به نام خدا

خلاصه فصل ۵

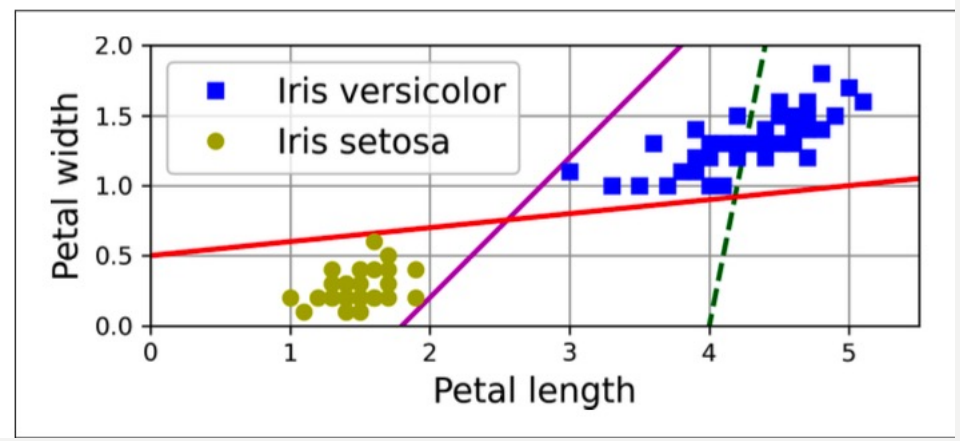
کتاب

HANDS-ON MACHINE LEARNING

SUPPORT VECTOR MACHINES

- یکی از قدرتمند ترین مدل هایی که وجود داره، svm هستش. میتونه linear or nonlinear classification، regression و ... رو انجام بده. حتی توانایی خوبی برای دیتاست های کوچیک و متوسط داره. اما بدی که داره اینه مقایس بندی خوبی روی دیتاست های خیلی بزرگ نداره.

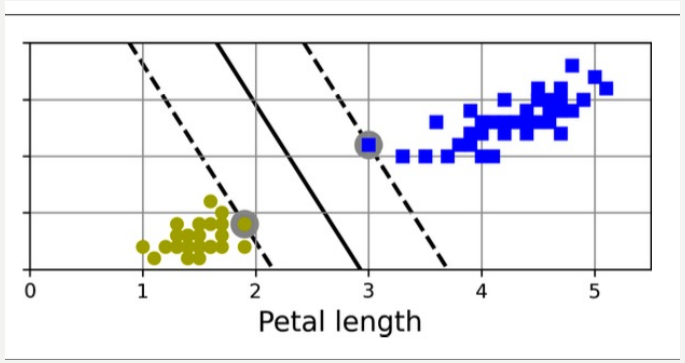
- Linear SVM Classification: اگه دیتاست iris رو به یاد بیاریم. میتونیم کلاس هایی که برای دو گل هستن رو به شکل زیر با استفاده از یک خط classification کنیم.



نکته ای که وجود داره خط راستی مثلا خیلی بده حتی داره اشتباه classification میکنه در صورتی که دو تا خط دیگه بهترن. ولی این خط ها هم خیلی خوب نیستن چون خیلی رو مرزن یه نمونه میتونه بیاد اونور خط ولی متعلق به کلاس اینور خط باشه پس این خط ها فقط روی training set دارن خوب کار میکنن

CONTINUE

- اما تو این خط که حاصل کار svm هستش، علاوه بر اینکه به درستی جدا کرده، خطی رو انتخاب کرده که بیشترین فاصله رو از نزدیک ترین نمونه به اون کلاس داشته. به این کار میگن large margin classification.



این دو تا نمونه که دورش خط کشیدیم باعث میشن خط رو به وجود بیاریم اسمش رو میذاریم support vectors.
و بین خط اصلی و خط های dash شده رو اگه نگاه کنیم مثل یه خیابونه.

- Svm ها به شدت نسبت به scale دیتاهامون حساسن. میتونیم دلیلش رو به این شکل ببینیم:

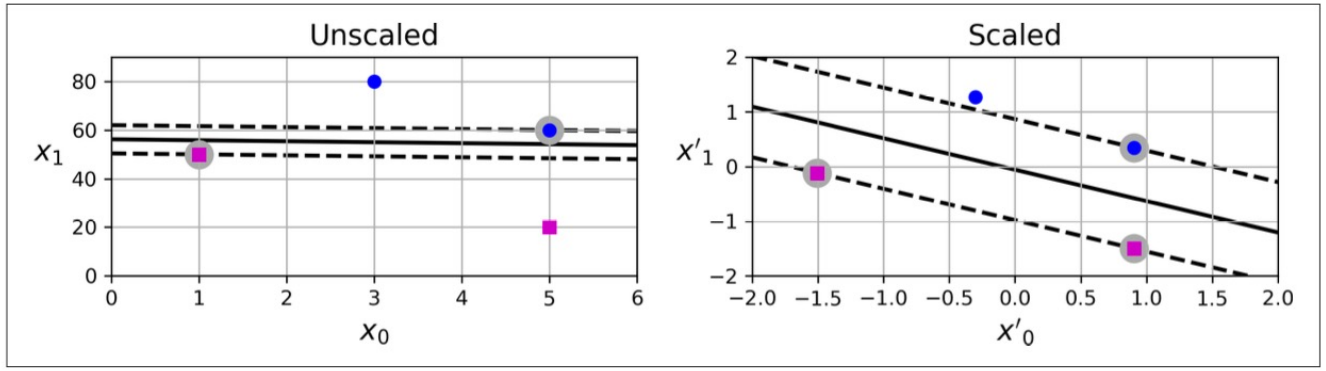


Figure 5-2. Sensitivity to feature scales

SOFT MARGIN CLASSIFICATION

- اگه ما بیایم این شکلی باشیم که خب همه دیتاها باید از قاعده پیروی کنن و حتما اینور داخل ساید درست باشن و فاصله از خط نسبت به support vectors بیشتر باشه بهش میگیم hard margin classification. اما این کار دو مشکل اصلی داره یکی اینکه اگه واقعا دیتامون با یه خط قابل جداسازی نباشه چی؟ و اینکه اگه دیتامون داده پرت داشته باشه مشکل ساز میشه.

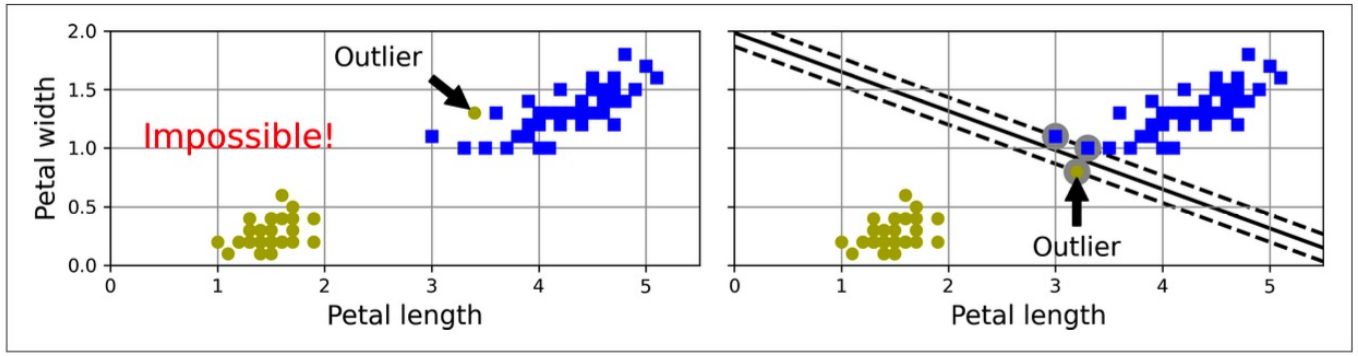


Figure 5-3. Hard margin sensitivity to outliers

- برای اینکه با همچین مشکلی مواجه نشیم، باید از یه مدل منعطف تری استفاده کنیم. هدف اصلی اینه همه نمونه ها تو سمت درست باشن و خیابونمون تا حد ممکن عریض باشه. و اگه بیایم یه بالانس بین این دو هدف برقرار کنیم اسم یه مدلی میشه به نام soft margin classification.

CONTINUE

- یه hyperparameter هستش به اسم C که میاد کنترل میکنه این trade off رو. هرچقدر بزرگ تر باشه بیشتر به درست classification همه نمونه ها توجه داره و هرچقدر کمترش کنیم بیشتر به این کمتر میپردازه که باعث میشه خیابون عریض تری داشته باشی (نمونه ای که باعث میشه خیابون رو تشکیل بدیم از نزدیک ترین حالت ممکن به یه سری نمونه دیگه انتقال پیدا کرده).

