false 8s. For example, you could try to gather more training data for digits that look like 8s (but are not) so that the classifier can learn to distinguish them from real 8s. Or you could engineer new features that would help the classifier—for example, writing an algorithm to count the number of closed loops (e.g., 8 has two, 6 has one, 5 has none). Or you could preprocess the images (e.g., using Scikit-Image, Pillow, or OpenCV) to make some patterns, such as closed loops, stand out more.

Analyzing individual errors can also be a good way to gain insights into what your classifier is doing and why it is failing. For example, let's plot examples of 3s and 5s in a confusion matrix style (Figure 3-11):

```
cl_a, cl_b = '3', '5'
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]
[...]  # plot all images in X_aa, X_ab, X_ba, X_bb in a confusion matrix style
```

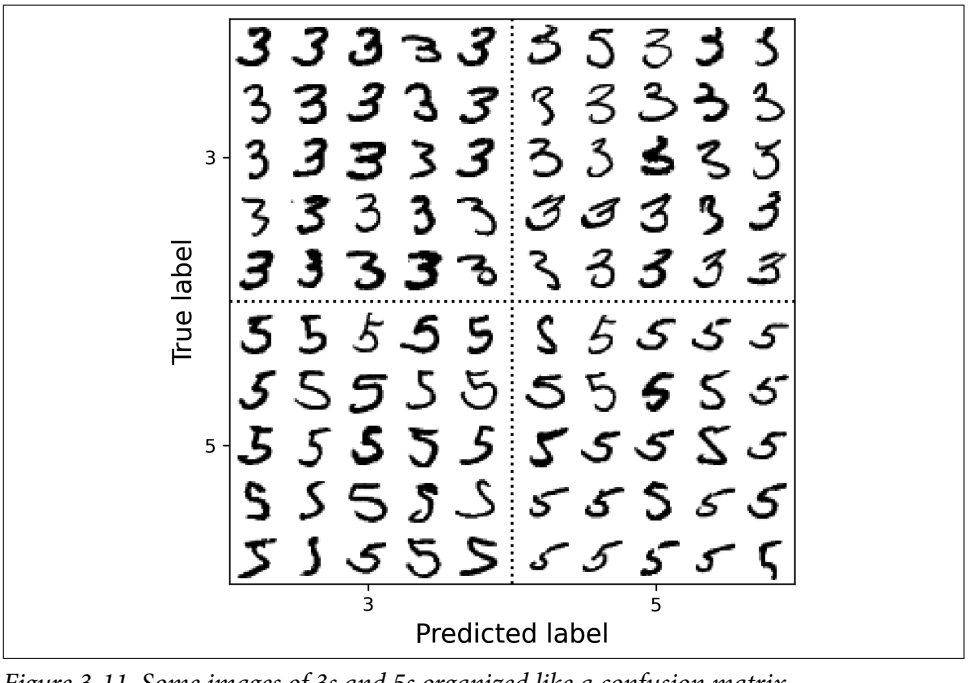

Figure 3-11. Some images of 3s and 5s organized like a confusion matrix

As you can see, some of the digits that the classifier gets wrong (i.e., in the bottom-left and top-right blocks) are so badly written that even a human would have trouble classifying them. However, most misclassified images seem like obvious errors to us.

It may be hard to understand why the classifier made the mistakes it did, but remember that the human brain is a fantastic pattern recognition system, and our visual system does a lot of complex preprocessing before any information even reaches our consciousness. So, the fact that this task feels simple does not mean that it is. Recall that we used a simple `SGDClassifier`, which is just a linear model: all it does is assign a weight per class to each pixel, and when it sees a new image it just sums up the weighted pixel intensities to get a score for each class. Since 3s and 5s differ by only a few pixels, this model will easily confuse them.

The main difference between 3s and 5s is the position of the small line that joins the top line to the bottom arc. If you draw a 3 with the junction slightly shifted to the left, the classifier might classify it as a 5, and vice versa. In other words, this classifier is quite sensitive to image shifting and rotation. One way to reduce the 3/5 confusion is to preprocess the images to ensure that they are well centered and not too rotated. However, this may not be easy since it requires predicting the correct rotation of each image. A much simpler approach consists of augmenting the training set with slightly shifted and rotated variants of the training images. This will force the model to learn to be more tolerant to such variations. This is called *data augmentation* (we'll cover this in Chapter 12; also see exercise 2 at the end of this chapter).

Multilabel Classification

Until now, each instance has always been assigned to just one class. But in some cases you may want your classifier to output multiple classes for each instance. Consider a face-recognition classifier: what should it do if it recognizes several people in the same picture? It should attach one tag per person it recognizes. Say the classifier has been trained to recognize three faces: Alice, Bob, and Charlie. Then when the classifier is shown a picture of Alice and Charlie, it should output `[True, False, True]` (meaning "Alice yes, Bob no, Charlie yes"). Such a classification system that outputs multiple binary tags is called a *multilabel classification* system.

We won't go into face recognition just yet, but let's look at a simpler example, just for illustration purposes:

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= '7')
y_train_odd = (y_train.astype('int8') % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

This code creates a `y_multilabel` array containing two target labels for each digit image: the first indicates whether the digit is large (7, 8, or 9), and the second

indicates whether it is odd. Then the code creates a `KNeighborsClassifier` instance, which supports multilabel classification (not all classifiers do), and trains this model using the multiple targets array. Now you can make a prediction, and notice that it outputs two labels:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]])
```

And it gets it right! The digit 5 is indeed not large (`False`) and odd (`True`).

There are many ways to evaluate a multilabel classifier, and selecting the right metric really depends on your project. One approach is to measure the F_1 score for each individual label (or any other binary classifier metric discussed earlier), then simply compute the average score. The following code computes the average F_1 score across all labels:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.976410265560605
```

This approach assumes that all labels are equally important, which may not be the case. In particular, if you have many more pictures of Alice than of Bob or Charlie, you may want to give more weight to the classifier's score on pictures of Alice. One simple option is to give each label a weight equal to its *support* (i.e., the number of instances with that target label). To do this, simply set `average="weighted"` when calling the `f1_score()` function.[6]

If you wish to use a classifier that does not natively support multilabel classification, such as `SVC`, one possible strategy is to train one model per label. However, this strategy may have a hard time capturing the dependencies between the labels. For example, a large digit (7, 8, or 9) is twice more likely to be odd than even, but the classifier for the "odd" label does not know what the classifier for the "large" label predicted. To solve this issue, the models can be organized in a chain: when a model makes a prediction, it uses the input features plus all the predictions of the models that come before it in the chain.

The good news is that Scikit-Learn has a class called `ClassifierChain` that does just that! By default it will use the true labels for training, feeding each model the appropriate labels depending on their position in the chain. But if you set the `cv` hyperparameter, it will use cross-validation to get "clean" (out-of-sample) predictions from each trained model for every instance in the training set, and these predictions will then be used to train all the models later in the chain. Note that the order of the classifiers in the chain may affect the final performance. Here's an example showing

6 Scikit-Learn offers a few other averaging options and multilabel classifier metrics; see the documentation for more details.

how to create and train a `ClassifierChain` using the cross-validation strategy. As earlier, we'll just use the first 2,000 images in the training set to speed things up:

```
from sklearn.multioutput import ClassifierChain

chain_clf = ClassifierChain(SVC(), cv=3, random_state=42)
chain_clf.fit(X_train[:2000], y_multilabel[:2000])
```

Now we can use this `ClassifierChain` to make predictions:

```
>>> chain_clf.predict([some_digit])
array([[0., 1.]])
```

Multioutput Classification

The last type of classification task we'll discuss here is called *multioutput–multiclass classification* (or just *multioutput classification*). It is a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).

To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). This is thus an example of a multioutput classification system.

The line between classification and regression is sometimes blurry, such as in this example. Arguably, predicting pixel intensity is more akin to regression than to classification. Moreover, multioutput systems are not limited to classification tasks; you could even have a system that outputs multiple labels per instance, including both class labels and value labels.

Let's start by creating the training and test sets by taking the MNIST images and adding noise to their pixel intensities, using a random number generator's `integers()` method. The target images will be the original images:

```
rng = np.random.default_rng(seed=42)
noise_train = rng.integers(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise_train
noise_test = rng.integers(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise_test
y_train_mod = X_train
y_test_mod = X_test
```

Let's take a peek at the first image from the test set (Figure 3-12). Yes, we're snooping on the test data, so you should be frowning right now.

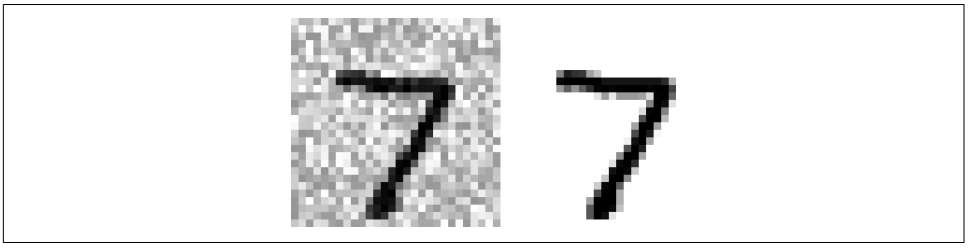

Figure 3-12. A noisy image (left) and the target clean image (right)

On the left is the noisy input image, and on the right is the clean target image. Now let's train the classifier and make it clean up this image (Figure 3-13):

```
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[0]])
plot_digit(clean_digit)
plt.show()
```

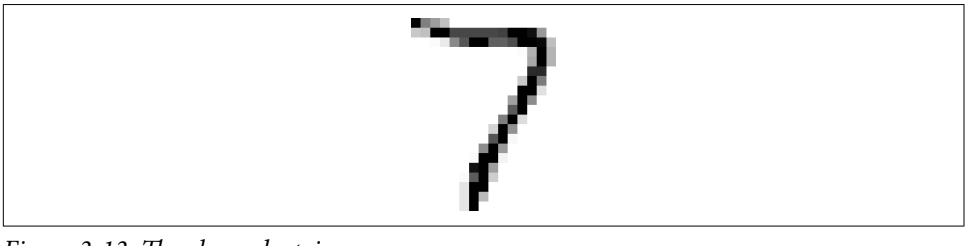

Figure 3-13. The cleaned-up image

Looks close enough to the target! This concludes our tour of classification. You now know how to select good metrics for classification tasks, pick the appropriate precision/recall trade-off, compare classifiers, and more generally build good classification systems for a variety of tasks. In the next chapters, you'll learn how all these machine learning models you've been using actually work.

Exercises

1. Try to build a classifier for the MNIST dataset that achieves over 97% accuracy on the test set. Hint: the `KNeighborsClassifier` works quite well for this task; you just need to find good hyperparameter values (try a grid search on the `weights` and `n_neighbors` hyperparameters).

2. Write a function that can shift an MNIST image in any direction (left, right, up, or down) by one pixel.[7] Then, for each image in the training set, create four shifted copies (one per direction) and add them to the training set. Finally, train your best model on this expanded training set and measure its accuracy on the test set. You should observe that your model performs even better now! This technique of artificially growing the training set is called *data augmentation* or *training set expansion*.
3. Tackle the Titanic dataset. A great place to start is on Kaggle. Alternatively, you can download the data from *https://homl.info/titanic.tgz* and unzip this tarball like you did for the housing data in Chapter 2. This will give you two CSV files, *train.csv* and *test.csv*, which you can load using `pandas.read_csv()`. The goal is to train a classifier that can predict the `Survived` column based on the other columns.
4. Build a spam classifier (a more challenging exercise):
   a. Download examples of spam and ham from Apache SpamAssassin's public datasets.
   b. Unzip the datasets and familiarize yourself with the data format.
   c. Split the data into a training set and a test set.
   d. Write a data preparation pipeline to convert each email into a feature vector. Your preparation pipeline should transform an email into a (sparse) vector that indicates the presence or absence of each possible word. For example, if all emails only ever contain four words, "Hello", "how", "are", "you", then the email "Hello you Hello Hello you" would be converted into a vector [1, 0, 0, 1] (meaning ["Hello" is present, "how" is absent, "are" is absent, "you" is present]), or [3, 0, 0, 2] if you prefer to count the number of occurrences of each word.

      You may want to add hyperparameters to your preparation pipeline to control whether to strip off email headers, convert each email to lowercase, remove punctuation, replace all URLs with "URL", replace all numbers with "NUMBER", or even perform *stemming* (i.e., trim off word endings; there are Python libraries available to do this).
   e. Finally, try out several classifiers and see if you can build a great spam classifier, with both high recall and high precision.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

---

7 You can use the `shift()` function from the `scipy.ndimage.interpolation` module. For example, `shift(image, [2, 1], cval=0)` shifts the image two pixels down and one pixel to the right.

CHAPTER 4
Training Models

So far we have treated machine learning models and their training algorithms mostly like black boxes. If you went through some of the exercises in the previous chapters, you may have been surprised by how much you can get done without knowing anything about what's under the hood: you optimized a regression system, you improved a digit image classifier, and you even built a spam classifier from scratch, all without knowing how they actually work. Indeed, in many situations you don't really need to know the implementation details.

However, having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task. Understanding what's under the hood will also help you debug issues and perform error analysis more efficiently. Lastly, most of the topics discussed in this chapter will be essential in understanding, building, and training neural networks (discussed in Part II of this book).

In this chapter we will start by looking at the linear regression model, one of the simplest models there is. We will discuss two very different ways to train it:

- Using a "closed-form" equation[1] that directly computes the model parameters that best fit the model to the training set (i.e., the model parameters that minimize the cost function over the training set).
- Using an iterative optimization approach called gradient descent (GD) that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method. We will look at a few variants of gradient descent that we will use again and

1 A closed-form equation is only composed of a finite number of constants, variables, and standard operations: for example, $a = \sin(b - c)$. No infinite sums, no limits, no integrals, etc.

again when we study neural networks in Part II: batch GD, mini-batch GD, and stochastic GD.

Next we will look at polynomial regression, a more complex model that can fit nonlinear datasets. Since this model has more parameters than linear regression, it is more prone to overfitting the training data. We will explore how to detect whether this is the case using learning curves, and then we will look at several regularization techniques that can reduce the risk of overfitting the training set.

Finally, we will examine two more models that are commonly used for classification tasks: logistic regression and softmax regression.

There will be quite a few math equations in this chapter using basic concepts of linear algebra and calculus. To understand these equations, you need to be familiar with vectors and matrices—how to transpose them, multiply them, and invert them—as well as partial derivatives. If these concepts are unfamiliar, please review the introductory Jupyter notebooks on linear algebra and calculus provided in the online supplemental material. If you are truly allergic to math, you can just skip the equations; the text should still help you grasp most of the concepts. That said, learning the mathematical formalism is extremely useful, as it will allow you to read ML papers. Although it may seem daunting at first, it's actually not that hard, and this chapter includes code that should help you make sense of the equations.

Linear Regression

In Chapter 1 we looked at a simple linear model of life satisfaction (Equation 4-1).

Equation 4-1. A simple linear model of life satisfaction

$$\text{life\_satisfaction} = \theta_0 + \theta_1 \times \text{GDP\_per\_capita}$$

This model is just a linear function of the input feature `GDP_per_capita`. θ_0 and θ_1 are the model's parameters.

More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*), as shown in Equation 4-2.

Equation 4-2. Linear regression model prediction

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

In this equation:

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter, including the bias term θ_0 and the feature weights θ_1, θ_2, ⋯, θ_n.

This can be written much more concisely using a vectorized form, as shown in Equation 4-3.

Equation 4-3. 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 $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$.

In machine learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column. If $\boldsymbol{\theta}$ and $\mathbf{x}$ are column vectors, then the prediction is $\hat{y} = \boldsymbol{\theta}^\intercal \mathbf{x}$, where $\boldsymbol{\theta}^\intercal$ is the *transpose* of $\boldsymbol{\theta}$ (a row vector instead of a column vector) and $\boldsymbol{\theta}^\intercal \mathbf{x}$ is the matrix multiplication of $\boldsymbol{\theta}^\intercal$ and $\mathbf{x}$. It is of course the same prediction, except that it is now represented as a single-cell matrix rather than a scalar value. In this book I will use this notation to avoid switching between dot products and matrix multiplications.

OK, that's the linear regression model—but how do we train it? Well, recall that training a model means setting its parameters so that the model best fits the training set. For this purpose, we first need a measure of how well (or poorly) the model fits the training data. In Chapter 2 we saw that the most common performance measure of a regression model is the root mean squared error (Equation 2-1). Therefore, to train a linear regression model, we need to find the value of $\boldsymbol{\theta}$ 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. For example, classifiers are often trained using a cost function such as the log loss (as you will see later in this chapter) 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 $h_{\boldsymbol{\theta}}$ on a training set $\mathbf{X}$ is calculated using Equation 4-4.

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

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

Most of these notations were presented in Chapter 2 (see "Notations" on page 45). The only difference is that we write $h_{\boldsymbol{\theta}}$ instead of just h to make it clear that the model is parametrized by the vector $\boldsymbol{\theta}$. To simplify notations, we will just write MSE($\boldsymbol{\theta}$) instead of MSE($\mathbf{X}$, $h_{\boldsymbol{\theta}}$).

The Normal Equation

To find the value of $\boldsymbol{\theta}$ that minimizes the MSE, there exists a *closed-form solution*—in other words, a mathematical equation that gives the result directly. This is called the *normal equation* (Equation 4-5).

Equation 4-5. Normal equation

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

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 $y^{(m)}$.

Let's generate some linear-looking data to test this equation on (Figure 4-1):

```
import numpy as np

rng = np.random.default_rng(seed=42)
m = 200  # number of instances
X = 2 * rng.random((m, 1))  # column vector
y = 4 + 3 * X + rng.standard_normal((m, 1))  # column vector
```

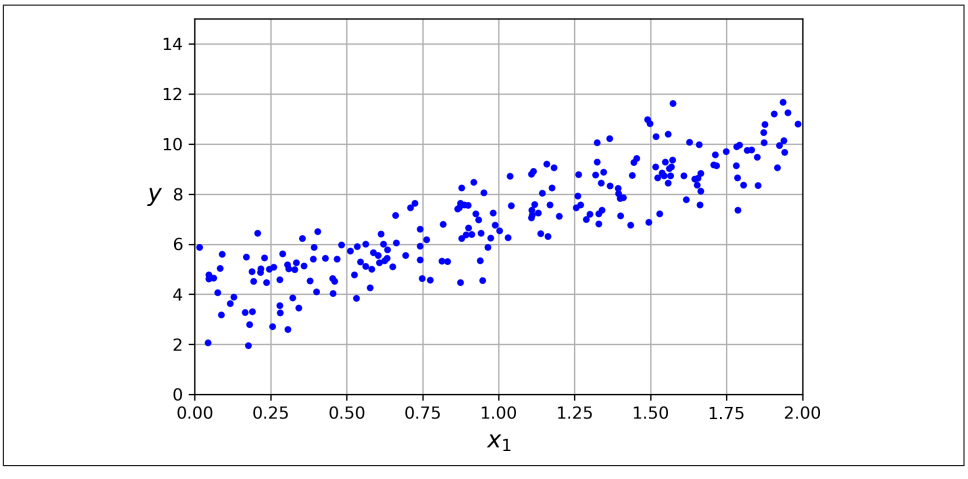

Figure 4-1. A randomly generated linear dataset

Now let's compute $\widehat{\boldsymbol{\theta}}$ using the normal equation. We will use the `inv()` function from NumPy's linear algebra module (`np.linalg`) to compute the inverse of a matrix, and the `@` operator 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
```

The `@` operator performs matrix multiplication. If `A` and `B` are NumPy arrays, then `A @ B` is equivalent to `np.matmul(A, B)`. Many other libraries, like TensorFlow, PyTorch, and JAX, support the `@` operator as well. However, you cannot use `@` on pure Python arrays (i.e., lists of lists).

The function that we used to generate the data is $y = 4 + 3x_1 + \text{Gaussian noise}$. Let's see what the equation found:

```
>>> theta_best
array([[3.69084138],
       [3.32960458]])
```

We would have hoped for $\theta_0 = 4$ and $\theta_1 = 3$ instead of $\theta_0 = 3.6908$ and $\theta_1 = 3.3296$. Close enough, but the noise made it impossible to recover the exact parameters of the original function. The smaller and noisier the dataset, the harder it gets.

Now we can make predictions using $\widehat{\boldsymbol{\theta}}$:

```
>>> 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([[ 3.69084138],
       [10.35005055]])
```

Let's plot this model's predictions (Figure 4-2):

```
import matplotlib.pyplot as plt

plt.plot(X_new, y_predict, "r-", label="Predictions")
plt.plot(X, y, "b.")
[...]  # beautify the figure: add labels, axis, grid, and legend
plt.show()
```

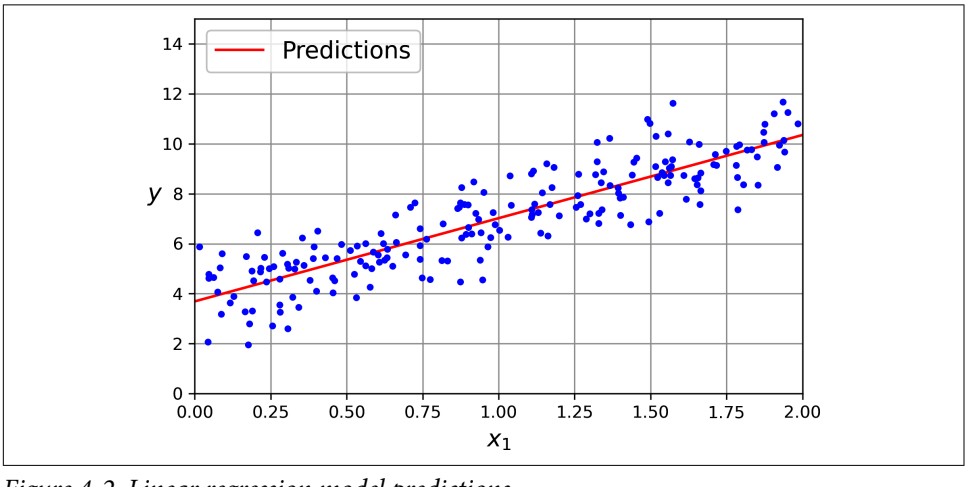

Figure 4-2. Linear regression model predictions

Performing linear regression using Scikit-Learn is relatively straightforward:

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([3.69084138]), array([[3.32960458]]))
>>> lin_reg.predict(X_new)
array([[ 3.69084138],
       [10.35005055]])
```

Notice that 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"), which you could call directly:

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[3.69084138],
       [3.32960458]])
```

This function computes $\widehat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$, where $\mathbf{X}^{+}$ is the *pseudoinverse* of $\mathbf{X}$ (specifically, the Moore–Penrose inverse). You can use `np.linalg.pinv()` to compute the pseudoinverse directly:

