




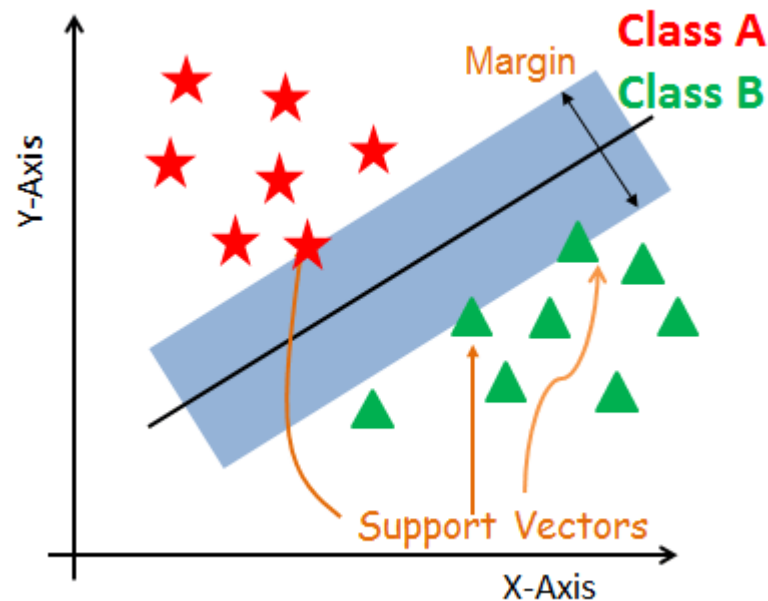
AI in Biomedical Data

Dr. M.B. Khodabakhshi

Amir Hossein Fouladi

Alireza Javadi

 github.com/mbkhodabakhshi/AI_in_BiomedicalData



یادگیری ماشین در زیست پزشکی

Chapter 5. Support Vector Machines

دکتر محمدباقر خدابخشی

mb.khodabakhshi@gmail.com

- A *Support Vector Machine* (SVM) is a powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, and regression.
- It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex small- or medium-sized datasets.
- Let's use the *iris dataset* to illustrate linear classification. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: *Iris setosa*, *Iris versicolor*, and *Iris virginica*.



Linear SVM Classification

- ❑ 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.**
- ❑ 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)

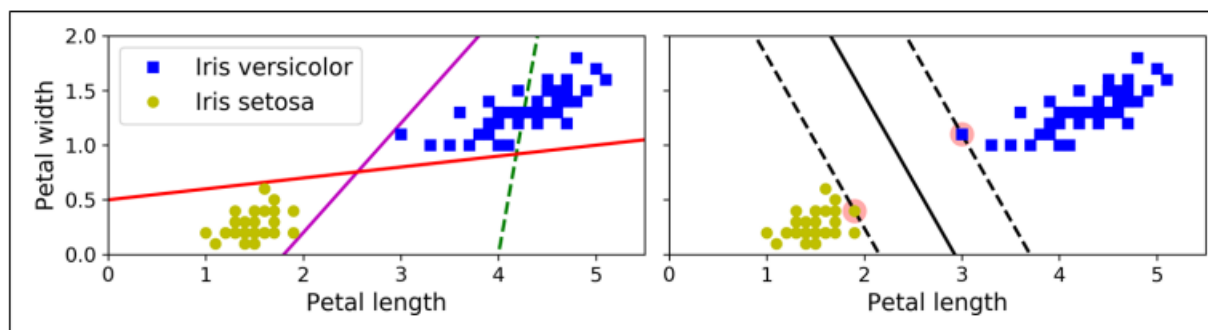


Figure 5-1. Large margin classification

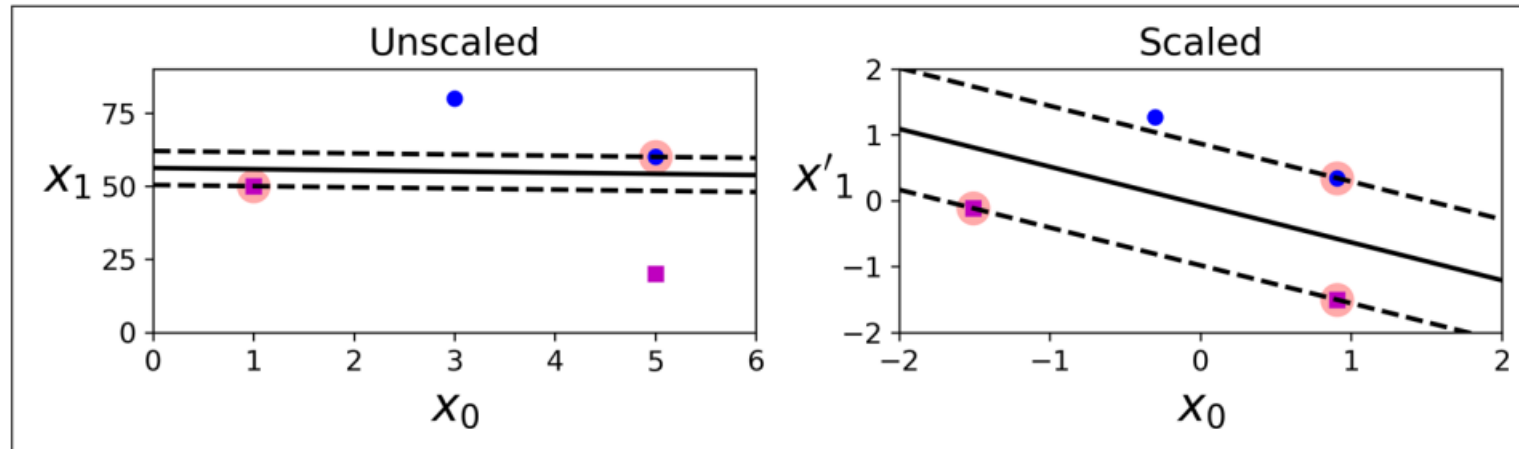


Figure 5-2. Sensitivity to feature scales



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.

