CHAPTER 5
Support Vector Machines

A *support vector machine* (SVM) is a powerful and versatile machine learning model, capable of performing linear or nonlinear classification, regression, and even novelty detection. SVMs shine with small to medium-sized nonlinear datasets (i.e., hundreds to thousands of instances), especially for classification tasks. However, they don't scale very well to very large datasets, as you will see.

This chapter will explain the core concepts of SVMs, how to use them, and how they work. Let's jump right in!

Linear SVM Classification

The fundamental idea behind SVMs is best explained with some visuals. Figure 5-1 shows part of the iris dataset that was introduced at the end of Chapter 4. The two classes can clearly be separated easily with a straight line (they are *linearly separable*). The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line is so bad that it does not even separate the classes properly. The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances. In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also stays as far away from the closest training instances as possible. You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.

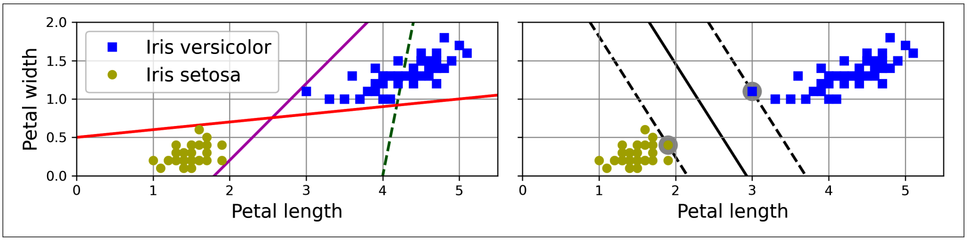

Figure 5-1. Large margin classification

Notice that adding more training instances "off the street" will not affect the decision boundary at all: it is fully determined (or "supported") by the instances located on the edge of the street. These instances are called the *support vectors* (they are circled in Figure 5-1).

SVMs are sensitive to the feature scales, as you can see in Figure 5-2. In the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using Scikit-Learn's `StandardScaler`), the decision boundary in the right plot looks much better.

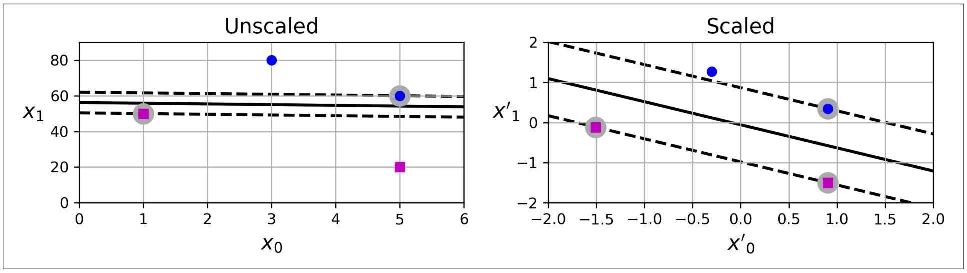

Figure 5-2. Sensitivity to feature scales

Soft Margin Classification

If we strictly impose that all instances must be off the street and on the correct side, this is called *hard margin classification*. There are two main issues with hard margin classification. First, it only works if the data is linearly separable. Second, it is sensitive to outliers. Figure 5-3 shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin; on the right, the decision boundary ends up very different from the one we saw in Figure 5-1 without the outlier, and the model will probably not generalize as well.

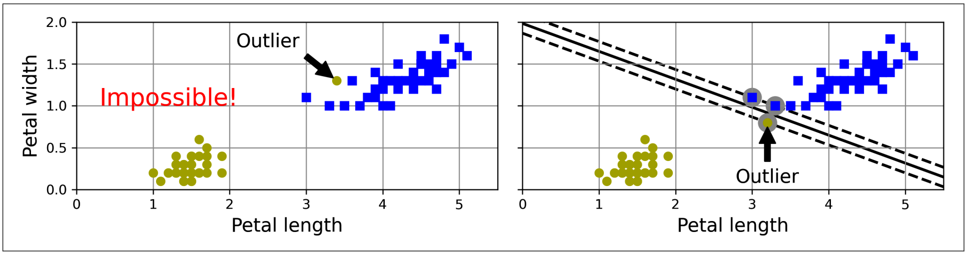

Figure 5-3. Hard margin sensitivity to outliers

To avoid these issues, we need to use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the *margin violations* (i.e., instances that end up in the middle of the street or even on the wrong side). This is called *soft margin classification*.

When creating an SVM model using Scikit-Learn, you can specify several hyperparameters, including the regularization hyperparameter `C`. If you set it to a low value, then you end up with the model on the left of Figure 5-4. With a high value, you get the model on the right. As you can see, reducing `C` makes the street larger, but it also leads to more margin violations. In other words, reducing `C` results in more instances supporting the street, so there's less risk of overfitting. But if you reduce it too much, then the model ends up underfitting, as seems to be the case here: the model with `C=100` looks like it will generalize better than the one with `C=1`.

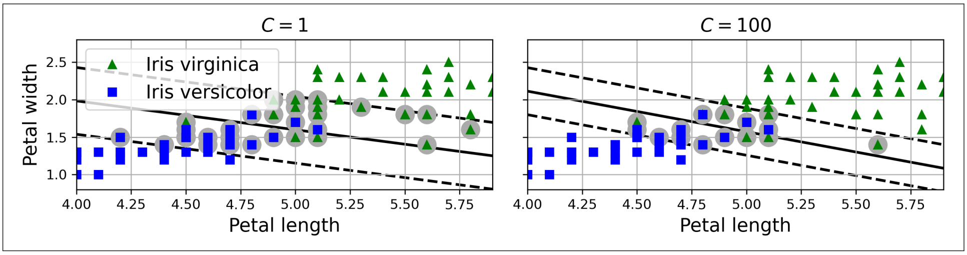

Figure 5-4. Large margin (left) versus fewer margin violations (right)

If your SVM model is overfitting, you can try regularizing it by reducing `C`.

The following Scikit-Learn code loads the iris dataset and trains a linear SVM classifier to detect *Iris virginica* flowers. The pipeline first scales the features, then uses a `LinearSVC` with `C=1`:

```
from sklearn.datasets import load_iris
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 2)  # Iris virginica

svm_clf = make_pipeline(StandardScaler(),
                        LinearSVC(C=1, random_state=42))
svm_clf.fit(X, y)
```

The resulting model is represented on the left in Figure 5-4.

Then, as usual, you can use the model to make predictions:

```
>>> X_new = [[5.5, 1.7], [5.0, 1.5]]
>>> svm_clf.predict(X_new)
array([ True, False])
```

The first plant is classified as an *Iris virginica*, while the second is not. Let's look at the scores that the SVM used to make these predictions. These measure the signed distance between each instance and the decision boundary:

```
>>> svm_clf.decision_function(X_new)
array([ 0.66163411, -0.22036063])
```

Unlike `LogisticRegression`, `LinearSVC` doesn't have a `predict_proba()` method to estimate the class probabilities. That said, if you use the `SVC` class (discussed shortly) instead of `LinearSVC`, and if you set its `probability` hyperparameter to `True`, then the model will fit an extra model at the end of training to map the SVM decision function scores to estimated probabilities. Under the hood, this requires using 5-fold cross-validation to generate out-of-sample predictions for every instance in the training set, then training a `LogisticRegression` model, so it will slow down training considerably. After that, the `predict_proba()` and `predict_log_proba()` methods will be available.

Nonlinear SVM Classification

Although linear SVM classifiers are efficient and often work surprisingly well, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features (as we did in Chapter 4); in some cases this can result in a linearly separable dataset. Consider the lefthand plot in Figure 5-5: it represents a simple dataset with just one feature, x_1. This dataset is not linearly separable, as you can see. But if you add a second feature $x_2 = (x_1)^2$, the resulting 2D dataset is perfectly linearly separable.