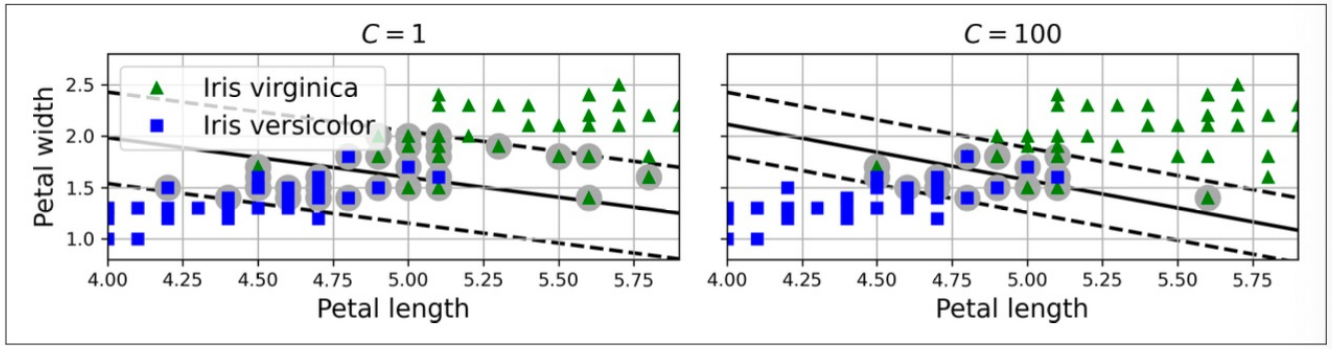


Figure 5-4. Large margin (left) versus fewer margin violations (right)

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

```
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])
```

NONLINEAR SVM CLASSIFICATION

- تا الان دیتاهامون با یه خط قابل جدا شدن بود، اما اگه یه سری دیتا داشته باشیم که نتونیم خطی جداشون کنیم چی کار کنیم؟ یه راه اینه بهشون فیچر اضافه کنیم تا بتونیم با یه خط جداشون کنیم مثل شکل زیر که توان ۲ فیچر x1 رو به عنوان x2 مشخص کردیم و قابل جدا شدن شد.

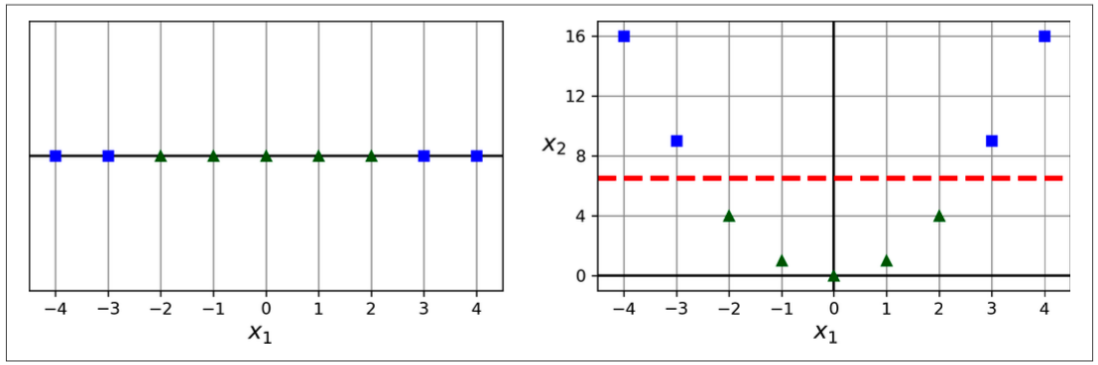


Figure 5-5. Adding features to make a dataset linearly separable

برای اینکه بتونیم همچین کاری انجام بدیم باید اول polynomial feature که داخل فصل ۲ استفاده کردیم رو هم اینجا به کار ببریم. پیاده سازی این شکلی داره (رو دیتاست جدیدی این کارو کردیم):

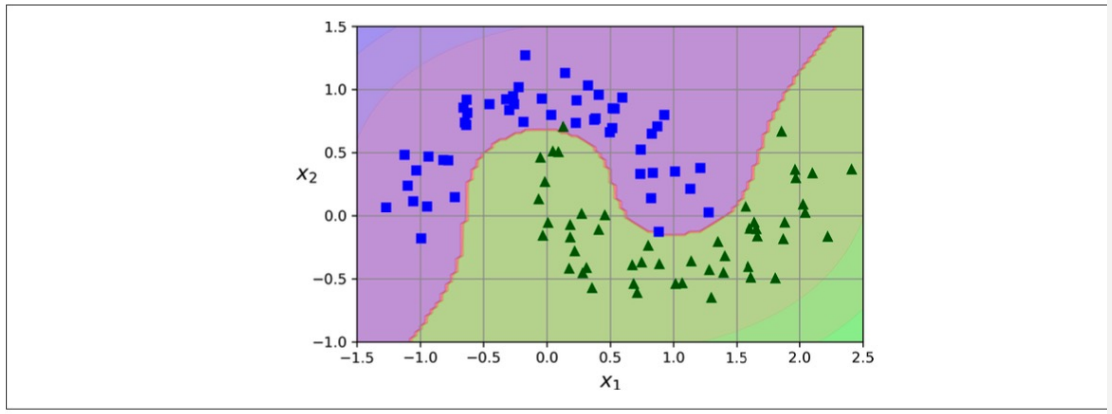


Figure 5-6. Linear SVM classifier using polynomial features

```python
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)
```