Soft Margin Classification

- If we strictly impose that all instances must be off the street and on the right 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 it will probably not generalize as well.

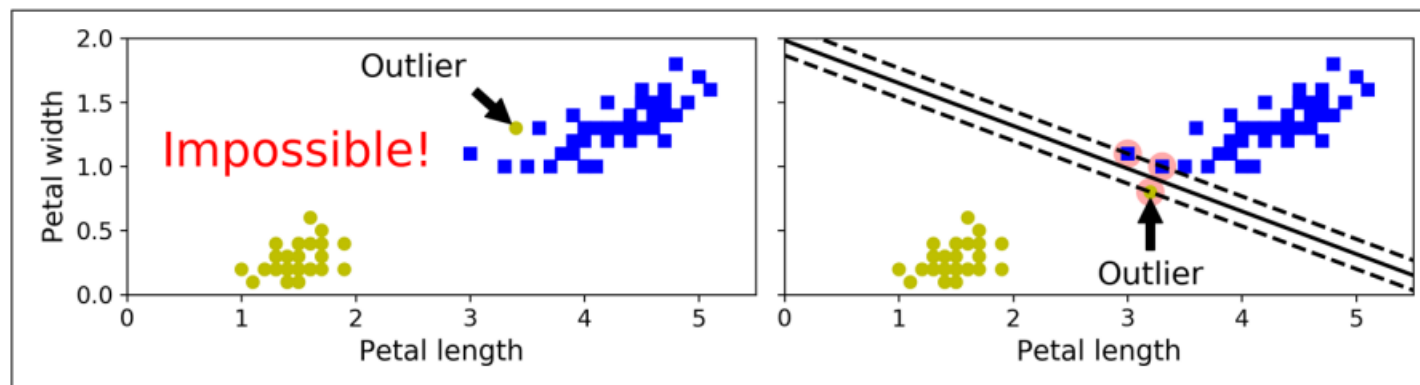


Figure 5-3. Hard margin sensitivity to outliers

- ❖ To avoid these issues, 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, we can specify a number of hyperparameters. C is one of those hyperparameters.
- ❖ If we set it to a low value, then we end up with the model on the left of Figure 5-4. With a high value, we get the model on the right. Margin violations are bad. It's usually better to have few of them. However, in this case the model on the left has a lot of margin violations but will probably generalize better.

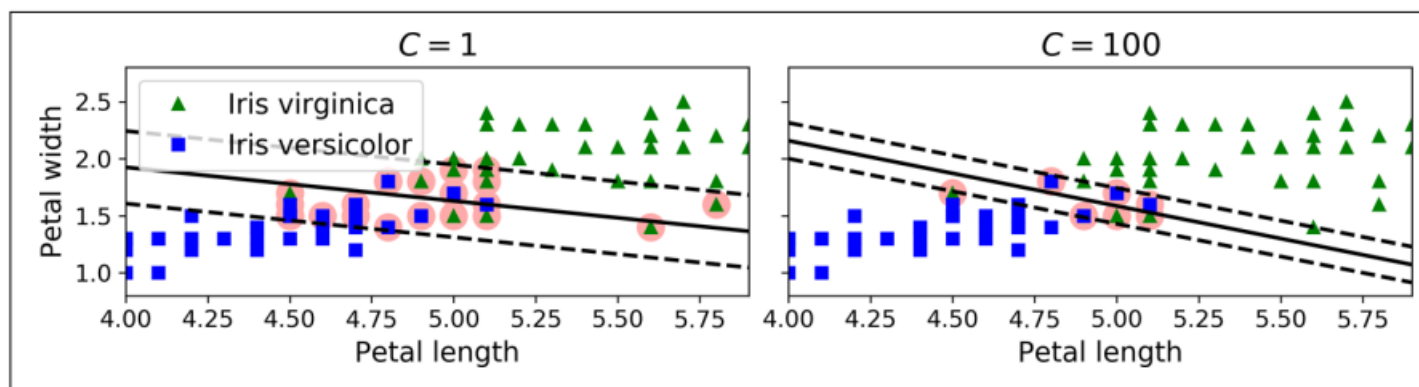


Figure 5-4. Large margin (left) versus fewer margin violations (right)

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

The following Scikit-Learn code loads the iris dataset, scales the features, and then trains a linear SVM model (using the LinearSVC class with $C=1$ and the hinge loss function, described shortly) to detect Iris virginica flowers:

