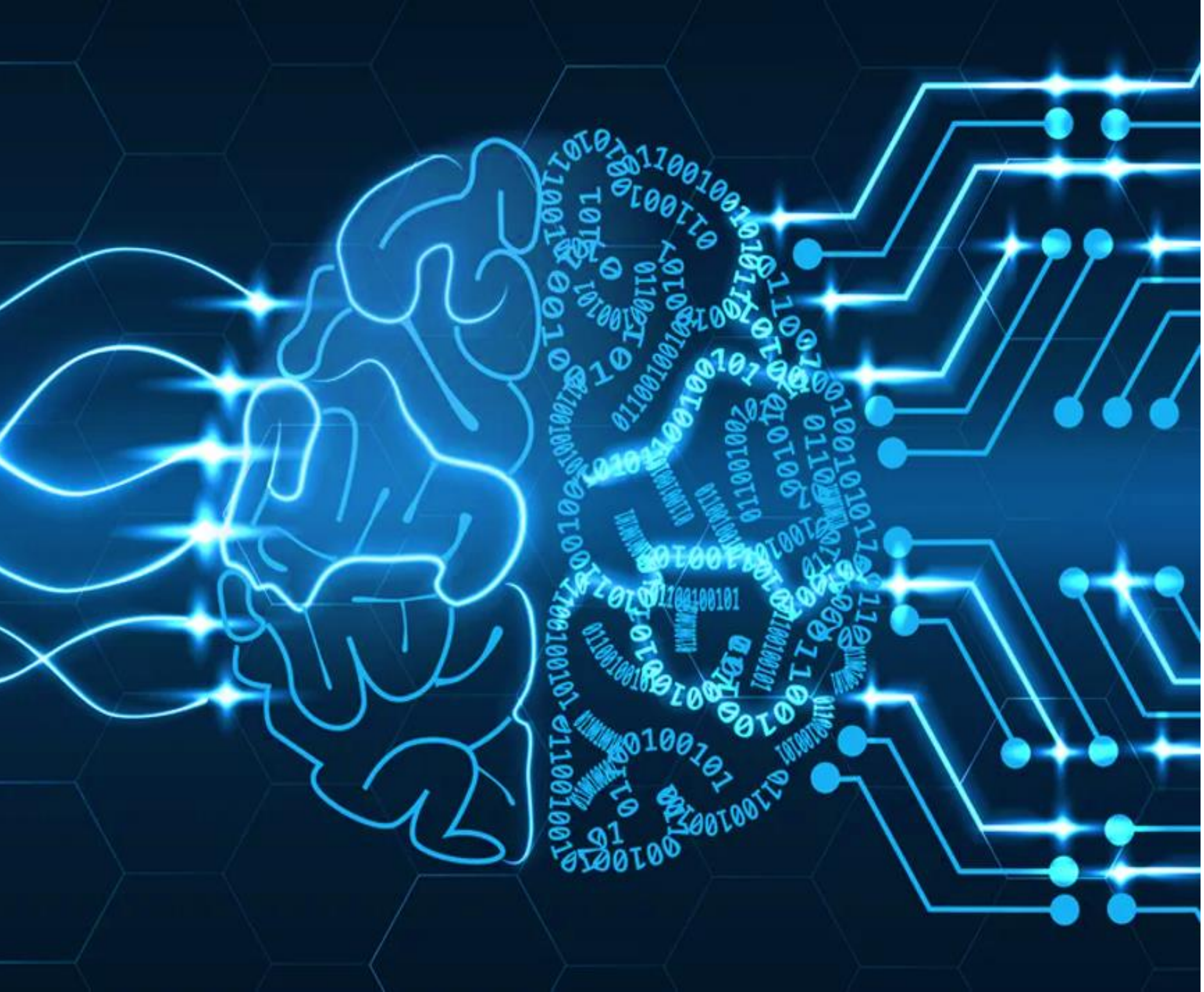


یادگیری ماشین:

ماشین بردار پشتیبان

فروردین ۱۴۰۲

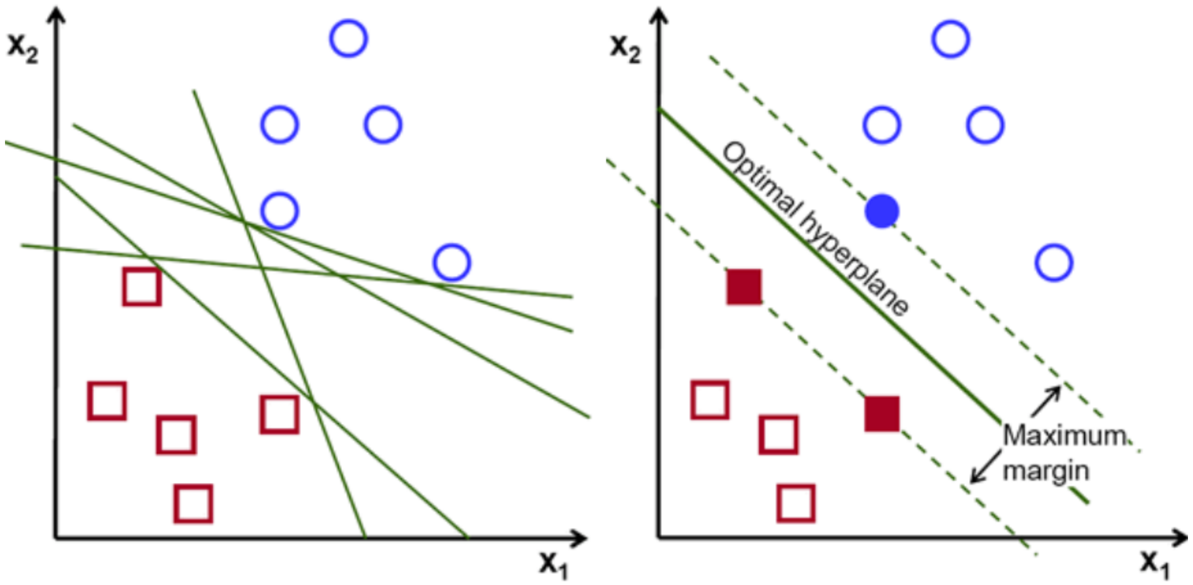
دانشکده مهندسی صنایع
دانشگاه صنعتی شریف

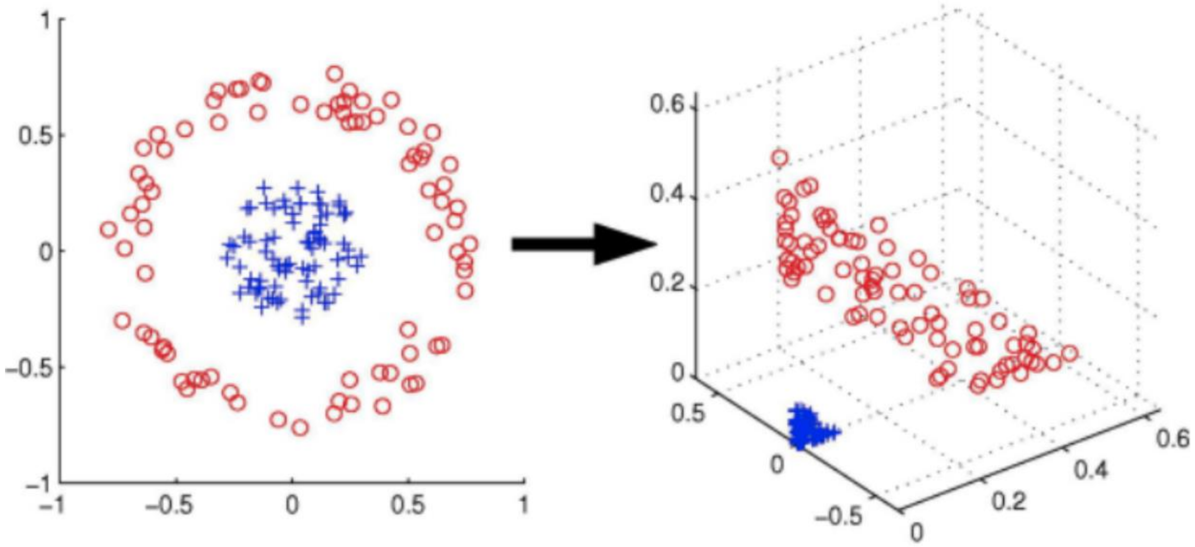
دکتر مهدی شریف‌زاده
محمدتقی دهقان نژاد
سعیدرضا زواشکیانی

بسم الله الرحمن الرحیم

What is support vector machine(SVM)?

A Support Vector Machine (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. 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 but small- or medium-sized datasets.



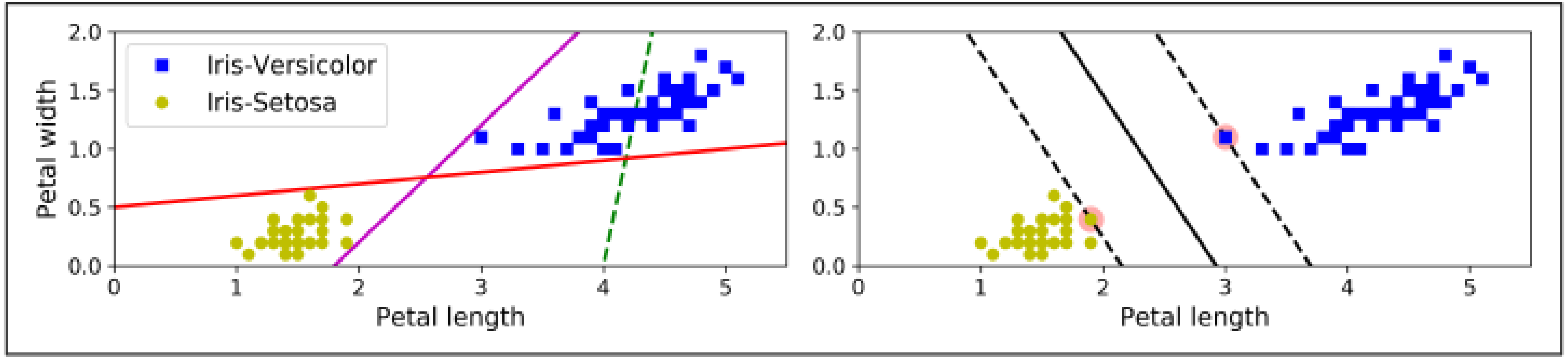


Dataset Introduction

The *Iris* flower data set or Fisher's *Iris* data set is a multivariate dataset collected the data to quantify the morphologic variation of *Iris* flowers of three related species. The data set consists of 50 samples from each of three species of *Iris* (*Iris setosa*, *Iris virginica* and *Iris versicolor, respectively from left to right*).
Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.

Linear SVM Classification

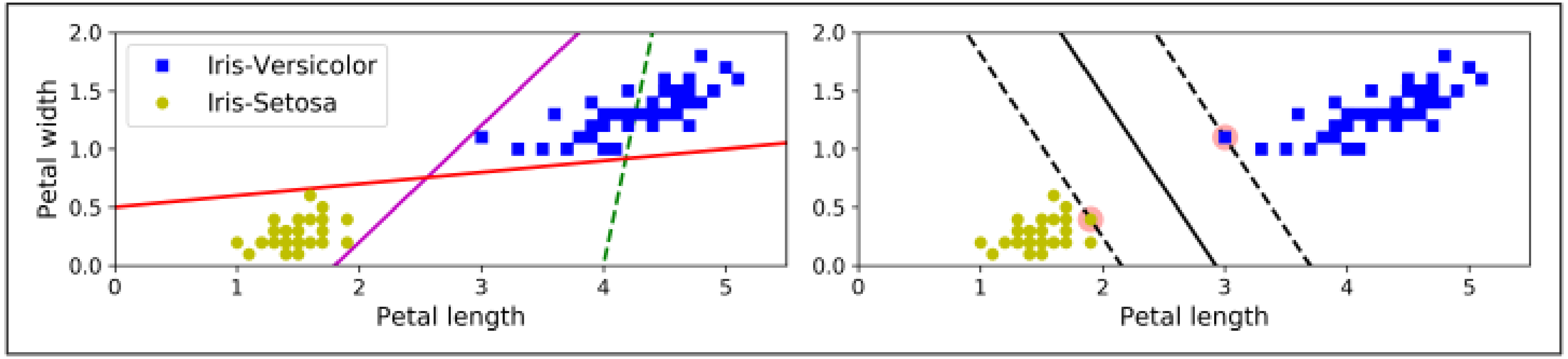
The fundamental idea behind SVMs is best explained with some pictures. The two classes can clearly be separated easily with a straight line (they are *linearly separable*). The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line is so bad that it does not even separate the classes properly.