```
>>> np.linalg.pinv(X_b) @ y
array([[3.69084138],
       [3.32960458]])
```

The pseudoinverse itself is computed using a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices $\mathbf{U}\ \boldsymbol{\Sigma}\ \mathbf{V}^{\intercal}$ (see `numpy.linalg.svd()`). The pseudoinverse is computed as $\mathbf{X}^{+} = \mathbf{V}\boldsymbol{\Sigma}^{+}\mathbf{U}^{\intercal}$. To compute the matrix $\boldsymbol{\Sigma}^{+}$, the algorithm takes $\boldsymbol{\Sigma}$ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix. This approach is more efficient than computing the normal equation, plus it handles edge cases nicely: indeed, the normal equation may not work if the matrix $\mathbf{X}^{\intercal}\mathbf{X}$ is not invertible (i.e., singular), such as if $m < n$ or if some features are redundant, but the pseudoinverse is always defined.

Computational Complexity

The normal equation computes the inverse of $\mathbf{X}^{\intercal}\ \mathbf{X}$, which is an $(n + 1) \times (n + 1)$ matrix (where n is the number of features). The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^{3})$, depending on the implementation. In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^{3} = 8$.

The SVD approach used by Scikit-Learn's `LinearRegression` class is about $O(n^{2})$. If you double the number of features, you multiply the computation time by roughly 4.

Both the normal equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000). On the positive side, both are linear with regard to the number of instances in the training set (they are $O(m)$), so they handle large training sets efficiently, provided they can fit in memory.

Also, once you have trained your linear regression model (using the normal equation or any other algorithm), predictions are very fast: the computational complexity is linear with 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.

Now we will look at a very different way to train a linear regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory.

Gradient Descent

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.

Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what gradient descent does: it measures the local gradient of the error function with regard to the parameter vector $\boldsymbol{\theta}$, 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 $\boldsymbol{\theta}$ 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 (see Figure 4-3).

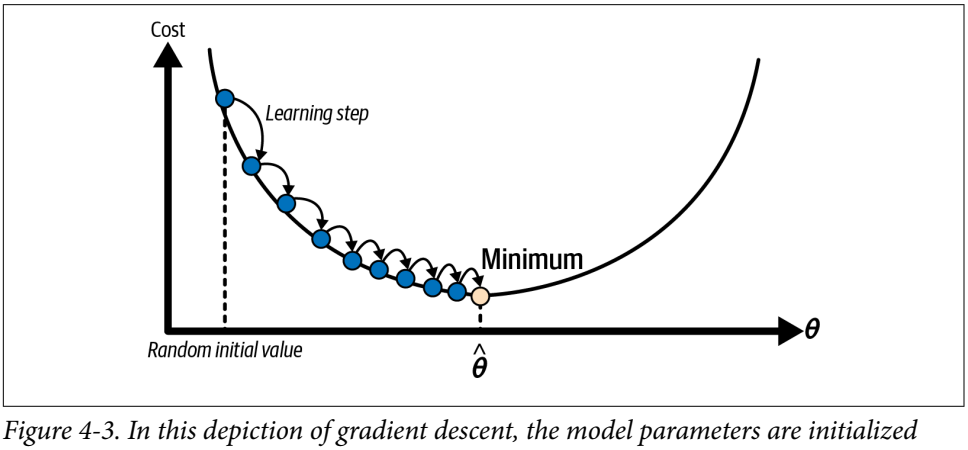

Figure 4-3. In this depiction of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the cost approaches the minimum

An important parameter in gradient descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 4-4).

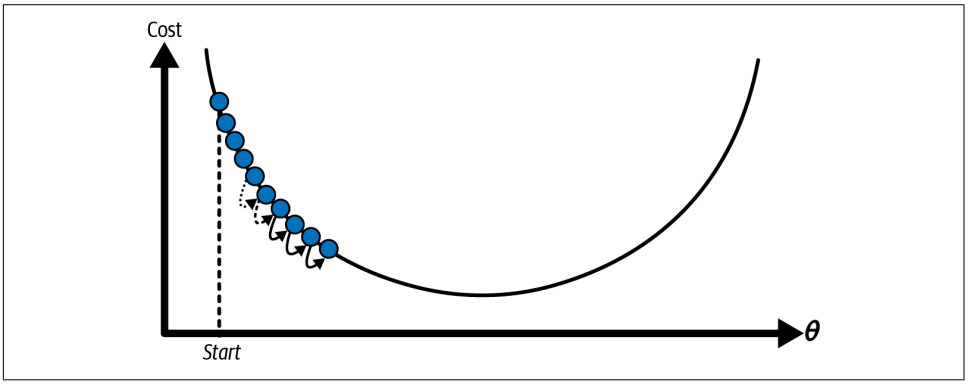

Figure 4-4. Learning rate too small

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 (see Figure 4-5).

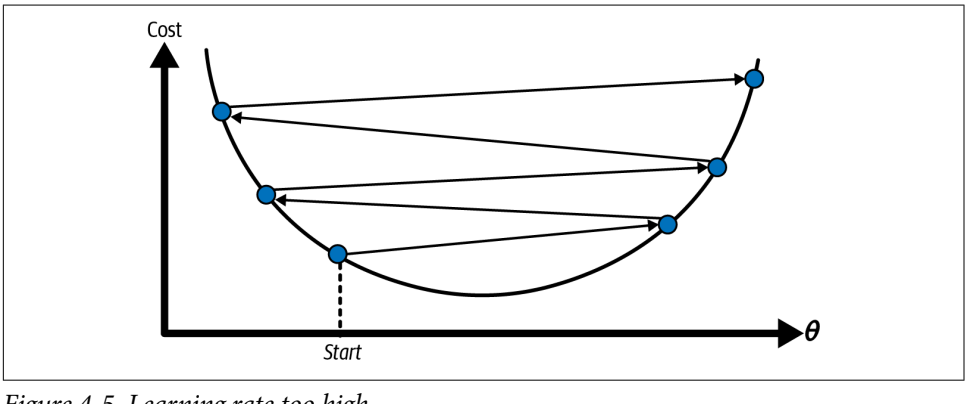

Figure 4-5. Learning rate too high

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.

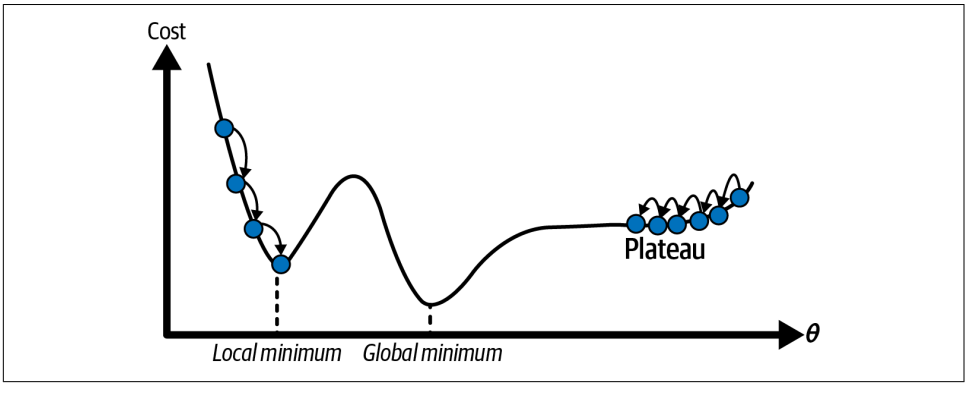

Figure 4-6. Gradient descent pitfalls

Fortunately, the MSE cost function for a linear regression model happens to be a *convex function*, which means that if you pick any two points on the curve, the line segment joining them 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.[2] 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).[3]

As you can see, on the left the gradient descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.

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: searching

2 Technically speaking, its derivative is *Lipschitz continuous*.

3 Since feature 1 is smaller, it takes a larger change in θ_1 to affect the cost function, which is why the bowl is elongated along the θ_1 axis.

for a needle in a 300-dimensional haystack is much trickier than in 3 dimensions. Fortunately, since the cost function is convex in the case of linear regression, the needle is simply at the bottom of the bowl.

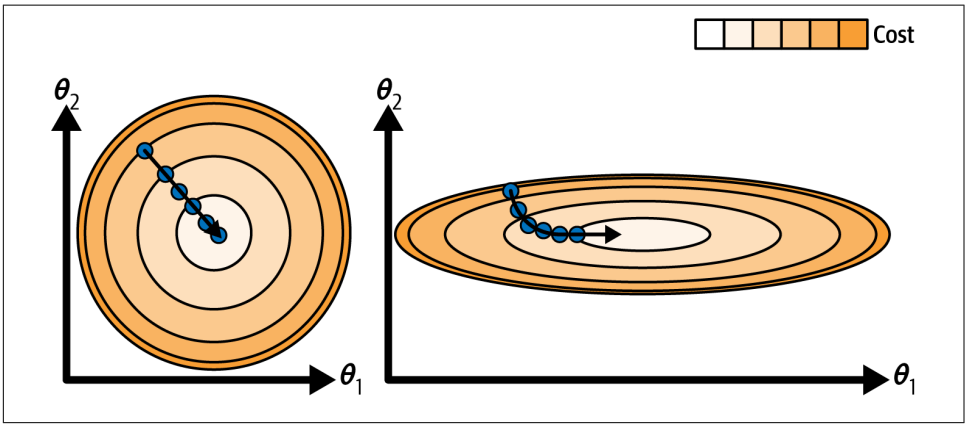

Figure 4-7. Gradient descent with (left) and without (right) feature scaling

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.

Batch Gradient Descent

Most models have more than one model parameter. Therefore, 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*. It is like asking, "What is the slope of the mountain toward the east?" and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions). Equation 4-6 computes the partial derivative of the MSE with regard to parameter θ_j, denoted $\partial \text{ MSE}(\boldsymbol{\theta}) / \partial\theta_j$.

Equation 4-6. Partial derivatives of the cost function

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

Instead of computing these partial derivatives individually, you can use Equation 4-7 to compute them all in one go. The gradient vector, denoted $\nabla_{\boldsymbol{\theta}}\text{MSE}(\boldsymbol{\theta})$, contains all the partial derivatives of the cost function (one for each model parameter).

Equation 4-7. Gradient vector of the cost function

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

Notice that this formula involves calculations over the full training set **X**, at each gradient descent step! This is why the algorithm is called *batch gradient descent*: it uses the whole batch of training data at every step (actually, *full gradient descent* would probably be a better name). As a result, it is terribly slow on very large training sets (we will look at some much faster gradient descent algorithms shortly). However, gradient descent scales well with the number of features; training a linear regression model when there are hundreds of thousands of features is much faster using gradient descent than using the normal equation or SVD decomposition.

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_{\boldsymbol{\theta}}\text{MSE}(\boldsymbol{\theta})$ from $\boldsymbol{\theta}$. This is where the learning rate η comes into play:[4] multiply the gradient vector by η to determine the size of the downhill step (Equation 4-8).

Equation 4-8. Gradient descent step

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

Let's look at a quick implementation of this algorithm:

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

rng = np.random.default_rng(seed=42)
theta = rng.standard_normal((2, 1))  # randomly initialized model parameters
```

4 Eta (η) is the seventh letter of the Greek alphabet.

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

That wasn't too hard! Each iteration over the training set is called an *epoch*. Let's look at the resulting `theta`:

```
>>> theta
array([[3.69084138],
       [3.32960458]])
```

Hey, that's exactly what the normal equation found! Gradient descent worked perfectly. But what if you had used a different learning rate (`eta`)? Figure 4-8 shows the first 20 steps of gradient descent using three different learning rates. The line at the bottom of each plot represents the random starting point, then each epoch is represented by a darker and darker line.

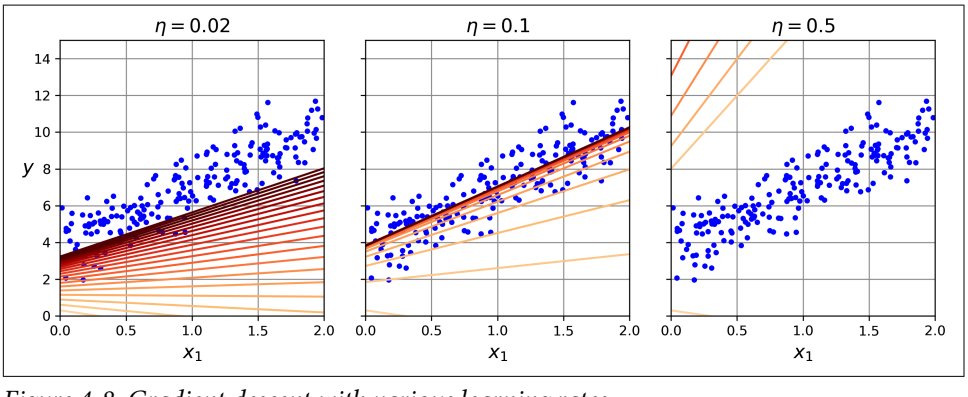

Figure 4-8. Gradient descent with various learning rates

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.

To find a good learning rate, you can use grid search (see Chapter 2). 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 vector 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.

Convergence Rate

When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), batch gradient descent with a fixed learning rate will eventually converge to the optimal solution, but you may have to wait a while: it can take $O(1/\varepsilon)$ iterations to reach the optimum within a range of ε, depending on the shape of the cost function. If you divide the tolerance by 10 to have a more precise solution, then the algorithm may have to run about 10 times longer.

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. At the opposite extreme, *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; see Chapter 1).

On the other hand, 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 (see Figure 4-9). Once the algorithm stops, the final parameter values will be good, but not optimal.

When the cost function is very irregular (as in Figure 4-6), this can actually help the algorithm jump out of local minima, so stochastic gradient descent has a better chance of finding the global minimum than batch gradient descent does.

Therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum. This process is akin to *simulated annealing*, an algorithm inspired by the process in metallurgy of annealing, where molten metal is slowly cooled down. 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.

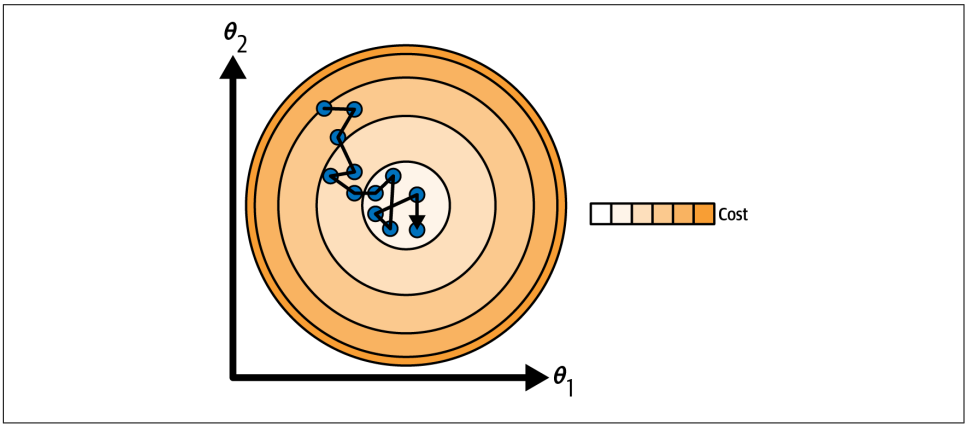

Figure 4-9. With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent

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)

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

for epoch in range(n_epochs):
    for iteration in range(m):
        random_index = rng.integers(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*, as earlier. 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([[3.69826475],
       [3.30748311]])
```

Figure 4-10 shows the first 20 steps of training (notice how irregular the steps are).

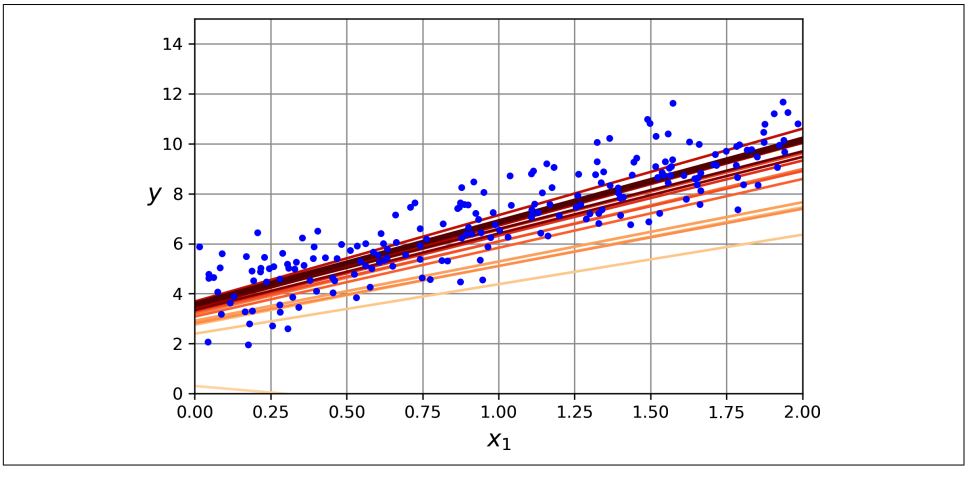

Figure 4-10. The first 20 steps of stochastic gradient descent

Note that since instances are picked randomly, some instances may be picked several times per epoch, while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, another approach is to shuffle the training set (making sure to shuffle the input features and the labels jointly), then go through it instance by instance, then shuffle it again, and so on. However, this approach is more complex, and it generally does not improve the result.

When using stochastic gradient descent, the training instances must be independent and identically distributed (IID) to ensure that the parameters get pulled toward the global optimum, on average. A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch). If you do not shuffle the instances—for example, if the instances are sorted by label—then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum.

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 a maximum of 1,000 epochs (`max_iter`) or until the loss drops by less than 10^{-5} (`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`; more details on this shortly):

```
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
```

Once again, you find a solution quite close to the one returned by the normal equation:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([3.68899733]), array([3.33054574]))
```

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

The last gradient descent algorithm we will look at is called *mini-batch gradient descent*. It is straightforward once you know batch and stochastic gradient descent: at each step, instead of computing the gradients based on the full training set (as in batch GD) or based on just one instance (as in stochastic GD), mini-batch GD computes the gradients on small random sets of instances called *mini-batches*. The main advantage of mini-batch GD over stochastic GD is that you can get a performance boost from hardware acceleration of matrix operations, especially when using *graphical processing units* (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 (in the case of problems that suffer from local minima, unlike linear regression with the MSE cost function). 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.

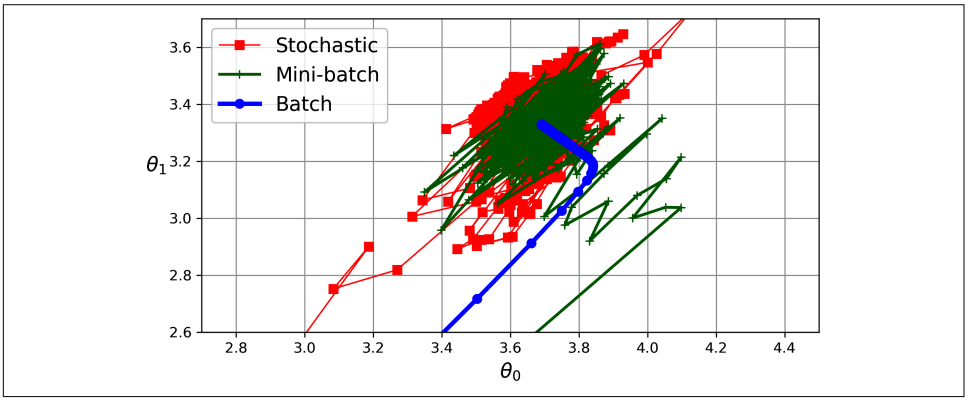

Figure 4-11. Gradient descent paths in parameter space

Table 4-1 compares the algorithms we've discussed so far for linear regression[5] (recall that m is the number of training instances and n is the number of features).

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	N/A
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	N/A

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

5 While the normal equation can only perform linear regression, the gradient descent algorithms can be used to train many other models, as you'll see.

Polynomial Regression

What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called *polynomial regression*.

Let's look at an example. First, we'll generate some nonlinear data (see Figure 4-12), based on a simple *quadratic equation*—that's an equation of the form $y = ax^2 + bx + c$—plus some noise:

```
rng = np.random.default_rng(seed=42)
m = 200  # number of instances
X = 6 * rng.random((m, 1)) - 3
y = 0.5 * X ** 2 + X + 2 + rng.standard_normal((m, 1))
```

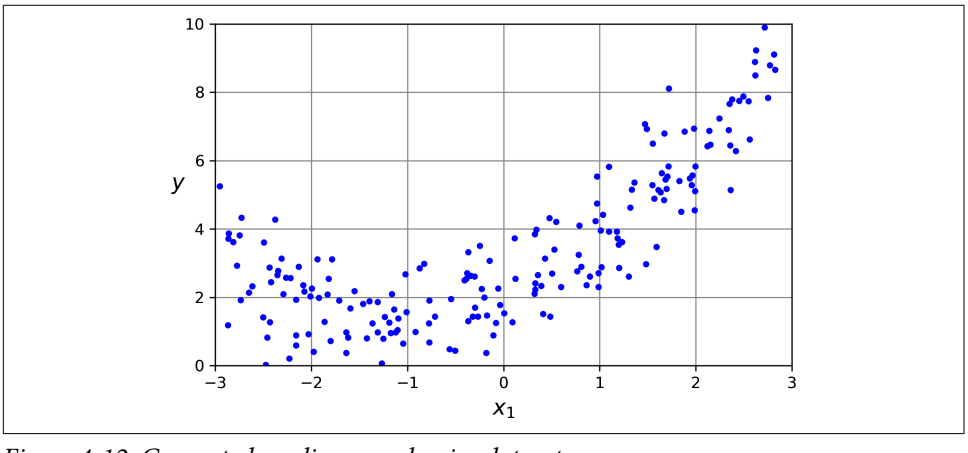

Figure 4-12. Generated nonlinear and noisy dataset

Clearly, a straight line will never fit this data properly. So let's 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([1.64373629])
>>> X_poly[0]
array([1.64373629, 2.701869  ])
```

`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 (Figure 4-13):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([2.00540719]), array([[1.11022126, 0.50526985]]))
```

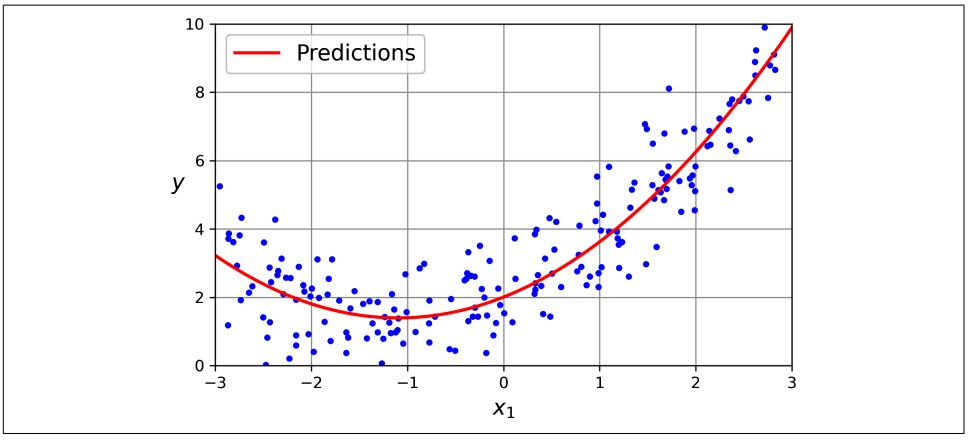

Figure 4-13. Polynomial regression model predictions

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

Note that when there are multiple features, polynomial regression is capable of finding relationships between features, which is something a plain linear regression model cannot do. This is made possible by the fact that `PolynomialFeatures` also adds all combinations of features up to the given degree. For example, if there were two features a and b, `PolynomialFeatures` with `degree=3` would not only add the features a^2, a^3, b^2, and b^3, but also the combinations ab, a^2b, and ab^2.

`PolynomialFeatures(degree=d)` transforms an array containing n features into an array containing $(n + d)! / d!n!$ features, where $n!$ is the *factorial* of n, equal to $1 \times 2 \times 3 \times \cdots \times n$. 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. For example, Figure 4-14 applies a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (second-degree polynomial). Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances.

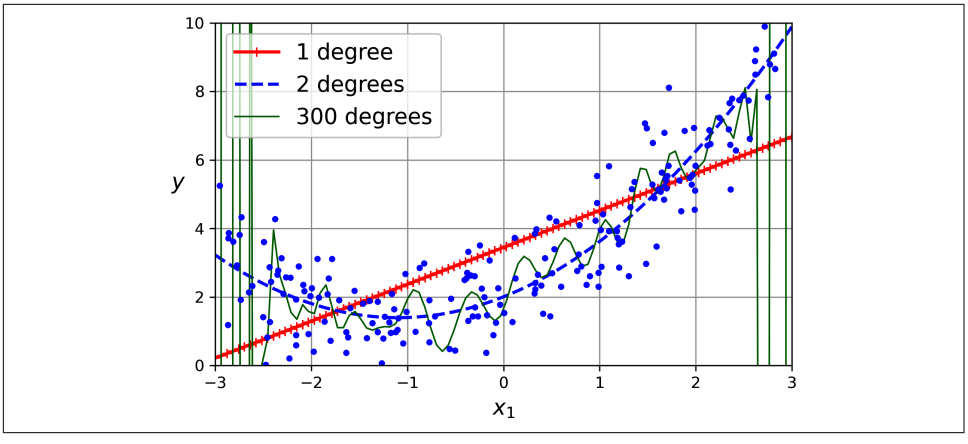

Figure 4-14. High-degree polynomial regression

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?

In Chapter 2 you used cross-validation to get an estimate of a model's generalization performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting. If it performs poorly on both, then it is underfitting. This is one way to tell when a model is too simple or too complex.

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. Let's use this function to look at the learning curves of the plain linear regression model (see Figure 4-15):

```
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()
```

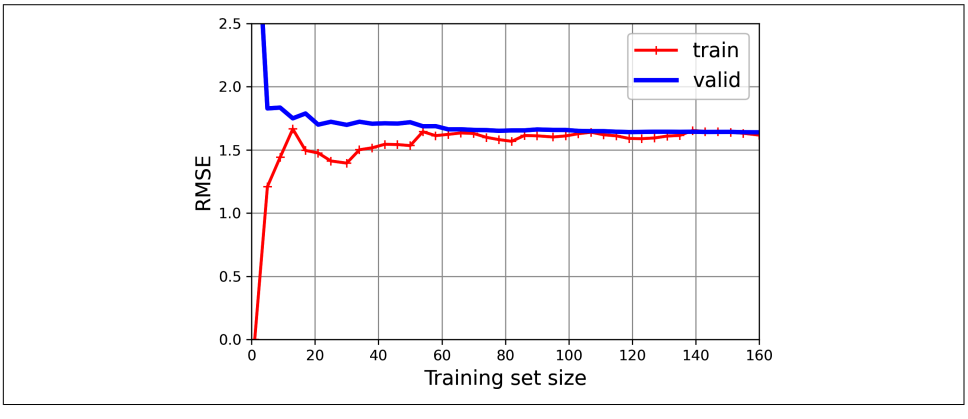

Figure 4-15. Learning curves

This model is underfitting, it's too simple for the data. How can we tell? Well, 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 set 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.

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.

Now let's look at the learning curves of a 10th-degree polynomial model on the same data (Figure 4-16):

```
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
```

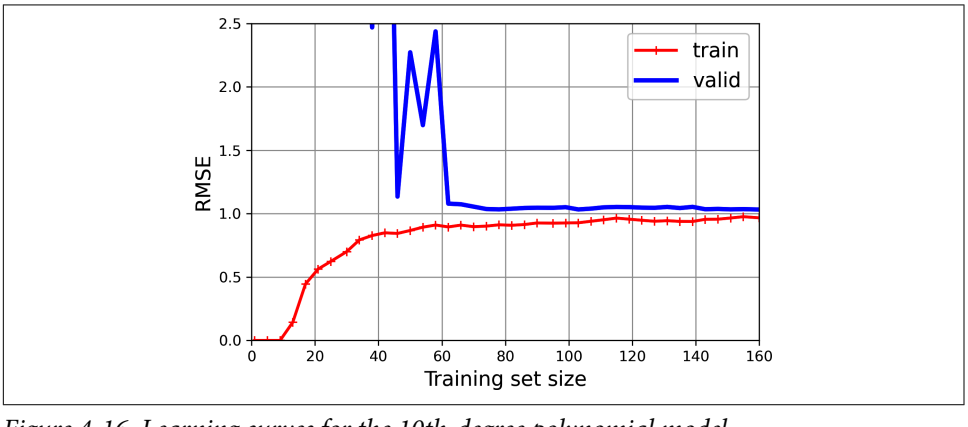

Figure 4-16. Learning curves for the 10th-degree polynomial model

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 better on the training data than on the validation data, which is the hallmark of an overfitting model. If you used a much larger training set, however, the two curves would continue to get closer.

