

Figure 5-10. SVM Regression

Adding more training instances within the margin does 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 of Figure 5-10 (the training data should be scaled and centered first):

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

To tackle nonlinear regression tasks, you can use a kernelized SVM model. For example, Figure 5-11 shows SVM Regression on a random quadratic training set, using a 2nd-degree polynomial kernel. There is little regularization on the left plot (i.e., a large C value), and much more regularization on the right plot (i.e., a small C value).

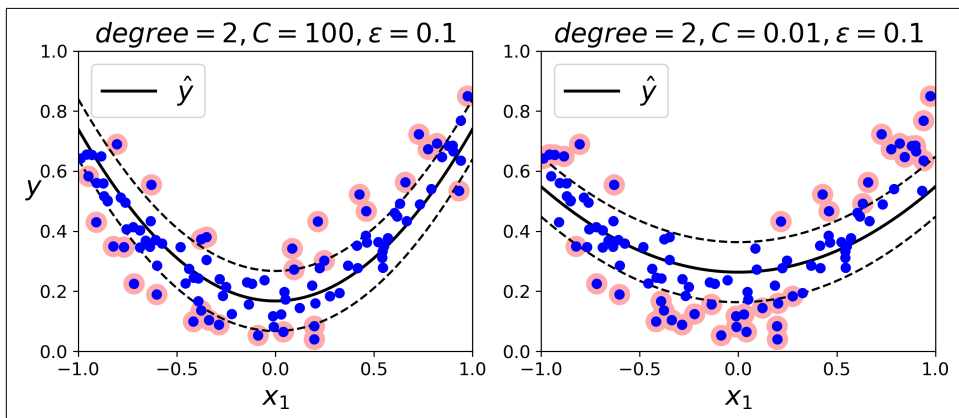


Figure 5-11. SVM regression using a 2nd-degree polynomial kernel

The following code produces the model represented on the left of [Figure 5-11](#) using Scikit-Learn's SVR class (which supports the kernel trick). 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 large (just like the SVC class).

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



SVMs can also be used for outlier detection; see Scikit-Learn's documentation for more details.

Under the Hood

This section explains how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. You can safely skip it and go straight to the exercises at the end of this chapter if you are just getting started with Machine Learning, and come back later when you want to get a deeper understanding of SVMs.

First, a word about notations: in [Chapter 4](#) we used the convention of putting all the model parameters in one vector θ , including the bias term θ_0 and the input feature weights θ_1 to θ_n , and adding a bias input $x_0 = 1$ to all instances. In this chapter, we will use a different convention, which is more convenient (and more common) when you are dealing with SVMs: the bias term will be called b and the feature weights vector will be called \mathbf{w} . No bias feature will be added to the input feature vectors.

Decision Function and Predictions

The linear SVM classifier model predicts the class of a new instance \mathbf{x} by simply computing the decision function $\mathbf{w}^T \mathbf{x} + b = w_1 x_1 + \dots + w_n x_n + b$: if the result is positive, the predicted class \hat{y} is the positive class (1), or else it is the negative class (0); see [Equation 5-2](#).

Equation 5-2. Linear SVM classifier prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^T \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^T \mathbf{x} + b \geq 0 \end{cases}$$

Figure 5-12 shows the decision function that corresponds to the model on the left of Figure 5-4: it is a two-dimensional plane since this dataset has two features (petal width and petal length). The decision boundary is the set of points where the decision function is equal to 0: it is the intersection of two planes, which is a straight line (represented by the thick solid line).³