Linear SVM Classification

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**.

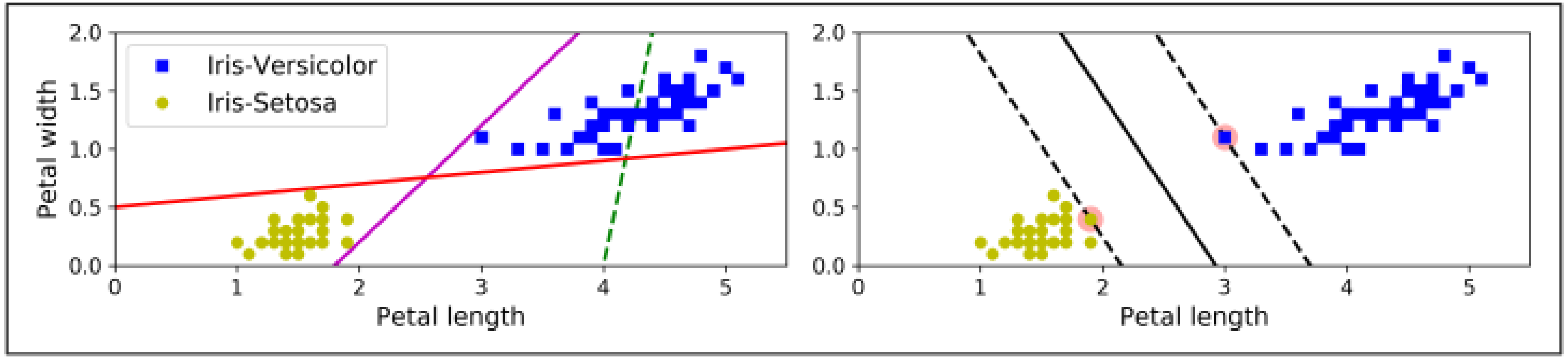
In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also **stays as far away from the closest training instances as possible**.



Linear SVM Classification

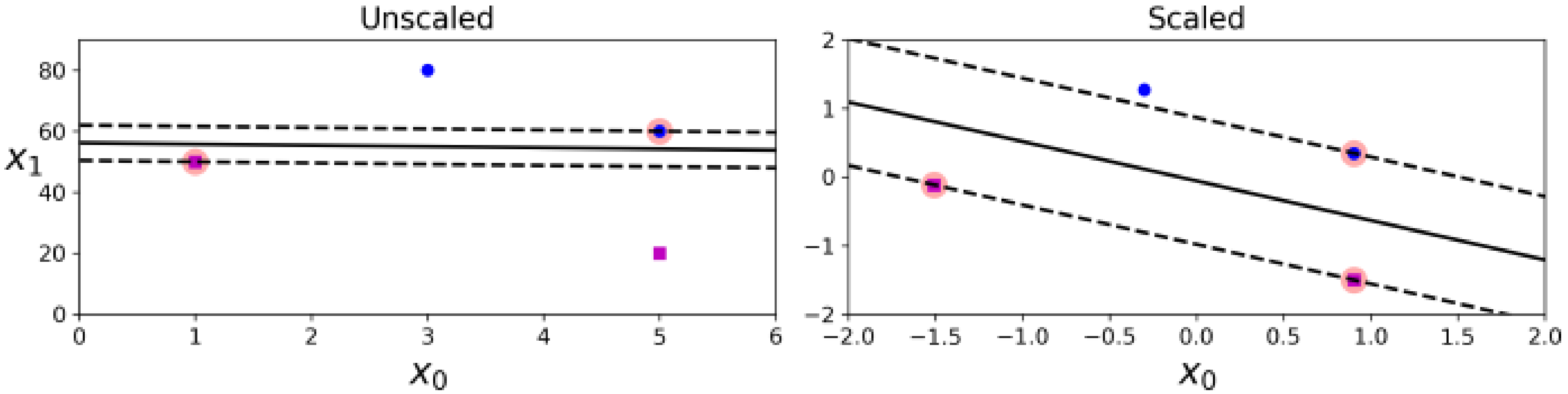
You can think of an SVM classifier as fitting **the widest possible street** (represented by the parallel dashed lines) between the classes. This is called large margin classification.

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).



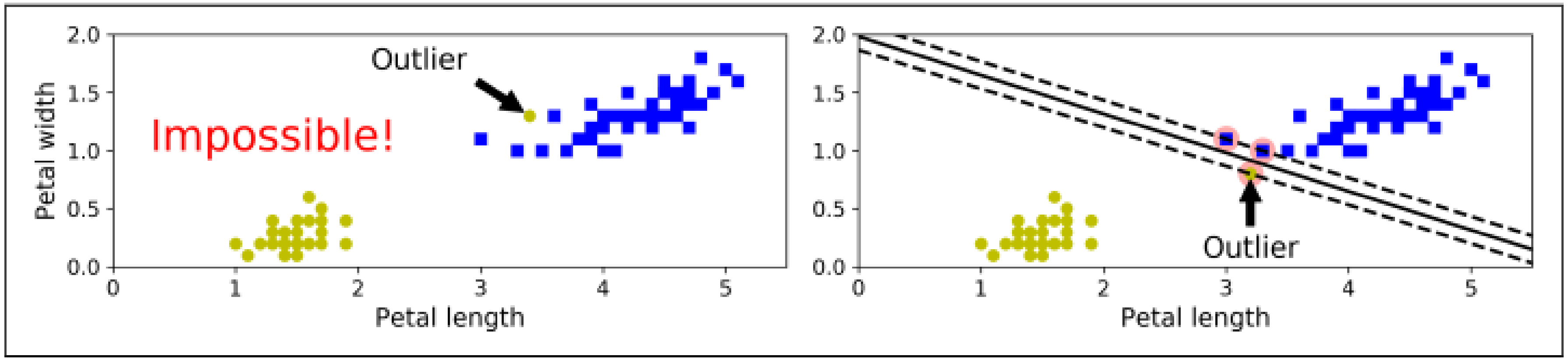
Importance of Good Scale

SVMs are sensitive to the **feature scales**, as you can see in below Figure: on the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street (represented by the parallel dashed lines) is close to horizontal. After feature scaling (e.g., using Scikit-Learn’s **StandardScaler**), the decision boundary looks much better (on the right plot)