POLYNOMIAL KERNEL

- اینکه بیایم همچین ایده ای بزنیم فیچر هارو به بعد های بالاتر ببریم کلا خوبه نه فقط برای svm حتی مدل های دیگه هم خوبه. یه بدی که فیچر های زیاد داره اینه که ممکنه فرآیند یادگیری تو همه مدل ها زیاد کنه. البته چرا میگیم ممکنه، چون داخل svm به خاطر kernel trick برای svm این اتفاق نمیفته. به طور مفصل توضیح میدیم بعدا ولی نکتش اینه با این trick نمیایم همه فیچر های درجه بالا رو درست کنیم و بدون ساختشون میتونیم یه برآوردی ازشون داشته باشیم.

```
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)
```

این کد میاد یه svm classifier با استفاده از کرنل درجه سه درست میکنه. اگه مدل داره اورفیت میکنه میتونیم این درجه رو کمتر کنیم. با استفاده از coef0 میاد مشخص میکنه چقدر به فیچر های درجه بالا اهمیت داده بشه تا فیچر های درجه پایین.

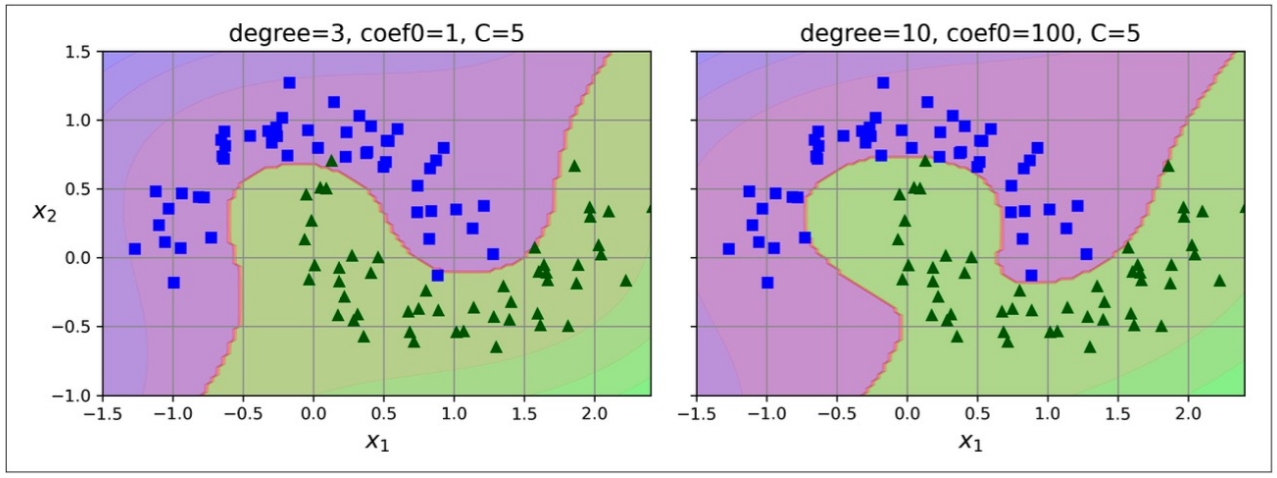


Figure 5-7. SVM classifiers with a polynomial kernel

SIMILARITY FEATURES

- یه روش دیگه برای اینکه با مسائل غیرخطی رو به رو بشیم، بیایم یه سری فیچر که توسط similarity function ساخته شدن رو اضافه کنیم. که نشون میده هر نمونه چقدر متعلق به یه فضا هستش. برای مثال داخل این عکس اومدیم بر اساس نقاط -۲ و ۱ دو منطقه که توزیع گوسی دارن تعریف کردیم و وقتی یه نقطه میاد میبینیم چقدر متعلق به این توزیع ها هستن.

$\times 2^2) \approx 0.30$. 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.