The Bias/Variance Trade-Off

An important theoretical result of statistics and machine learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

Bias
: This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.[6]

Variance
: This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.

Irreducible error
: This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity (or increasing regularization) increases its bias and reduces its variance (see Figure 4-17). This is why it is called a trade-off.

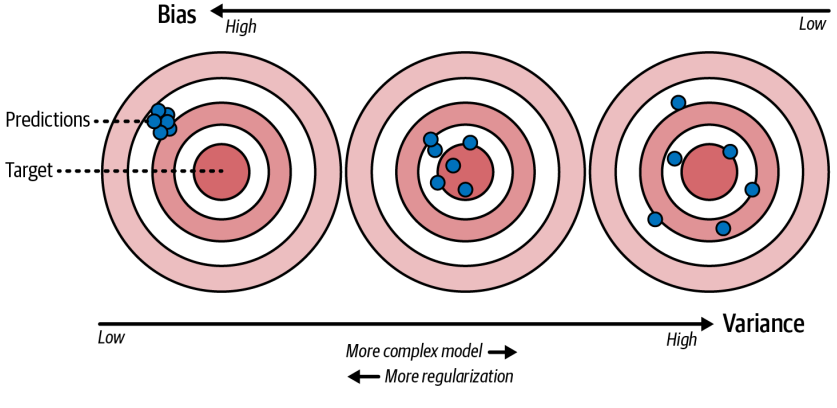

Figure 4-17. Bias/variance trade-off

6 This notion of bias is not to be confused with the bias term of linear models.

One way to improve an overfitting model is to feed it more training data until the validation error gets close enough to the training error.

Regularized Linear Models

As you saw in Chapters 1 and 2, 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.

What about linear models? Can we regularize them too? You may wonder why we may want to do that: aren't linear models constrained enough already? Well, linear regression makes a few assumptions, including the fact that the true relationship between the inputs and the outputs is linear, the noise has zero mean, constant variance, and is independent of the inputs, plus the input matrix has full rank, meaning that the inputs are not colinear[7] and there at least as many samples as parameters. In practice, some assumptions don't hold perfectly. For example, some inputs may be close to colinear, which makes linear regression numerically unstable, meaning that very small differences in the training set can have a big impact on the trained model. Regularization can stabilize linear models and make them more accurate.

So how can we regularize a linear model? This is usually done by constraining its weights. In this section, we will discuss ridge regression, lasso regression, and elastic net regression, which implement three different ways to do that.

Ridge Regression

Ridge regression (also called *Tikhonov regularization*) is a regularized version of linear regression: a *regularization term* equal to $\frac{\alpha}{m}\sum_{i=1}^{n}\theta_i^2$ is added to the MSE. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. This constraint makes the model less flexible, preventing it from stretching itself too much to fit every data point: this reduces the risk of overfitting. 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.

7 Inputs are colinear when one input is equal to a linear combination of some other inputs. For example, the temperature in Celsius degrees is colinear with the temperature in Fahrenheit degrees.

The hyperparameter α controls how much you want to regularize the model. If $\alpha = 0$, then ridge regression is just linear regression. If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean. Equation 4-9 presents the ridge regression cost function.[8]

Equation 4-9. Ridge regression cost function

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

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define $\mathbf{w}$ 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.[9] For batch gradient descent, just add $2\alpha\mathbf{w} / m$ 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 4-7).

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 4-18 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.

8 It is common to use the notation $J(\mathbf{\theta})$ for cost functions that don't have a short name; I'll often use this notation throughout the rest of this book. The context will make it clear which cost function is being discussed.

9 Norms are discussed in Chapter 2.

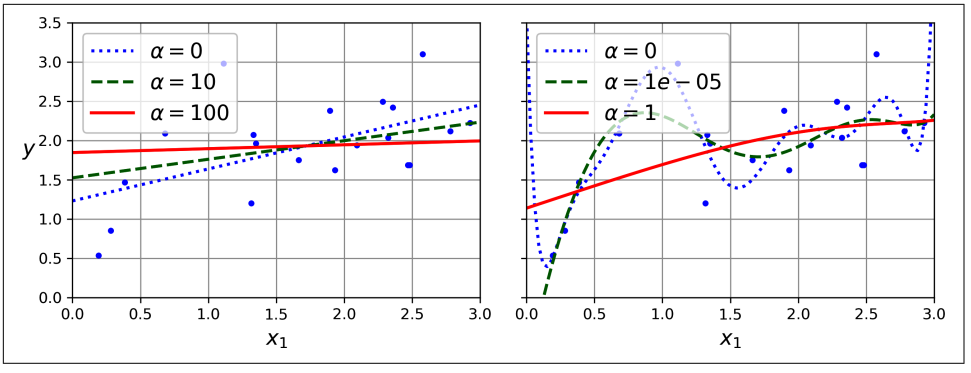

Figure 4-18. Linear (left) and polynomial (right) models, both with various levels of ridge regularization

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 4-10 shows the closed-form solution, where **A** is the $(n + 1) \times (n + 1)$ *identity matrix*,[10] except with a 0 in the top-left cell, corresponding to the bias term.

Equation 4-10. Ridge regression closed-form solution

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

Here is how to perform ridge regression with Scikit-Learn using a closed-form solution (a variant of Equation 4-10 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([1.84414523])
```

And using stochastic gradient descent:[11]

```
>>> 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([1.83659707])
```

10 A square matrix full of 0s except for 1s on the main diagonal (top left to bottom right).

11 Alternatively, you can use the `Ridge` class with the `"sag"` solver. Stochastic average GD is a variant of stochastic GD. For more details, see the presentation "Minimizing Finite Sums with the Stochastic Average Gradient Algorithm" by Mark Schmidt et al. from the University of British Columbia.

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 ℓ_2 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)`.

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 4-11). 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 (see Scikit-Learn issue #15657 for more details).

Equation 4-11. Lasso regression cost function

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

Figure 4-19 shows the same thing as Figure 4-18 but replaces the ridge models with lasso models and uses different α values.

An important characteristic of lasso regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero). For example, the dashed line in the righthand plot in Figure 4-19 (with $\alpha = 0.01$) 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. Of course, there's a trade-off: if you increase α too much, the model will be very sparse, but its performance will plummet.

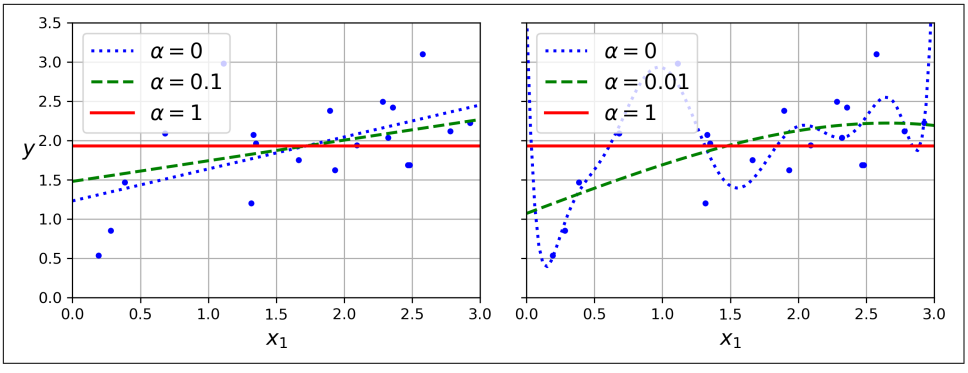

Figure 4-19. Linear (left) and polynomial (right) models, both using various levels of lasso regularization

You can get a sense of why the ℓ_1 norm induces sparsity by looking at Figure 4-20: 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 ($|\theta_1| + |\theta_2|$), which drops linearly as you get closer to any axis. For example, if you initialize the model parameters to $\theta_1 = 2$ and $\theta_2 = 0.5$, running gradient descent will decrement both parameters equally (as represented by the dashed yellow line); therefore θ_2 will reach 0 first (since it was closer to 0 to begin with). After that, gradient descent will roll down the gutter until it reaches $\theta_1 = 0$ (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). 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 $\theta_1 = 0.25$ and $\theta_2 = -1$: notice again how the path quickly reaches $\theta_2 = 0$, 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 $\theta_1 = 2$ and $\theta_2 = 0.5$).

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 descent 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.

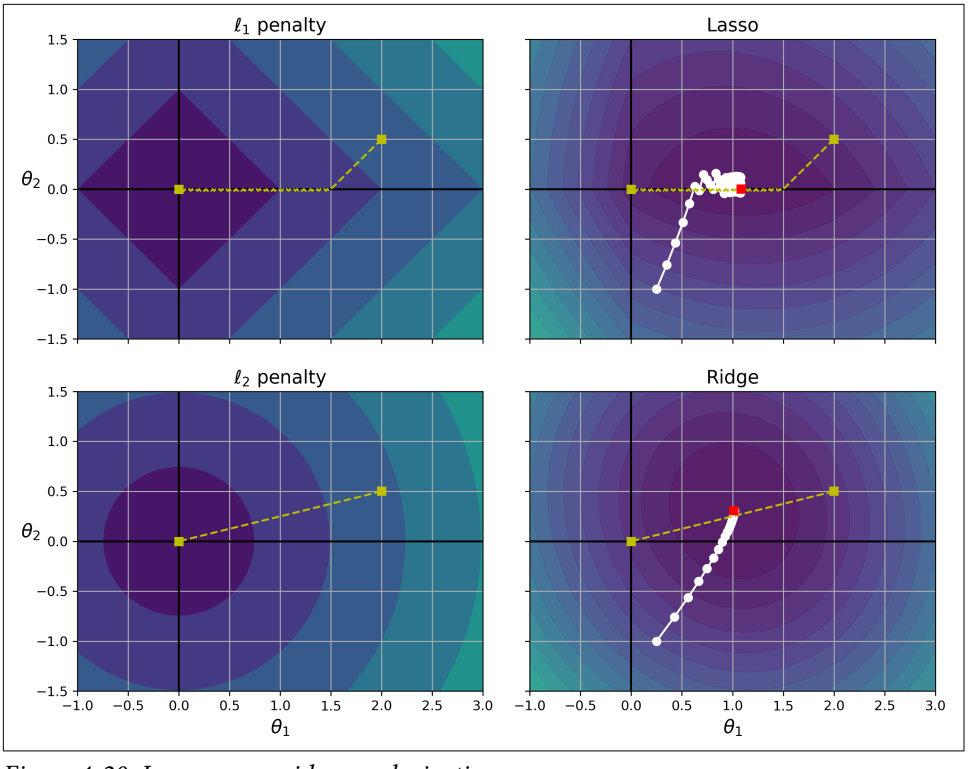

Figure 4-20. Lasso versus ridge regularization

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 $i = 1, 2, \cdots, n$), but gradient descent still works if you use a *subgradient vector* $\mathbf{g}$[12] instead when any $\theta_i = 0$. Equation 4-12 shows a subgradient vector equation you can use for gradient descent with the lasso cost function.

12 You can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point.

Equation 4-12. Lasso regression subgradient vector

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

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

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.87550211])
```

Note that you could instead use `SGDRegressor(penalty="l1", alpha=0.1)`.

Elastic Net Regression

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 4-13).

Equation 4-13. Elastic net cost function

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

So 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, but if you suspect that only a few features are useful, you should prefer lasso or elastic net because they tend to reduce the useless features' weights down to zero, as discussed earlier. In general, elastic net is preferred over lasso because lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

Here is a short example that uses Scikit-Learn's `ElasticNet` (`l1_ratio` corresponds to the mix ratio r):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
```

```
>>> elastic_net.predict([[1.5]])
array([1.8645014])
```

Early Stopping

A 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 popular technique is called *early stopping*. Figure 4-21 shows a complex model (in this case, a high-degree 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. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a "beautiful free lunch".[13] That said, the validation error sometimes comes back down after a while: this is called *double descent*. It's fairly common with large neural networks, and is an area of active research.

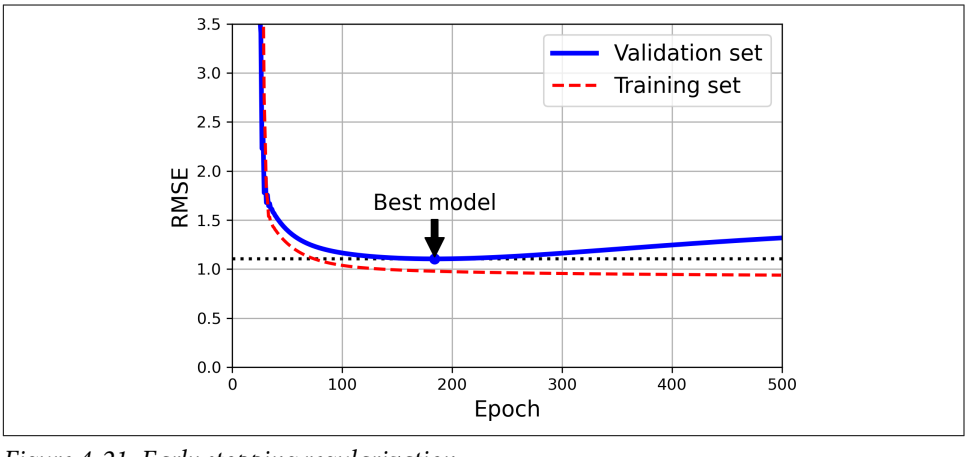

Figure 4-21. Early stopping regularization

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

13 Slide #63 of the NeurIPS 2015 Deep Learning Tutorial.

Here is a basic implementation of early stopping:

```
from copy import deepcopy
from sklearn.metrics import root_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 = root_mean_squared_error(y_valid, y_valid_predict)
    if val_error < best_valid_rmse:
        best_valid_rmse = val_error
        best_model = deepcopy(sgd_reg)
```

This code first adds the polynomial features and scales all the input features, both for the training set and for the validation set (the code assumes that you have split the original training set into a smaller training set and a 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. Note that 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.

Logistic Regression

As discussed in Chapter 1, 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 (e.g., what is the probability that this email is spam?). If the estimated probability is greater than a given threshold (typically 50%), then the model predicts that the instance belongs to that class (called the *positive class*, labeled "1"), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled "0"). This makes it a binary classifier.

Estimating Probabilities

So 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 (see Equation 4-14).

Equation 4-14. Logistic regression model estimated probability (vectorized form)

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

The logistic—denoted $\sigma(\cdot)$—is a *sigmoid function* (i.e., *S*-shaped) that outputs a number between 0 and 1. It is defined as shown in Equation 4-15 and Figure 4-22.

Equation 4-15. Logistic function

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

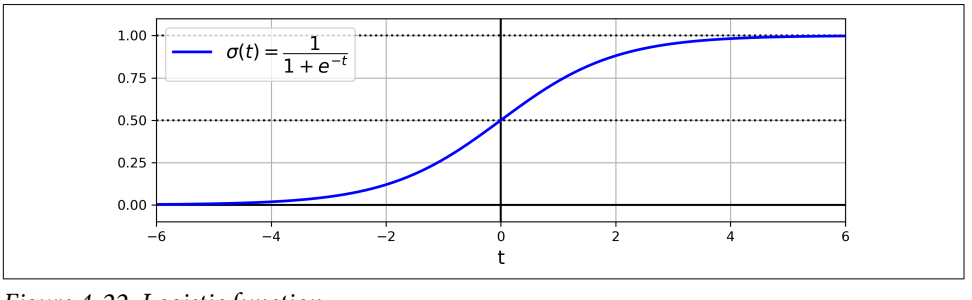

Figure 4-22. Logistic function

Once the logistic regression model has estimated the probability $\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x})$ that an instance $\mathbf{x}$ belongs to the positive class, it can make its prediction $\hat{y}$ easily (see Equation 4-16).

Equation 4-16. 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 $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a logistic regression model using the default threshold of 50% probability predicts 1 if $\boldsymbol{\theta}^\intercal \mathbf{x}$ is positive and 0 if it is negative.

The score t is often called the *logit*. The name comes from the fact that the logit function, defined as $\text{logit}(p) = \log(p / (1 - p))$, 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

Now you know how a logistic regression model estimates probabilities and makes predictions. But how is it trained? The objective of training is to set the parameter vector $\boldsymbol{\theta}$ 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 shown in Equation 4-17 for a single training instance $\mathbf{x}$.

Equation 4-17. 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*, shown in Equation 4-18.

Equation 4-18. Logistic regression cost function (log loss)

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

The log loss was not just pulled out of a hat. It 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.

The bad news is that there is no known closed-form equation to compute the value of $\boldsymbol{\theta}$ that minimizes this cost function (there is no equivalent of the normal equation). But the good news is that this cost function is convex, so gradient descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough). The partial derivatives of the cost function with regard to the j^{th} model parameter θ_j are given by Equation 4-19.

Equation 4-19. Logistic cost function partial derivatives

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

This equation looks very much like Equation 4-6: for each instance it computes the prediction error and multiplies it by the j^{th} feature value, and then it computes the average over all training instances. Once you have the gradient vector containing all the partial derivatives, you can use it in the batch gradient descent algorithm. That's it: you now know how to train a logistic regression model. 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

We can use the iris dataset to illustrate logistic regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: *Iris setosa*, *Iris versicolor*, and *Iris virginica* (see Figure 4-23).

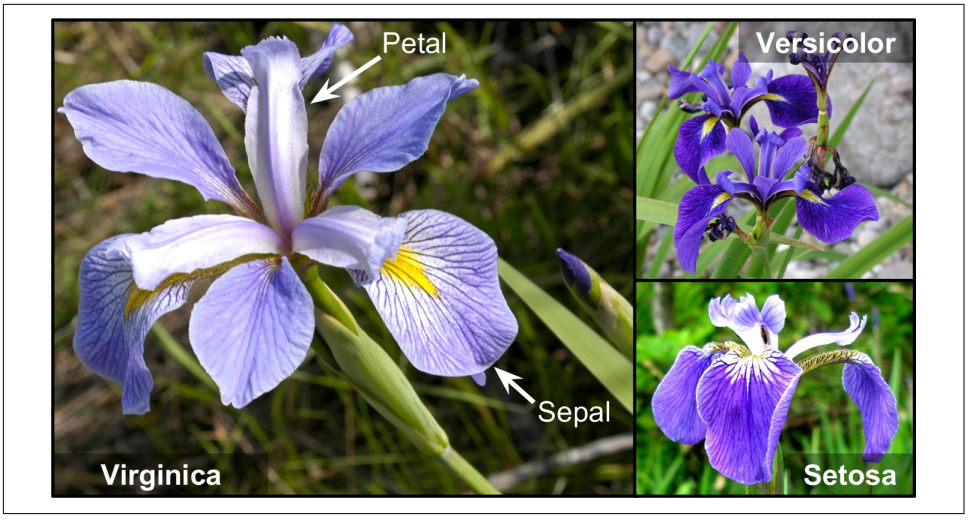

Figure 4-23. Flowers of three iris plant species[14]

Let's try to build a classifier to detect the *Iris virginica* 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
```

14 Photos reproduced from the corresponding Wikipedia pages. *Iris virginica* photo by Frank Mayfield (Creative Commons BY-SA 2.0), *Iris versicolor* photo by D. Gordon E. Robertson (Creative Commons BY-SA 3.0), *Iris setosa* photo public domain.

```
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)
```

Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 cm to 3 cm (Figure 4-24):[15]

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)  # reshape to get a column vector
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0, 0]

plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2,
         label="Not Iris virginica proba")
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica proba")
plt.plot([decision_boundary, decision_boundary], [0, 1], "k:", linewidth=2,
         label="Decision boundary")
[...] # beautify the figure: add grid, labels, axis, legend, arrows, and samples
plt.show()
```

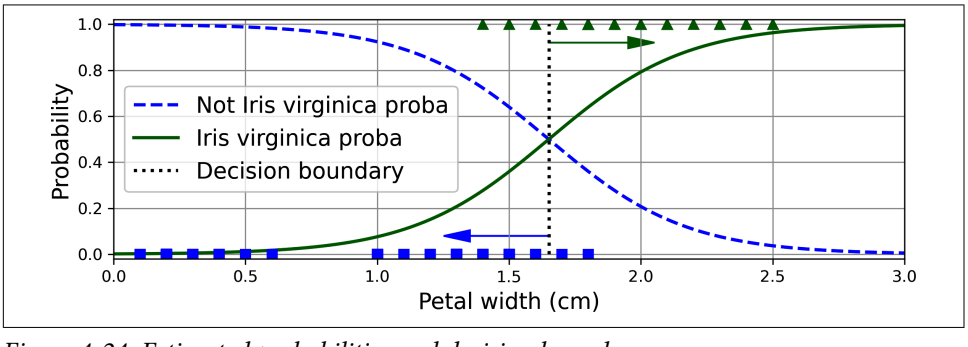

Figure 4-24. Estimated probabilities and decision boundary

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

15 NumPy's `reshape()` function allows one dimension to be –1, which means "automatic": the value is inferred from the length of the array and the remaining dimensions.

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
np.float64(1.6516516516516517)
>>> log_reg.predict([[1.7], [1.5]])
array([ True, False])
```

Figure 4-25 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.[16] 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 a 90% chance of being *Iris virginica*, according to the model.

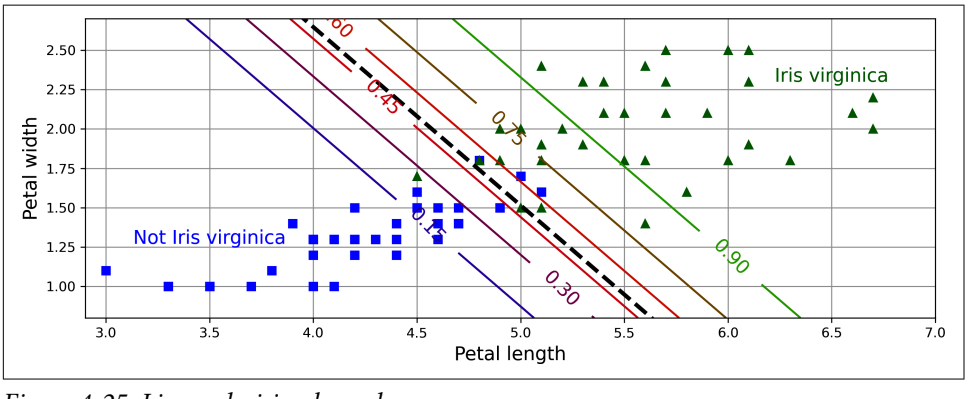

Figure 4-25. Linear decision boundary

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.

16 It is the set of points $\mathbf{x}$ such that $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, which defines a straight line.

Softmax Regression

The logistic regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers (as discussed in Chapter 3). This is called *softmax regression*, or *multinomial logistic regression*.

The idea is simple: when given an instance $\mathbf{x}$, the softmax regression model first computes a score $s_k(\mathbf{x})$ for each class k, then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores. The equation to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equation for linear regression prediction (see Equation 4-20).

Equation 4-20. Softmax score for class k

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

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* $\boldsymbol{\Theta}$.

Once you have computed the score of every class for the instance $\mathbf{x}$, you can estimate the probability $\hat{p}_k$ that the instance belongs to class k by running the scores through the softmax function (Equation 4-21). 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 4-21. Softmax function

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

In this equation:

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k, given the scores of each class for that instance.

Just like the logistic regression classifier, by default the softmax regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score), as shown in Equation 4-22.

Equation 4-22. 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 $\sigma(\mathbf{s}(\mathbf{x}))_k$.

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.

Now that you know how the model estimates probabilities and makes predictions, let's take a look at training. The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes). Minimizing the cost function shown in Equation 4-23, 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 4-23. 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, $y_k^{(i)}$ is the target probability that the i^{th} instance belongs to class k. In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.

Notice that when there are just two classes ($K = 2$), this cost function is equivalent to the logistic regression cost function (log loss; see Equation 4-18).

Cross Entropy

Cross entropy originated from Claude Shannon's *information theory*. Suppose you want to efficiently transmit information about the weather every day. If there are eight options (sunny, rainy, etc.), you could encode each option using 3 bits, because $2^3 = 8$. However, if you think it will be sunny almost every day, it would be much more efficient to code "sunny" on just one bit (0) and the other seven options on four bits (starting with a 1). Cross entropy measures the average number of bits you actually send per option. If your assumption about the weather is perfect, cross entropy will be equal to the entropy of the weather itself (i.e., its intrinsic unpredictability). But if

your assumption is wrong (e.g., if it rains often), cross entropy will be greater by an amount called the *Kullback–Leibler (KL) divergence*.

The cross entropy between two probability distributions p and q is defined as $H(p,q) = -\Sigma_x p(x) \log q(x)$ (at least when the distributions are discrete). For more details, check out my video on the subject.

The gradient vector of this cost function with regard to $\boldsymbol{\theta}^{(k)}$ is given by Equation 4-24.

Equation 4-24. Cross entropy gradient vector for class k

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

Now you can compute the gradient vector for every class, then use gradient descent (or any other optimization algorithm) to find the parameter matrix $\boldsymbol{\Theta}$ that minimizes the cost function.

Let's use softmax regression to classify the iris plants 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`: decrease `C` to increase regularization, as mentioned earlier.

```
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)
```

So the next time you find an iris with petals that are 5 cm long and 2 cm wide, you can ask your model to tell you what type of iris it is, and it will answer *Iris virginica* (class 2) with 96% probability (or *Iris versicolor* with 4% probability):

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

Figure 4-26 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%.