Effects of Outliers on Classification

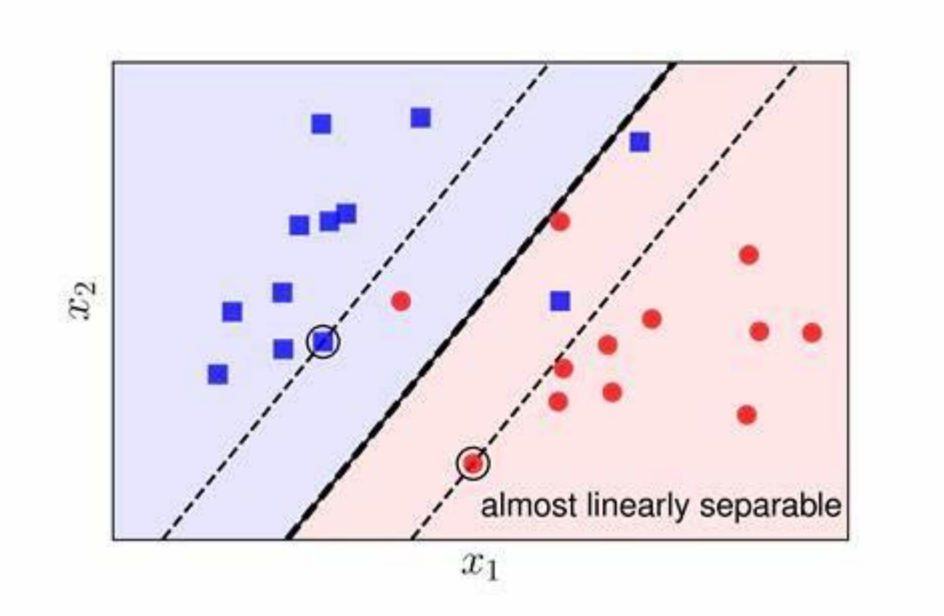
If we strictly impose that all instances be off the street and on the right side, this is called ***hard margin classification***. There are two main issues with hard margin classification. First, it only works if the data is linearly separable, and second it is quite sensitive to outliers. Left Figure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

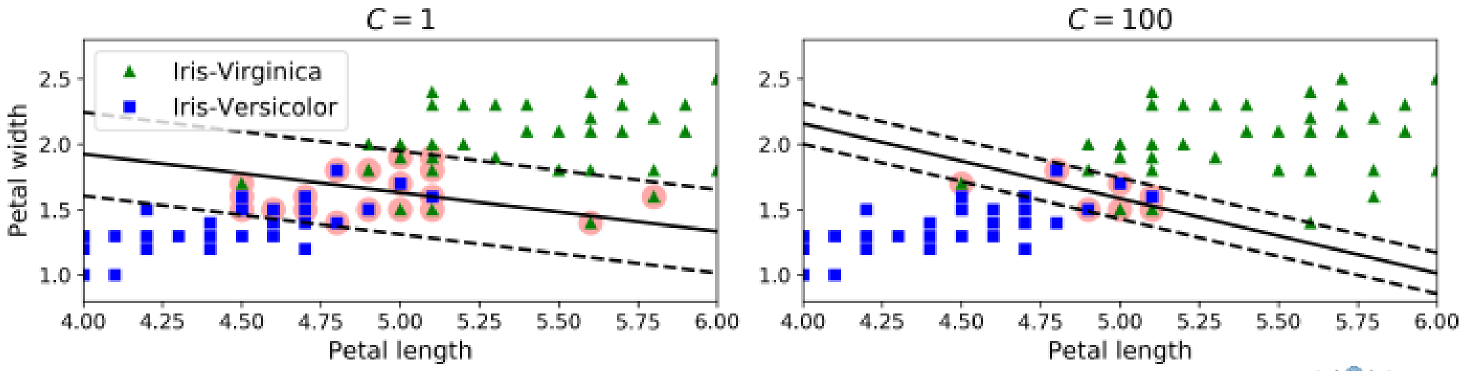


Soft-Margin Classification

To avoid these issues it is preferable 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*** .

In Scikit-Learn's SVM classes, you can control this balance using the C hyperparameter: a smaller C value leads to a wider street but more margin violations.