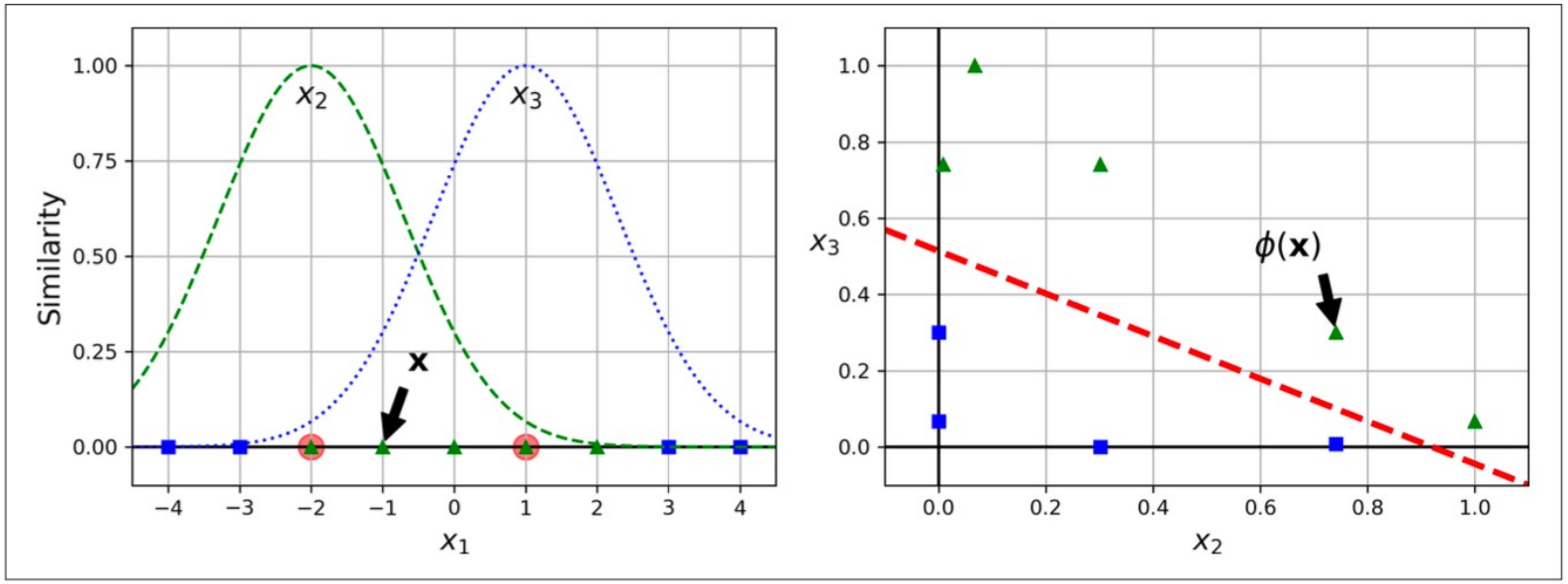


Figure 5-8. Similarity features using the Gaussian RBF

CONTINUE

- شاید سوال پیش بیاد خب از کجا این landmark ها و نقاط رو پیدا کنیم. ساده ترین راه اینه بیایم هر نمونه داخل training set رو یه landmark بهش بدیم. اینجوری چون خیلی فیچر اضافه میشه میتونیم مطمئن بشیم دیگه دیتا خطی قابل جدا شدن هستش. یعنی اگه training set شامل m عضو باشه. فضای جدید m بعدی میشه.

- **Gauusian RBF Kernel: درست مثل polynomial features method، این similarity features method** هم میتونه توسط هر مدل دیگه ای به کار برده بشه اما چون svm از kernel trick بهره میبره باعث میشه استفاده ازش اینجا منطقی بشه. بیایم یه svc که کرنل rbf داره بزنیم.

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

- گاما در اصل یه hyperparameter هستش کاری که میکنه اینه که کرنل گوسی که به هر نمونه میده رو چقدر دست باز یا بسته میذاره. هرچی مقدارش بیشتر میشه فضای نمونه های یه کلاس کوچیک تر میشه و احتمال overfit بیشتر مثال صفحه بعد به خوبی توضیح میده.