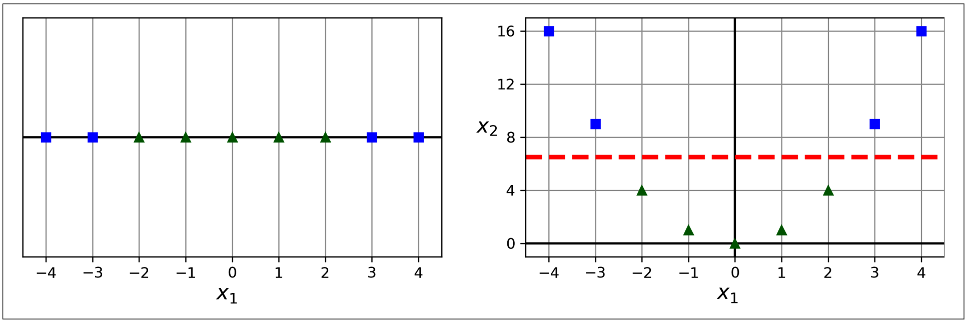

Figure 5-5. Adding features to make a dataset linearly separable

To implement this idea using Scikit-Learn, you can create a pipeline containing a `PolynomialFeatures` transformer (discussed in "Polynomial Regression" on page 149), followed by a `StandardScaler` and a `LinearSVC` classifier. Let's test this on the moons dataset, a toy dataset for binary classification in which the data points are shaped as two interleaving crescent moons (see Figure 5-6). You can generate this dataset using the `make_moons()` function:

```
from sklearn.datasets import make_moons
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

polynomial_svm_clf = make_pipeline(
    PolynomialFeatures(degree=3),
    StandardScaler(),
    LinearSVC(C=10, max_iter=10_000, random_state=42)
)
polynomial_svm_clf.fit(X, y)
```

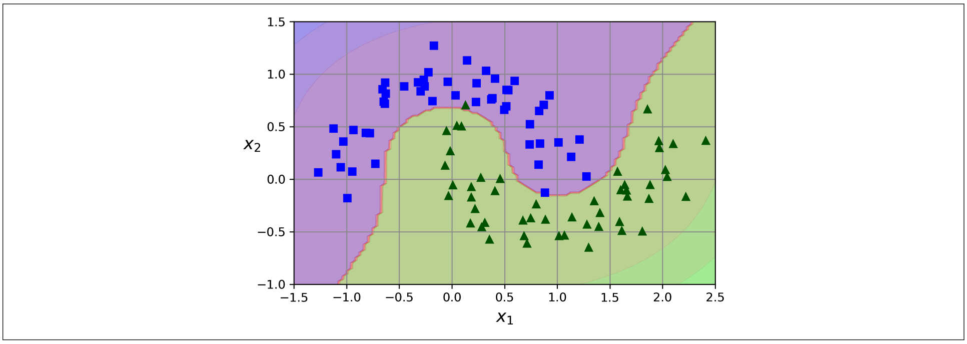

Figure 5-6. Linear SVM classifier using polynomial features

Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of machine learning algorithms (not just SVMs). That said, at a low polynomial degree this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the *kernel trick* (which is explained later in this chapter). The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with a very high degree, without actually having to add them. This means there's no combinatorial explosion of the number of features. This trick is implemented by the `SVC` class. Let's test it on the moons dataset:

```python
from sklearn.svm import SVC

poly_kernel_svm_clf = make_pipeline(StandardScaler(),
                                    SVC(kernel="poly", degree=3, coef0=1, C=5))
poly_kernel_svm_clf.fit(X, y)
```

This code trains an SVM classifier using a third-degree polynomial kernel, represented on the left in Figure 5-7. On the right is another SVM classifier using a 10th-degree polynomial kernel. Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter `coef0` controls how much the model is influenced by high-degree terms versus low-degree terms.

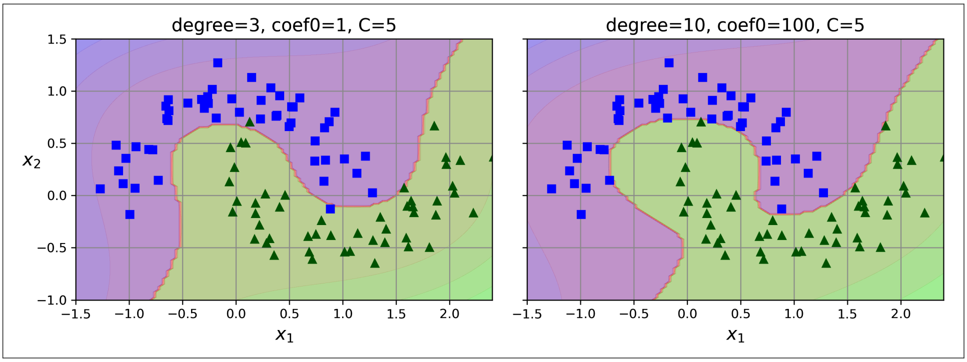

Figure 5-7. SVM classifiers with a polynomial kernel

Although hyperparameters will generally be tuned automatically (e.g., using randomized search), it's good to have a sense of what each hyperparameter actually does and how it may interact with other hyperparameters: this way, you can narrow the search to a much smaller space.

Similarity Features

Another technique to tackle nonlinear problems is to add features computed using a similarity function, which measures how much each instance resembles a particular *landmark*, as we did in Chapter 2 when we added the geographic similarity features. For example, let's take the 1D dataset from earlier and add two landmarks to it at x_1 = –2 and x_1 = 1 (see the left plot in Figure 5-8). Next, we'll define the similarity function to be the Gaussian RBF with $\gamma = 0.3$. This is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark).

Now we are ready to compute the new features. For example, let's look at the instance $x_1 = -1$: it is located at a distance of 1 from the first landmark and 2 from the second landmark. Therefore, its new features are $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ and $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. The plot on the right in Figure 5-8 shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

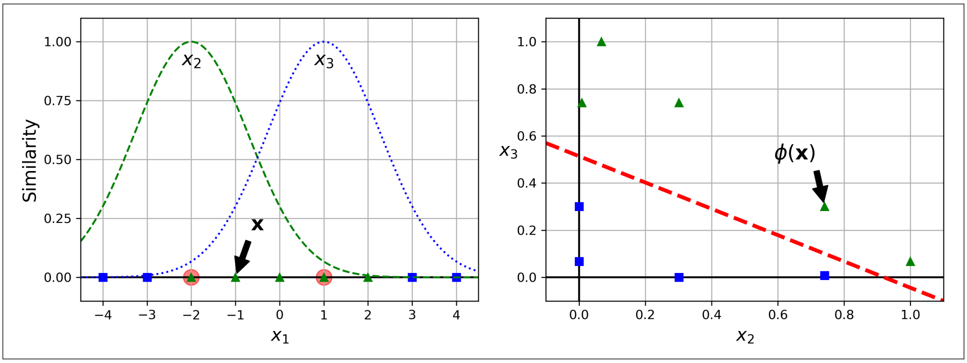

Figure 5-8. Similarity features using the Gaussian RBF

You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset. Doing that creates many dimensions and thus increases the chances that the transformed training set will be linearly separable. The downside is that a training set with *m* instances and *n* features gets transformed into a training set with *m* instances and *m* features (assuming you drop the original features). If your training set is very large, you end up with an equally large number of features.