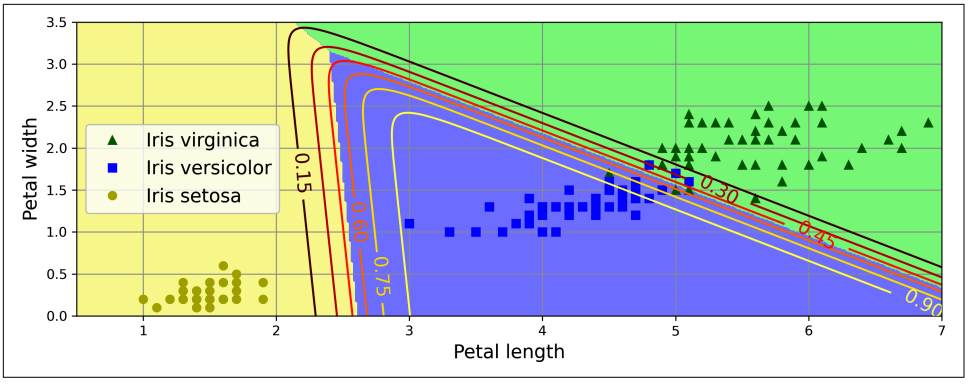

Figure 4-26. Softmax regression decision boundaries

In this chapter, you learned various ways to train linear models, both for regression and for classification. You used a closed-form equation to solve linear regression, as well as gradient descent, and you learned how various penalties can be added to the cost function during training to regularize the model. Along the way, you also learned how to plot learning curves and analyze them, and how to implement early stopping. Finally, you learned how logistic regression and softmax regression work. We've opened up the first machine learning black boxes! In the next chapters we will open many more, starting with support vector machines.

Exercises

1. Which linear regression training algorithm can you use if you have a training set with millions of features?
2. 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?
3. Can gradient descent get stuck in a local minimum when training a logistic regression model?
4. Do all gradient descent algorithms lead to the same model, provided you let them run long enough?
5. Suppose you use batch gradient descent and you plot the validation error at every epoch. If you notice that the validation error consistently goes up, what is likely going on? How can you fix this?
6. Is it a good idea to stop mini-batch gradient descent immediately when the validation error goes up?
7. Which gradient descent algorithm (among those we discussed) will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?

8. 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?
9. Suppose you are using ridge regression and you notice that the training error and the validation error are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter α or reduce it?
10. Why would you want to use:
    a. Ridge regression instead of plain linear regression (i.e., without any regularization)?
    b. Lasso instead of ridge regression?
    c. Elastic net instead of lasso regression?
11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two logistic regression classifiers or one softmax regression classifier?
12. Implement batch gradient descent with early stopping for softmax regression without using Scikit-Learn, only NumPy. Use it on a classification task such as the iris dataset.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

CHAPTER 5

Decision Trees

Decision trees are versatile machine learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are powerful algorithms, capable of fitting complex datasets. For example, in Chapter 2 you trained a `DecisionTreeRegressor` model on the California housing dataset, fitting it perfectly (actually, overfitting it).

Decision trees are also the fundamental components of random forests (see Chapter 6), which are among the most powerful machine learning algorithms available today.

In this chapter we will start by discussing how to train, visualize, and make predictions with decision trees. Then we will go through the CART training algorithm used by Scikit-Learn, and we will explore how to regularize trees and use them for regression tasks. Finally, we will discuss some of the limitations of decision trees.

Training and Visualizing a Decision Tree

To understand decision trees, let's build one and take a look at how it makes predictions. The following code trains a `DecisionTreeClassifier` on the iris dataset (see Chapter 4):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)
```

You can visualize the trained decision tree by first using the `export_graphviz()` function to output a graph definition file called *iris_tree.dot*:

```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file="iris_tree.dot",
        feature_names=["petal length (cm)", "petal width (cm)"],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

Then you can use `graphviz.Source.from_file()` to load and display the file in a Jupyter notebook:

```
from graphviz import Source

Source.from_file("iris_tree.dot")
```

Graphviz is an open source graph visualization software package. It also includes a `dot` command-line tool to convert *.dot* files to a variety of formats, such as PDF or PNG.

Your first decision tree looks like Figure 5-1.

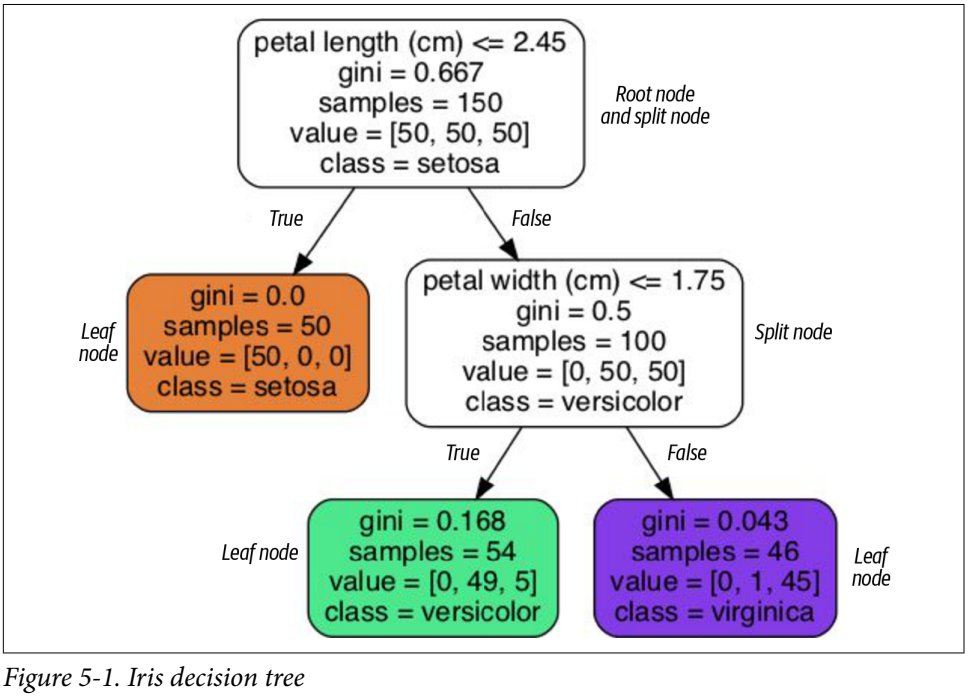

Figure 5-1. Iris decision tree

Making Predictions

Let's see how the tree represented in Figure 5-1 makes predictions. Suppose you find an iris flower and you want to classify it based on its petals. You start at the *root node* (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a *leaf node* (i.e., it does not have any child nodes), so it does not ask any questions: simply look at the predicted class for that node, and the decision tree predicts that your flower is an *Iris setosa* (`class=setosa`).

Now suppose you find another flower, and this time the petal length is greater than 2.45 cm. You again start at the root but now move down to its right child node (depth 1, right). This is not a leaf node, it's a *split node*, so it asks another question: is the petal width smaller than 1.75 cm? If it is, then your flower is most likely an *Iris versicolor* (depth 2, left). If not, it is likely an *Iris virginica* (depth 2, right). It's really that simple.

One of the many qualities of decision trees is that they require very little data preparation. In fact, they don't require feature scaling or centering at all.

A node's `samples` attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), and of those 100, 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's `value` attribute tells you how many training instances of each class this node applies to: for example, the bottom-right node applies to 0 *Iris setosa*, 1 *Iris versicolor*, and 45 *Iris virginica*. Finally, a node's `gini` attribute measures its *Gini impurity*: a node is "pure" (`gini=0`) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies only to *Iris setosa* training instances, its Gini impurity is 0. Conversely, the other nodes all apply to instances of multiple classes, so they are "impure". Equation 5-1 shows how the training algorithm computes the Gini impurity G_i of the i^{th} node. The more classes and the more mixed they are, the larger the impurity. For example, the depth-2 left node has a Gini impurity equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$.

Equation 5-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

In this equation:

- G_i is the Gini impurity of the i^{th} node.
- $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.

Scikit-Learn uses the CART algorithm (discussed shortly), which produces only *binary trees*, meaning trees where split nodes always have exactly two children (i.e., questions only have yes/no answers). However, other algorithms, such as ID3, can produce decision trees with nodes that have more than two children.

Figure 5-2 shows this decision tree's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm. Since the lefthand area is pure (only *Iris setosa*), it cannot be split any further. However, the righthand area is impure, so the depth-1 right node splits it at petal width = 1.75 cm (represented by the dashed line). Since `max_depth` was set to 2, the decision tree stops right there. If you set `max_depth` to 3, then the two depth-2 nodes would each add another decision boundary (represented by the two vertical dotted lines).

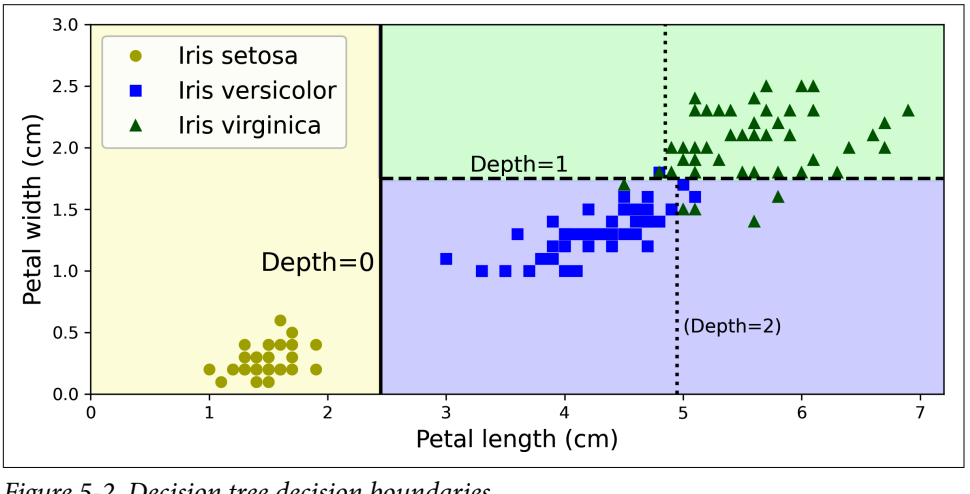

Figure 5-2. Decision tree decision boundaries

The tree structure, including all the information shown in Figure 5-1, is available via the classifier's `tree_` attribute. Type **`help(tree_clf.tree_)`** for details, and see this chapter's notebook for an example.

Model Interpretation: White Box Versus Black Box

Decision trees are intuitive, and their decisions are easy to interpret. Such models are often called *white box models*. In contrast, as you will see, random forests and neural networks are generally considered *black box models*. They make great predictions, and you can easily check the calculations that they performed to make these predictions; nevertheless, it is usually hard to explain in simple terms why the predictions were made. For example, if a neural network says that a particular person appears in a picture, it is hard to know what contributed to this prediction: Did the model recognize that person's eyes? Their mouth? Their nose? Their shoes? Or even the couch that they were sitting on? Conversely, decision trees provide nice, simple classification rules that can even be applied manually if need be (e.g., for flower classification). The field of *interpretable ML* aims at creating ML systems that can explain their decisions in a way humans can understand. This is important in many domains, for example in healthcare, to let a doctor review the diagnosis; in finance, to let analysts understand the risks; in a judicial system, to let a human make the final call; or in human resources, to ensure decisions aren't biased.

Estimating Class Probabilities

A decision tree can also estimate the probability that an instance belongs to a particular class *k*. First it traverses the tree to find the leaf node for this instance, and then it returns the proportion of instances of class *k* among the training instances that would also reach this leaf node. For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth-2 left node, so the decision tree outputs the following probabilities: 0% for *Iris setosa* (0/54), 90.7% for *Iris versicolor* (49/54), and 9.3% for *Iris virginica* (5/54). And if you ask it to predict the class, it outputs *Iris versicolor* (class 1) because it has the highest probability. Let's check this:

```
>>> tree_clf.predict_proba([[5, 1.5]]).round(3)
array([[0.   , 0.907, 0.093]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfect! Notice that the estimated probabilities would be identical anywhere else in the bottom-right rectangle of Figure 5-2—for example, if the petals were 6 cm long and 1.5 cm wide (even though it seems obvious that it would most likely be an *Iris virginica* in this case).

The CART Training Algorithm

Scikit-Learn uses the *Classification and Regression Tree* (CART) algorithm to train decision trees (also called "growing" trees). The algorithm works by first splitting

the training set into two subsets using a single feature k and a threshold t_k (e.g., "petal length ≤ 2.45 cm"). How does it choose k and t_k? It searches for the pair (k, t_k) that produces the purest subsets, weighted by their size. Equation 5-2 gives the cost function that the algorithm tries to minimize.

Equation 5-2. CART cost function for classification

$$J(k,t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where $\begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset} \\ m = m_{\text{left}} + m_{\text{right}} \end{cases}$

Once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the `max_depth` hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters (described in a moment) control additional stopping conditions: `min_samples_split`, `min_samples_leaf`, `max_leaf_nodes`, and more.

As you can see, the CART algorithm is a *greedy algorithm*: it greedily searches for an optimum split at the top level, then repeats the process at each subsequent level. It does not check whether the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a solution that's reasonably good but not guaranteed to be optimal.

Unfortunately, finding the optimal tree is known to be an *NP-complete* problem.[1] It requires $O(\exp(m))$ time,[2] making the problem intractable even for small training sets. This is why we must settle for a "reasonably good" solution when training decision trees.

1 P is the set of problems that can be solved in *polynomial time* (i.e., a polynomial of the dataset size). NP is the set of problems whose solutions can be verified in polynomial time. An NP-hard problem is a problem that can be reduced to a known NP-hard problem in polynomial time. An NP-complete problem is both NP and NP-hard. A major open mathematical question is whether P = NP. If P ≠ NP (which seems likely), then no polynomial algorithm will ever be found for any NP-complete problem (except perhaps one day on a quantum computer).

2 This *big O notation* means that as m (i.e., the number of training instances) gets larger, the computation time becomes proportional to the exponential of m (it's actually an upper bound, but we make it as small as we can). This tells us how "fast" the computation grows with m, and $O(\exp(m))$ is very fast.

Computational Complexity

Making predictions requires traversing the decision tree from the root to a leaf. Decision trees are generally approximately balanced, so traversing the decision tree requires going through roughly $O(\log_2(m))$ nodes, where m is the number of training instances, and $\log_2(m)$ is the *binary logarithm* of m, equal to $\log(m) / \log(2)$. Since each node only requires checking the value of one feature, the overall prediction complexity is $O(\log_2(m))$, independent of the number of features. So predictions are very fast, even when dealing with large training sets.

By default, the training algorithm compares all features on all samples at each node, which results in a training complexity of $O(n \times m \log_2(m))$.

It's possible to set a maximum tree depth using the `max_depth` hyperparameter, and/or set a maximum number of features to consider at each node (the features are then chosen randomly). Doing so will help speed up training considerably, and it can also reduce the risk of overfitting (but as always, going too far would result in underfitting).

Gini Impurity or Entropy?

By default, the `DecisionTreeClassifier` class uses the Gini impurity measure, but you can select the *entropy* impurity measure instead by setting the `criterion` hyperparameter to `"entropy"`. The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. Entropy later spread to a wide variety of domains, including in Shannon's information theory, where it measures the average information content of a message, as we saw in Chapter 4. Entropy is zero when all messages are identical. In machine learning, entropy is frequently used as an impurity measure: a set's entropy is zero when it contains instances of only one class. Equation 5-3 shows the definition of the entropy of the i^{th} node. For example, the depth-2 left node in Figure 5-1 has an entropy equal to $-(49/54) \log_2 (49/54) - (5/54) \log_2 (5/54) \approx 0.445$.

Equation 5-3. Entropy

$$H_i = - \sum_{\substack{k = 1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log_2 \left(p_{i,k}\right)$$

So, should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to

isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.[3]

Regularization Hyperparameters

Decision trees make very few assumptions about the training data (as opposed to linear models, which assume that the data is linear, for example). If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely—indeed, most likely overfitting it. Such a model is often called a *nonparametric model*, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a *parametric model*, such as a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

To avoid overfitting the training data, you need to restrict the decision tree's freedom during training. As you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the decision tree. In Scikit-Learn, this is controlled by the `max_depth` hyperparameter. The default value is `None`, which means unlimited. Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the decision tree:

`max_features`
: Maximum number of features that are evaluated for splitting at each node

`max_leaf_nodes`
: Maximum number of leaf nodes

`min_samples_split`
: Minimum number of samples a node must have before it can be split

`min_samples_leaf`
: Minimum number of samples a leaf node must have to be created

`min_weight_fraction_leaf`
: Same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances

`min_impurity_decrease`
: Only split a node if this split results in at least this reduction in impurity

3 See Sebastian Raschka's interesting analysis for more details.

`ccp_alpha`
: Controls *minimal cost-complexity pruning* (MCCP), i.e., pruning subtrees that don't reduce impurity enough compared to their number of leaves; a larger `ccp_alpha` value leads to more pruning, resulting in a smaller tree (the default is 0—no pruning)

To limit the model's complexity and thereby regularize the model, you can increase `min_*` hyperparameters or `ccp_alpha`, or decrease `max_*` hyperparameters. Tuning `max_depth` is usually a good default: it provides effective regularization, and it keeps the tree small and easy to interpret. Setting `min_samples_leaf` is also a good idea, especially for small datasets. And `max_features` is great when working with high-dimensional datasets.

Other algorithms work by first training the decision tree without restrictions, then *pruning* (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistical tests, such as the χ^2 *test* (chi-squared test), are used to estimate the probability that the improvement is purely the result of chance (which is called the *null hypothesis*). If this probability, called the *p-value*, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

Let's test regularization on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving crescent moons (see Figure 5-3). You can generate this dataset using the `make_moons()` function.

We'll train one decision tree without regularization, and another with `min_samples_leaf=5`. Here's the code; Figure 5-3 shows the decision boundaries of each tree:

```
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=150, noise=0.2, random_state=42)

tree_clf1 = DecisionTreeClassifier(random_state=42)
tree_clf2 = DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
tree_clf1.fit(X_moons, y_moons)
tree_clf2.fit(X_moons, y_moons)
```

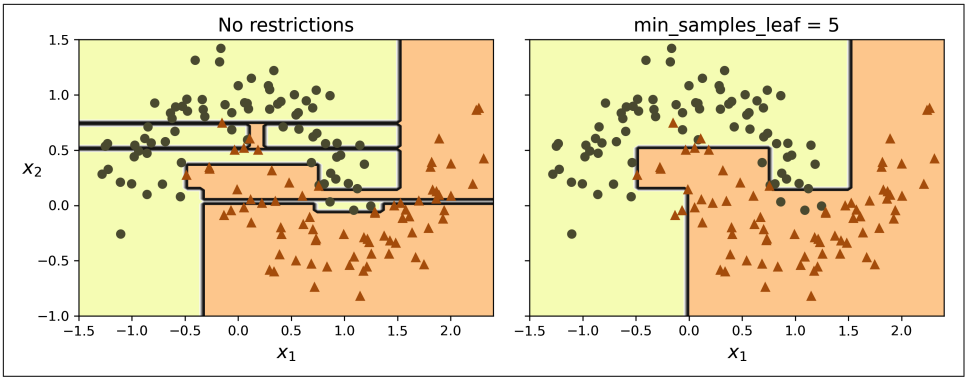

Figure 5-3. Decision boundaries of an unregularized tree (left) and a regularized tree (right)

The unregularized model on the left is clearly overfitting, and the regularized model on the right will probably generalize better. We can verify this by evaluating both trees on a test set generated using a different random seed:

```
>>> X_moons_test, y_moons_test = make_moons(n_samples=1000, noise=0.2,
...                                         random_state=43)
...
>>> tree_clf1.score(X_moons_test, y_moons_test)
0.898
>>> tree_clf2.score(X_moons_test, y_moons_test)
0.92
```

Indeed, the second tree has a better accuracy on the test set.

Regression

Decision trees are also capable of performing regression tasks. While linear regression only works well with linear data, decision trees can fit all sorts of complex datasets. Let's build a regression tree using Scikit-Learn's `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with `max_depth=2`:

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

rng = np.random.default_rng(seed=42)
X_quad = rng.random((200, 1)) - 0.5  # a single random input feature
y_quad = X_quad ** 2 + 0.025 * rng.standard_normal((200, 1))

tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg.fit(X_quad, y_quad)
```

The resulting tree is represented in Figure 5-4.

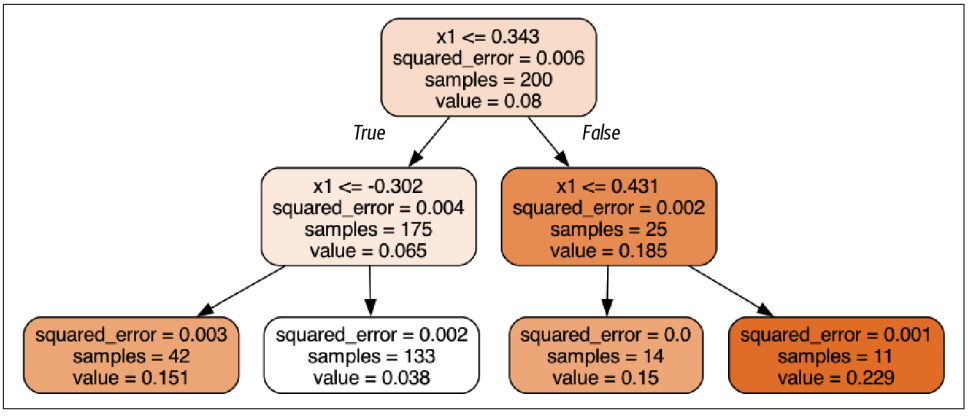

Figure 5-4. A decision tree for regression

This tree looks very similar to the classification tree you built earlier. The main difference is that instead of predicting a class in each node, it predicts a value. For example, suppose you want to make a prediction for a new instance with $x_1 = 0.2$. The root node asks whether $x_1 \leq 0.343$. Since it is, the algorithm goes to the left child node, which asks whether $x_1 \leq -0.302$. Since it is not, the algorithm goes to the right child node. This is a leaf node, and it predicts `value=0.038`. This prediction is the average target value of the 133 training instances associated with this leaf node, and it results in a mean squared error equal to 0.002 over these 133 instances.

This model's predictions are represented on the left in Figure 5-5. If you set `max_depth=3`, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.

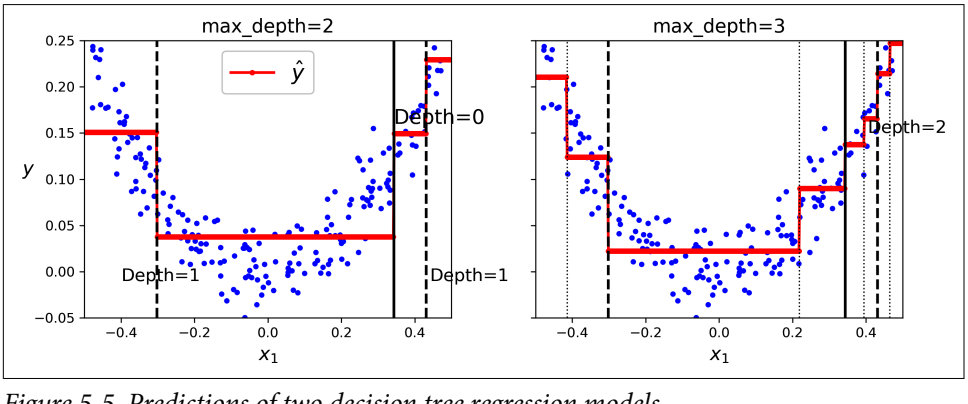

Figure 5-5. Predictions of two decision tree regression models

The CART algorithm works as described earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. Equation 5-4 shows the cost function that the algorithm tries to minimize.

Equation 5-4. CART cost function for regression

$$J(k,t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \quad \text{where} \begin{cases} \text{MSE}_{\text{node}} = \dfrac{\sum_{i \in \text{node}} \left(\widehat{y}_{\text{node}} - y^{(i)}\right)^2}{m_{\text{node}}} \\ \widehat{y}_{\text{node}} = \dfrac{\sum_{i \in \text{node}} y^{(i)}}{m_{\text{node}}} \end{cases}$$

Just like for classification tasks, decision trees are prone to overfitting when dealing with regression tasks. Without any regularization (i.e., using the default hyperparameters), you get the predictions on the left in Figure 5-6. These predictions are obviously overfitting the training set very badly. Just setting `min_samples_leaf=10` results in a much more reasonable model, represented on the right in Figure 5-6.

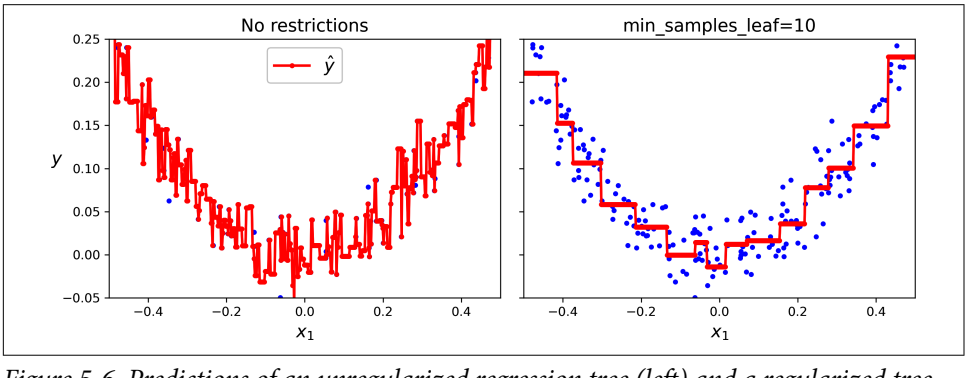

Figure 5-6. Predictions of an unregularized regression tree (left) and a regularized tree (right)

Sensitivity to Axis Orientation

Hopefully by now you are convinced that decision trees have a lot going for them: they are relatively easy to understand and interpret, simple to use, versatile, and powerful. However, they do have a few limitations. First, as you may have noticed, decision trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to the data's orientation. For example, Figure 5-7 shows a simple linearly separable dataset: on the left, a decision tree can split it easily, while on the right, after the dataset is rotated by 45°, the decision boundary looks

unnecessarily convoluted. Although both decision trees fit the training set perfectly, it is very likely that the model on the right will not generalize well.