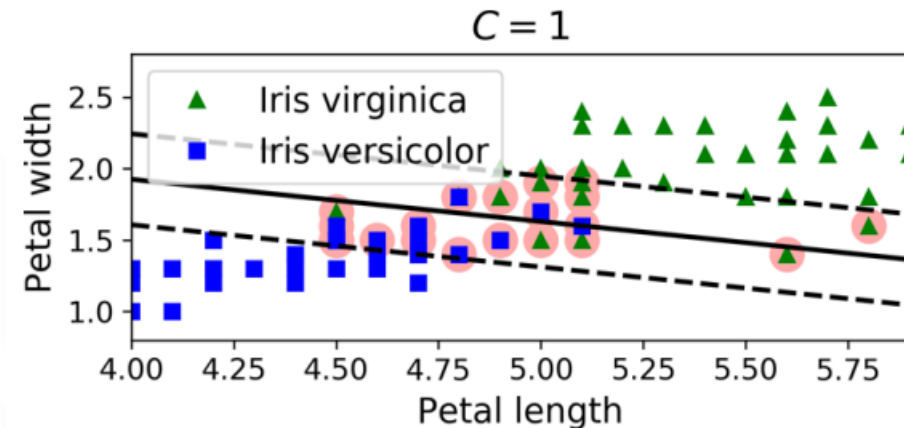
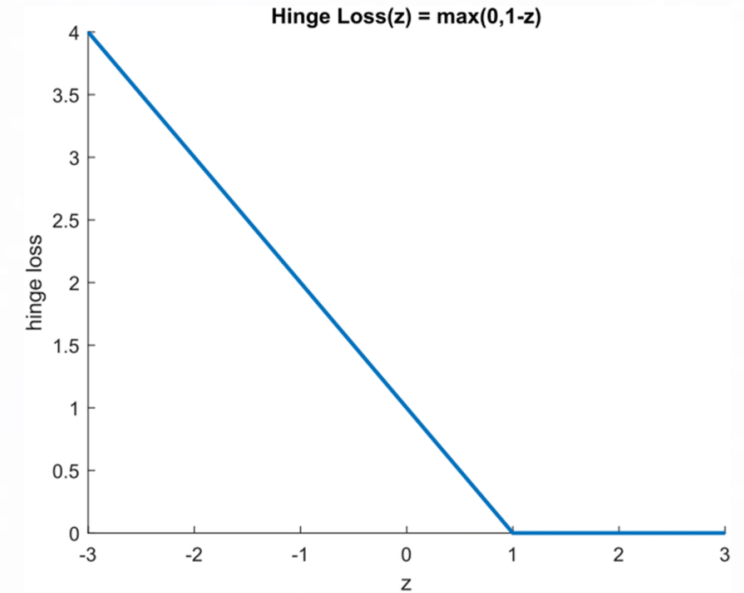
```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris virginica

svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")),
])

svm_clf.fit(X, y)

>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```



- Instead of using the *LinearSVC* class, we could use the *SVC* class with a linear kernel. When creating the *SVC* model, we would write *SVC(kernel="linear", C=1)*.
- Or we could use the *SGDClassifier* class, with *SGDClassifier(loss="hinge", alpha=1/(m*C))*. This applies regular *Stochastic Gradient Descent* (see Chapter 4) to train a linear SVM classifier.
- It does not converge as fast as the *LinearSVC* class, but it can be useful to handle online classification tasks or huge datasets that do not fit in memory (out-of-core training).

Imagine you have a massive dataset of images that you want to use to train a deep learning model. This dataset is so large that it doesn't fit entirely into your computer's RAM.

Here's how out-of-core training would work:

1. **Divide the dataset:** You divide the dataset into smaller, manageable chunks or batches. For example, you might split it into batches of 1,000 images each.
2. **Load a batch:** Load the first batch of images into memory.
3. **Train the model:** Train the deep learning model on this batch of images.
4. **Save the model:** Save the trained model to disk.
5. **Load the next batch:** Load the next batch of images into memory, overwriting the previous batch.
6. **Continue training:** Continue training the model on the new batch, updating its parameters based on the previous training.
7. **Repeat:** Repeat steps 5 and 6 until all batches have been processed.

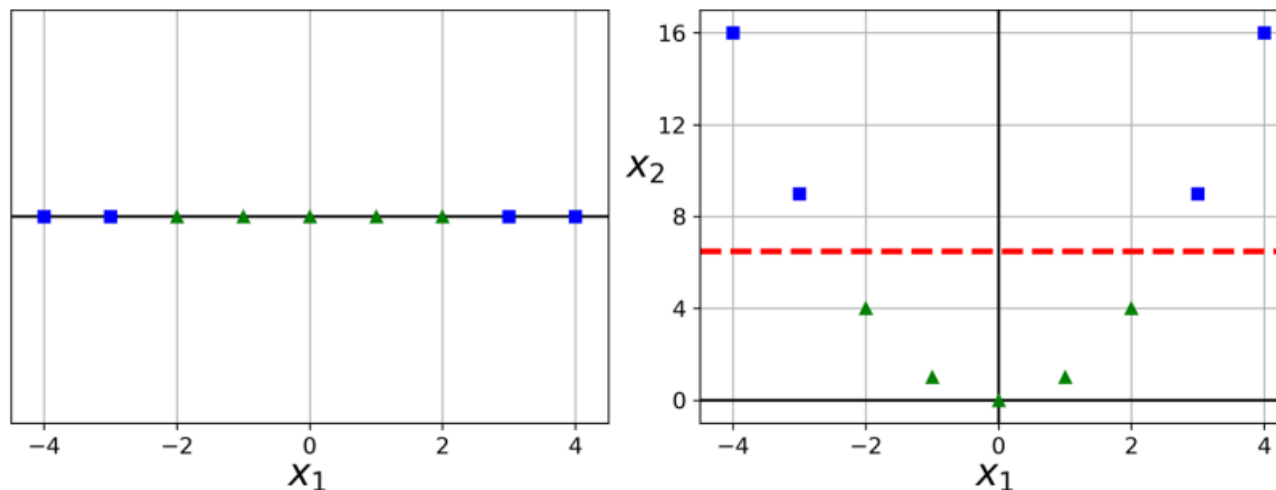
In essence, out-of-core training allows you to train a model on a large dataset without running out of memory by processing it in smaller, manageable chunks.

Nonlinear SVM Classification

Many datasets are not even close to being linearly separable.

Consider the left 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.**

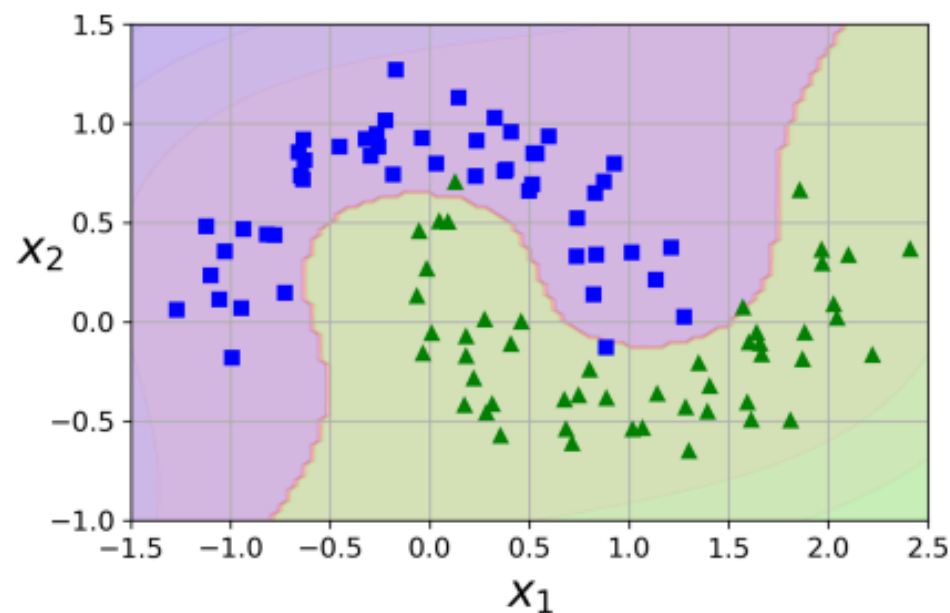


To implement this idea using Scikit-Learn, create a Pipeline containing a PolynomialFeatures transformer (discussed in “Polynomial Regression” on page 128), followed by a StandardScaler and a LinearSVC. Let’s test this on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving half circles (see Figure 5-6). You can generate this dataset using the `make_moons()` function:

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

X, y = make_moons(n_samples=100, noise=0.15)
polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])

polynomial_svm_clf.fit(X, y)
```