CONTINUE

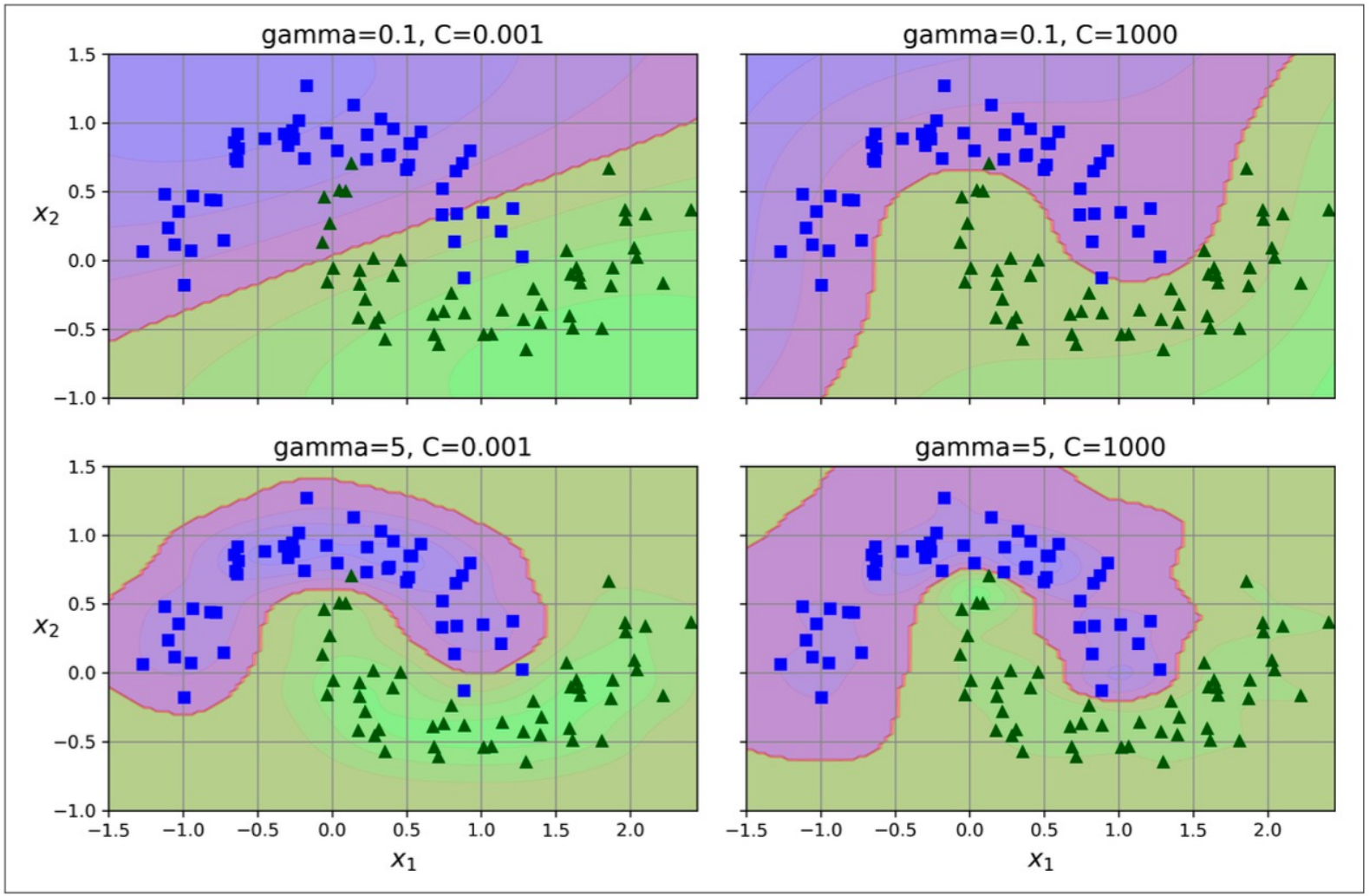


Figure 5-9. SVM classifiers using an RBF kernel

SVM CLASSES AND COMPUTATIONAL COMPLEXITY

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

- اینجا هم مثل classification هدف اینه یه خیابون داشته باشیم ولی این خیابون هدفش اینه حالا نمونه های بیشتری داخلش باشن. با استفاده از یه hyperparameter اپسیلون میایم عرض خیابون رو کنترل میکنیم.