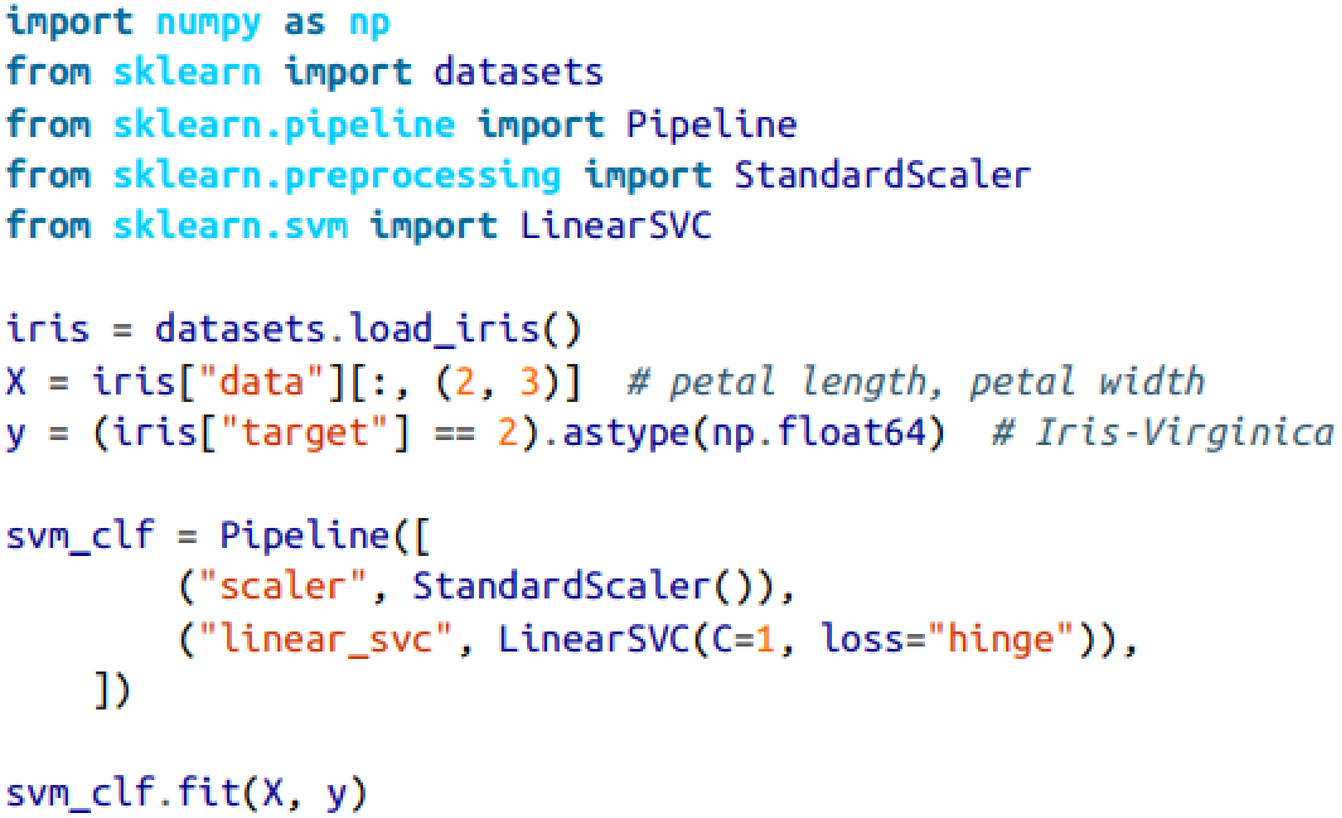


Linear SVM Model Code

Create a Pipeline containing

- StandardScaler
- LinearSVC

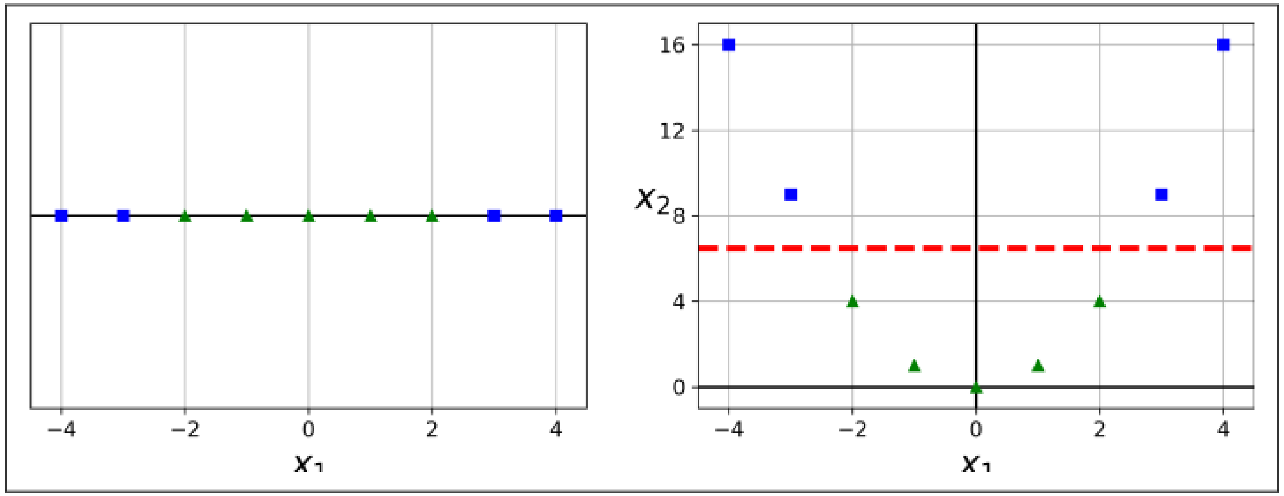
The following Scikit-Learn code loads the iris dataset, scales the features, and then trains a linear SVM model (using the LinearSVC class with C = 0.1) to detect Iris-Virginica flowers.



```
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)
```