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*.
- The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with very high-degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features because you don't actually add any features. This trick is implemented by the SVC class. Let's test it on the moons dataset:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", 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. It is 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 polynomials versus low-degree polynomials.

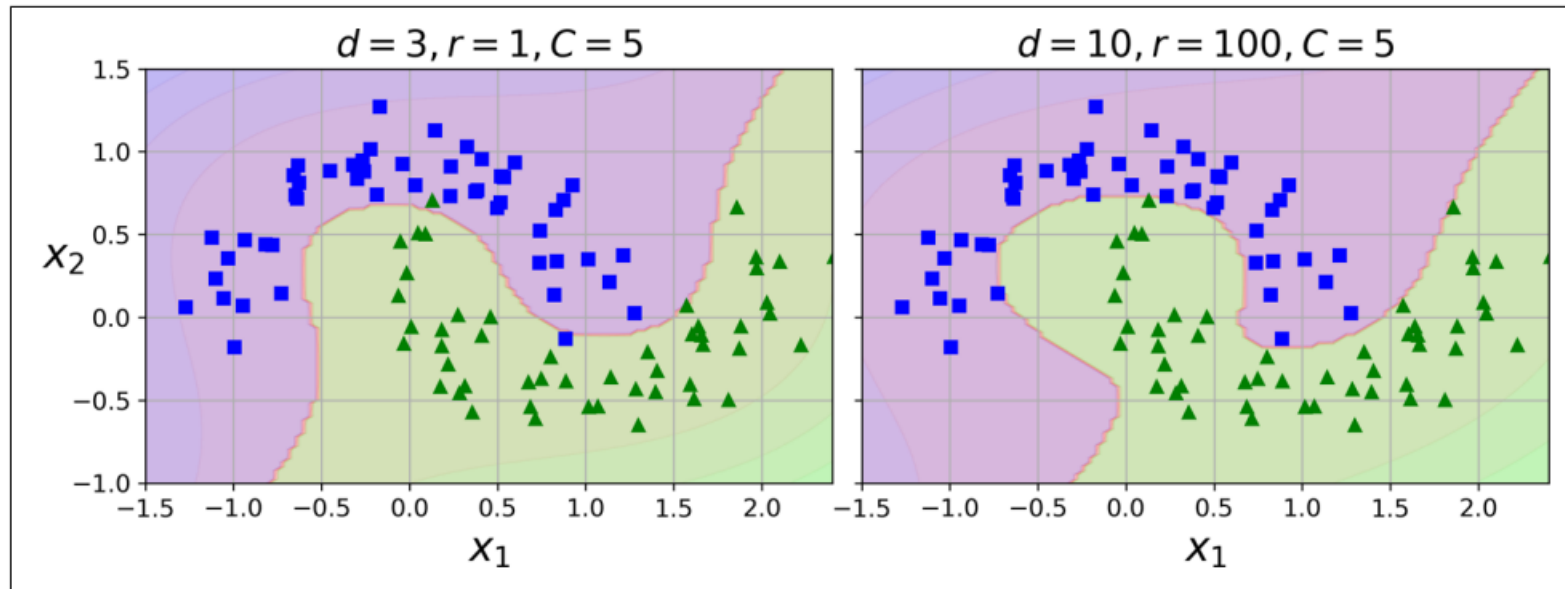


Figure 5-7. SVM classifiers with a polynomial kernel

A common approach to finding the right hyperparameter values is to use grid search (see Chapter 2).

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*. For example, let's take the 1D dataset discussed earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot in **Figure 5-8**). Next, let's define the similarity function to be the Gaussian *Radial Basis Function* (RBF) with $\gamma = 0.3$ (see **Equation 5-1**).

Equation 5-1. Gaussian RBF

$$\phi_{\gamma}(\mathbf{x}, \ell) = \exp \left(-\gamma \| \mathbf{x} - \ell \|^2 \right)$$

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.