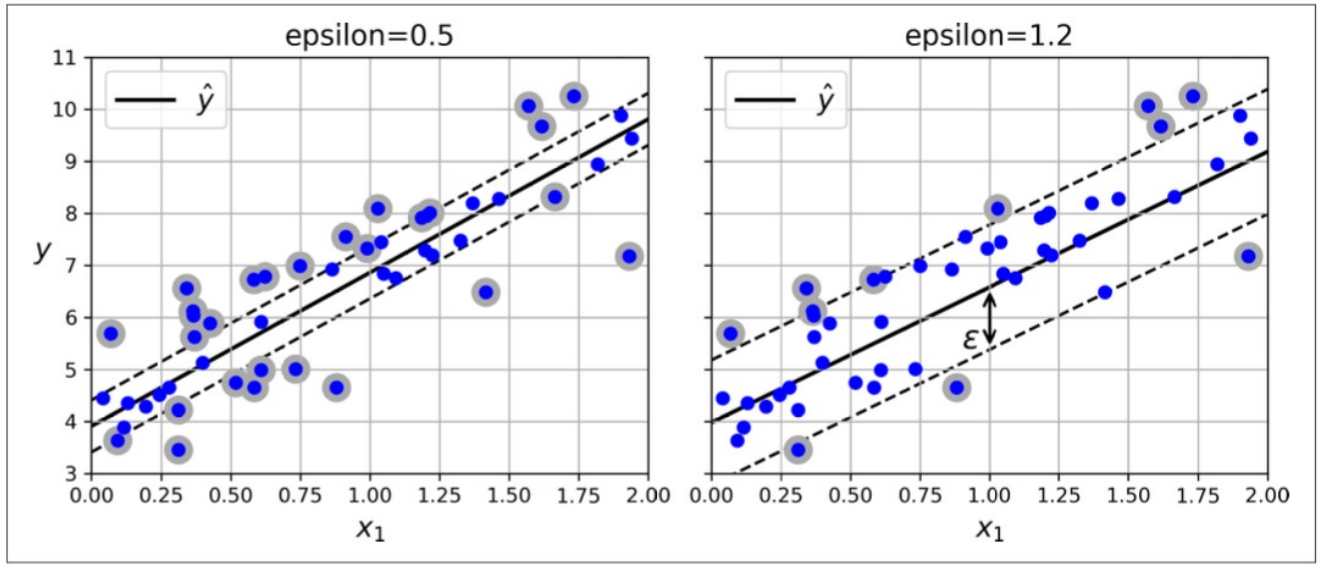


Figure 5-10. SVM regression

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*.

```
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)
```

CONTINUE

- حالا اگه regression غیرخطی داشته باشیم چی کار کنیم؟ میایم از polynomial درجه n استفاده میکنیم.