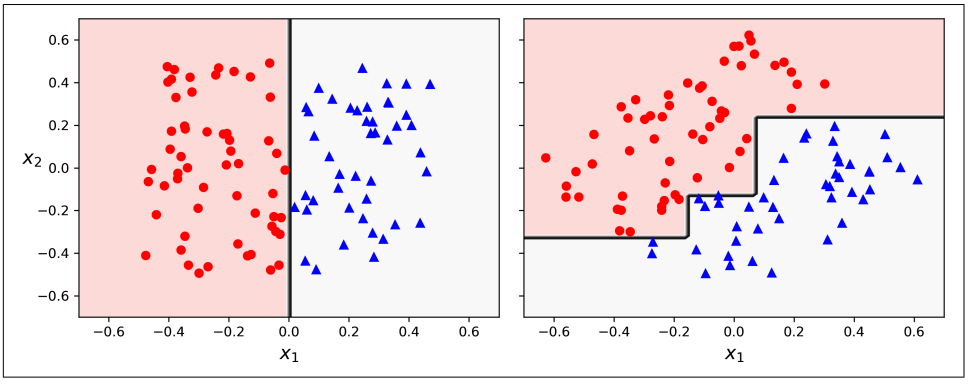

Figure 5-7. Sensitivity to training set rotation

One way to limit this problem is to scale the data, then apply a principal component analysis (PCA) transformation. We will look at PCA in detail in Chapter 7, but for now you only need to know that it rotates the data in a way that reduces the correlation between the features, which often (not always) makes things easier for trees.

Let's create a small pipeline that scales the data and rotates it using PCA, then train a `DecisionTreeClassifier` on that data. Figure 5-8 shows the decision boundaries of that tree: as you can see, the rotation makes it possible to fit the dataset pretty well using only one feature, z_1, which is a linear function of the original petal length and width. Here's the code:

```
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

pca_pipeline = make_pipeline(StandardScaler(), PCA())
X_iris_rotated = pca_pipeline.fit_transform(X_iris)
tree_clf_pca = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf_pca.fit(X_iris_rotated, y_iris)
```

The `DecisionTreeClassifier` and `DecisionTreeRegressor` classes both support missing values natively, no need for an imputer.

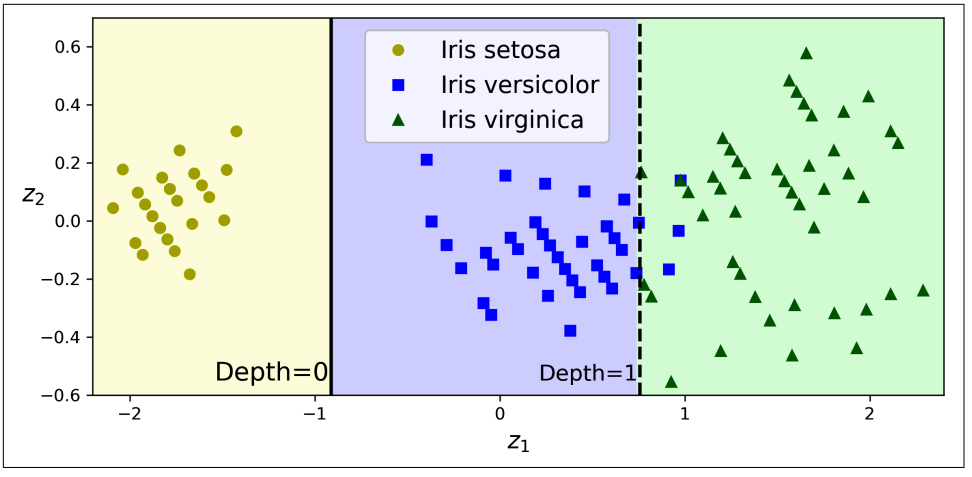

Figure 5-8. A tree's decision boundaries on the scaled and PCA-rotated iris dataset

Decision Trees Have a High Variance

More generally, the main issue with decision trees is that they have quite a high variance: small changes to the hyperparameters or to the data may produce very different models. In fact, since the training algorithm used by Scikit-Learn is stochastic—it randomly selects the set of features to evaluate at each node—even retraining the same decision tree on the exact same data may produce a very different model, such as the one represented in Figure 5-9 (unless you set the `random_state` hyperparameter). As you can see, it looks very different from the previous decision tree (Figure 5-2).

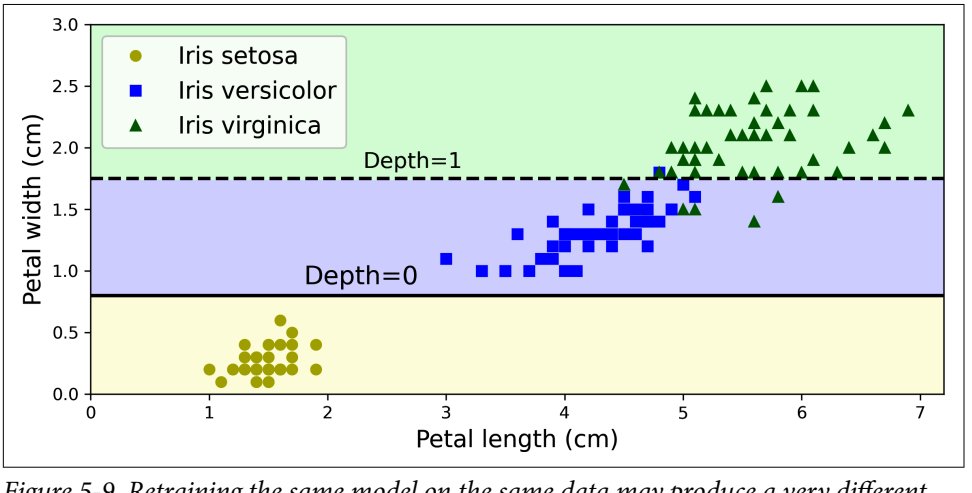

Figure 5-9. Retraining the same model on the same data may produce a very different model

Luckily, by averaging predictions over many trees, it's possible to reduce variance significantly. Such an *ensemble* of trees is called a *random forest*, and it's one of the most powerful types of models available today, as you will see in the next chapter.

Exercises

1. What is the approximate depth of a decision tree trained (without restrictions) on a training set with one million instances?
2. Is a node's Gini impurity generally lower or higher than its parent's? Is it *generally* lower/higher, or *always* lower/higher?
3. If a decision tree is overfitting the training set, is it a good idea to try decreasing `max_depth`?
4. If a decision tree is underfitting the training set, is it a good idea to try scaling the input features?
5. If it takes one hour to train a decision tree on a training set containing one million instances, roughly how much time will it take to train another decision tree on a training set containing ten million instances? Hint: consider the CART algorithm's computational complexity.
6. If it takes one hour to train a decision tree on a given training set, roughly how much time will it take if you double the number of features?
7. Train and fine-tune a decision tree for the moons dataset by following these steps:
   a. Use `make_moons(n_samples=10000, noise=0.4)` to generate a moons dataset.
   b. Use `train_test_split()` to split the dataset into a training set and a test set.
   c. Use grid search with cross-validation (with the help of the `GridSearchCV` class) to find good hyperparameter values for a `DecisionTreeClassifier`. Hint: try various values for `max_leaf_nodes`.
   d. Train it on the full training set using these hyperparameters, and measure your model's performance on the test set. You should get roughly 85% to 87% accuracy.
8. Grow a forest by following these steps:
   a. Continuing the previous exercise, generate 1,000 subsets of the training set, each containing 100 instances selected randomly. Hint: you can use Scikit-Learn's `ShuffleSplit` class for this.
   b. Train one decision tree on each subset, using the best hyperparameter values found in the previous exercise. Evaluate these 1,000 decision trees on the test set. Since they were trained on smaller sets, these decision trees will likely perform worse than the first decision tree, achieving only about 80% accuracy.

c. Now comes the magic. For each test set instance, generate the predictions of the 1,000 decision trees, and keep only the most frequent prediction (you can use SciPy's `mode()` function for this). This approach gives you *majority-vote predictions* over the test set.

d. Evaluate these predictions on the test set: you should obtain a slightly higher accuracy than your first model (about 0.5 to 1.5% higher). Congratulations, you have trained a random forest classifier!

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

CHAPTER 6

Ensemble Learning and Random Forests

Suppose you pose a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer. This is called the *wisdom of the crowd*. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *ensemble learning*, and an ensemble learning algorithm is called an *ensemble method*.

As an example of an ensemble method, you can train a group of decision tree classifiers, each on a different random subset of the training set. You can then obtain the predictions of all the individual trees, and the class that gets the most votes is the ensemble's prediction (see the last exercise in Chapter 5). Such an ensemble of decision trees is called a *random forest*, and despite its simplicity, this is one of the most powerful machine learning algorithms available today.

As discussed in Chapter 2, you will often use ensemble methods near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor. In fact, the winning solutions in machine learning competitions often involve several ensemble methods—most famously in the Netflix Prize competition. There are some downsides, however: ensemble learning requires much more computing resources than using a single model (both for training and for inference), it can be more complex to deploy and manage, and the predictions are harder to interpret. But the pros often outweigh the cons.

In this chapter we will examine the most popular ensemble methods, including voting classifiers, bagging and pasting ensembles, random forests, boosting, and stacking ensembles.

Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a logistic regression classifier, an SVM classifier, a random forest classifier, a *k*-nearest neighbors classifier, and perhaps a few more (see Figure 6-1).

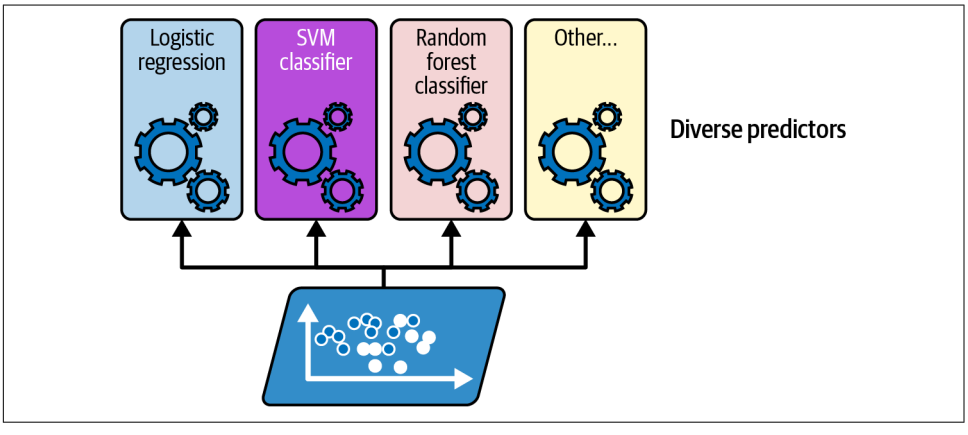

Figure 6-1. Training diverse classifiers

A very simple way to create an even better classifier is to aggregate the predictions of each classifier: the class that gets the most votes is the ensemble's prediction. This majority-vote classifier is called a *hard voting* classifier (see Figure 6-2).

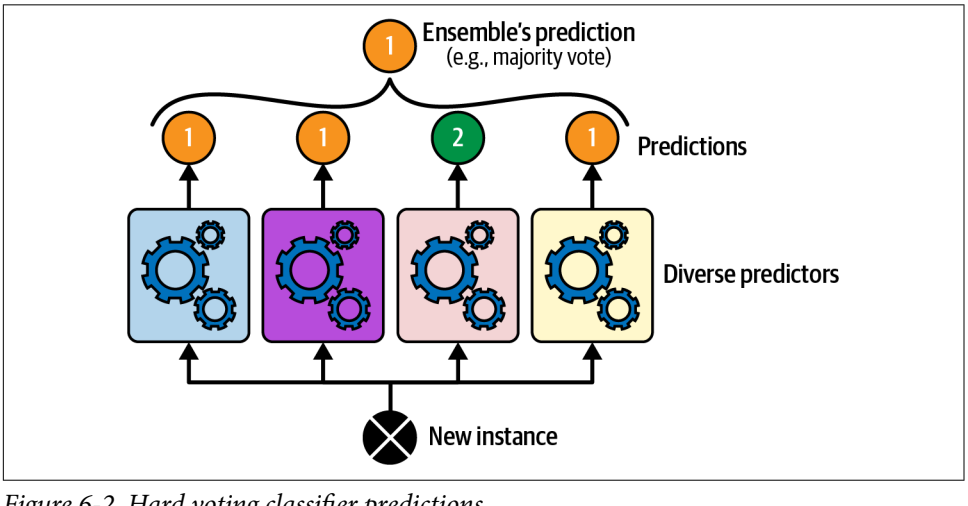

Figure 6-2. Hard voting classifier predictions

Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners in the ensemble and they are sufficiently diverse (i.e., if they focus on different aspects of the data and make different kinds of errors).

How is this possible? The following analogy can help shed some light on this mystery. Suppose you have a slightly biased coin that has a 51% chance of coming up heads and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the *law of large numbers*: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%). Figure 6-3 shows 10 series of biased coin tosses. You can see that as the number of tosses increases, the ratio of heads approaches 51%. Eventually all 10 series end up so close to 51% that they are consistently above 50%.

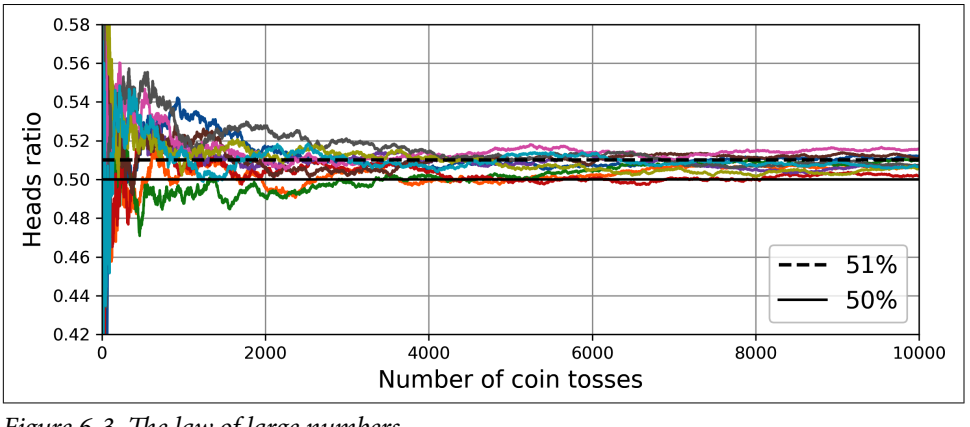

Figure 6-3. The law of large numbers

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy. You can also play with the model hyperparameters to get diverse models, or train the models on different subsets of the data, as we will see.

Scikit-Learn provides a `VotingClassifier` class that's quite easy to use: just give it a list of name/predictor pairs, and use it like a normal classifier. Let's try it on the moons dataset (introduced in Chapter 5). We will load and split the moons dataset into a training set and a test set, then we'll create and train a voting classifier composed of three diverse classifiers:

```
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

When you fit a `VotingClassifier`, it clones every estimator and fits the clones. The original estimators are available via the `estimators` attribute, while the fitted clones are available via the `estimators_` attribute. If you prefer a dict rather than a list, you can use `named_estimators` or `named_estimators_` instead. To begin, let's look at each fitted classifier's accuracy on the test set:

```
>>> for name, clf in voting_clf.named_estimators_.items():
...     print(name, "=", clf.score(X_test, y_test))
...
lr = 0.864
rf = 0.896
svc = 0.896
```

When you call the voting classifier's `predict()` method, it performs hard voting. For example, the voting classifier predicts class 1 for the first instance of the test set, because two out of three classifiers predict that class:

```
>>> voting_clf.predict(X_test[:1])
array([1])
>>> [clf.predict(X_test[:1]) for clf in voting_clf.estimators_]
[array([1]), array([1]), array([0])]
```

Now let's look at the performance of the voting classifier on the test set:

```
>>> voting_clf.score(X_test, y_test)
0.912
```

There you have it! The voting classifier outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities (i.e., if they all have a `predict_proba()` method), then you should generally tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*. It often achieves higher performance than hard voting because it gives more weight to highly confident votes. All you need to do is set the voting classifier's `voting` hyperparameter to `"soft"`, and ensure that all classifiers can estimate class probabilities. This is not the case for the `SVC` class by default, so you need to set its `probability` hyperparameter to `True` (this will make the `SVC` class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method). Let's try that:

```
>>> voting_clf.voting = "soft"
>>> voting_clf.named_estimators["svc"].probability = True
>>> voting_clf.fit(X_train, y_train)
>>> voting_clf.score(X_test, y_test)
0.92
```

We reach 92% accuracy simply by using soft voting—not bad!

Soft voting works best when the estimated probabilities are well-calibrated. If they are not, you can use `sklearn.calibration.CalibratedClassifierCV` to calibrate them (see Chapter 3).

Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another way is to use the same training algorithm for every predictor but train them on different random subsets of the training set. When sampling

is performed *with* replacement,[1] this method is called *bagging*[2] (short for *bootstrap aggregating*[3]). When sampling is performed *without* replacement, it is called *pasting*.[4]

In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in Figure 6-4.

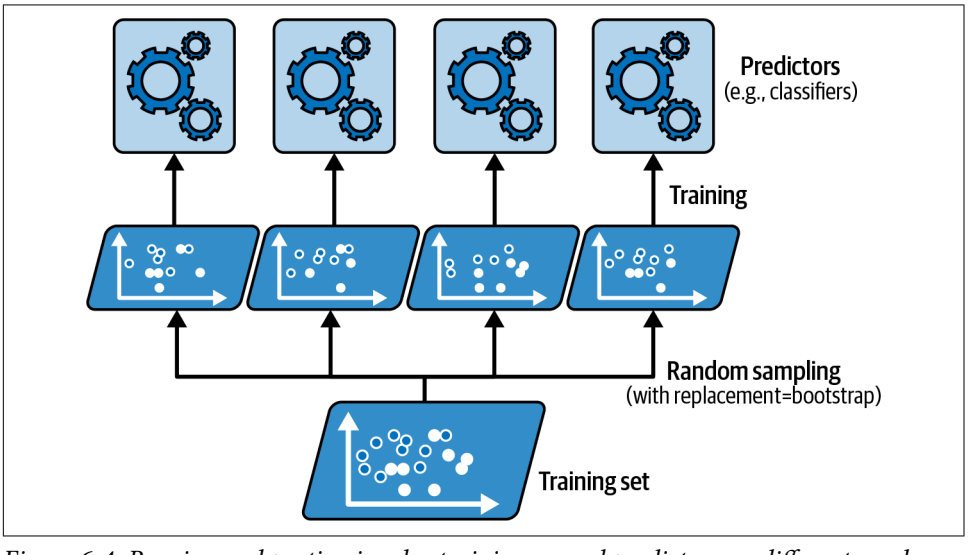

Figure 6-4. Bagging and pasting involve training several predictors on different random samples of the training set

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. For classification, the aggregation function is typically the *statistical mode* (i.e., the most frequent prediction, just like with a hard voting classifier), and for regression it's usually just the average. Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.[5]

1 Imagine picking a card randomly from a deck of cards, writing it down, then placing it back in the deck before picking the next card: the same card could be sampled multiple times.

2 Leo Breiman, "Bagging Predictors", *Machine Learning* 24, no. 2 (1996): 123–140.

3 In statistics, resampling with replacement is called *bootstrapping*.

4 Leo Breiman, "Pasting Small Votes for Classification in Large Databases and On-Line", *Machine Learning* 36, no. 1–2 (1999): 85–103.

5 Bias and variance were introduced in Chapter 4. Recall that a high bias means that the average prediction is far off target, while a high variance means that the predictions are very scattered. We want both to be low.

To get an intuition of why this is the case, imagine that you trained two regressors to predict house prices. The first underestimates the prices by $40,000 on average, while the second overestimates them by $50,000 on average. Assuming these regressors are 100% independent and their predictions follow a normal distribution, if you compute the average of the two predictions, the result will overestimate the prices by only (–40,000 + 50,000)/2 = $5,000 on average: that's a much lower bias! Similarly, if both predictors have a $10,000 standard deviation (i.e., a variance of 100,000,000), then the average prediction will have a variance of $(10{,}000^2 + 10{,}000^2)/2^2 = 50{,}000{,}000$ (i.e., the standard deviation will be $7,071). The variance is halved!

In practice, the ensemble often ends up with a similar bias but a lower variance than a single predictor trained on the original training set. Therefore it works best with high-variance and low-bias models (e.g., ensembles of decision trees, not ensembles of linear regressors).

Prefer bagging when your dataset is noisy or your model is prone to overfitting (e.g., deep decision tree). Otherwise, prefer pasting as it avoids redundancy during training, making it a bit more computationally efficient.

As you can see in Figure 6-4, predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons bagging and pasting are such popular methods: they scale very well.

Bagging and Pasting in Scikit-Learn

Scikit-Learn offers a simple API for both bagging and pasting: the `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 decision tree classifiers:[6] each is trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions, and `-1` tells Scikit-Learn to use all available cores:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

6 `max_samples` can alternatively be set to a float between 0.0 and 1.0, in which case the max number of sampled instances is equal to the size of the training set times `max_samples`.

A `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with decision tree classifiers.

Figure 6-5 compares the decision boundary of a single decision tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset. As you can see, the ensemble's predictions will likely generalize much better than the single decision tree's predictions: the ensemble has a comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

Bagging introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting; but the extra diversity also means that the predictors end up being less correlated, so the ensemble's variance is reduced. Overall, bagging often results in better models, which explains why it's generally preferred. But if you have spare time and CPU power, you can use cross-validation to evaluate both bagging and pasting, and select the one that works best.

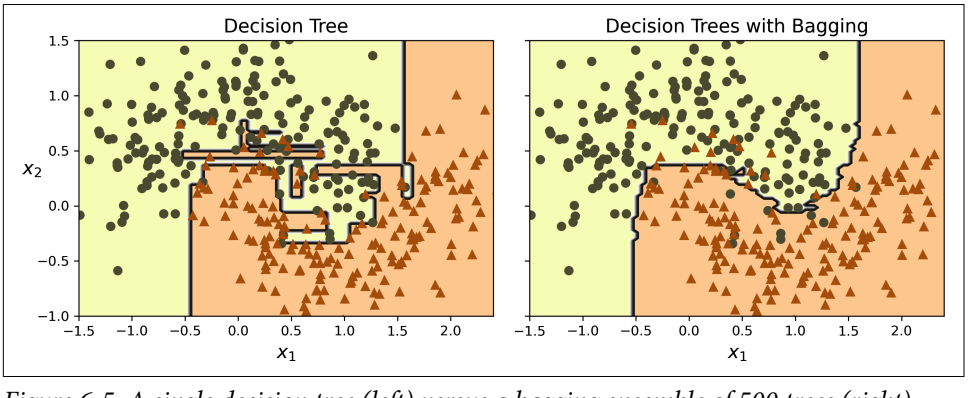

Figure 6-5. A single decision tree (left) versus a bagging ensemble of 500 trees (right)

Out-of-Bag Evaluation

With bagging, some training instances may be sampled several times for any given predictor, while others may not be sampled at all. By default, a `BaggingClassifier` samples m training instances with replacement (`bootstrap=True`), where m is the size of the training set. With this process, it can be shown mathematically that only about 63% of the training instances are sampled on average for each predictor.[7] The

7 As m grows, this ratio approaches $1 - \exp(-1) \approx 63\%$.

remaining 37% of the training instances that are not sampled are called *out-of-bag* (OOB) instances. Note that they are not the same 37% for all predictors.

A bagging ensemble can be evaluated using OOB instances, without the need for a separate validation set: indeed, if there are enough estimators, then each instance in the training set will likely be an OOB instance of several estimators, so these estimators can be used to make a fair ensemble prediction for that instance. Once you have a prediction for each instance, you can compute the ensemble's prediction accuracy (or any other metric).

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic OOB evaluation after training. The following code demonstrates this. The resulting evaluation score is available in the `oob_score_` attribute:

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
...                             oob_score=True, n_jobs=-1, random_state=42)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.896
```

According to this OOB evaluation, this `BaggingClassifier` is likely to achieve about 89.6% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.92
```

We get 92.0% accuracy on the test. The OOB evaluation was a bit too pessimistic.

The OOB decision function for each training instance is also available through the `oob_decision_function_` attribute. Since the base estimator has a `predict_proba()` method, the decision function returns the class probabilities for each training instance. For example, the OOB evaluation estimates that the first training instance has a 67.6% probability of belonging to the positive class and a 32.4% probability of belonging to the negative class:

```
>>> bag_clf.oob_decision_function_[:3]  # probas for the first 3 instances
array([[0.32352941, 0.67647059],
       [0.3375    , 0.6625    ],
       [1.        , 0.        ]])
```

Random Patches and Random Subspaces

The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two hyperparameters: `max_features` and `bootstrap_features`. They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead

of instance sampling. Thus, each predictor will be trained on a random subset of the input features.

This technique is particularly useful when you are dealing with high-dimensional inputs (such as images), as it can considerably speed up training. Sampling both training instances and features is called the *random patches* method.[8] Keeping all training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features (by setting `bootstrap_features` to `True` and/or `max_features` to a value smaller than `1.0`) is called the *random subspaces* method.[9]

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forests

As we have discussed, a random forest[10] is an ensemble of decision trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can use the `RandomForestClassifier` class, which is more convenient and optimized for decision trees[11] (similarly, there is a `RandomForestRegressor` class for regression tasks). The following code trains a random forest classifier with 500 trees, each limited to maximum 16 leaf nodes, using all available CPU cores:

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                 n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.

8 Gilles Louppe and Pierre Geurts, "Ensembles on Random Patches", *Lecture Notes in Computer Science* 7523 (2012): 346–361.

9 Tin Kam Ho, "The Random Subspace Method for Constructing Decision Forests", *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, no. 8 (1998): 832–844.

10 Tin Kam Ho, "Random Decision Forests", *Proceedings of the Third International Conference on Document Analysis and Recognition* 1 (1995): 278.

11 The `BaggingClassifier` class remains useful if you want a bag of something other than decision trees.

The `RandomForestClassifier` class introduces extra randomness when growing trees: instead of searching for the very best feature when splitting a node (see Chapter 5), it searches for the best feature among a random subset of features. By default, it samples $\sqrt{n}$ features (where n is the total number of features). The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model. So, the following `BaggingClassifier` is equivalent to the previous `RandomForestClassifier`:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),
    n_estimators=500, n_jobs=-1, random_state=42)
```

Extra-Trees

When you are growing a tree in a random forest, at each node only a random subset of the features is considered for splitting (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular decision trees do). For this, simply set `splitter="random"` when creating a `DecisionTreeClassifier`.