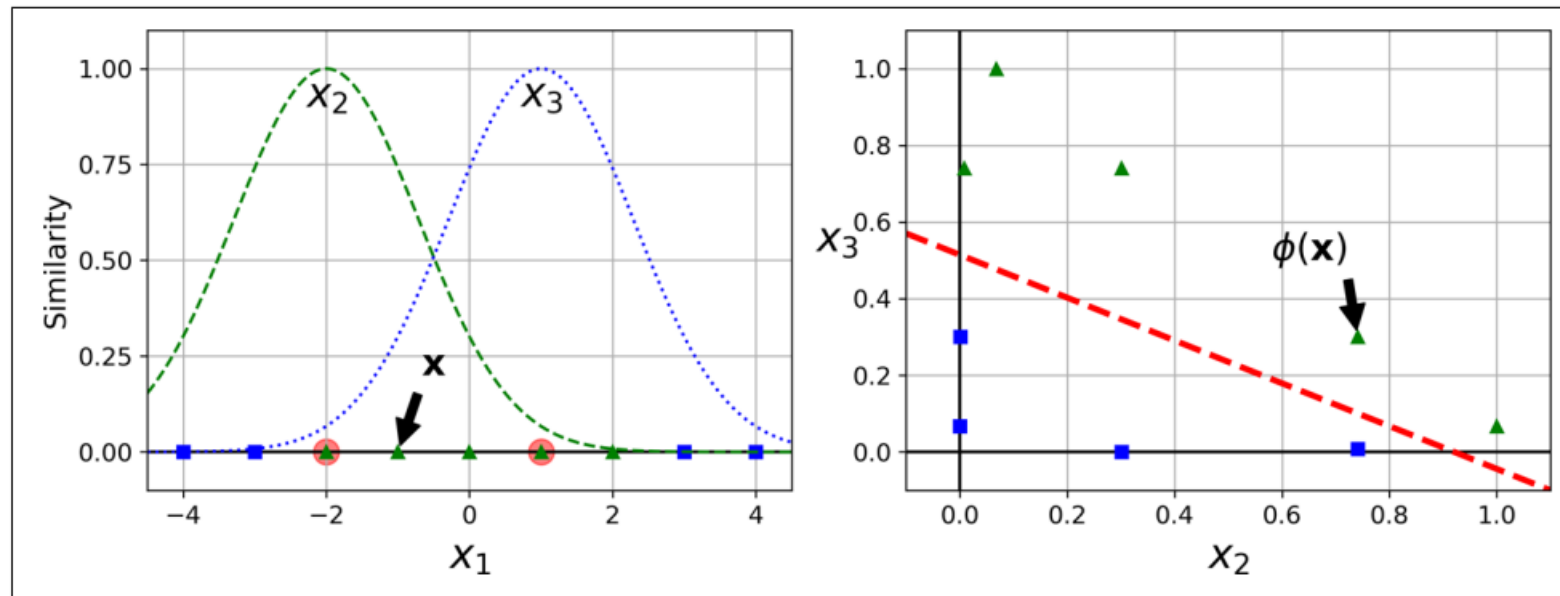


Figure 5-8. Similarity features using the Gaussian RBF

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. Let's try the SVC class with the Gaussian RBF kernel:

```
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", 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 γ and C . Increasing γ 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 γ 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 it; if it is underfitting, you should increase it (similar to the C hyperparameter).

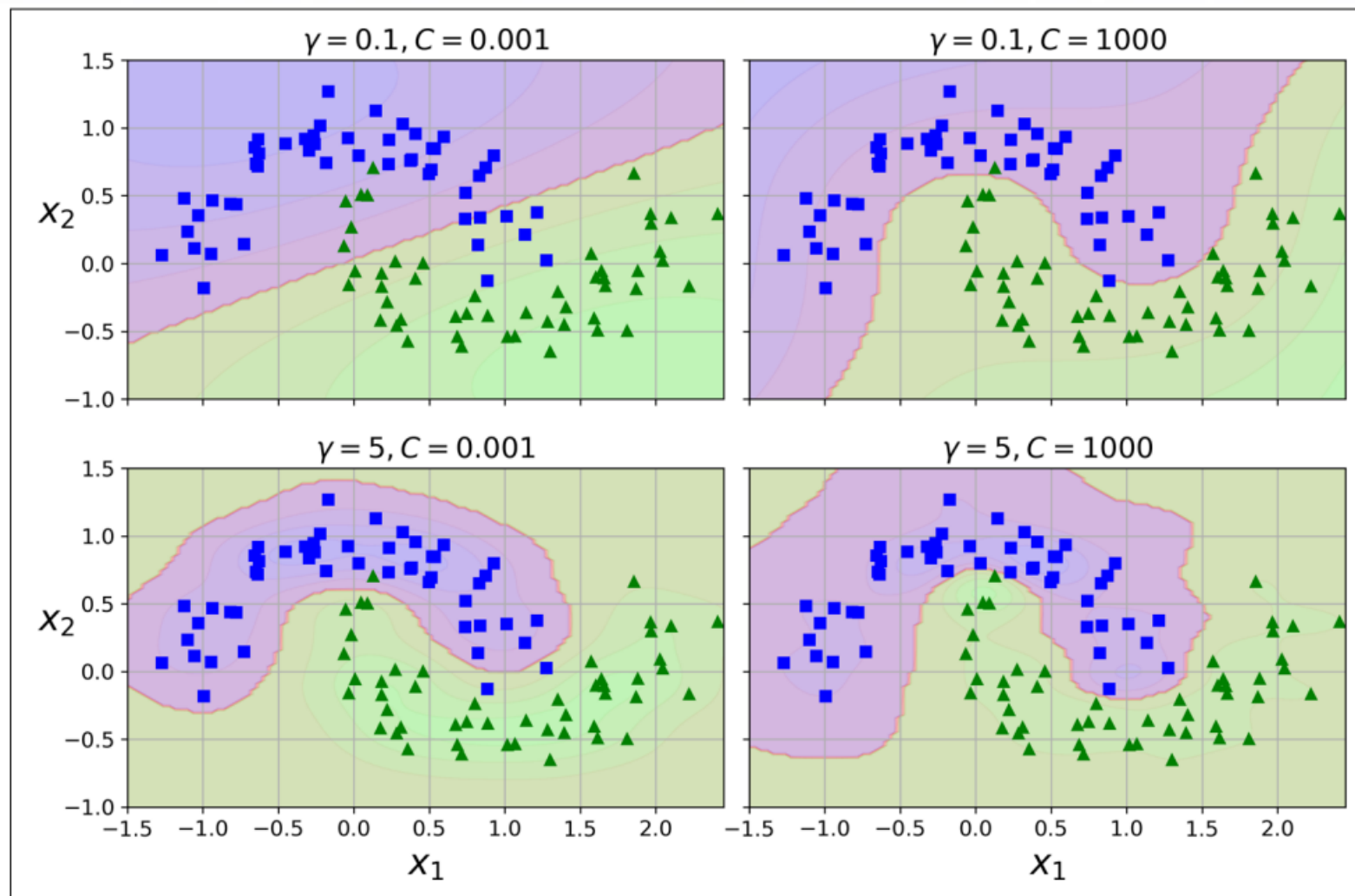


Figure 5-9. SVM classifiers using an RBF kernel

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 (remember that *LinearSVC* is much faster than *SVC(kernel="linear")*), especially if the training set is very large or if it has plenty of features.

If the training set is not too large, you should also try the *Gaussian RBF kernel*; it works well in most cases.

Then if you have spare time and computing power, you can experiment with a few other kernels, using cross-validation and grid search.

You'd want to experiment like that especially if there are kernels specialized for your training set's data structure.

SVM Regression

As mentioned earlier, the SVM algorithm is versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression. To use SVMs for regression instead of classification, the trick is to reverse 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 random linear data, one with a large margin ($\epsilon = 1.5$) and the other with a small margin ($\epsilon = 0.5$).

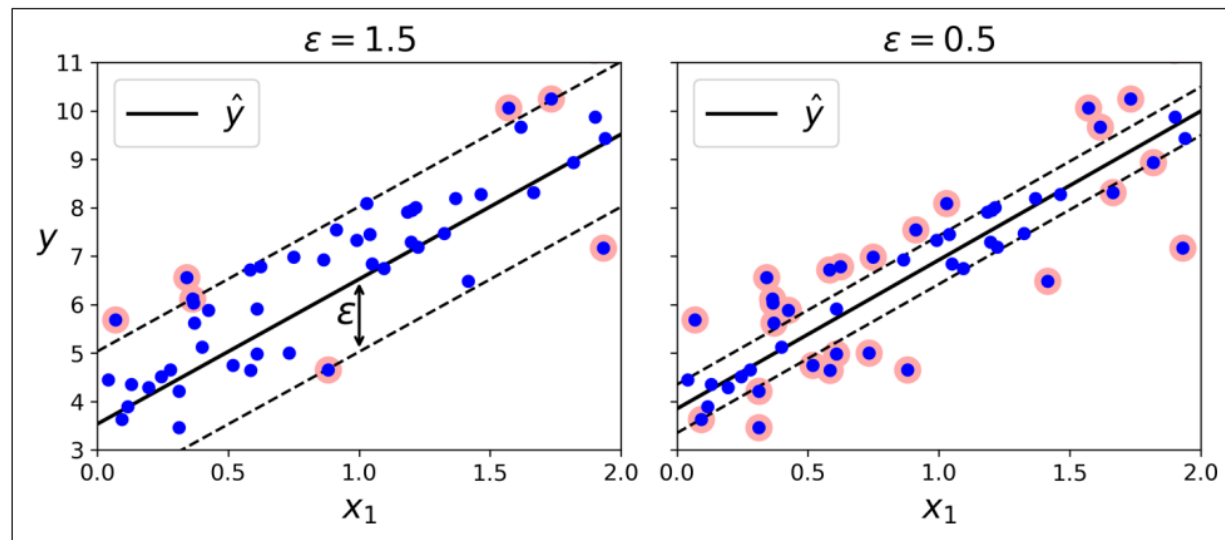


Figure 5-10. SVM Regression

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** (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. **Figure 5-11** shows SVM Regression on a random quadratic training set, using a second-degree polynomial kernel. There is little regularization in the left plot (i.e., a large C value), and much more regularization in the right plot (i.e., a small C value).