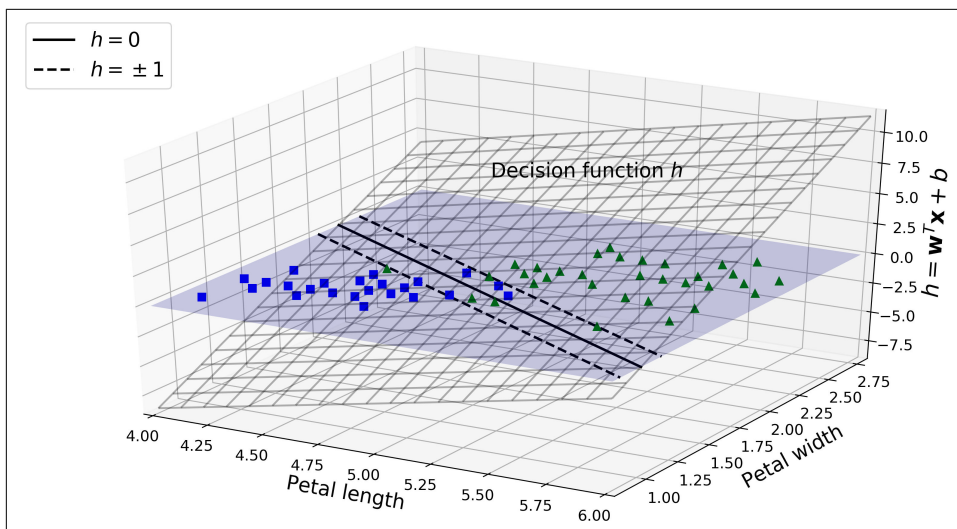


Figure 5-12. Decision function for the iris dataset

The dashed lines represent the points where the decision function is equal to 1 or -1: they are parallel and at equal distance to the decision boundary, forming a margin around it. Training a linear SVM classifier means finding the value of \mathbf{w} and b that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

Training Objective

Consider the slope of the decision function: it is equal to the norm of the weight vector, $\|\mathbf{w}\|$. If we divide this slope by 2, the points where the decision function is equal to ± 1 are going to be twice as far away from the decision boundary. In other words, dividing the slope by 2 will multiply the margin by 2. Perhaps this is easier to visualize in 2D in Figure 5-13. The smaller the weight vector \mathbf{w} , the larger the margin.

³ More generally, when there are n features, the decision function is an n -dimensional *hyperplane*, and the decision boundary is an $(n - 1)$ -dimensional hyperplane.

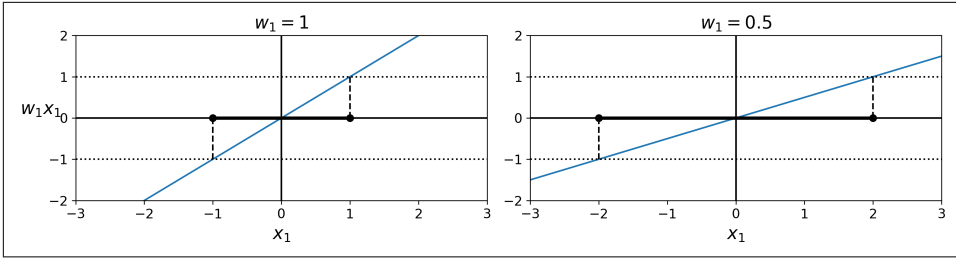


Figure 5-13. A smaller weight vector results in a larger margin

So we want to minimize $\| \mathbf{w} \|$ to get a large margin. However, if we also want to avoid any margin violation (hard margin), then 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 (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express this constraint as $t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1$ for all instances.

We can therefore express the hard margin linear SVM classifier objective as the *constrained optimization* problem in [Equation 5-3](#).

Equation 5-3. Hard margin linear SVM classifier objective

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



We are minimizing $\frac{1}{2} \mathbf{w}^T \mathbf{w}$, which is equal to $\frac{1}{2} \| \mathbf{w} \|^2$, rather than minimizing $\| \mathbf{w} \|$. Indeed, $\frac{1}{2} \| \mathbf{w} \|^2$ has a nice and simple derivative (it is just \mathbf{w}) while $\| \mathbf{w} \|$ is not differentiable at $\mathbf{w} = \mathbf{0}$. Optimization algorithms 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:⁴ $\zeta^{(i)}$ measures how much the i^{th} instance is allowed to violate the margin. We now have two conflicting objectives: making the slack variables as small as possible to reduce the margin violations, and making $\frac{1}{2} \mathbf{w}^T \mathbf{w}$ as small as possible to increase the margin. This is where the C hyperparameter comes in: it allows us to define the trade-

⁴ Zeta (ζ) is the 6th letter of the Greek alphabet.

off between these two objectives. This gives us the constrained optimization problem in [Equation 5-4](#).