A forest of such extremely random trees is called an *extremely randomized trees*[12] (or *extra-trees* for short) ensemble. Once again, this technique trades more bias for a lower variance, so they may perform better than regular random forests if you encounter overfitting, especially with noisy and/or high-dimensional datasets. Extra-trees classifiers are also much faster to train than regular random forests, because finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree in a random forest.

You can create an extra-trees classifier using Scikit-Learn's `ExtraTreesClassifier` class. Its API is identical to the `RandomForestClassifier` class, except `bootstrap` defaults to `False`. Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class, except `bootstrap` defaults to `False`.

Just like decision tree classes, random forest classes and extra-trees classes in recent Scikit-Learn versions support missing values natively, no need for an imputer.

12 Pierre Geurts et al., "Extremely Randomized Trees", *Machine Learning* 63, no. 1 (2006): 3–42.

Feature Importance

Yet another great quality of random forests is that they make it easy to measure the relative importance of each feature. Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average, across all trees in the forest. More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with it (see Chapter 5).

Scikit-Learn computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1. You can access the result using the `feature_importances_` variable. For example, the following code trains a `RandomForestClassifier` on the iris dataset (introduced in Chapter 4) and outputs each feature's importance. It seems that the most important features are the petal length (44%) and width (42%), while sepal length and width are rather unimportant in comparison (11% and 2%, respectively):

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
>>> rnd_clf.fit(iris.data, iris.target)
>>> for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
...     print(round(score, 2), name)
...
0.11 sepal length (cm)
0.02 sepal width (cm)
0.44 petal length (cm)
0.42 petal width (cm)
```

Similarly, if you train a random forest classifier on the MNIST dataset (introduced in Chapter 3) and plot each pixel's importance, you get the image represented in Figure 6-6.

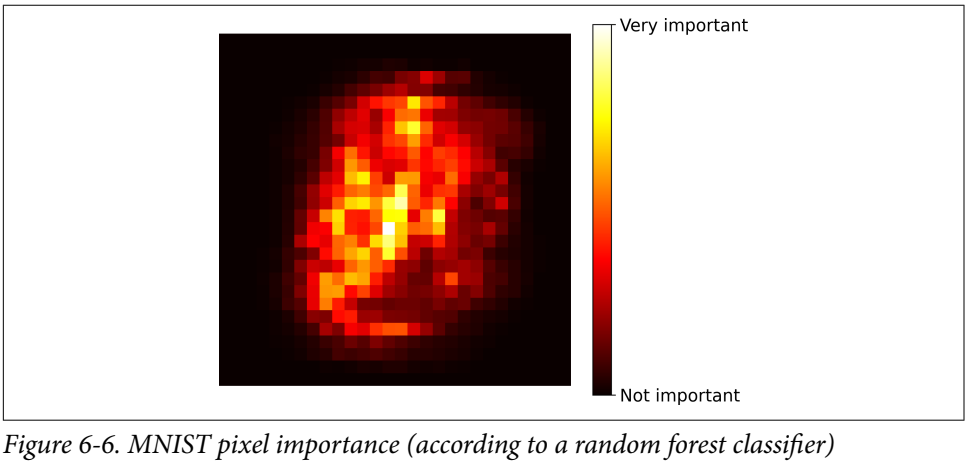

Figure 6-6. MNIST pixel importance (according to a random forest classifier)

Random forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

Boosting

Boosting (originally called *hypothesis boosting*) refers to any ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular are *AdaBoost*[13] (short for *adaptive boosting*) and *gradient boosting*. Let's start with AdaBoost.

AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfit. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, when training an AdaBoost classifier, the algorithm first trains a base classifier (such as a decision tree) and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on (see Figure 6-7).

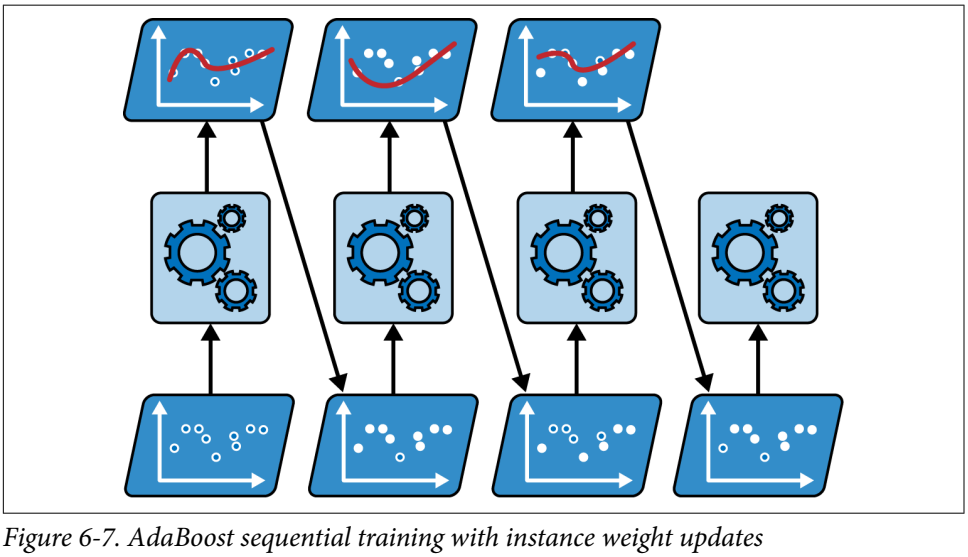

Figure 6-7. AdaBoost sequential training with instance weight updates

13 Yoav Freund and Robert E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting", *Journal of Computer and System Sciences* 55, no. 1 (1997): 119–139.

Figure 6-8 shows the decision boundaries of five consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel).[14] The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does a better job on these instances, and so on. The plot on the right represents the same sequence of predictors, except that the learning rate is halved (i.e., the misclassified instance weights are boosted much less at every iteration). As you can see, this sequential learning technique has some similarities with gradient descent, except that instead of tweaking a single predictor's parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.

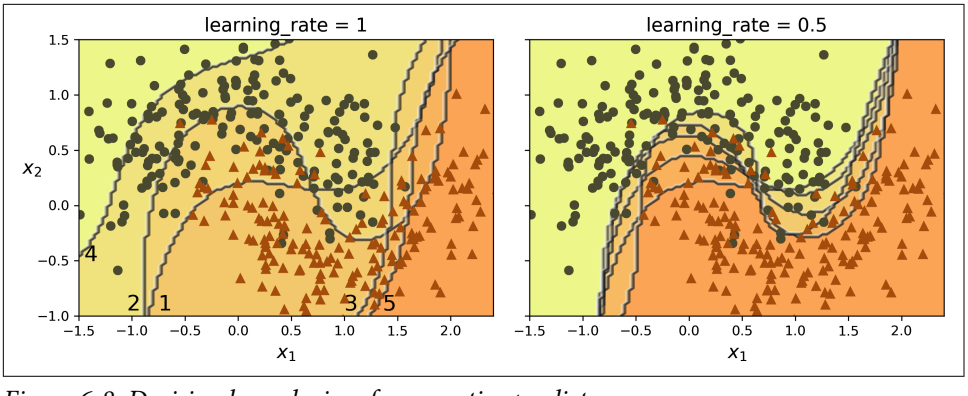

Figure 6-8. Decision boundaries of consecutive predictors

There is one important drawback to this sequential learning technique: training cannot be parallelized since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.

Let's take a closer look at the AdaBoost algorithm. Each instance weight $w^{(i)}$ is initially set to $1/m$, so their sum is 1. A first predictor is trained, and its weighted error rate r_1 is computed on the training set; see Equation 6-1.

14 This is just for illustrative purposes. SVMs are generally not good base predictors for AdaBoost; they are slow and tend to be unstable with it.

Equation 6-1. Weighted error rate of the j^{th} predictor

$$r_j = \sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance}$$

The predictor's weight α_j is then computed using Equation 6-2, where η is the learning rate hyperparameter (defaults to 1).[15] The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

Equation 6-2. Predictor weight

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Next, the AdaBoost algorithm updates the instance weights, using Equation 6-3, which boosts the weights of the misclassified instances and encourages the next predictor to pay more attention to them.

Equation 6-3. Weight update rule

$$\text{for } i = 1,2,\ldots,m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}^{(i)} \neq y^{(i)} \end{cases}$$

Then all the instance weights are normalized to ensure that their sum is once again 1 (i.e., they are divided by $\sum_{i=1}^{m} w^{(i)}$).

Finally, a new predictor is trained using the updated weights, and the whole process is repeated: the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on. The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j. The predicted class is the one that receives the majority of weighted votes (see Equation 6-4).

15 The original AdaBoost algorithm does not use a learning rate hyperparameter.

Equation 6-4. AdaBoost predictions

$$\widehat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \widehat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors}$$

Scikit-Learn uses a multiclass version of AdaBoost called *SAMME*[16] (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*). When there are just two classes, SAMME is equivalent to AdaBoost.

The following code trains an AdaBoost classifier based on 30 *decision stumps* using Scikit-Learn's `AdaBoostClassifier` class (as you might expect, there is also an `AdaBoostRegressor` class). A decision stump is a decision tree with `max_depth=1`—in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=30,
    learning_rate=0.5, random_state=42, algorithm="SAMME")
ada_clf.fit(X_train, y_train)
```

If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

Gradient Boosting

Another very popular boosting algorithm is *gradient boosting*.[17] Just like AdaBoost, gradient boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Let's go through a simple regression example, using decision trees as the base predictors; this is called *gradient tree boosting*, or *gradient boosted regression trees* (GBRT). First, let's generate a noisy quadratic dataset and fit a `DecisionTreeRegressor` to it:

16 For more details, see Ji Zhu et al., "Multi-Class AdaBoost", *Statistics and Its Interface* 2, no. 3 (2009): 349–360.

17 Gradient boosting was first introduced in Leo Breiman's 1997 paper "Arcing the Edge" and was further developed in the 1999 paper "Greedy Function Approximation: A Gradient Boosting Machine" by Jerome H. Friedman.

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

m = 100  # number of instances
rng = np.random.default_rng(seed=42)
X = rng.random((m, 1)) - 0.5
noise = 0.05 * rng.standard_normal(m)
y = 3 * X[:, 0] ** 2 + noise  # y = 3x² + Gaussian noise

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
tree_reg2.fit(X, y2)
```

And then we'll train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
>>> X_new = np.array([[-0.4], [0.], [0.5]])
>>> sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
array([0.57356534, 0.0405142 , 0.66914249])
```

Figure 6-9 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

You can use Scikit-Learn's `GradientBoostingRegressor` class to train GBRT ensembles more easily (there's also a `GradientBoostingClassifier` class for classification). Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of decision trees (e.g., `max_depth`, `min_samples_leaf`), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`). The following code creates the same ensemble as the previous one:

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                 learning_rate=1.0, random_state=42)
gbrt.fit(X, y)
```

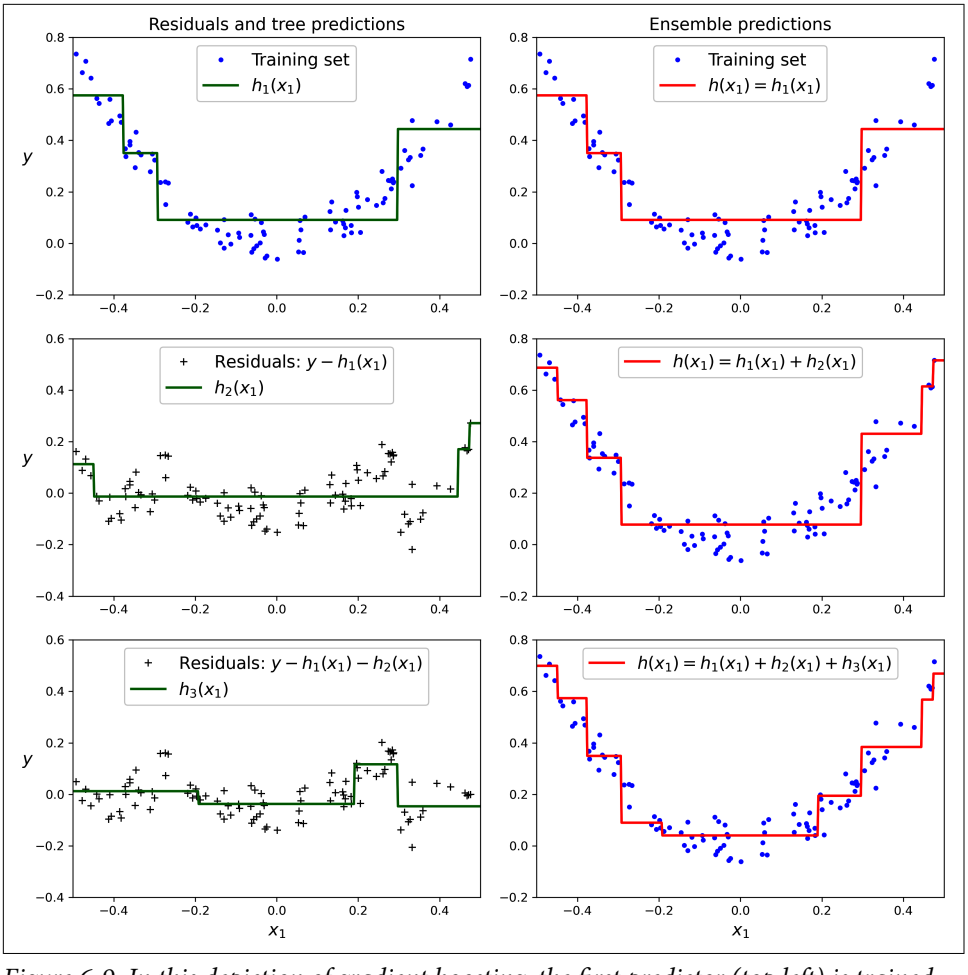

Figure 6-9. In this depiction of gradient boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as `0.05`, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*. Figure 6-10 shows two GBRT ensembles trained with different hyperparameters: the one on the left does not have enough trees to fit the

training set, while the one on the right has about the right amount. If we added more trees, the GBRT would start to overfit the training set. As usual, you can use cross-validation to find the optimal learning rate, using `GridSearchCV` or `RandomizedSearchCV`.

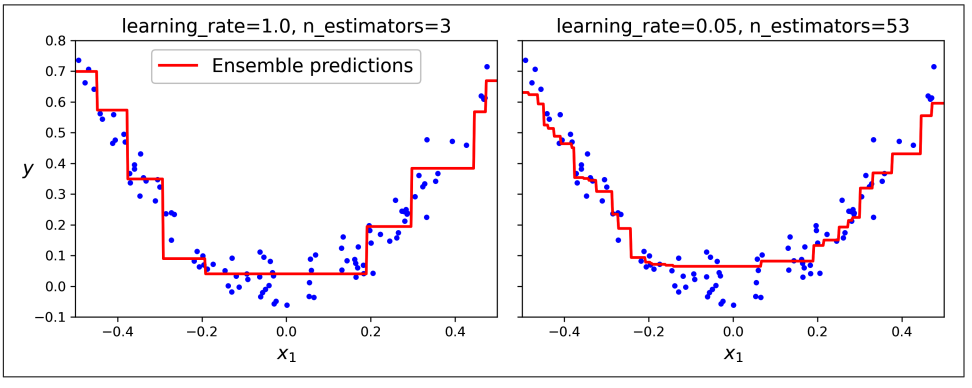

Figure 6-10. GBRT ensembles with not enough predictors (left) and just enough (right)

To find the optimal number of trees, you could also perform cross-validation, but there's a simpler way: if you set the `n_iter_no_change` hyperparameter to an integer value, say 10, then the `GradientBoostingRegressor` will automatically stop adding more trees during training if it sees that the last 10 trees didn't help. This is simply early stopping (introduced in Chapter 4), but with a little bit of patience: it tolerates having no progress for a few iterations before it stops. Let's train the ensemble using early stopping:

```
gbrt_best = GradientBoostingRegressor(
    max_depth=2, learning_rate=0.05, n_estimators=500,
    n_iter_no_change=10, random_state=42)
gbrt_best.fit(X, y)
```

If you set `n_iter_no_change` too low, training may stop too early and the model will underfit. But if you set it too high, it will overfit instead. We also set a fairly small learning rate and a high number of estimators, but the actual number of estimators in the trained ensemble is much lower, thanks to early stopping:

```
>>> gbrt_best.n_estimators_
53
```

When `n_iter_no_change` is set, the `fit()` method automatically splits the training set into a smaller training set and a validation set: this allows it to evaluate the model's performance each time it adds a new tree. The size of the validation set is controlled by the `validation_fraction` hyperparameter, which is 10% by default. The `tol` hyperparameter determines the maximum performance improvement that still counts as negligible. It defaults to 0.0001.

The `GradientBoostingRegressor` class also supports a `subsample` hyperparameter, which specifies the fraction of training instances to be used for training each tree. For example, if `subsample=0.25`, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this technique trades a higher bias for a lower variance. It also speeds up training considerably. This is called *stochastic gradient boosting*.

Histogram-Based Gradient Boosting

Scikit-Learn also provides another GBRT implementation, optimized for large datasets: *histogram-based gradient boosting* (HGB). It works by binning the input features, replacing them with integers. The number of bins is controlled by the `max_bins` hyperparameter, which defaults to 255 and cannot be set any higher than this. Binning can greatly reduce the number of possible thresholds that the training algorithm needs to evaluate. Moreover, working with integers makes it possible to use faster and more memory-efficient data structures. And the way the bins are built removes the need for sorting the features when training each tree.

As a result, this implementation has a computational complexity of $O(b \times m)$ instead of $O(n \times m \times \log(m))$, where b is the number of bins, m is the number of training instances, and n is the number of features. In practice, this means that HGB can train hundreds of times faster than regular GBRT on large datasets. However, binning causes a precision loss, which acts as a regularizer: depending on the dataset, this may help reduce overfitting, or it may cause underfitting.

Scikit-Learn provides two classes for HGB: `HistGradientBoostingRegressor` and `HistGradientBoostingClassifier`. They're similar to `GradientBoostingRegressor` and `GradientBoostingClassifier`, with a few notable differences:

- Early stopping is automatically activated if the number of instances is greater than 10,000. You can turn early stopping always on or always off by setting the `early_stopping` hyperparameter to `True` or `False`.
- Subsampling is not supported.
- `n_estimators` is renamed to `max_iter`.
- The only decision tree hyperparameters that can be tweaked are `max_leaf_nodes`, `min_samples_leaf`, `max_depth`, and `max_features`.

The HGB classes support missing values natively, as well as categorical features. This simplifies preprocessing quite a bit. However, the categorical features must be represented as integers ranging from 0 to a number lower than `max_bins`. You can use an `OrdinalEncoder` for this. For example, here's how to build and train a complete pipeline for the California housing dataset introduced in Chapter 2:

```
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder

hgb_reg = make_pipeline(
    make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
                            remainder="passthrough",
                            force_int_remainder_cols=False),
    HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
)
hgb_reg.fit(housing, housing_labels)
```

The whole pipeline is almost as short as the imports! No need for an imputer, scaler, or a one-hot encoder, it's really convenient. Note that `categorical_features` must be set to the categorical column indices (or a Boolean array). Without any hyperparameter tuning, this model yields an RMSE of about 47,600, which is not too bad.

In short, HGB is a great choice when you have a fairly large dataset, especially when it contains categorical features and missing values: it performs well, doesn't require much preprocessing work, and training is fast. However, it can be a bit less accurate than GBRT, due to the binning, so you might want to try both.

Several other optimized implementations of gradient boosting are available in the Python ML ecosystem: in particular, XGBoost, CatBoost, and LightGBM. These libraries have been around for several years. They are all specialized for gradient boosting, their APIs are very similar to Scikit-Learn's, and they provide many additional features, including hardware acceleration using GPUs; you should definitely check them out! Moreover, Yggdrasil Decision Forests (YDF) provides optimized implementations of a variety of random forest algorithms.

Stacking

The last ensemble method we will discuss in this chapter is called *stacking* (short for *stacked generalization*).[18] It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation? Figure 6-11 shows such an ensemble performing a regression task on a new instance. Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and then the final predictor

18 David H. Wolpert, "Stacked Generalization", *Neural Networks* 5, no. 2 (1992): 241–259.

(called a *blender*, or a *meta learner*) takes these predictions as inputs and makes the final prediction (3.0).

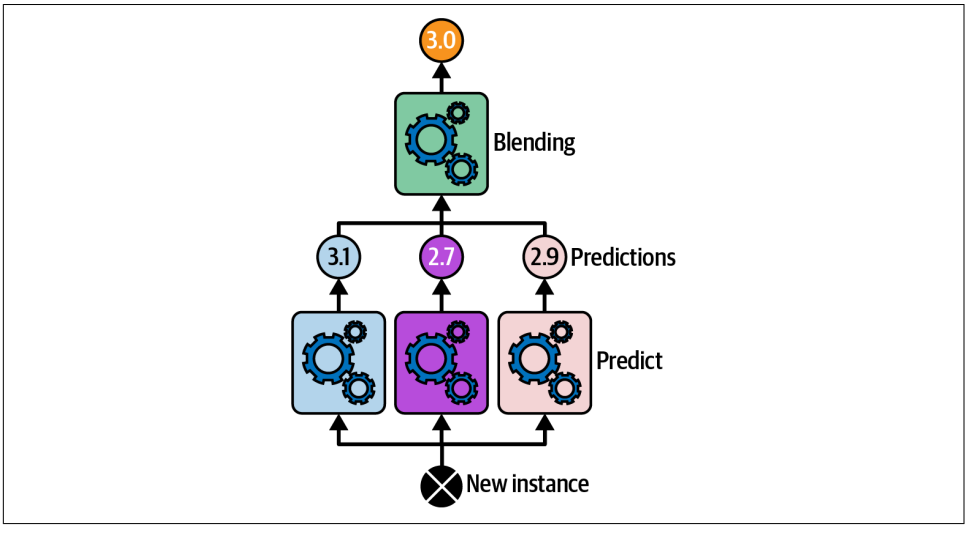

Figure 6-11. Aggregating predictions using a blending predictor

To train the blender, you first need to build the blending training set. You can use `cross_val_predict()` on every predictor in the ensemble to get out-of-sample predictions for each instance in the original training set (Figure 6-12), and use these as the input features to train the blender; the targets can simply be copied from the original training set. Note that regardless of the number of features in the original training set (just one in this example), the blending training set will contain one input feature per predictor (three in this example). Once the blender is trained, the base predictors must be retrained one last time on the full original training set (since `cross_val_predict()` does not give access to the trained estimators).

It is actually possible to train several different blenders this way (e.g., one using linear regression, another using random forest regression) to get a whole layer of blenders, and then add another blender on top of that to produce the final prediction, as shown in Figure 6-13. You may be able to squeeze out a few more drops of performance by doing this, but it will cost you in both training time and system complexity.

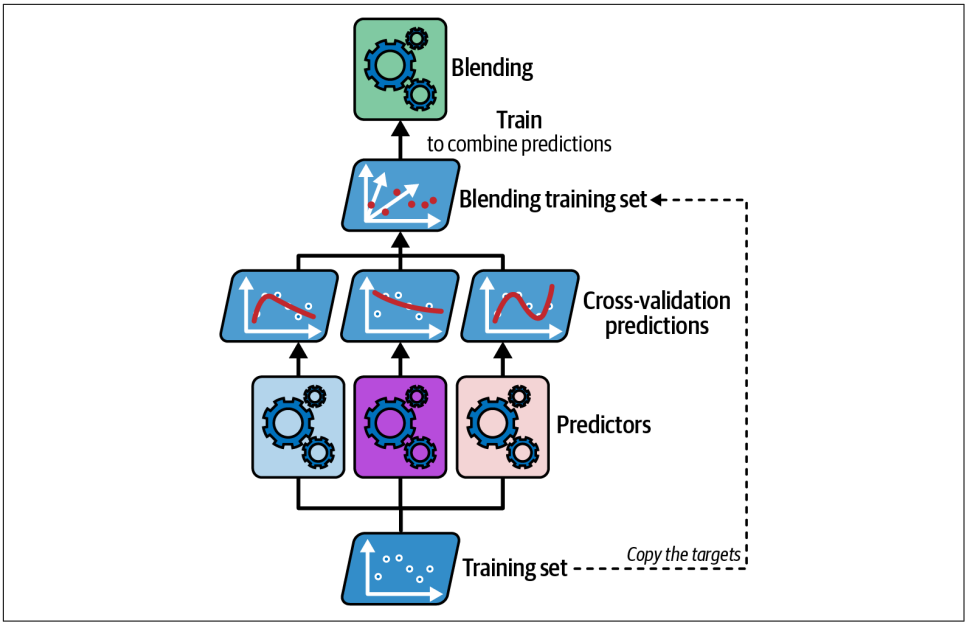

Figure 6-12. Training the blender in a stacking ensemble

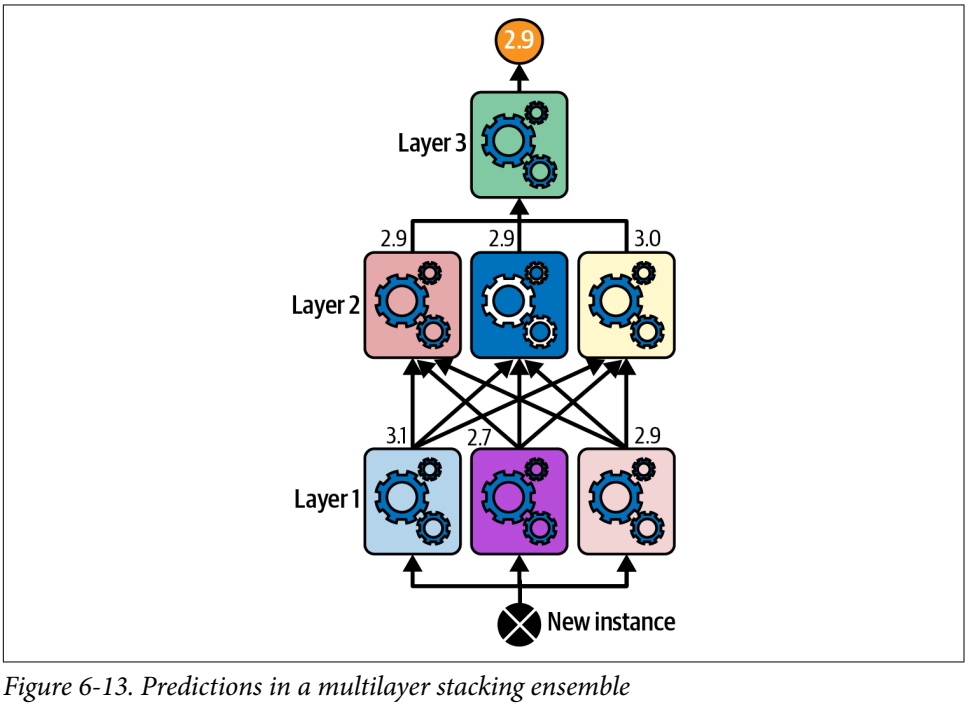

Figure 6-13. Predictions in a multilayer stacking ensemble

Scikit-Learn provides two classes for stacking ensembles: `StackingClassifier` and `StackingRegressor`. For example, we can replace the `VotingClassifier` we used at the beginning of this chapter on the moons dataset with a `StackingClassifier`:

```
from sklearn.ensemble import StackingClassifier

stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),
    cv=5  # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

For each predictor, the stacking classifier will call `predict_proba()` if available; if not it will fall back to `decision_function()` or, as a last resort, call `predict()`. If you don't provide a final estimator, `StackingClassifier` will use `LogisticRegression` and `StackingRegressor` will use `RidgeCV`.