Gaussian RBF Kernel

Just like the polynomial features method, the similarity features method can be useful with any machine learning algorithm, but it may be computationally expensive to compute all the additional features (especially on large training sets). Once again the kernel trick does its SVM magic, making it possible to obtain a similar result as if you had added many similarity features, but without actually doing so. Let's try the SVC class with the Gaussian RBF kernel:

```
rbf_kernel_svm_clf = make_pipeline(StandardScaler(),
                                   SVC(kernel="rbf", gamma=5, C=0.001))
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented at the bottom left in Figure 5-9. The other plots show models trained with different values of hyperparameters gamma (γ) and C. Increasing gamma makes the bell-shaped curve narrower (see the lefthand plots in Figure 5-8). As a result, each instance's range of influence is smaller: the decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small gamma value makes the bell-shaped curve wider: instances have a larger range of influence, and the decision boundary ends up smoother. So γ acts like a regularization hyperparameter: if your model is overfitting, you should reduce γ; if it is underfitting, you should increase γ (similar to the C hyperparameter).

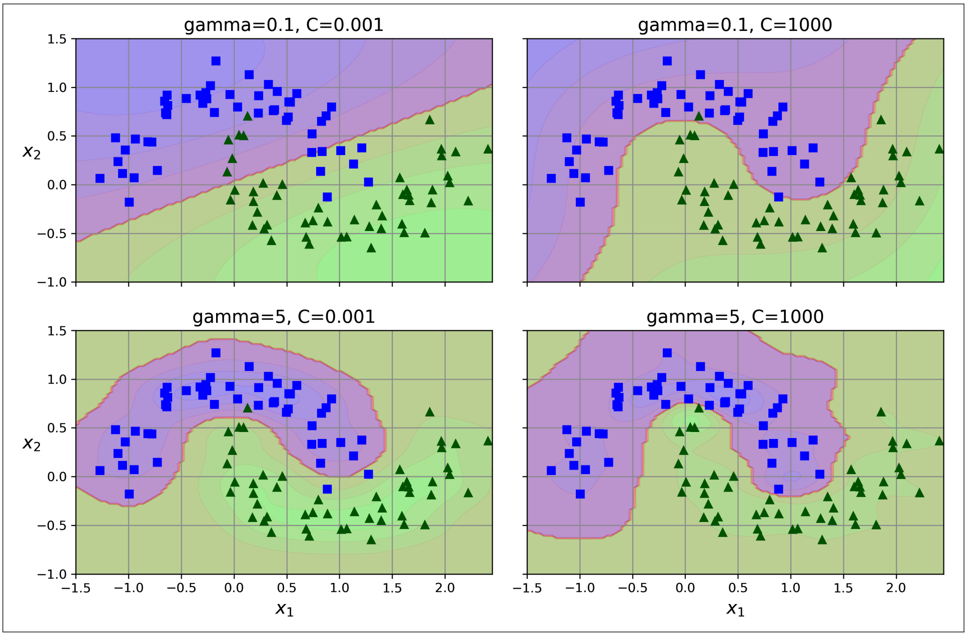

Figure 5-9. SVM classifiers using an RBF kernel

Other kernels exist but are used much more rarely. Some kernels are specialized for specific data structures. *String kernels* are sometimes used when classifying text documents or DNA sequences (e.g., using the string subsequence kernel or kernels based on the Levenshtein distance).

With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first. The `LinearSVC` class is much faster than `SVC(kernel="linear")`, especially if the training set is very large. If it is not too large, you should also try kernelized SVMs, starting with the Gaussian RBF kernel; it often works really well. Then, if you have spare time and computing power, you can experiment with a few other kernels using hyperparameter search. If there are kernels specialized for your training set's data structure, make sure to give them a try too.

SVM Classes and Computational Complexity

The `LinearSVC` class is based on the `liblinear` library, which implements an optimized algorithm for linear SVMs.[1] It does not support the kernel trick, but it scales almost linearly with the number of training instances and the number of features. Its training time complexity is roughly $O(m \times n)$. The algorithm takes longer if you require very high precision. This is controlled by the tolerance hyperparameter ϵ (called `tol` in Scikit-Learn). In most classification tasks, the default tolerance is fine.

The `SVC` class is based on the `libsvm` library, which implements an algorithm that supports the kernel trick.[2] The training time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$. Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (e.g., hundreds of thousands of instances), so this algorithm is best for small or medium-sized nonlinear training sets. It scales well with the number of features, especially with sparse features (i.e., when each instance has few nonzero features). In this case, the algorithm scales roughly with the average number of nonzero features per instance.

The `SGDClassifier` class also performs large margin classification by default, and its hyperparameters–especially the regularization hyperparameters (`alpha` and `penalty`) and the `learning_rate`–can be adjusted to produce similar results as the linear SVMs. For training it uses stochastic gradient descent (see Chapter 4), which allows incremental learning and uses little memory, so you can use it to train a model on a large dataset that does not fit in RAM (i.e., for out-of-core learning). Moreover, it scales very well, as its computational complexity is $O(m \times n)$. Table 5-1 compares Scikit-Learn's SVM classification classes.

[1] Chih-Jen Lin et al., "A Dual Coordinate Descent Method for Large-Scale Linear SVM", *Proceedings of the 25th International Conference on Machine Learning* (2008): 408–415.

[2] John Platt, "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines" (Microsoft Research technical report, April 21, 1998).

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No

Now let's see how the SVM algorithms can also be used for linear and nonlinear regression.

SVM Regression

To use SVMs for regression instead of classification, the trick is to tweak the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM regression tries to fit as many instances as possible *on* the street while limiting margin violations (i.e., instances *off* the street). The width of the street is controlled by a hyperparameter, ϵ. Figure 5-10 shows two linear SVM regression models trained on some linear data, one with a small margin ($\epsilon = 0.5$) and the other with a larger margin ($\epsilon = 1.2$).

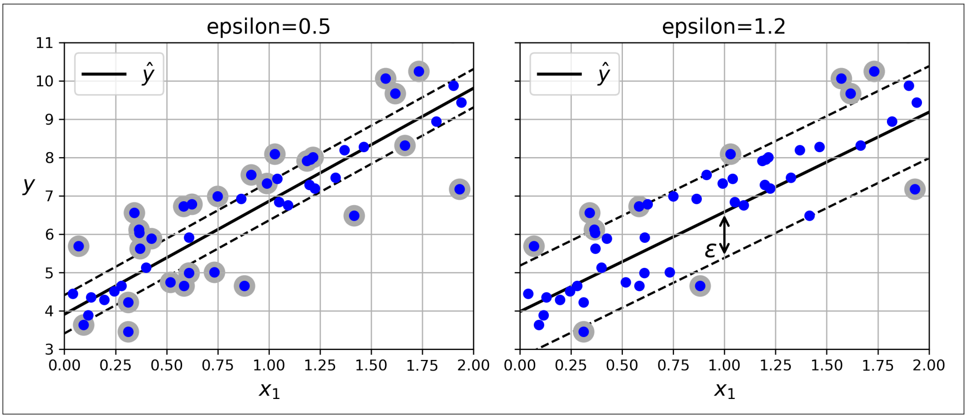

Figure 5-10. SVM regression

Reducing ϵ increases the number of support vectors, which regularizes the model. Moreover, if you add more training instances within the margin, it will not affect the model's predictions; thus, the model is said to be *ϵ-insensitive*.

You can use Scikit-Learn's `LinearSVR` class to perform linear SVM regression. The following code produces the model represented on the left in Figure 5-10:

```
from sklearn.svm import LinearSVR

X, y = [...]  # a linear dataset
svm_reg = make_pipeline(StandardScaler(),
                        LinearSVR(epsilon=0.5, random_state=42))
svm_reg.fit(X, y)
```

To tackle nonlinear regression tasks, you can use a kernelized SVM model. Figure 5-11 shows SVM regression on a random quadratic training set, using a second-degree polynomial kernel. There is some regularization in the left plot (i.e., a small `C` value), and much less in the right plot (i.e., a large `C` value).

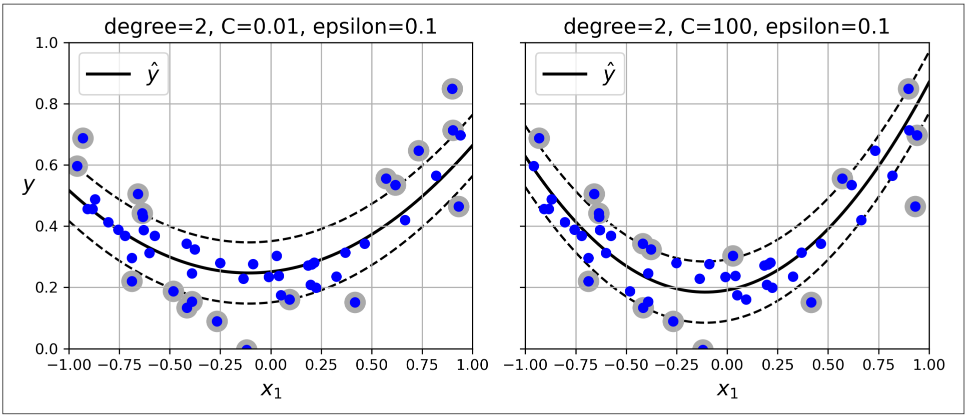

Figure 5-11. SVM regression using a second-degree polynomial kernel

The following code uses Scikit-Learn's `SVR` class (which supports the kernel trick) to produce the model represented on the left in Figure 5-11:

```
from sklearn.svm import SVR

X, y = [...]  # a quadratic dataset
svm_poly_reg = make_pipeline(StandardScaler(),
                             SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1))
svm_poly_reg.fit(X, y)
```

The `SVR` class is the regression equivalent of the `SVC` class, and the `LinearSVR` class is the regression equivalent of the `LinearSVC` class. The `LinearSVR` class scales linearly with the size of the training set (just like the `LinearSVC` class), while the `SVR` class gets much too slow when the training set grows very large (just like the `SVC` class).

SVMs can also be used for novelty detection, as you will see in Chapter 9.

The rest of this chapter explains how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. If you are just getting started with machine learning, you can safely skip this and go straight to the exercises at the end of this chapter, and come back later when you want to get a deeper understanding of SVMs.

Under the Hood of Linear SVM Classifiers

A linear SVM classifier predicts the class of a new instance **x** by first computing the decision function $\boldsymbol{\theta}^\intercal \mathbf{x} = \theta_0 x_0 + \cdots + \theta_n x_n$, where x_0 is the bias feature (always equal to 1). If the result is positive, then the predicted class $\hat{y}$ is the positive class (1); otherwise it is the negative class (0). This is exactly like `LogisticRegression` (discussed in Chapter 4).

> Up to now, I have used the convention of putting all the model parameters in one vector $\boldsymbol{\theta}$, including the bias term $\boldsymbol{\theta}_0$ and the input feature weights $\boldsymbol{\theta}_1$ to $\boldsymbol{\theta}_n$. This required adding a bias input $x_0 = 1$ to all instances. Another very common convention is to separate the bias term b (equal to $\boldsymbol{\theta}_0$) and the feature weights vector **w** (containing $\boldsymbol{\theta}_1$ to $\boldsymbol{\theta}_n$). In this case, no bias feature needs to be added to the input feature vectors, and the linear SVM's decision function is equal to $\mathbf{w}^\intercal \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b$. I will use this convention throughout the rest of this book.

So, making predictions with a linear SVM classifier is quite straightforward. How about training? This requires finding the weights vector **w** and the bias term b that make the street, or margin, as wide as possible while limiting the number of margin violations. Let's start with the width of the street: to make it larger, we need to make **w** smaller. This may be easier to visualize in 2D, as shown in Figure 5-12. Let's define the borders of the street as the points where the decision function is equal to –1 or +1. In the left plot the weight w_1 is 1, so the points at which $w_1 x_1 = -1$ or $+1$ are $x_1 = -1$ and $+1$: therefore the margin's size is 2. In the right plot the weight is 0.5, so the points at which $w_1 x_1 = -1$ or $+1$ are $x_1 = -2$ and $+2$: the margin's size is 4. So, we need to keep **w** as small as possible. Note that the bias term b has no influence on the size of the margin: tweaking it just shifts the margin around, without affecting its size.

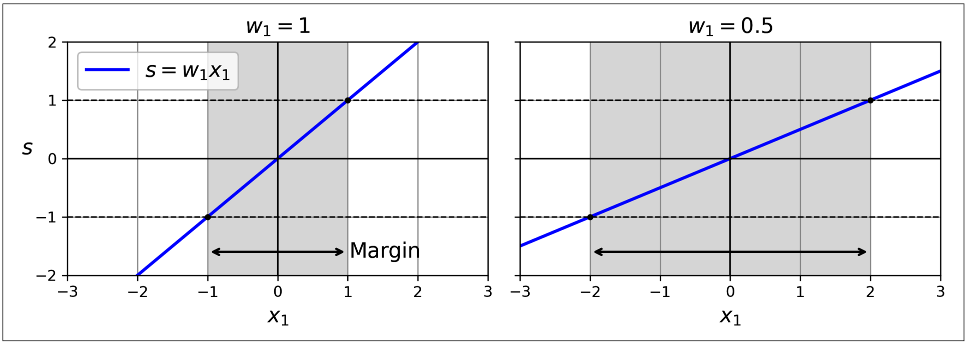

Figure 5-12. A smaller weight vector results in a larger margin

We also want to avoid margin violations, so we need the decision function to be greater than 1 for all positive training instances and lower than −1 for negative training instances. If we define $t^{(i)} = -1$ for negative instances (when $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (when $y^{(i)} = 1$), then we can write this constraint as $t^{(i)}(\mathbf{w}^\intercal \mathbf{x}^{(i)} + b) \geq 1$ for all instances.

We can therefore express the hard margin linear SVM classifier objective as the constrained optimization problem in Equation 5-1.

Equation 5-1. Hard margin linear SVM classifier objective

$$\begin{aligned} &\underset{\mathbf{w}, b}{\text{minimize}} && \frac{1}{2}\mathbf{w}^\intercal \mathbf{w} \\ &\text{subject to} && t^{(i)}\left(\mathbf{w}^\intercal \mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m \end{aligned}$$

We are minimizing ½ $\mathbf{w}^\intercal \mathbf{w}$, which is equal to ½$\| \mathbf{w} \|^2$, rather than minimizing $\| \mathbf{w} \|$ (the norm of $\mathbf{w}$). Indeed, ½$\| \mathbf{w} \|^2$ has a nice, simple derivative (it is just $\mathbf{w}$), while $\| \mathbf{w} \|$ is not differentiable at $\mathbf{w} = 0$. Optimization algorithms often work much better on differentiable functions.

To get the soft margin objective, we need to introduce a *slack variable* $\zeta^{(i)} \geq 0$ for each instance:[3] $\zeta^{(i)}$ measures how much the i^{th} instance is allowed to violate the margin. We now have two conflicting objectives: make the slack variables as small as possible to reduce the margin violations, and make ½ $\mathbf{w}^\intercal\ \mathbf{w}$ as small as possible to increase the margin. This is where the `C` hyperparameter comes in: it allows us to define the trade-off between these two objectives. This gives us the constrained optimization problem in Equation 5-2.

Equation 5-2. Soft margin linear SVM classifier objective

$$\begin{aligned}&\underset{\mathbf{w}, b, \zeta}{\text{minimize}} && \frac{1}{2}\mathbf{w}^\intercal\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}\\ &\text{subject to} && t^{(i)}\left(\mathbf{w}^\intercal\mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m\end{aligned}$$

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. Such problems are known as *quadratic programming* (QP) problems. Many off-the-shelf solvers are available to solve QP problems by using a variety of techniques that are outside the scope of this book.[4]

Using a QP solver is one way to train an SVM. Another is to use gradient descent to minimize the *hinge loss* or the *squared hinge loss* (see Figure 5-13). Given an instance $\mathbf{x}$ of the positive class (i.e., with $t = 1$), the loss is 0 if the output s of the decision function ($s = \mathbf{w}^\intercal\ \mathbf{x} + b$) is greater than or equal to 1. This happens when the instance is off the street and on the positive side. Given an instance of the negative class (i.e., with $t = -1$), the loss is 0 if $s \leq -1$. This happens when the instance is off the street and on the negative side. The further away an instance is from the correct side of the margin, the higher the loss: it grows linearly for the hinge loss, and quadratically for the squared hinge loss. This makes the squared hinge loss more sensitive to outliers. However, if the dataset is clean, it tends to converge faster. By default, `LinearSVC` uses the squared hinge loss, while `SGDClassifier` uses the hinge loss. Both classes let you choose the loss by setting the `loss` hyperparameter to `"hinge"` or `"squared_hinge"`. The `SVC` class's optimization algorithm finds a similar solution as minimizing the hinge loss.

[3] Zeta (ζ) is the sixth letter of the Greek alphabet.

[4] To learn more about quadratic programming, you can start by reading Stephen Boyd and Lieven Vandenberghe's book *Convex Optimization* (Cambridge University Press) or watching Richard Brown's series of video lectures.

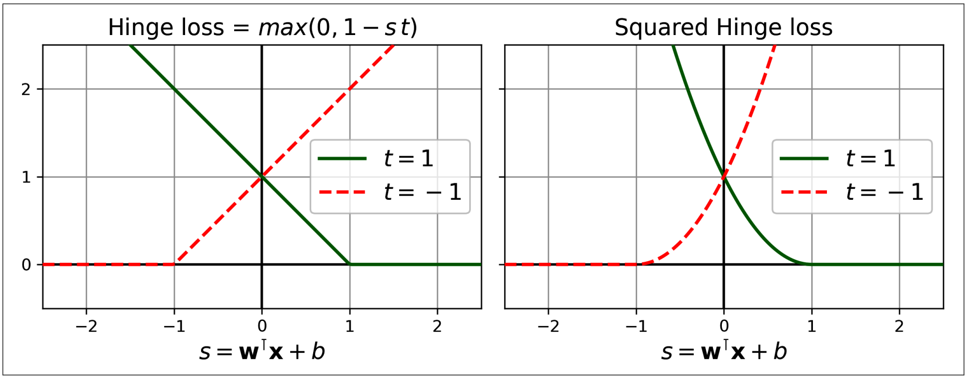

Figure 5-13. The hinge loss (left) and the squared hinge loss (right)

Next, we'll look at yet another way to train a linear SVM classifier: solving the dual problem.

The Dual Problem

Given a constrained optimization problem, known as the *primal problem*, it is possible to express a different but closely related problem, called its *dual problem*. The solution to the dual problem typically gives a lower bound to the solution of the primal problem, but under some conditions it can have the same solution as the primal problem. Luckily, the SVM problem happens to meet these conditions,[5] so you can choose to solve the primal problem or the dual problem; both will have the same solution. Equation 5-3 shows the dual form of the linear SVM objective. If you are interested in knowing how to derive the dual problem from the primal problem, see the extra material section in this chapter's notebook.

Equation 5-3. Dual form of the linear SVM objective

$$\underset{\boldsymbol{\alpha}}{\text{minimize}} \quad \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\mathbf{x}^{(i)\top}\mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m}\alpha^{(i)}$$

$$\text{subject to } \alpha^{(i)} \geq 0 \text{ for all } i = 1, 2, \ldots, m \text{ and } \sum_{i=1}^{m}\alpha^{(i)}t^{(i)} = 0$$

[5] The objective function is convex, and the inequality constraints are continuously differentiable and convex functions.

Once you find the vector $\hat{\boldsymbol{\alpha}}$ that minimizes this equation (using a QP solver), use Equation 5-4 to compute the $\hat{\mathbf{w}}$ and $\hat{b}$ that minimize the primal problem. In this equation, n_s represents the number of support vectors.

Equation 5-4. From the dual solution to the primal solution

$$\hat{\mathbf{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(t^{(i)} - \hat{\mathbf{w}}^\top \mathbf{x}^{(i)} \right)$$

The dual problem is faster to solve than the primal one when the number of training instances is smaller than the number of features. More importantly, the dual problem makes the kernel trick possible, while the primal problem does not. So what is this kernel trick, anyway?

Kernelized SVMs

Suppose you want to apply a second-degree polynomial transformation to a two-dimensional training set (such as the moons training set), then train a linear SVM classifier on the transformed training set. Equation 5-5 shows the second-degree polynomial mapping function ϕ that you want to apply.

Equation 5-5. Second-degree polynomial mapping

$$\varphi(\mathbf{x}) = \varphi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} {x_1}^2 \\ \sqrt{2}\, x_1 x_2 \\ {x_2}^2 \end{pmatrix}$$

Notice that the transformed vector is 3D instead of 2D. Now let's look at what happens to a couple of 2D vectors, **a** and **b**, if we apply this second-degree polynomial mapping and then compute the dot product[6] of the transformed vectors (see Equation 5-6).

[6] As explained in Chapter 4, the dot product of two vectors **a** and **b** is normally noted **a** · **b**. However, in machine learning, vectors are frequently represented as column vectors (i.e., single-column matrices), so the dot product is achieved by computing $\mathbf{a}^\top\mathbf{b}$. To remain consistent with the rest of the book, we will use this notation here, ignoring the fact that this technically results in a single-cell matrix rather than a scalar value.

Equation 5-6. Kernel trick for a second-degree polynomial mapping

$$\begin{aligned}
\varphi(\mathbf{a})^\intercal\varphi(\mathbf{b}) \quad &= \begin{pmatrix} {a_1}^2 \\ \sqrt{2}\, a_1 a_2 \\ {a_2}^2 \end{pmatrix}^\intercal \begin{pmatrix} {b_1}^2 \\ \sqrt{2}\, b_1 b_2 \\ {b_2}^2 \end{pmatrix} = {a_1}^2{b_1}^2 + 2a_1 b_1 a_2 b_2 + {a_2}^2{b_2}^2 \\
&= \left(a_1 b_1 + a_2 b_2\right)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^\intercal \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)^2 = \left(\mathbf{a}^\intercal\mathbf{b}\right)^2
\end{aligned}$$

How about that? The dot product of the transformed vectors is equal to the square of the dot product of the original vectors: $\phi(\mathbf{a})^\intercal\ \phi(\mathbf{b}) = (\mathbf{a}^\intercal\ \mathbf{b})^2$.