Nonlinear SVM Classification

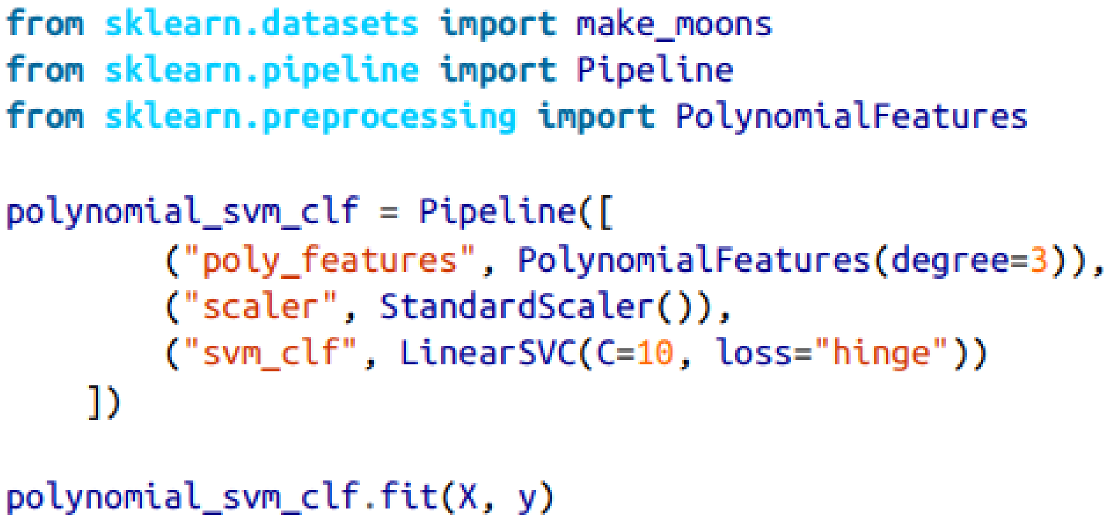
Although linear SVM classifiers are efficient and work surprisingly well in many cases, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to **add more features**, such as polynomial features; Consider the left plot in Figure: it represents a simple dataset with just one feature x_1. This dataset is not linearly separable But if you add a second feature $x_2 = x_1^2$, the resulting 2D dataset is perfectly linearly separable.



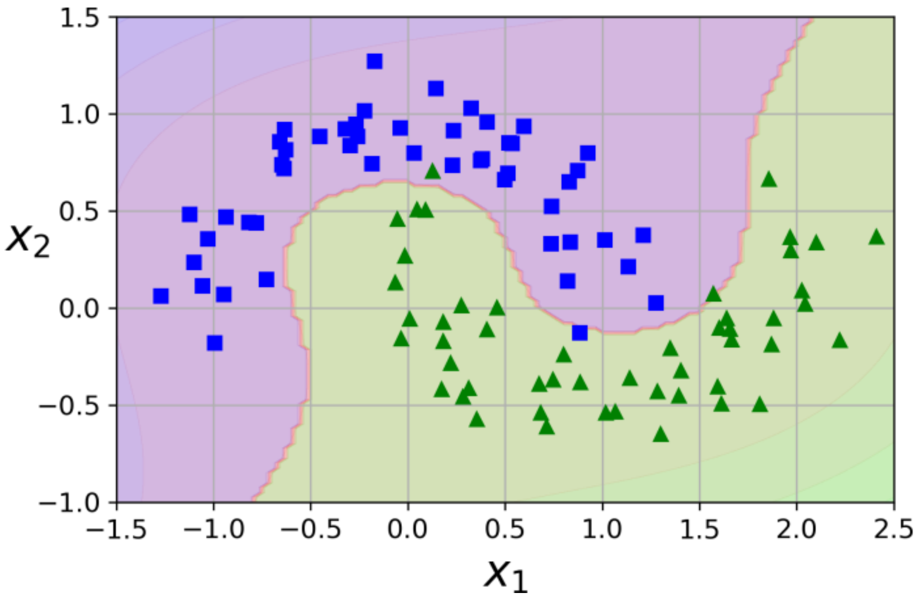
Nonlinear SVM Classification Code

Create a **Pipeline** containing

- **PolynomialFeatures** transformer
- **StandardScaler**
- **LinearSVC**



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

polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ])

polynomial_svm_clf.fit(X, y)
```

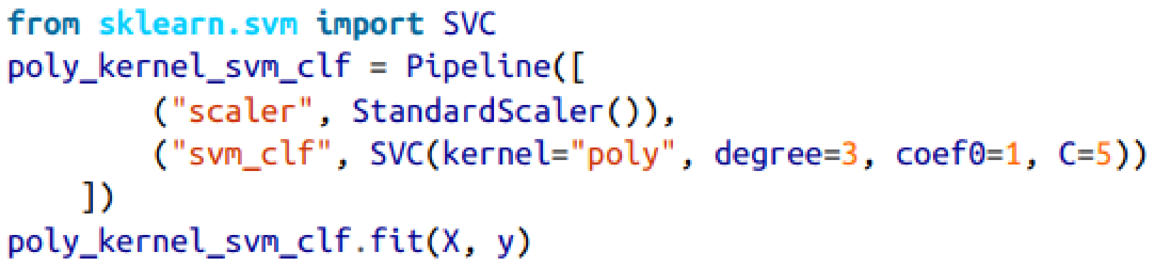


Polynomial Kernel

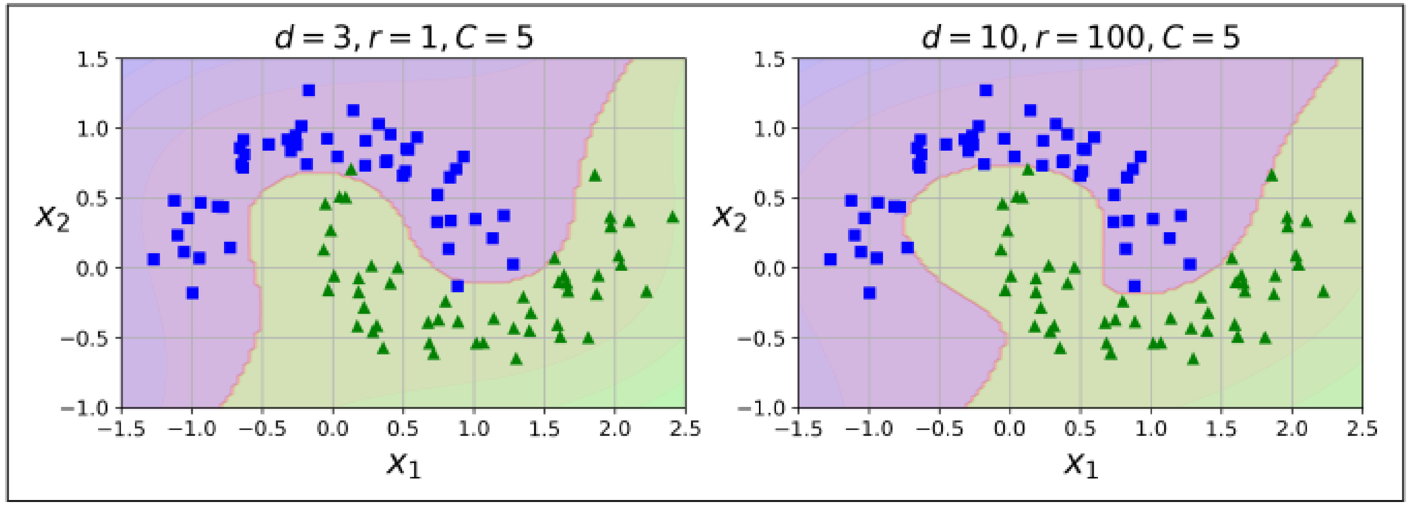
Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs), but at a low polynomial degree it cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow. Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the **kernel trick** (it is explained in a moment). It makes it possible to get the same result as if you added many polynomial features, even with very high degree polynomials, without actually having to add them.

Polynomial Kernel Code

This code trains an SVM classifier using a third-degree polynomial kernel. It is represented on the left. 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.

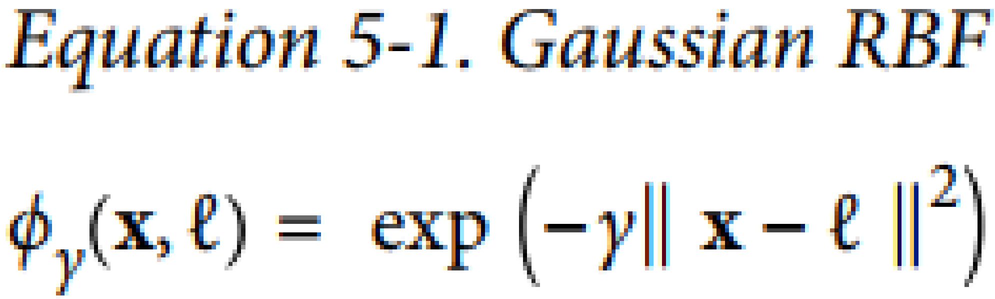