Equation 5-4. Soft margin linear SVM classifier objective

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

Quadratic Programming

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 using a variety of techniques that are outside the scope of this book.⁵ The general problem formulation is given by [Equation 5-5](#).

Equation 5-5. Quadratic Programming problem

$$\begin{aligned} & \underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2} \mathbf{p}^T \mathbf{H} \mathbf{p} + \mathbf{f}^T \mathbf{p} \\ & \text{subject to} \quad \mathbf{A} \mathbf{p} \leq \mathbf{b} \end{aligned}$$

where

$$\left\{ \begin{array}{l} \mathbf{p} \text{ is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}), \\ \mathbf{H} \text{ is an } n_p \times n_p \text{ matrix,} \\ \mathbf{f} \text{ is an } n_p\text{-dimensional vector,} \\ \mathbf{A} \text{ is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\ \mathbf{b} \text{ is an } n_c\text{-dimensional vector.} \end{array} \right.$$

Note that the expression $\mathbf{A} \mathbf{p} \leq \mathbf{b}$ actually defines n_c constraints: $\mathbf{p}^T \mathbf{a}^{(i)} \leq b^{(i)}$ for $i = 1, 2, \dots, n_c$, where $\mathbf{a}^{(i)}$ is the vector containing the elements of the i^{th} row of \mathbf{A} and $b^{(i)}$ is the i^{th} element of \mathbf{b} .

You can easily verify that if you set the QP parameters in the following way, you get the hard margin linear SVM classifier objective:

- $n_p = n + 1$, where n is the number of features (the +1 is for the bias term).

⁵ To learn more about Quadratic Programming, you can start by reading Stephen Boyd and Lieven Vandenberghe, *Convex Optimization* (Cambridge, UK: Cambridge University Press, 2004) or watch Richard Brown's [series of video lectures](#).

- $n_c = m$, where m is the number of training instances.
- \mathbf{H} is the $n_p \times n_p$ identity matrix, except with a zero in the top-left cell (to ignore the bias term).
- $\mathbf{f} = \mathbf{0}$, an n_p -dimensional vector full of 0s.
- $\mathbf{b} = -\mathbf{1}$, an n_c -dimensional vector full of -1s.
- $\mathbf{a}^{(i)} = -t^{(i)} \dot{\mathbf{x}}^{(i)}$, where $\dot{\mathbf{x}}^{(i)}$ is equal to $\mathbf{x}^{(i)}$ with an extra bias feature $\dot{\mathbf{x}}_0 = 1$.

So one way to train a hard margin linear SVM classifier is just to use an off-the-shelf QP solver by passing it the preceding parameters. The resulting vector \mathbf{p} will contain the bias term $b = p_0$ and the feature weights $w_i = p_i$ for $i = 1, 2, \dots, n$. Similarly, you can use a QP solver to solve the soft margin problem (see the exercises at the end of the chapter).

However, to use the kernel trick we are going to look at a different constrained optimization 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 even have the same solutions as the primal problem. Luckily, the SVM problem happens to meet these conditions,⁶ so you can choose to solve the primal problem or the dual problem; both will have the same solution. Equation 5-6 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 ???).

Equation 5-6. Dual form of the linear SVM objective

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^m \alpha^{(i)} \\ \text{subject to} \quad & \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

⁶ The objective function is convex, and the inequality constraints are continuously differentiable and convex functions.

Once you find the vector $\hat{\alpha}$ that minimizes this equation (using a QP solver), you can compute $\hat{\mathbf{w}}$ and \hat{b} that minimize the primal problem by using [Equation 5-7](#).

Equation 5-7. From the dual solution to the primal solution

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

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

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

Kernelized SVM

Suppose you want to apply a 2nd-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 5-8](#) shows the 2nd-degree polynomial mapping function ϕ that you want to apply.

Equation 5-8. Second-degree polynomial mapping

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

Notice that the transformed vector is three-dimensional instead of two-dimensional. Now let's look at what happens to a couple of two-dimensional vectors, \mathbf{a} and \mathbf{b} , if we apply this 2nd-degree polynomial mapping and then compute the dot product⁷ of the transformed vectors (See [Equation 5-9](#)).