If you evaluate this stacking model on the test set, you will find 92.8% accuracy, which is a bit better than the voting classifier using soft voting, which got 92%. Depending on your use case, this may or may not be worth the extra complexity and computational cost (since there's an extra model to run after all the others).

In conclusion, ensemble methods are versatile, powerful, and fairly simple to use. They can overfit if you're not careful, but that's true of every powerful model. Table 6-1 summarizes all the techniques we discussed in this chapter, and when to use each one.

Table 6-1. When to use each ensemble learning method

Ensemble method	When to use it	Example use cases
Hard voting	Balanced classification dataset with multiple strong but diverse classifiers.	Spam detection, sentiment analysis, disease classification
Soft voting	Classification dataset with probabilistic models, where confidence scores matter.	Medical diagnosis, credit risk analysis, fake news detection
Bagging	Structured or semi-structured dataset with high variance and overfitting-prone models.	Financial risk modeling, ecommerce recommendation
Pasting	Structured or semi-structured dataset where more independent models are needed.	Customer segmentation, protein classification
Random forest	High-dimensional structured datasets with potentially noisy features.	Customer churn prediction, genetic data analysis, fraud detection
Extra-trees	Large structured datasets with many features, where speed is critical and reducing variance is important.	Real-time fraud detection, sensor data analysis

Ensemble method	When to use it	Example use cases
AdaBoost	Small to medium-sized, low-noise, structured datasets with weak learners (e.g., decision stumps), where interpretability is helpful.	Credit scoring, anomaly detection, predictive maintenance
Gradient boosting	Medium to large structured datasets where high predictive power is required, even at the cost of extra tuning.	Housing price prediction, risk assessment, demand forecasting
Histogram-based gradient boosting (HGB)	Large structured datasets where training speed and scalability are key.	Click-through rate prediction, ranking algorithms, real-time bidding in advertising
Stacking	Complex, high-dimensional datasets where combining multiple diverse models can maximize accuracy.	Recommendation engines, autonomous vehicle decision-making, Kaggle competitions

Random forests, AdaBoost, GBRT, and HGB are among the first models you should test for most machine learning tasks, particularly with heterogeneous tabular data. Moreover, as they require very little preprocessing, they're great for getting a prototype up and running quickly.

So far, we have looked only at supervised learning techniques. In the next chapter, we will turn to the most common unsupervised learning task: dimensionality reduction.

Exercises

1. If you have trained five different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?
2. What is the difference between hard and soft voting classifiers?
3. Is it possible to speed up training of a bagging ensemble by distributing it across multiple servers? What about pasting ensembles, boosting ensembles, random forests, or stacking ensembles?
4. What is the benefit of out-of-bag evaluation?
5. What makes extra-trees ensembles more random than regular random forests? How can this extra randomness help? Are extra-trees classifiers slower or faster than regular random forests?
6. If your AdaBoost ensemble underfits the training data, which hyperparameters should you tweak, and how?
7. If your gradient boosting ensemble overfits the training set, should you increase or decrease the learning rate?

8. Load the MNIST dataset (introduced in Chapter 3), and split it into a training set, a validation set, and a test set (e.g., use 50,000 instances for training, 10,000 for validation, and 10,000 for testing). Then train various classifiers, such as a random forest classifier, an extra-trees classifier, and an SVM classifier. Next, try to combine them into an ensemble that outperforms each individual classifier on the validation set, using soft or hard voting. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?
9. Run the individual classifiers from the previous exercise to make predictions on the validation set, and create a new training set with the resulting predictions: each training instance is a vector containing the set of predictions from all your classifiers for an image, and the target is the image's class. Train a classifier on this new training set. Congratulations—you have just trained a blender, and together with the classifiers it forms a stacking ensemble! Now evaluate the ensemble on the test set. For each image in the test set, make predictions with all your classifiers, then feed the predictions to the blender to get the ensemble's predictions. How does it compare to the voting classifier you trained earlier? Now try again using a `StackingClassifier` instead. Do you get better performance? If so, why?

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

CHAPTER 7
Dimensionality Reduction

Many machine learning problems involve thousands or even millions of features for each training instance. Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution, as you will see. This problem is often referred to as the *curse of dimensionality*.

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in Chapter 3): the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. As we saw in the previous chapter, Figure 6-6 confirms that these pixels are utterly unimportant for the classification task. Additionally, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information, removing redundancy and sometimes even noise.

Reducing dimensionality can also drop some useful information, just like compressing an image to JPEG can degrade its quality: it can make your system perform slightly worse, especially if you reduce dimensionality too much. Moreover, some models—such as neural networks—can handle high-dimensional data efficiently and learn to reduce its dimensionality while preserving the useful information for the task at hand. In short, adding an extra preprocessing step for dimensionality reduction will not always help.

Apart from speeding up training and possibly improving your model's performance, dimensionality reduction is also extremely useful for data visualization. Reducing the number of dimensions down to two (or three) makes it possible to plot a condensed view of a high-dimensional training set on a graph and often gain some important

insights by visually detecting patterns, such as clusters. Moreover, data visualization is essential to communicate your conclusions to people who are not data scientists—in particular, decision makers who will use your results.

In this chapter we will first discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then we will consider the two main approaches to dimensionality reduction (projection and manifold learning), and we will go through three of the most popular dimensionality reduction techniques: PCA, random projection, and locally linear embedding (LLE).

The Curse of Dimensionality

We are so used to living in three dimensions[1] that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our minds (see Figure 7-1), let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.

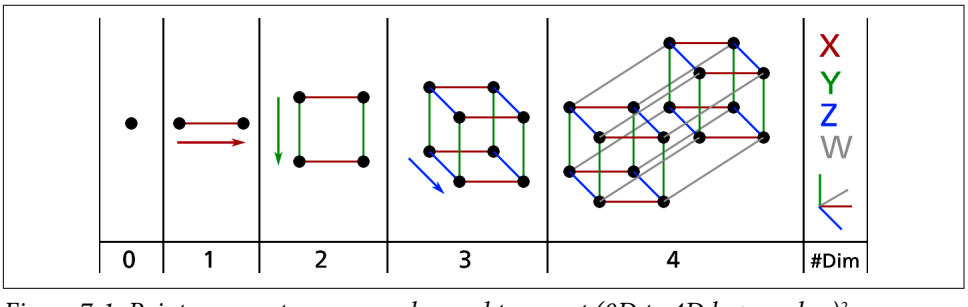

Figure 7-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

It turns out that many things behave very differently in high-dimensional space. For example, if you pick a random point in a unit square (a 1 × 1 square), it will have only about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be "extreme" along any dimension). But in a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.[3]

Here is a more troublesome difference: if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a 3D unit cube, the average distance will be roughly

1 Well, four dimensions if you count time, and a few more if you are a string theorist.

2 Watch a rotating tesseract projected into 3D space at *https://homl.info/30*. Image by Wikipedia user Nerd-Boy1392 (Creative Commons BY-SA 3.0). Reproduced from *https://en.wikipedia.org/wiki/Tesseract*.

3 Fun fact: anyone you know is probably an extremist in at least one dimension (e.g., how much sugar they put in their coffee), if you consider enough dimensions.

0.66. But what about two points picked randomly in a 1,000,000-dimensional unit hypercube? The average distance, believe it or not, will be about 408.25 (roughly $\sqrt{\frac{1{,}000{,}000}{6}}$)! This is counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? Well, there's just plenty of space in high dimensions.

As a result, high-dimensional datasets are often very sparse: most training instances are likely to be far away from each other, so training methods based on distance or similarity (such as *k*-nearest neighbors) will be much less effective. And some types of models will not be usable at all because they scale poorly with the dataset's dimensionality (e.g., SVMs or dense neural networks). And new instances will likely be far away from any training instance, making predictions much less reliable than in lower dimensions since they will be based on much larger extrapolations. Since patterns in the data will become harder to identify, models will tend to fit the noise more frequently than in lower dimensions; regularization will become all the more important. Lastly, models will become even harder to interpret.

In theory, some of these issues can be resolved by increasing the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features—significantly fewer than in the MNIST problem—all ranging from 0 to 1, you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

Main Approaches for Dimensionality Reduction

Before diving into specific dimensionality reduction algorithms, let's look at the two main approaches to reducing dimensionality: projection and manifold learning.

Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated (as discussed earlier for MNIST). As a result, all training instances lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space. This sounds abstract, so let's look at an example. In Figure 7-2, a 3D dataset is represented by small spheres (I will refer to this figure several times in the following sections).

Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the higher-dimensional (3D) space. If we project every training instance perpendicularly onto this subspace (as represented by the short dashed lines connecting the instances to the plane), we get the new 2D dataset shown in

Figure 7-3. Ta-da! We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2: they are the coordinates of the projections on the plane.

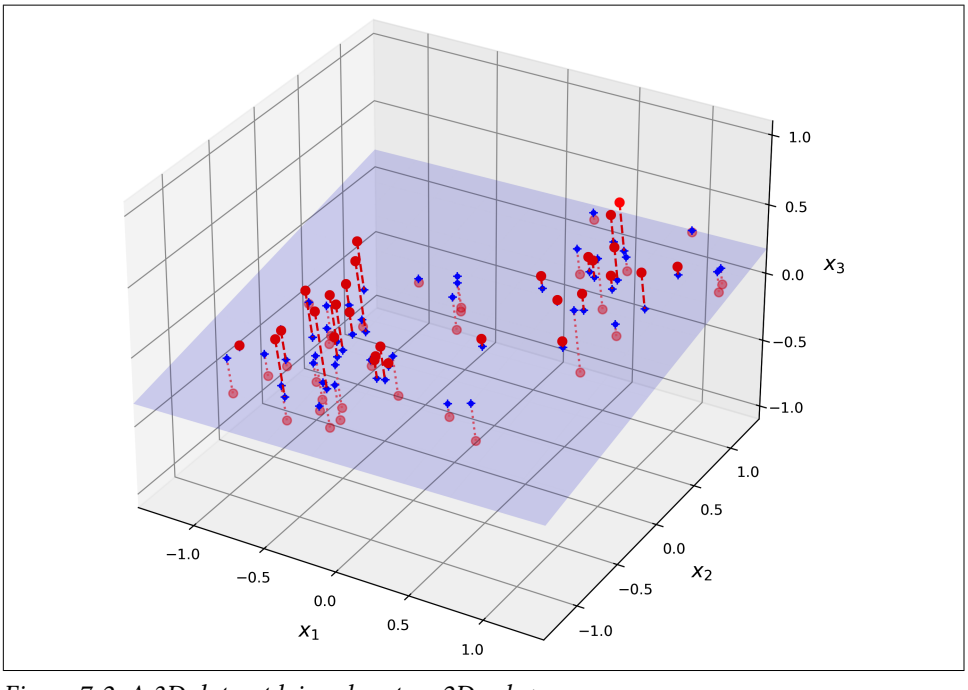

Figure 7-2. A 3D dataset lying close to a 2D subspace

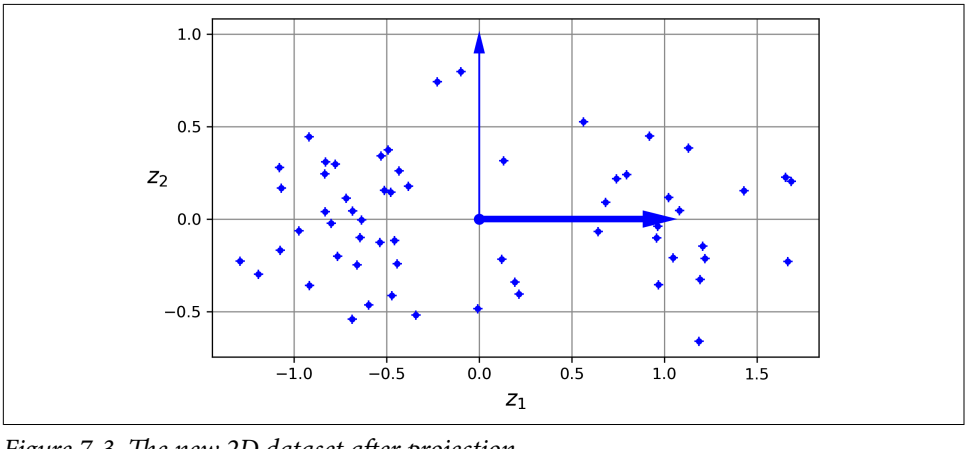

Figure 7-3. The new 2D dataset after projection

Manifold Learning

Although projection is fast and often works well, it's not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the Swiss roll dataset represented in Figure 7-4: this is a toy dataset containing 3D points in the shape of a Swiss roll.

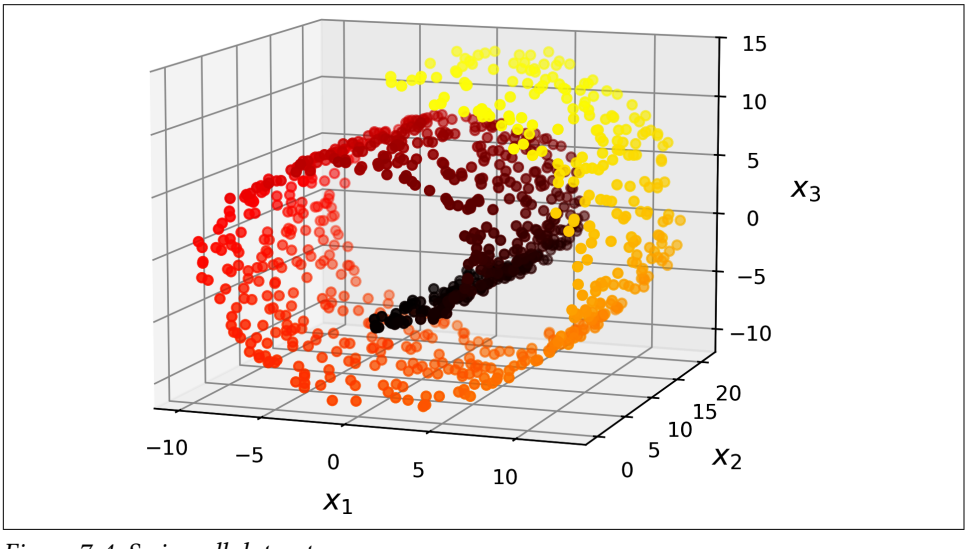

Figure 7-4. Swiss roll dataset

Simply projecting onto a plane (e.g., by dropping x_3) would squash different layers of the Swiss roll together, as shown on the left side of Figure 7-5. What you probably want instead is to unroll the Swiss roll to obtain the 2D dataset on the righthand side of Figure 7-5.

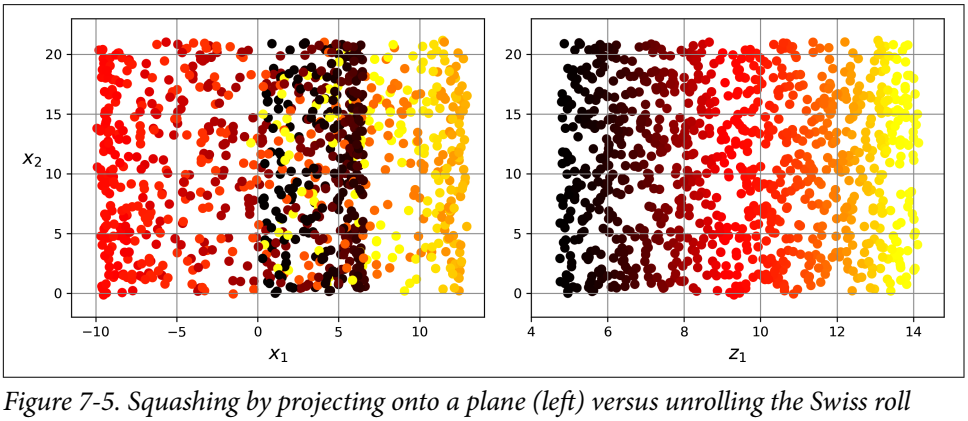

Figure 7-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

The Swiss roll is an example of a 2D *manifold*. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d-dimensional manifold is a part of an n-dimensional space (where $d < n$) that locally resembles a d-dimensional hyperplane. In the case of the Swiss roll, $d = 2$ and $n = 3$: it locally resembles a 2D plane, but it is rolled in the third dimension.

Many dimensionality reduction algorithms (e.g., LLE, Isomap, t-SNE, or UMAP), work by modeling the manifold on which the training instances lie; this is called *manifold learning*. It relies on the *manifold assumption*, also called the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

Once again, think about the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, and they are more or less centered. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you have if you are allowed to generate any image you want. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of Figure 7-6 the Swiss roll is split into two classes: in the 3D space (on the left) the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right) the decision boundary is a straight line.

However, this implicit assumption does not always hold. For example, in the bottom row of Figure 7-6, the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

In short, reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset. Dimensionality reduction is typically more effective when the dataset is small relative to the number of features, especially if it's noisy, or many features are highly correlated to one another (i.e., redundant). And if you have domain knowledge about the process that generated the data, and you know it's simple, then the manifold assumption certainly holds, and dimensionality reduction is likely to help.

Hopefully, you now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold

assumption holds. The rest of this chapter will go through some of the most popular algorithms for dimensionality reduction.

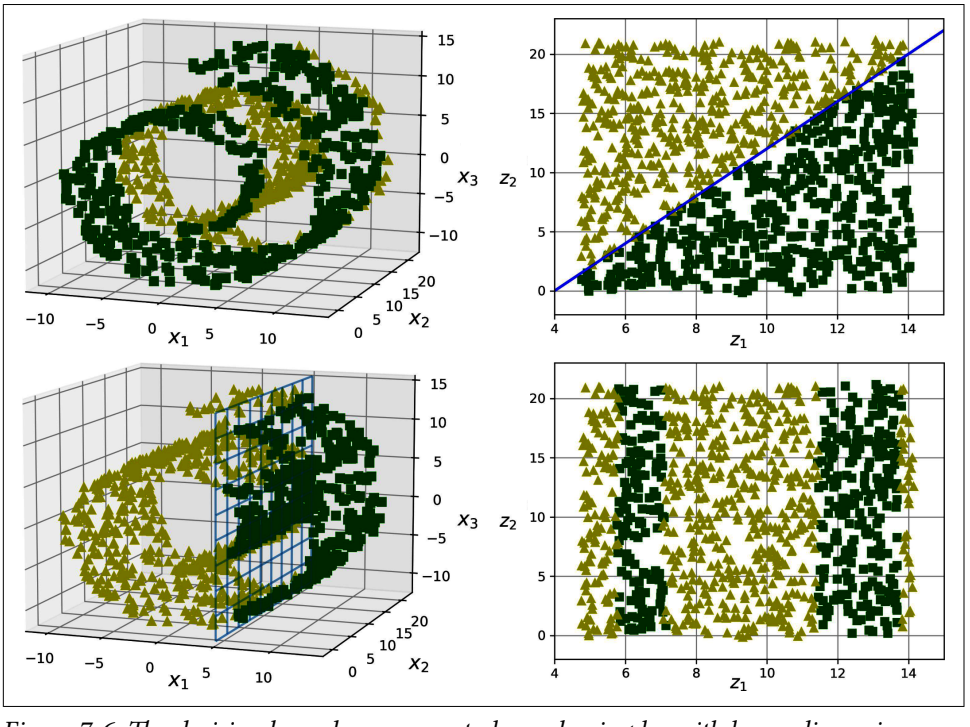

Figure 7-6. The decision boundary may not always be simpler with lower dimensions

PCA

Principal component analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it, as shown back in Figure 7-2.

Preserving the Variance

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left in Figure 7-7, along with three different axes (i.e., 1D hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum variance (top), while the projection onto the dotted line preserves very little variance (bottom), and the projection onto the dashed line preserves an intermediate amount of variance (middle).

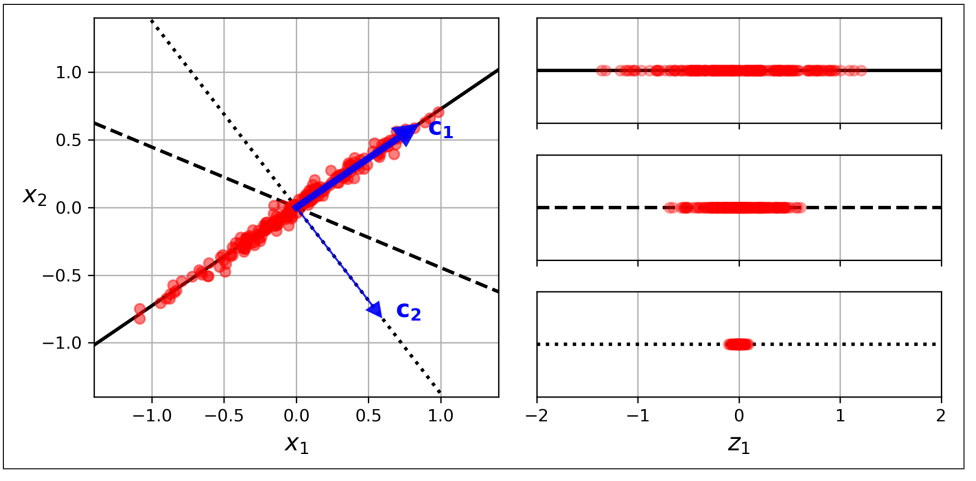

Figure 7-7. Selecting the subspace on which to project

It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Consider your shadow on the ground when the sun is directly overhead: it's a small blob that doesn't look anything like you. But your shadow on a wall at sunrise is much larger and it *does* look like you. Another way to justify choosing the axis that maximizes the variance is that it is also the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind PCA, introduced way back in 1901![4]

Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. In Figure 7-7, it is the solid line. It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of the remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

The i^{th} axis is called the i^{th} *principal component* (PC) of the data. In Figure 7-7, the first PC is the axis on which vector $\mathbf{c}_1$ lies, and the second PC is the axis on which vector $\mathbf{c}_2$ lies. In Figure 7-2, the first two PCs are on the projection plane, and the third PC is the axis orthogonal to that plane. After the projection, back in Figure 7-3, the first PC corresponds to the z_1 axis, and the second PC corresponds to the z_2 axis.

4 Karl Pearson, "On Lines and Planes of Closest Fit to Systems of Points in Space", *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, no. 11 (1901): 559–572.