```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```



Adding Similarity Features

Another technique to tackle nonlinear problems is to add features computed using a *similarity function* that measures how much each instance resembles a particular *landmark*. let’s define the similarity function to be the Gaussian *Radial Basis Function* (*RBF*)



Equation 5-1. Gaussian RBF

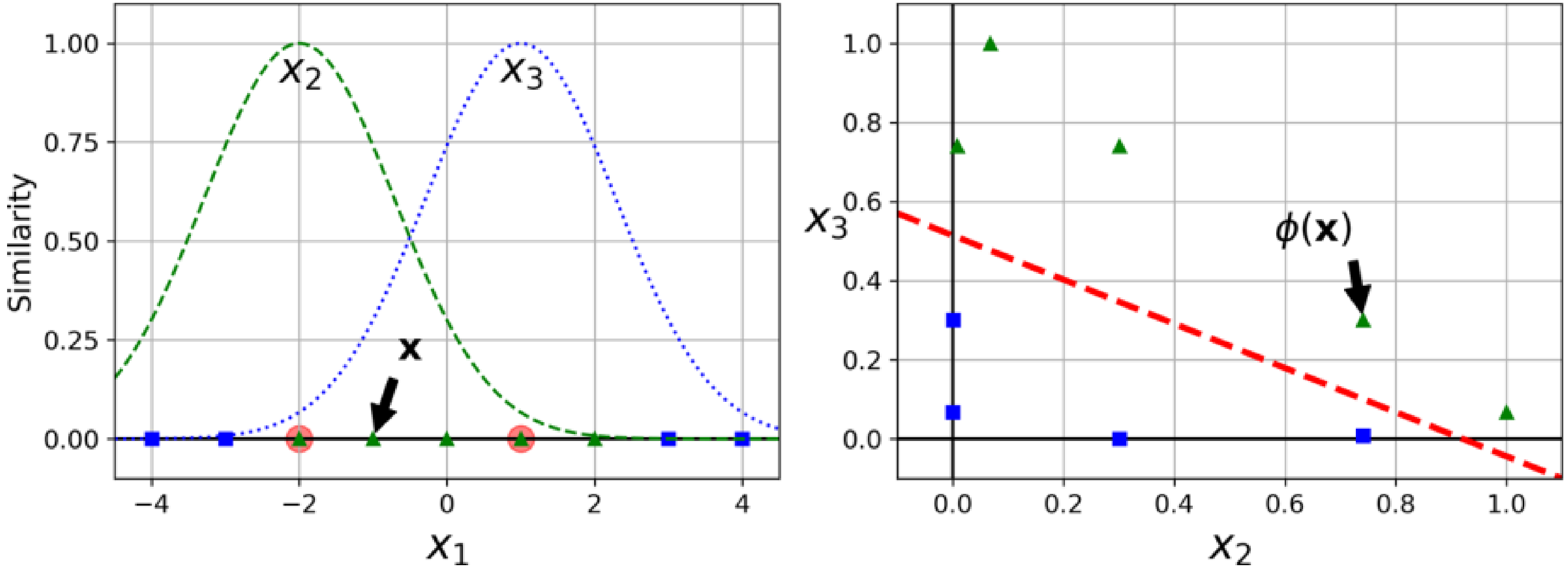
$$\phi_{\gamma}(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

RBF Kernel

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. It is located at a distance of 1 from the first landmark and 2 from the second landmark. Therefore, its new features are x = exp(–0.3 × 1 × 1) ≈ 0.74 and x = exp(–0.3 × 2 × 2) ≈ 0.30.