Here is the key insight: if you apply the transformation ϕ to all training instances, then the dual problem (see Equation 5-3) will contain the dot product $\phi(\mathbf{x}^{(i)})^\intercal\ \phi(\mathbf{x}^{(j)})$. But if ϕ is the second-degree polynomial transformation defined in Equation 5-5, then you can replace this dot product of transformed vectors simply by $\left(\mathbf{x}^{(i)\intercal}\mathbf{x}^{(j)}\right)^2$. So, you don't need to transform the training instances at all; just replace the dot product by its square in Equation 5-3. The result will be strictly the same as if you had gone through the trouble of transforming the training set and then fitting a linear SVM algorithm, but this trick makes the whole process much more computationally efficient.

The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^\intercal\ \mathbf{b})^2$ is a second-degree polynomial kernel. In machine learning, a *kernel* is a function capable of computing the dot product $\phi(\mathbf{a})^\intercal\ \phi(\mathbf{b})$, based only on the original vectors **a** and **b**, without having to compute (or even to know about) the transformation ϕ. Equation 5-7 lists some of the most commonly used kernels.

Equation 5-7. Common kernels

$$\begin{aligned}
\text{Linear:} \quad & K(\mathbf{a},\mathbf{b}) = \mathbf{a}^\intercal\mathbf{b} \\
\text{Polynomial:} \quad & K(\mathbf{a},\mathbf{b}) = \left(\gamma\mathbf{a}^\intercal\mathbf{b} + r\right)^d \\
\text{Gaussian RBF:} \quad & K(\mathbf{a},\mathbf{b}) = \exp\left(-\gamma \|\, \mathbf{a} - \mathbf{b} \,\|^2\right) \\
\text{Sigmoid:} \quad & K(\mathbf{a},\mathbf{b}) = \tanh\left(\gamma\mathbf{a}^\intercal\mathbf{b} + r\right)
\end{aligned}$$

Mercer's Theorem

According to *Mercer's theorem*, if a function $K(\mathbf{a}, \mathbf{b})$ respects a few mathematical conditions called *Mercer's conditions* (e.g., K must be continuous and symmetric in its arguments so that $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), then there exists a function ϕ that maps $\mathbf{a}$ and $\mathbf{b}$ into another space (possibly with much higher dimensions) such that $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^\intercal \phi(\mathbf{b})$. You can use K as a kernel because you know ϕ exists, even if you don't know what ϕ is. In the case of the Gaussian RBF kernel, it can be shown that ϕ maps each training instance to an infinite-dimensional space, so it's a good thing you don't need to actually perform the mapping!

Note that some frequently used kernels (such as the sigmoid kernel) don't respect all of Mercer's conditions, yet they generally work well in practice.

There is still one loose end we must tie up. Equation 5-4 shows how to go from the dual solution to the primal solution in the case of a linear SVM classifier. But if you apply the kernel trick, you end up with equations that include $\phi(x^{(i)})$. In fact, $\widehat{\mathbf{w}}$ must have the same number of dimensions as $\phi(x^{(i)})$, which may be huge or even infinite, so you can't compute it. But how can you make predictions without knowing $\widehat{\mathbf{w}}$? Well, the good news is that you can plug the formula for $\widehat{\mathbf{w}}$ from Equation 5-4 into the decision function for a new instance $\mathbf{x}^{(n)}$, and you get an equation with only dot products between input vectors. This makes it possible to use the kernel trick (Equation 5-8).

Equation 5-8. Making predictions with a kernelized SVM

$$\begin{aligned}
h_{\widehat{\mathbf{w}}, \widehat{b}}\left(\varphi\left(\mathbf{x}^{(n)}\right)\right) &= \widehat{\mathbf{w}}^\intercal \varphi\left(\mathbf{x}^{(n)}\right) + \widehat{b} = \left(\sum_{i=1}^{m} \widehat{\alpha}^{(i)} t^{(i)} \varphi\left(\mathbf{x}^{(i)}\right)\right)^\intercal \varphi\left(\mathbf{x}^{(n)}\right) + \widehat{b} \\
&= \sum_{i=1}^{m} \widehat{\alpha}^{(i)} t^{(i)} \left(\varphi\left(\mathbf{x}^{(i)}\right)^\intercal \varphi\left(\mathbf{x}^{(n)}\right)\right) + \widehat{b} \\
&= \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^{m} \widehat{\alpha}^{(i)} t^{(i)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}\right) + \widehat{b}
\end{aligned}$$

Note that since $\alpha^{(i)} \neq 0$ only for support vectors, making predictions involves computing the dot product of the new input vector $\mathbf{x}^{(n)}$ with only the support vectors, not all the training instances. Of course, you need to use the same trick to compute the bias term $\widehat{b}$ (Equation 5-9).

Equation 5-9. Using the kernel trick to compute the bias term

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1\\ \hat{\alpha}^{(i)}>0}}^{m} \left(t^{(i)} - \hat{\mathbf{w}}^\top \varphi\left(\mathbf{x}^{(i)}\right)\right) = \frac{1}{n_s}\sum_{\substack{i=1\\ \hat{\alpha}^{(i)}>0}}^{m} \left(t^{(i)} - \left(\sum_{j=1}^{m} \hat{\alpha}^{(j)} t^{(j)} \varphi\left(\mathbf{x}^{(j)}\right)\right)^\top \varphi\left(\mathbf{x}^{(i)}\right)\right)$$

$$= \frac{1}{n_s}\sum_{\substack{i=1\\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \sum_{\substack{j=1\\ \hat{\alpha}^{(j)}>0}}^{m} \hat{\alpha}^{(j)} t^{(j)} K\left(\mathbf{x}^{(i)},\mathbf{x}^{(j)}\right)\right)$$

If you are starting to get a headache, that's perfectly normal: it's an unfortunate side effect of the kernel trick.

It is also possible to implement online kernelized SVMs, capable of incremental learning, as described in the papers "Incremental and Decremental Support Vector Machine Learning"[7] and "Fast Kernel Classifiers with Online and Active Learning".[8] These kernelized SVMs are implemented in Matlab and C++. But for large-scale nonlinear problems, you may want to consider using random forests (see Chapter 7) or neural networks (see Part II).

Exercises

1. What is the fundamental idea behind support vector machines?
2. What is a support vector?
3. Why is it important to scale the inputs when using SVMs?
4. Can an SVM classifier output a confidence score when it classifies an instance? What about a probability?
5. How can you choose between `LinearSVC`, `SVC`, and `SGDClassifier`?
6. Say you've trained an SVM classifier with an RBF kernel, but it seems to underfit the training set. Should you increase or decrease γ (`gamma`)? What about `C`?
7. What does it mean for a model to be *ϵ-insensitive*?
8. What is the point of using the kernel trick?

[7] Gert Cauwenberghs and Tomaso Poggio, "Incremental and Decremental Support Vector Machine Learning", *Proceedings of the 13th International Conference on Neural Information Processing Systems* (2000): 388–394.

[8] Antoine Bordes et al., "Fast Kernel Classifiers with Online and Active Learning", *Journal of Machine Learning Research* 6 (2005): 1579–1619.

9. Train a `LinearSVC` on a linearly separable dataset. Then train an `SVC` and a `SGDClassifier` on the same dataset. See if you can get them to produce roughly the same model.

10. Train an SVM classifier on the wine dataset, which you can load using `sklearn.datasets.load_wine()`. This dataset contains the chemical analyses of 178 wine samples produced by 3 different cultivators: the goal is to train a classification model capable of predicting the cultivator based on the wine's chemical analysis. Since SVM classifiers are binary classifiers, you will need to use one-versus-all to classify all three classes. What accuracy can you reach?

11. Train and fine-tune an SVM regressor on the California housing dataset. You can use the original dataset rather than the tweaked version we used in Chapter 2, which you can load using `sklearn.datasets.fetch_california_housing()`. The targets represent hundreds of thousands of dollars. Since there are over 20,000 instances, SVMs can be slow, so for hyperparameter tuning you should use far fewer instances (e.g., 2,000) to test many more hyperparameter combinations. What is your best model's RMSE?

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab3*.

CHAPTER 6

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 7), 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 6-1.

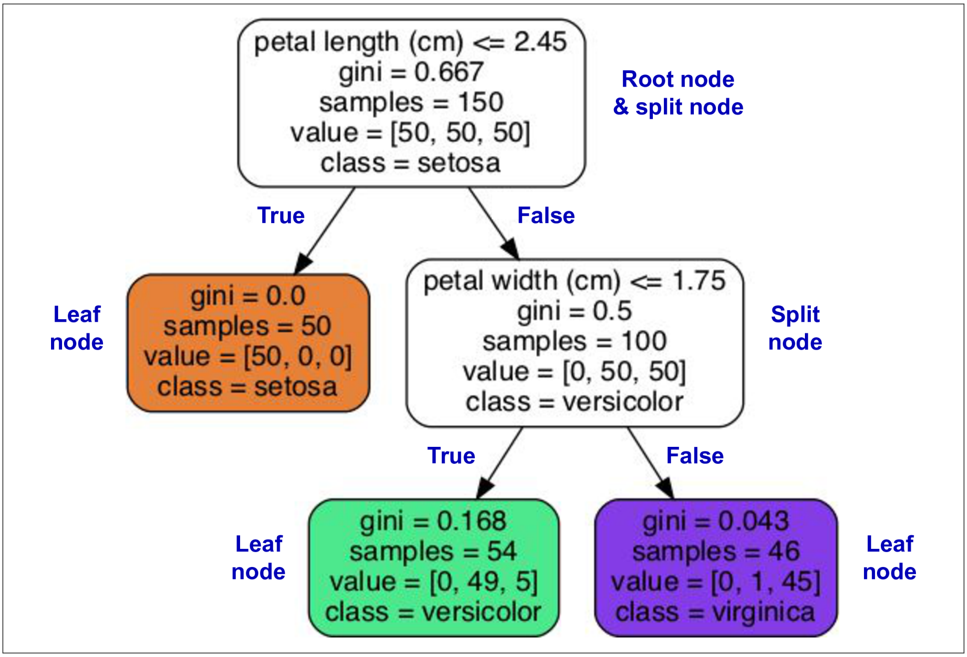

Figure 6-1. Iris decision tree

Making Predictions

Let's see how the tree represented in Figure 6-1 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, it is pure and its Gini impurity is 0. Equation 6-1 shows how the training algorithm computes the Gini impurity G_i of the i^{th} node. 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 6-1. 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, 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 6-2 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).

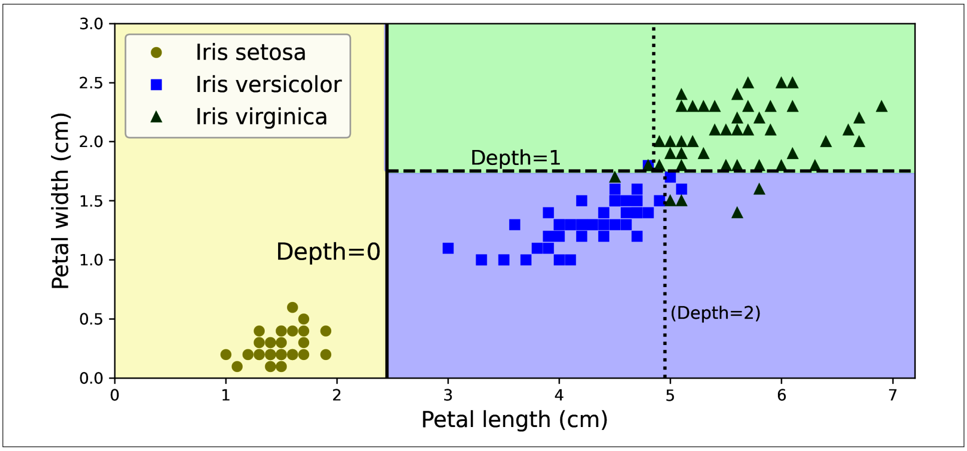

Figure 6-2. Decision tree decision boundaries

> The tree structure, including all the information shown in Figure 6-1, is available via the classifier's `tree_` attribute. Type **`help(tree_clf.tree_)`** for details, and see the 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, to ensure the system does not make unfair decisions.

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 ratio of training instances of class k in this 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 6-2—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 6-2 gives the cost function that the algorithm tries to minimize.

Equation 6-2. CART cost function for classification

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

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

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`, `min_weight_fraction_leaf`, and `max_leaf_nodes`.

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 or not 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, making the problem intractable even for small training sets. This is why we must settle for a "reasonably good" solution when training decision trees.

Computational Complexity

Making predictions requires traversing the decision tree from the root to a leaf. Decision trees generally are approximately balanced, so traversing the decision tree requires going through roughly $O(\log_2(m))$ nodes, where $\log_2(m)$ is the *binary logarithm* of m, equal to $\log(m) / \log(2)$. Since each node only requires checking the

[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 or not 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).

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.

The training algorithm compares all features (or less if `max_features` is set) on all samples at each node. Comparing all features on all samples at each node results in a training complexity of $O(n \times m \log_2(m))$.

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 6-3 shows the definition of the entropy of the i^{th} node. For example, the depth-2 left node in Figure 6-1 has an entropy equal to $-(49/54) \log_2 (49/54) - (5/54) \log_2 (5/54) \approx 0.445$.

Equation 6-3. 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.[2]

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

[2] See Sebastian Raschka's interesting analysis for more details.

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

Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.

> 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, introduced in Chapter 5. We'll train one decision tree without regularization, and another with `min_samples_leaf=5`. Here's the code; Figure 6-3 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)
```