For each principal component, PCA finds a zero-centered unit vector pointing along the direction of the PC. Unfortunately, its direction is not guaranteed: if you perturb the training set slightly and run PCA again, the unit vector may point in the opposite direction. In fact, a pair of unit vectors may even rotate or swap if the variances along these two axes are very close. So if you use PCA as a preprocessing step before a model, make sure you always retrain the model entirely every time you update the PCA transformer: if you don't and if the PCA's output doesn't align with the previous version, the model will be very confused.

So how can you find the principal components of a training set? Luckily, there is a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the product of three matrices $\mathbf{U}\ \mathbf{\Sigma}\ \mathbf{V}^{\intercal}$, where $\mathbf{V}$ contains the unit vectors that define all the principal components that you are looking for, in the correct order, as shown in Equation 7-1.[5]

Equation 7-1. Principal components matrix

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}$$

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the 3D training set represented back in Figure 7-2, then it extracts the two unit vectors that define the first two PCs:

```
import numpy as np

X = [...]  # create a small 3D dataset
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```

PCA assumes that the dataset is centered around the origin. As you will see, Scikit-Learn's PCA classes take care of centering the data for you. If you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

5 The proof that SVD happens to give us exactly the principal components we need for PCA requires some prerequisite math knowledge, such as eigenvectors and covariance matrices. If you are curious, you will find all the details in this 2014 paper by Jonathon Shlens.

Projecting Down to d Dimensions

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components (we will discuss how to choose the number of dimensions d shortly). Selecting this hyperplane ensures that the projection will preserve as much variance as possible. For example, in Figure 7-2 the 3D dataset is projected down to the 2D plane defined by the first two principal components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane and obtain a reduced dataset $\mathbf{X}_{d\text{-proj}}$ of dimensionality d, compute the matrix multiplication of the training set matrix $\mathbf{X}$ by the matrix $\mathbf{W}_d$, defined as the matrix containing the first d columns of $\mathbf{V}$, as shown in Equation 7-2.

Equation 7-2. Projecting the training set down to d dimensions

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$$

The following Python code projects the training set onto the plane defined by the first two principal components:

```
W2 = Vt[:2].T
X2D = X_centered @ W2
```

There you have it! You now know how to reduce the dimensionality of any dataset by projecting it down to any number of dimensions, while preserving as much variance as possible.

Using Scikit-Learn

Scikit-Learn's `PCA` class uses SVD to implement PCA, just like we did earlier in this chapter. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X2D = pca.fit_transform(X)
```

After fitting the `PCA` transformer to the dataset, its `components_` attribute holds the transpose of $\mathbf{W}_d$: it contains one row for each of the first d principal components.

Explained Variance Ratio

Another useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in Figure 7-2:

```
>>> pca.explained_variance_ratio_
array([0.82279334, 0.10821224])
```

This output tells us that about 82% of the dataset's variance lies along the first PC, and about 11% lies along the second PC. This leaves about 7% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance—say, 95%. (An exception to this rule, of course, is if you are reducing dimensionality for data visualization, in which case you will want to reduce the dimensionality down to 2 or 3.)

The following code loads and splits the MNIST dataset (introduced in Chapter 3) and performs PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1  # d equals 154
```

You could then set `n_components=d` and run PCA again, but there's a better option. Instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

The actual number of components is determined during training, and it is stored in the `n_components_` attribute:

```
>>> pca.n_components_
np.int64(154)
```

Yet another option is to plot the explained variance as a function of the number of dimensions (simply plot `cumsum`; see Figure 7-8). There will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

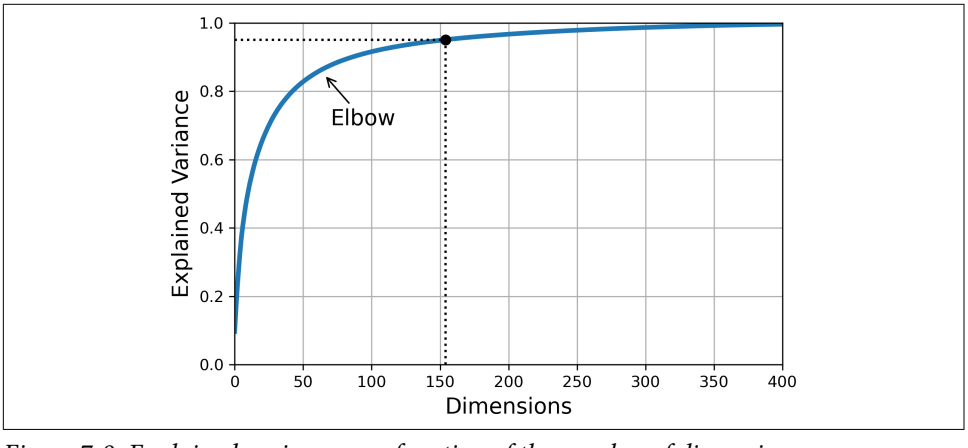

Figure 7-8. Explained variance as a function of the number of dimensions

Alternatively, if you are using dimensionality reduction as a preprocessing step for a supervised learning task (e.g., classification), then you can tune the number of dimensions as you would any other hyperparameter (see Chapter 2). For example, the following code example creates a two-step pipeline, first reducing dimensionality using PCA, then classifying using a random forest. Next, it uses `RandomizedSearchCV` to find a good combination of hyperparameters for both PCA and the random forest classifier. This example does a quick search, tuning only 2 hyperparameters, training on just 1,000 instances, and running for just 10 iterations, but feel free to do a more thorough search if you have the time:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

clf = make_pipeline(PCA(random_state=42),
                    RandomForestClassifier(random_state=42))
param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)
}
rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,
                                random_state=42)
rnd_search.fit(X_train[:1000], y_train[:1000])
```

Let's look at the best hyperparameters found:

```
>>> print(rnd_search.best_params_)
{'randomforestclassifier__n_estimators': np.int64(475),
 'pca__n_components': np.int64(57)}
```

It's interesting to note how low the optimal number of components is: we reduced a 784-dimensional dataset to just 57 dimensions! This is tied to the fact that we used a random forest, which is a pretty powerful model. If we used a linear model instead, such as an `SGDClassifier`, the search would find that we need to preserve more dimensions (about 75).

You may also care about the model's size and speed, not just it's performance. The fewer dimensions, the smaller the model, and the faster training and inference will be. But if you shrink the data too much, then you will lose too much signal and your model will underfit. You need to choose the right balance of speed, size, and performance for your particular use case.

PCA for Compression

After dimensionality reduction, the training set takes up much less space. For example, after applying PCA to the MNIST dataset while preserving 95% of its variance, we are left with 154 features, instead of the original 784 features. So the dataset is now less than 20% of its original size, and we only lost 5% of its variance! This is a reasonable compression ratio, and it's easy to see how such a size reduction would speed up a classification algorithm tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

The `inverse_transform()` method lets us decompress the reduced MNIST dataset back to 784 dimensions:

```
X_recovered = pca.inverse_transform(X_reduced)
```

Figure 7-9 shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.

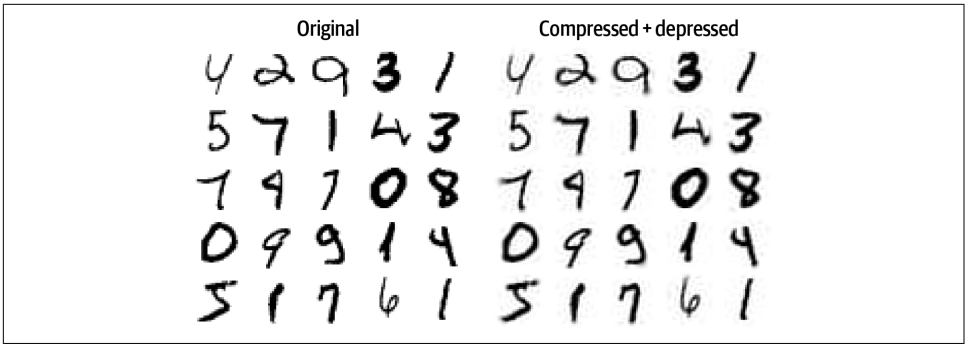

Figure 7-9. MNIST compression that preserves 95% of the variance

The equation for the inverse transformation is shown in Equation 7-3.

Equation 7-3. PCA inverse transformation, back to the original number of dimensions

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}}\mathbf{W}_d^{\mathrm{T}}$$

Randomized PCA

If you set the `svd_solver` hyperparameter to `"randomized"`, Scikit-Learn uses a stochastic algorithm called *randomized PCA* that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$ for the full SVD approach, so it is dramatically faster than full SVD when d is much smaller than n:

```
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

By default, `svd_solver` is set to `"auto"`: if the input data has few features ($n < 1{,}000$) and at least 10 times more samples ($m > 10n$), then the `"covariance_eigh"` solver is used, which is very fast in these conditions. Otherwise, if $\max(m, n) > 500$ and `n_components` is an integer smaller than 80% of $\min(m, n)$, it uses the `"randomized"` solver. In other cases, it uses the full SVD approach. If you want to force Scikit-Learn to use full SVD, trading compute time for a slightly more precise result, you can set `svd_solver="full"`.

Incremental PCA

One problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run. Fortunately, *incremental PCA* (IPCA) algorithms have been developed that allow you to split the

training set into mini-batches and feed these in one mini-batch at a time. This is useful for large training sets and for applying PCA online (i.e., on the fly, as new instances arrive).

The following code splits the MNIST training set into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` class[6] to reduce the dimensionality of the MNIST dataset down to 154 dimensions, just like before. Note that you must call the `partial_fit()` method with each mini-batch, rather than the `fit()` method with the whole training set:

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Alternatively, you can use NumPy's `memmap` class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it. To demonstrate this, let's first create a memory-mapped (memmap) file and copy the MNIST training set to it, then call `flush()` to ensure that any data still in the cache gets saved to disk. In real life, `X_train` would typically not fit in memory, so you would load it chunk by chunk and save each chunk to the right part of the memmap array:

```
filename = "my_mnist.mmap"
X_mmap = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)
X_mmap[:] = X_train  # could be a loop instead, saving the data chunk by chunk
X_mmap.flush()
```

Next, we can load the memmap file and use it like a regular NumPy array. Let's use the `IncrementalPCA` class to reduce its dimensionality. Since this algorithm uses only a small part of the array at any given time, memory usage remains under control. This makes it possible to call the usual `fit()` method instead of `partial_fit()`, which is quite convenient:

```
X_mmap = np.memmap(filename, dtype="float32", mode="readonly").reshape(-1, 784)
batch_size = X_mmap.shape[0] // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mmap)
```

6 Scikit-Learn uses the algorithm described in David A. Ross et al., "Incremental Learning for Robust Visual Tracking", *International Journal of Computer Vision* 77, no. 1–3 (2008): 125–141.

Only the raw binary data is saved to disk, so you need to specify the data type and shape of the array when you load it. If you omit the shape, `np.memmap()` returns a 1D array.

For very high-dimensional datasets, PCA can be too slow. As you saw earlier, even if you use randomized PCA, its computational complexity is still $O(m \times d^2) + O(d^3)$, so the target number of dimensions d must not be too large. If you are dealing with a dataset with tens of thousands of features or more (e.g., images), then training may become much too slow: in this case, you should consider using random projection instead.

Random Projection

As its name suggests, the random projection algorithm projects the data to a lower-dimensional space using a random linear projection. This may sound crazy, but it turns out that such a random projection is actually very likely to preserve distances fairly well, as was demonstrated mathematically by William B. Johnson and Joram Lindenstrauss in a famous lemma. So, two similar instances will remain similar after the projection, and two very different instances will remain very different.

Obviously, the more dimensions you drop, the more information is lost, and the more distances get distorted. So how can you choose the optimal number of dimensions? Well, Johnson and Lindenstrauss came up with an equation that determines the minimum number of dimensions to preserve in order to ensure—with high probability—that distances won't change by more than a given tolerance. For example, if you have a dataset containing m = 5,000 instances with n = 20,000 features each, and you don't want the squared distance between any two instances to change by more than ε = 10%,[7] then you should project the data down to d dimensions, with $d \geq 4 \log(m) / (½ \varepsilon^2 - ⅓ \varepsilon^3)$, which is 7,300 dimensions. That's quite a significant dimensionality reduction! Notice that the equation does not use n, it only relies on m and ε. This equation is implemented by the `johnson_lindenstrauss_min_dim()` function:

```
>>> from sklearn.random_projection import johnson_lindenstrauss_min_dim
>>> m, ε = 5_000, 0.1
>>> d = johnson_lindenstrauss_min_dim(m, eps=ε)
>>> d
7300
```

7 ε is the Greek letter epsilon, often used for tiny values.

Now we can just generate a random matrix **P** of shape [*d*, *n*], where each item is sampled randomly from a Gaussian distribution with mean 0 and variance 1 / *d*, and use it to project a dataset from *n* dimensions down to *d*:

```
n = 20_000
rng = np.random.default_rng(seed=42)
P = rng.standard_normal((d, n)) / np.sqrt(d)  # std dev = sqrt(variance)

X = rng.standard_normal((m, n))  # generate a fake dataset
X_reduced = X @ P.T
```

That's all there is to it! It's simple and efficient, and training is almost instantaneous: the only thing the algorithm needs to create the random matrix is the dataset's shape. The data itself is not used at all. This makes random projection particularly well suited for very high-dimensional data such as text or genomics with millions of features, or very sparse data, for which even randomized PCA may take too long to train and require too much memory. At inference time, random projection is just as fast as PCA (i.e., one matrix multiplication). That said, random projection is not a silver bullet: it loses a bit more signal than PCA, so there's a trade-off between training speed and performance.

Scikit-Learn offers a `GaussianRandomProjection` class to do exactly what we just did: when you call its `fit()` method, it uses `johnson_lindenstrauss_min_dim()` to determine the output dimensionality, then it generates a random matrix, which it stores in the `components_` attribute. Then when you call `transform()`, it uses this matrix to perform the projection. When creating the transformer, you can set `eps` to tweak ε (it defaults to 0.1), and `n_components` to force a specific target dimensionality *d* (you will probably want to fine-tune these hyperparameters using cross-validation). The following code example gives the same result as the preceding code (you can also verify that `gaussian_rnd_proj.components_` is equal to `P`):

```
from sklearn.random_projection import GaussianRandomProjection

gaussian_rnd_proj = GaussianRandomProjection(eps=ε, random_state=42)
X_reduced = gaussian_rnd_proj.fit_transform(X)  # same result as above
```

Scikit-Learn also provides a second random projection transformer, known as `SparseRandomProjection`. It determines the target dimensionality in the same way, generates a random matrix of the same shape, and performs the projection identically. The main difference is that the random matrix is sparse. This means it uses much less memory: about 25 MB instead of almost 1.2 GB in the preceding example! And it's also much faster, both to generate the random matrix and to reduce dimensionality: about 50% faster in this case. Moreover, if the input is sparse, the transformation keeps it sparse (unless you set `dense_output=True`). Lastly, it enjoys the same distance-preserving property as the previous approach, and the quality of the dimensionality reduction is comparable (only very slightly less accurate). In short,

it's usually preferable to use this transformer instead of the first one, especially for large or sparse datasets.

The ratio r of nonzero items in the sparse random matrix is called its *density*. By default, it is equal to $\frac{1}{\sqrt{n}}$. With 20,000 features, this means that only 1 in ~141 cells in the random matrix is nonzero: that's quite sparse! You can set the `density` hyperparameter to another value if you prefer. Each cell in the sparse random matrix has a probability r of being nonzero, and each nonzero value is either $-v$ or $+v$ (both equally likely), where $v = \frac{1}{\sqrt{dr}}$.

If you want to perform the inverse transform, you first need to compute the pseudoinverse of the components matrix using SciPy's `pinv()` function, then multiply the reduced data by the transpose of the pseudoinverse:

```
components_pinv = np.linalg.pinv(gaussian_rnd_proj.components_)
X_recovered = X_reduced @ components_pinv.T
```

Computing the pseudoinverse may take a very long time if the components matrix is large, as the computational complexity of `pinv()` is $O(dn^2)$ if $d < n$, or $O(nd^2)$ otherwise.

In summary, random projection is a simple, fast, memory-efficient, and surprisingly powerful dimensionality reduction algorithm that you should keep in mind, especially when you deal with high-dimensional datasets.

Random projection is not always used to reduce the dimensionality of large datasets. For example, a 2017 paper[8] by Sanjoy Dasgupta et al. showed that the brain of a fruit fly implements an analog of random projection to map dense low-dimensional olfactory inputs to sparse high-dimensional binary outputs: for each odor, only a small fraction of the output neurons get activated, but similar odors activate many of the same neurons. This is similar to a well-known algorithm called *locality sensitive hashing* (LSH), which is typically used in search engines to group similar documents (see Chapter 17).

8 Sanjoy Dasgupta et al., "A neural algorithm for a fundamental computing problem", *Science* 358, no. 6364 (2017): 793–796.

LLE

Locally linear embedding (LLE)[9] is a *nonlinear dimensionality reduction* (NLDR) technique. It is a manifold learning technique that does not rely on projections, unlike PCA and random projection. In a nutshell, LLE first determines how each training instance linearly relates to its nearest neighbors, then it looks for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise. However, it does not scale well so it is mostly for small or medium sized datasets.

The following code makes a Swiss roll, then uses Scikit-Learn's `LocallyLinearEmbedding` class to unroll it:

```
from sklearn.datasets import make_swiss_roll
from sklearn.manifold import LocallyLinearEmbedding

X_swiss, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=42)
X_unrolled = lle.fit_transform(X_swiss)
```

The variable `t` is a 1D NumPy array containing the position of each instance along the rolled axis of the Swiss roll. We don't use it in this example, but it can be used as a target for a nonlinear regression task. The resulting 2D dataset is shown in Figure 7-10.

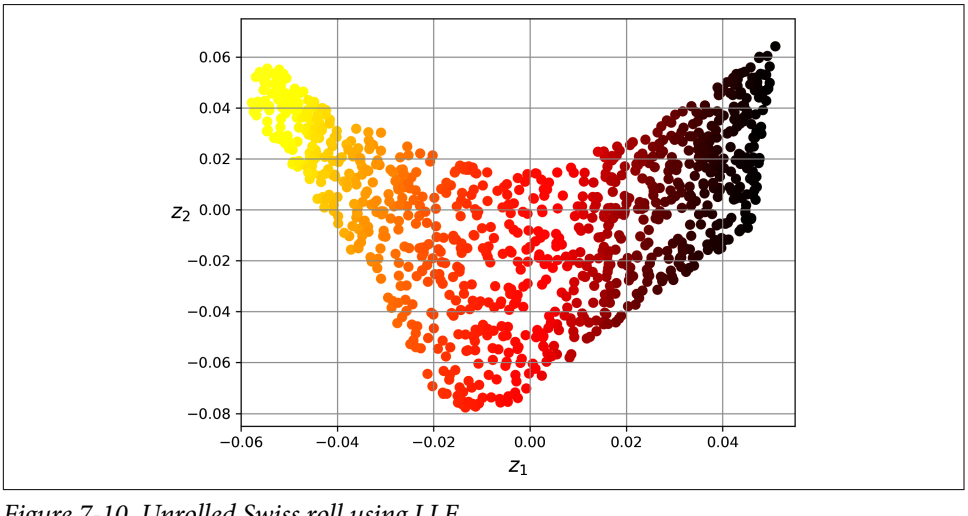

Figure 7-10. Unrolled Swiss roll using LLE

9 Sam T. Roweis and Lawrence K. Saul, "Nonlinear Dimensionality Reduction by Locally Linear Embedding", *Science* 290, no. 5500 (2000): 2323–2326.

As you can see, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the unrolled Swiss roll should be a rectangle, not this kind of stretched and twisted band. Nevertheless, LLE did a pretty good job of modeling the manifold.

Here's how LLE works: for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k-nearest neighbors (in the preceding code $k = 10$), then tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors. More specifically, it tries to find the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}^{(i)}$ and $\sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)}$ is as small as possible, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not one of the k-nearest neighbors of $\mathbf{x}^{(i)}$. Thus the first step of LLE is the constrained optimization problem described in Equation 7-4, where $\mathbf{W}$ is the weight matrix containing all the weights $w_{i,j}$. The second constraint simply normalizes the weights for each training instance $\mathbf{x}^{(i)}$.

Equation 7-4. LLE step 1: linearly modeling local relationships

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)} \right)^2$$

$$\text{subject to } \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ n.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1,2,\cdots,m \end{cases}$$

After this step, the weight matrix $\widehat{\mathbf{W}}$ (containing the weights $\widehat{w}_{i,j}$) encodes the local linear relationships between the training instances. The second step is to map the training instances into a d-dimensional space (where $d < n$) while preserving these local relationships as much as possible. If $\mathbf{z}^{(i)}$ is the image of $\mathbf{x}^{(i)}$ in this d-dimensional space, then we want the squared distance between $\mathbf{z}^{(i)}$ and $\sum_{j=1}^{m} \widehat{w}_{i,j}\mathbf{z}^{(j)}$ to be as small as possible. This idea leads to the unconstrained optimization problem described in Equation 7-5. It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that $\mathbf{Z}$ is the matrix containing all $\mathbf{z}^{(i)}$.

Equation 7-5. LLE step 2: reducing dimensionality while preserving relationships

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j}\mathbf{z}^{(j)} \right)^2$$

Scikit-Learn's LLE implementation has the following computational complexity: $O(m \log(m)n \log(k))$ for finding the k-nearest neighbors, $O(mnk^3)$ for optimizing the weights, and $O(dm^2)$ for constructing the low-dimensional representations. Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

As you can see, LLE is quite different from the projection techniques, and it's significantly more complex, but it can also construct much better low-dimensional representations, especially if the data is nonlinear.

Other Dimensionality Reduction Techniques

Before we conclude this chapter, let's take a quick look at a few other popular dimensionality reduction techniques available in Scikit-Learn:

`sklearn.manifold.MDS`
: *Multidimensional scaling* (MDS) reduces dimensionality while trying to preserve the distances between the instances. Random projection does that for high-dimensional data, but it doesn't work well on low-dimensional data.

`sklearn.manifold.Isomap`
: *Isomap* creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances* between the instances. The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes. This approach works best when the data lies on a fairly smooth and low-dimensional manifold with a single global structure (e.g., the Swiss roll).

`sklearn.manifold.TSNE`
: *t-distributed stochastic neighbor embedding* (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space. For example, in the exercises at the end of this chapter you will use t-SNE to visualize a 2D map of the MNIST images. However, it is not meant to be used as a preprocessing stage for an ML model.

`sklearn.discriminant_analysis.LinearDiscriminantAnalysis`
: *Linear discriminant analysis* (LDA) is a linear classification algorithm that, during training, learns the most discriminative axes between the classes. These axes can then be used to define a hyperplane onto which to project the data. The benefit of this approach is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm (unless LDA alone is sufficient).

Uniform Manifold Approximation and Projection (UMAP) is another popular dimensionality reduction technique for visualization. While t-SNE is better at preserving the local structure, especially clusters, UMAP tries to preserve both the local and global structures. Moreover, it scales better to large datasets. Sadly, it is not available in Scikit-Learn, but there's a good implementation in the umap-learn package.

Figure 7-11 shows the results of MDS, Isomap, and t-SNE on the Swiss roll. MDS manages to flatten the Swiss roll without losing its global curvature, while Isomap drops it entirely. Depending on the downstream task, preserving the large-scale structure may be good or bad. t-SNE does a reasonable job of flattening the Swiss roll, preserving a bit of curvature, and it also amplifies clusters, tearing the roll apart. Again, this might be good or bad, depending on the downstream task.

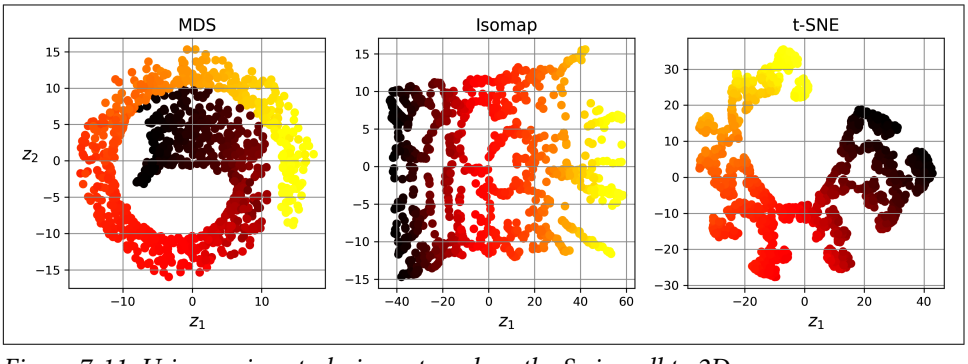

Figure 7-11. Using various techniques to reduce the Swiss roll to 2D

Exercises

1. What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?
2. What is the curse of dimensionality?
3. Once a dataset's dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
4. Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
5. Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%. How many dimensions will the resulting dataset have?
6. In what cases would you use regular PCA, incremental PCA, randomized PCA, or random projection?
7. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?

8. Does it make any sense to chain two different dimensionality reduction algorithms?
9. Load the MNIST dataset (introduced in Chapter 3) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a random forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new random forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next, evaluate the classifier on the test set. How does it compare to the previous classifier? Try again with an `SGDClassifier`. How much does PCA help now?
10. Use t-SNE to reduce the first 5,000 images of the MNIST dataset down to 2 dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class. Alternatively, you can replace each dot in the scatterplot with the corresponding instance's class (a digit from 0 to 9), or even plot scaled-down versions of the digit images themselves (if you plot all digits the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits. Try using other dimensionality reduction algorithms, such as PCA, LLE, or MDS, and compare the resulting visualizations.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.