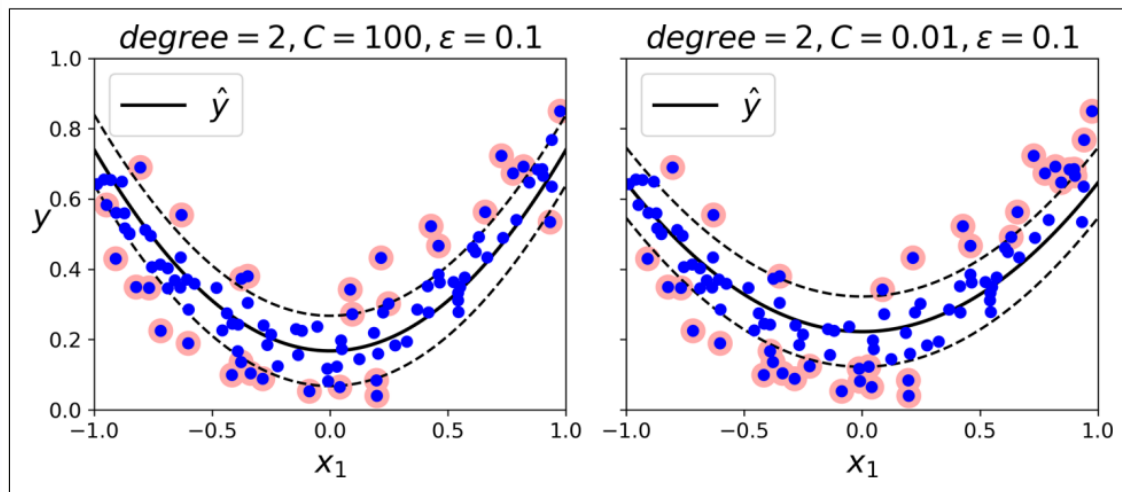


Figure 5-11. SVM Regression using a second-degree polynomial kernel

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

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, 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 large (just like the SVC class).

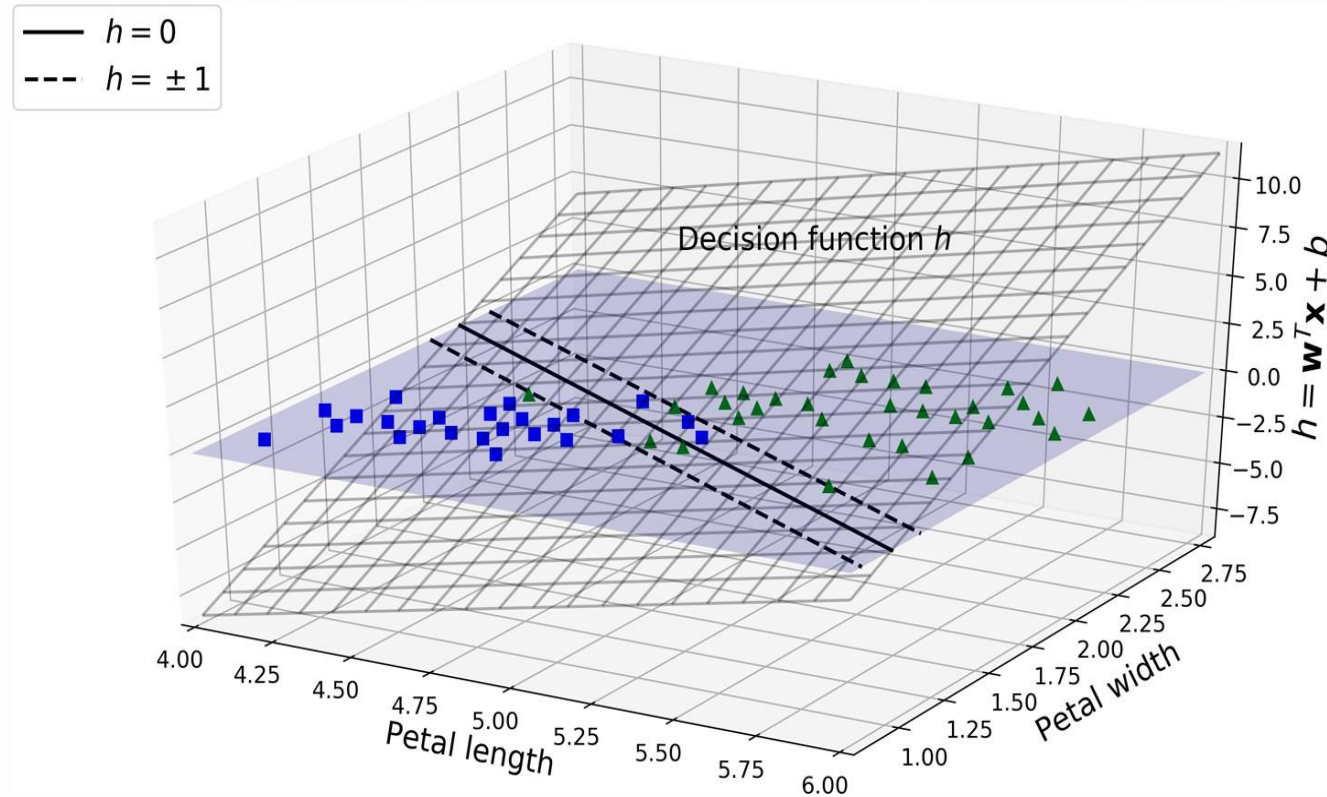
Beyond the Black Box

- Delving into the underlying mathematical principles and algorithms that power the SVM model.
- In this chapter we will use a convention that is more convenient (and more common) when dealing with SVM:
- 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.

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), and otherwise it is the negative class (0); see Equation 5-2.

Equation 5-2. Linear SVM classifier prediction

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

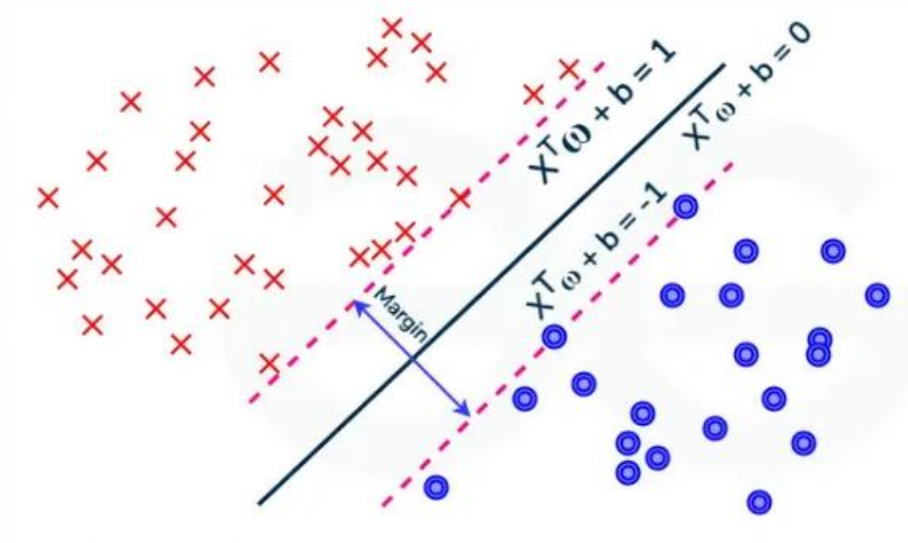


- The dashed lines represent the points where the decision function is equal to 1 or -1 : they are parallel and at equal distance to the decision boundary, and they form a margin around it.
- Training a linear SVM classifier means finding the values of w and b that make this margin as wide as possible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

Training Objective

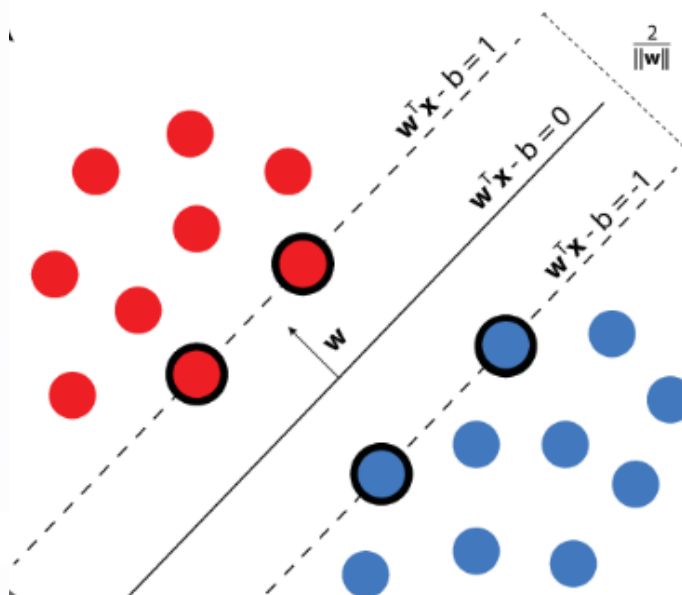
Consider the **slope of the decision function**: it is equal to the norm of the weight vector, \mathbf{w} .

- ❖ 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**. The smaller the weight vector \mathbf{w} , the larger the margin.



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

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



Soft Margin

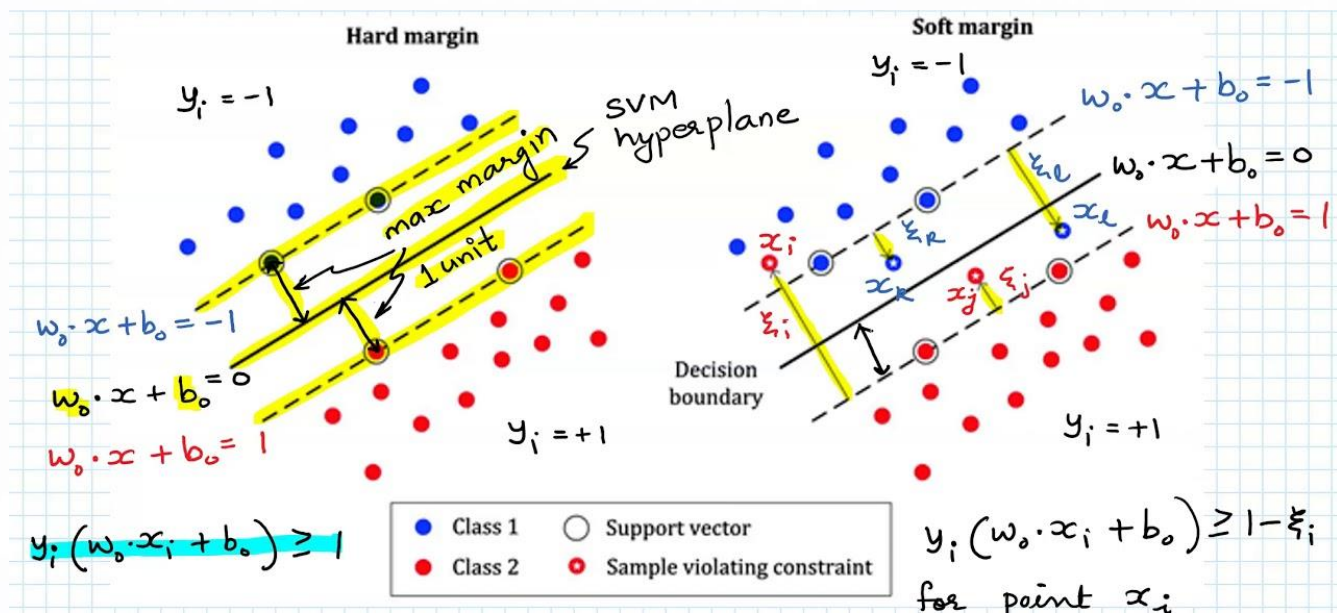