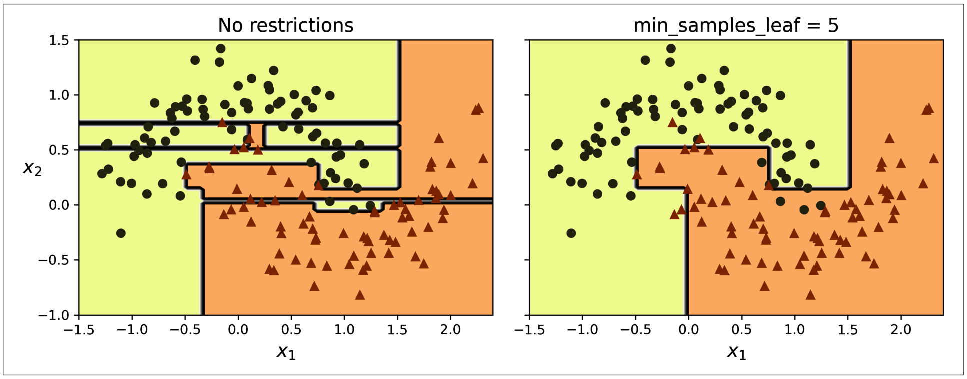

Figure 6-3. Decision boundaries of an unregularized tree (left) and a regularized tree (right)

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. 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

np.random.seed(42)
X_quad = np.random.rand(200, 1) - 0.5  # a single random input feature
y_quad = X_quad ** 2 + 0.025 * np.random.randn(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 6-4.

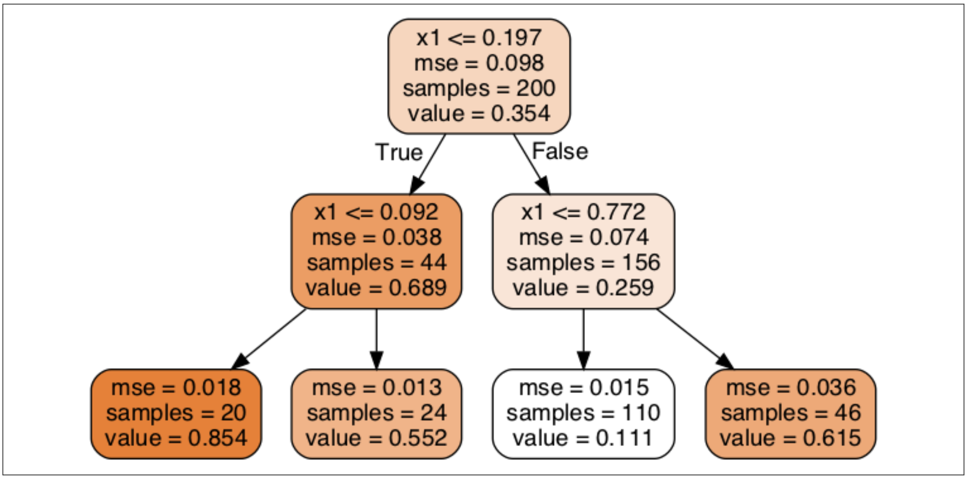

Figure 6-4. A decision tree for regression

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 $x_1 \leq 0.197$. Since it is not, the algorithm goes to the right child node, which asks whether $x_1 \leq 0.772$. Since it is, the algorithm goes to the left child node. This is a leaf node, and it predicts `value=0.111`. This prediction is the average target value of the 110 training instances associated with this leaf node, and it results in a mean squared error equal to 0.015 over these 110 instances.