⁷ As explained in [Chapter 4](#), the dot product of two vectors \mathbf{a} and \mathbf{b} is normally noted $\mathbf{a} \cdot \mathbf{b}$. However, in Machine Learning, vectors are frequently represented as column vectors (i.e., single-column matrices), so the dot product is achieved by computing $\mathbf{a}^T \mathbf{b}$. 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 5-9. Kernel trick for a 2nd-degree polynomial mapping

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

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

Now here is the key insight: if you apply the transformation ϕ to all training instances, then the dual problem (see Equation 5-6) will contain the dot product $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$. But if ϕ is the 2nd-degree polynomial transformation defined in Equation 5-8, then you can replace this dot product of transformed vectors simply by $\left(\mathbf{x}^{(i)T} \mathbf{x}^{(j)} \right)^2$. So you don't actually need to transform the training instances at all: just replace the dot product by its square in Equation 5-6. The result will be strictly the same as if you went through the trouble of actually transforming the training set then fitting a linear SVM algorithm, but this trick makes the whole process much more computationally efficient. This is the essence of the kernel trick.

The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$ is called a 2nd-degree *polynomial kernel*. In Machine Learning, a *kernel* is a function capable of computing the dot product $\phi(\mathbf{a})^T \phi(\mathbf{b})$ based only on the original vectors \mathbf{a} and \mathbf{b} , without having to compute (or even to know about) the transformation ϕ . Equation 5-10 lists some of the most commonly used kernels.

Equation 5-10. Common kernels

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

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* (K must be continuous, symmetric in its arguments so $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), then there exists a function ϕ that maps \mathbf{a} and \mathbf{b} into another space (possibly with much higher dimensions) such that $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \phi(\mathbf{b})$. So you can use K as a kernel since 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 actually 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. [Equation 5-7](#) 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 $\phi(\mathbf{x}^{(i)})$. In fact, $\hat{\mathbf{w}}$ must have the same number of dimensions as $\phi(\mathbf{x}^{(i)})$, which may be huge or even infinite, so you can't compute it. But how can you make predictions without knowing $\hat{\mathbf{w}}$? Well, the good news is that you can plug in the formula for $\hat{\mathbf{w}}$ from [Equation 5-7](#) 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, once again ([Equation 5-11](#)).

Equation 5-11. Making predictions with a kernelized SVM

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

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 also need to compute the bias term \hat{b} , using the same trick ([Equation 5-12](#)).

Equation 5-12. Computing the bias term using the kernel trick

$$\begin{aligned}\hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \hat{\mathbf{w}}^T \phi(\mathbf{x}^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \left(\sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^T \phi(\mathbf{x}^{(i)}) \right) \\ &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^m \hat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right)\end{aligned}$$

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

Online SVMs

Before concluding this chapter, let's take a quick look at online SVM classifiers (recall that online learning means learning incrementally, typically as new instances arrive).

For linear SVM classifiers, one method is to use Gradient Descent (e.g., using `SGDClassifier`) to minimize the cost function in [Equation 5-13](#), which is derived from the primal problem. Unfortunately it converges much more slowly than the methods based on QP.

Equation 5-13. Linear SVM classifier cost function

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b))$$

The first sum in the cost function will push the model to have a small weight vector \mathbf{w} , leading to a larger margin. The second sum computes the total of all margin violations. An instance's margin violation is equal to 0 if it is located off the street and on the correct side, or else it is proportional to the distance to the correct side of the street. Minimizing this term ensures that the model makes the margin violations as small and as few as possible

Hinge Loss

The function $\max(0, 1 - t)$ is called the *hinge loss* function (represented below). It is equal to 0 when $t \geq 1$. Its derivative (slope) is equal to -1 if $t < 1$ and 0 if $t > 1$. It is not differentiable at $t = 1$, but just like for Lasso Regression (see [“Lasso Regression” on page 139](#)) you can still use Gradient Descent using any *subderivative* at $t = 1$ (i.e., any value between -1 and 0).