- ❖ To get the soft margin objective, we need to introduce a *slack variable* $\zeta^{(i)} \geq 0$ for each instance:
- ❖ $\zeta^{(i)}$ measures how much the i^{th} instance is allowed to violate the margin.
- ❖ We now have **two conflicting objectives**: make the slack variables as small as possible to reduce the **margin violations**, and make $\frac{1}{2} \mathbf{w}^T \mathbf{w}$ as small as possible to increase the margin.
- ❖ This is where the **C** hyperparameter comes in: it allows us to define the tradeoff between these two objectives. This gives us the constrained optimization problem in Equation 5-4.

Equation 5-4. Soft margin linear SVM classifier objective

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

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

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



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, so *you can choose to solve the primal problem or the dual problem*:

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

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

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

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 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 does not.

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 5-8** shows the second-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 3D instead of 2D. Now let's look at what happens to a couple of 2D vectors, \mathbf{a} and \mathbf{b} , if we apply this second-degree polynomial mapping and then compute the dot product⁷ of the transformed vectors (See **Equation 5-9**).

Equation 5-9. Kernel trick for a second-degree polynomial mapping

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

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

Here is the key insight: if you apply the transformation ϕ to all training instances, then the dual problem (see Equation 5-6) 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 5-8, then you can replace this dot product of transformed vectors simply by $(\mathbf{x}^{(i)\top} \mathbf{x}^{(j)})^2$. So, you don't 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 had gone through the trouble of transforming the training set then fitting a linear SVM algorithm, but this trick makes the whole process much more computationally efficient.

The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^\top \mathbf{b})^2$ is a second-degree polynomial kernel. In Machine Learning, a *kernel* is a function capable of computing the dot product $\phi(\mathbf{a})^\top \phi(\mathbf{b})$, 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

Linear: $K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b}$

Polynomial: $K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^\top \mathbf{b} + r)^d$

Gaussian RBF: $K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \|\mathbf{a} - \mathbf{b}\|^2)$

Sigmoid: $K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^\top \mathbf{b} + r)$

Making predictions with a kernelized SVM

There is still one loose end we must tie up. **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 $\phi(x^{(i)})$. In fact, $\hat{\mathbf{w}}$ 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 $\hat{\mathbf{w}}$? Well, the good news is that you can plug 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 (**Equation 5-11**).

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

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 need to use the same trick to compute the bias term \hat{b} (Equation 5-12).

Equation 5-12. Using the kernel trick to compute the bias term

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

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