This model's predictions are represented on the left in Figure 6-5. 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.

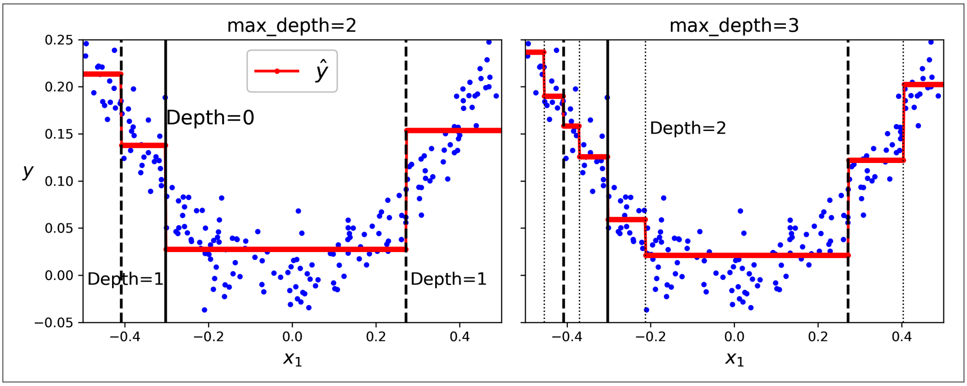

Figure 6-5. Predictions of two decision tree regression models

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 6-4 shows the cost function that the algorithm tries to minimize.

Equation 6-4. 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 6-6. 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 6-6.

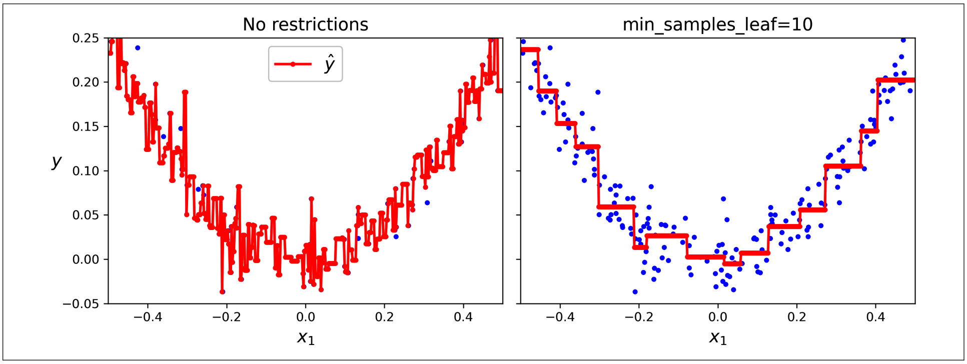

Figure 6-6. Predictions of an unregularized regression tree (left) and a regularized tree (right)

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 6-7 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.

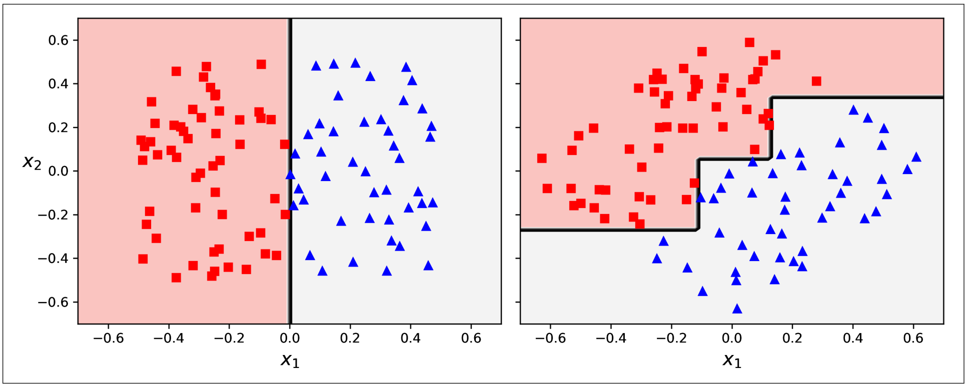

Figure 6-7. Sensitivity to training set rotation

One way to limit this problem is to scale the data, then apply a principal component analysis transformation. We will look at PCA in detail in Chapter 8, 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 6-8 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)
```

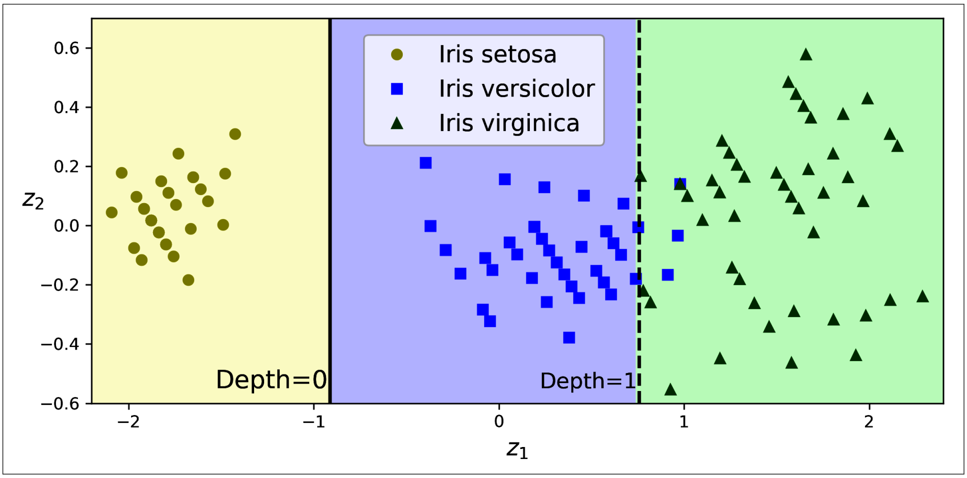

Figure 6-8. A tree's decision boundaries on the scaled and PCA-rotated iris dataset

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 6-9 (unless you set the `random_state` hyperparameter). As you can see, it looks very different from the previous decision tree (Figure 6-2).

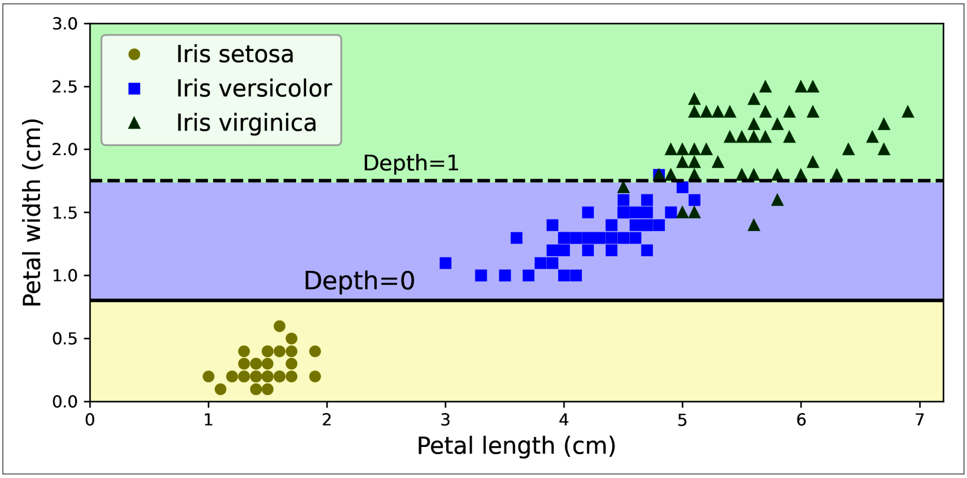

Figure 6-9. Retraining the same model on the same data may produce a very different model

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/colab3*.