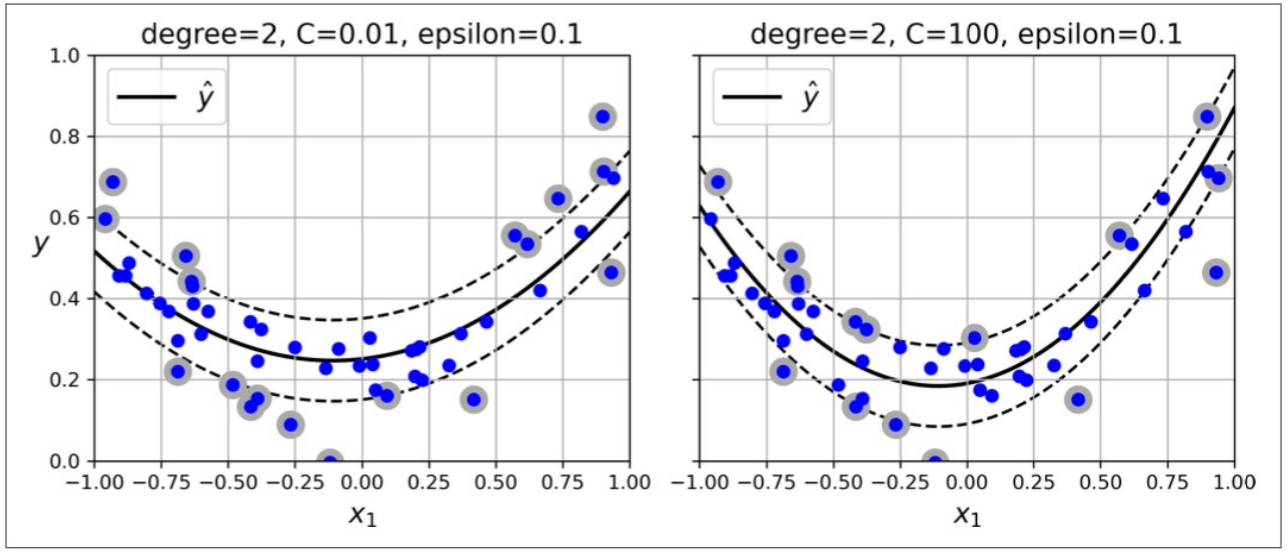


Figure 5-11. SVM regression using a second-degree polynomial kernel

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)
```

UNDER THE HODD OF LINEAR SVM CLASSIFIERS

- حالا میخوایم بریم تو بطن ریاضی ماجرا و ببینیم چه خبر بوده.اول از همه بحث classification رو ببینیم:

A linear SVM classifier predicts the class of a new instance $\mathbf{x}$ by first computing the decision function $\boldsymbol{\theta}^T \mathbf{x} = \theta_0 x_0 + \cdots + \theta_n x_n$, 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).

- الان که متوجه شدیم چطوری پیش‌بینی انجام میداده باید بفهمیم چطوری یادگیری رخ میداده براش. نیاز داریم ضریب پارامتر های به همراه بایاس ترم رو برای یه خط پیدا کنیم.

violations. Let's start with the width of the street: to make it larger, we need to make $\mathbf{w}$ 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 $\mathbf{w}$ 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.

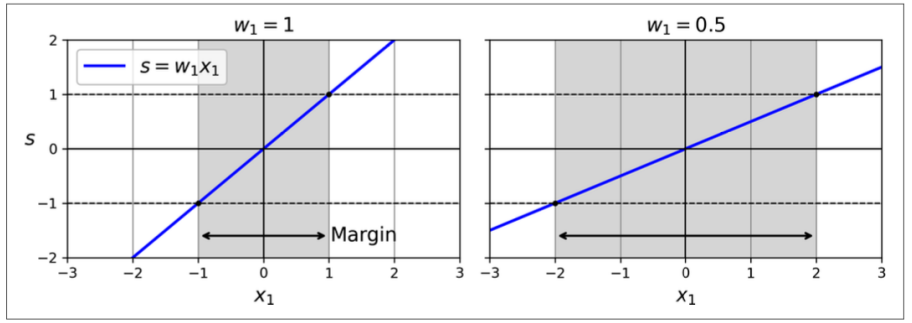


Figure 5-12. A smaller weight vector results in a larger margin

HARD MARGIN CLASSIFIER

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 $t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1$ 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

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\top \mathbf{w}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

We are minimizing ½ $\mathbf{w}^\top$ $\mathbf{w}$, 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.

SOFT MARGIN CLASSIFIER

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

$$\underset{\mathbf{w},\, b,\, \zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\intercal \mathbf{w} + C \sum_{i=1}^{m} \zeta^{(i)}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\intercal \mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

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 GRADIENT DESCENT TO TRAIN

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 ($s = \mathbf{w}^T \mathbf{x} + b$) 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.

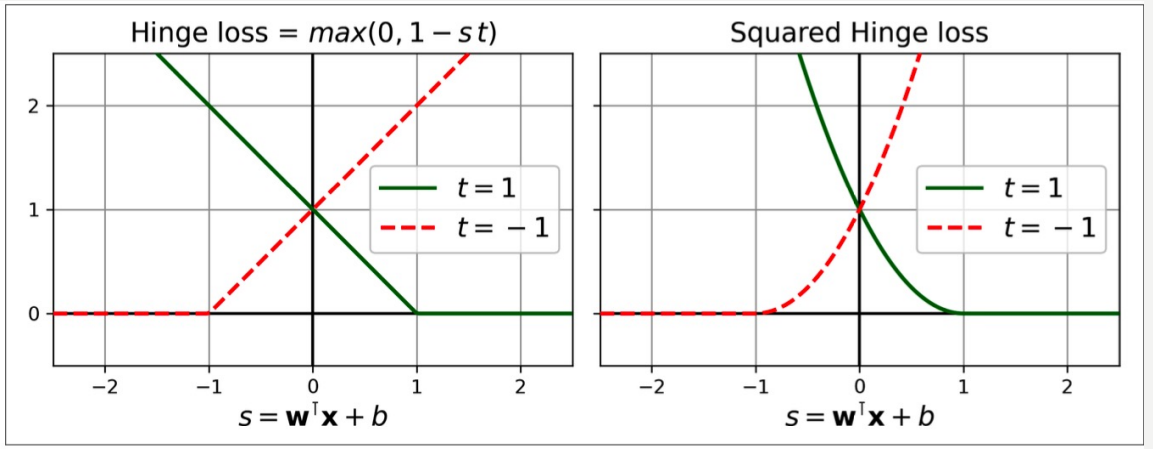


Figure 5-13. The hinge loss (left) and the squared hinge loss (right)

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

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

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

Once you find the vector $\widehat{\boldsymbol{\alpha}}$ that minimizes this equation (using a QP solver), use Equation 5-4 to compute the $\widehat{\mathbf{w}}$ and $\widehat{b}$ 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

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

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

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?

KERNEL TRICK

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).

KERNEL TRICK

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

$$\varphi(\mathbf{a})^\top \varphi(\mathbf{b}) = \begin{pmatrix} {a_1}^2 \\ \sqrt{2}\, a_1 a_2 \\ {a_2}^2 \end{pmatrix}^\top \begin{pmatrix} {b_1}^2 \\ \sqrt{2}\, b_1 b_2 \\ {b_2}^2 \end{pmatrix} = {a_1}^2 {b_1}^2 + 2 a_1 b_1 a_2 b_2 + {a_2}^2 {b_2}^2$$

$$= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^\top \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^\top \mathbf{b})^2$$

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

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 $\phi(\mathbf{x}^{(i)})^\top \phi(\mathbf{x}^{(j)})$. 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 $\left(\mathbf{x}^{(i)\top} \mathbf{x}^{(j)}\right)^2$. 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.

KERNEL TRICK

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

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

KERNEL TRICK

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

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

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).

FINSIH

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

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

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

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).