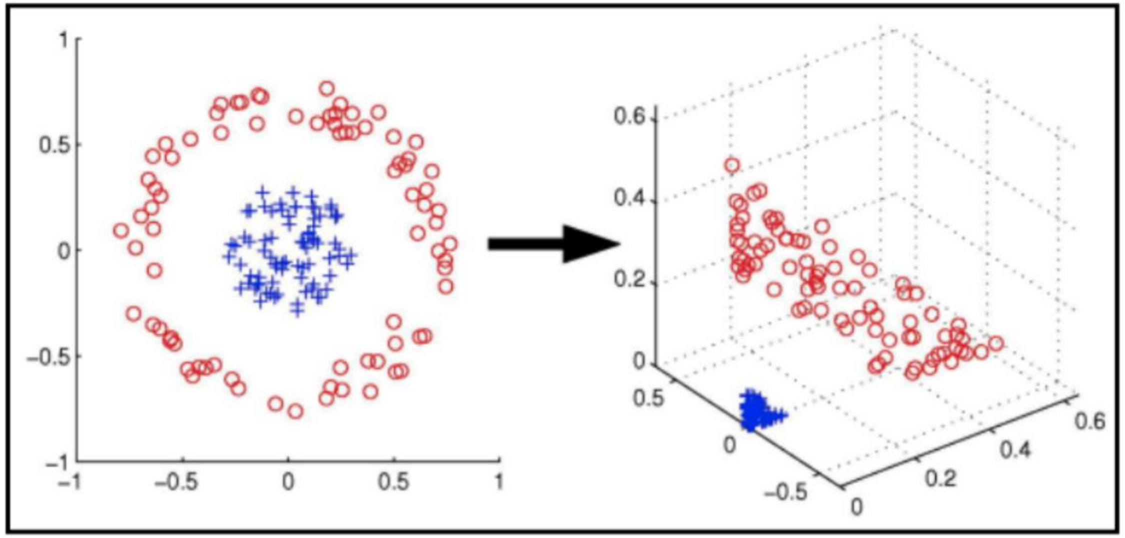
The plot on the right shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

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



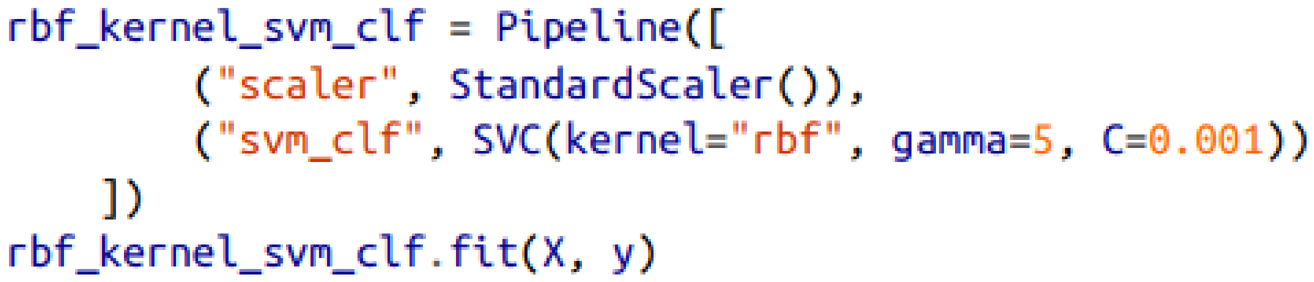
RBF Kernel

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 code

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)
```

Gaussian RBF Kernel code

Increasing **gamma** makes the bell-shaped curve narrower (see the righthand plots). 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 **gamma** 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)

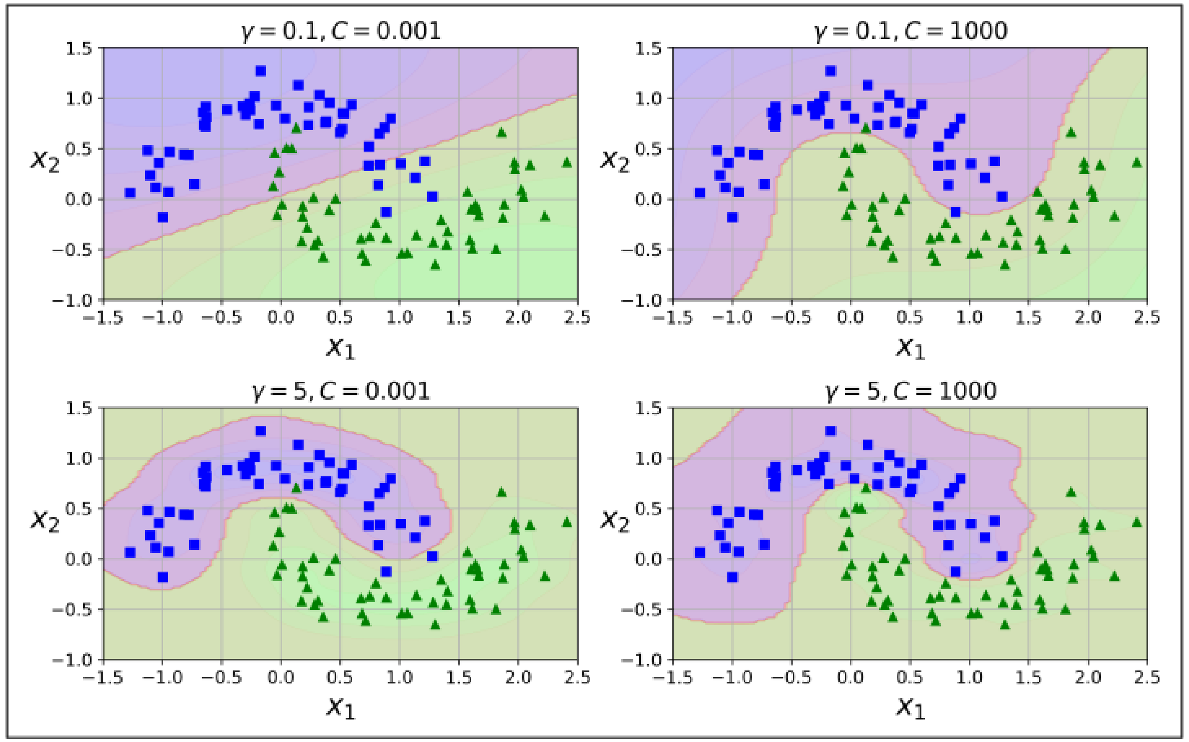


Figure 5-9. SVM classifiers using an RBF kernel

Kernel Selection

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 try the Gaussian RBF kernel as well; it works well in most cases. Then, if you have spare time and computing power, you can also experiment with a few other kernels using cross-validation and grid search, especially if there are kernels specialized for your training set’s data structure.

Kernel Selection

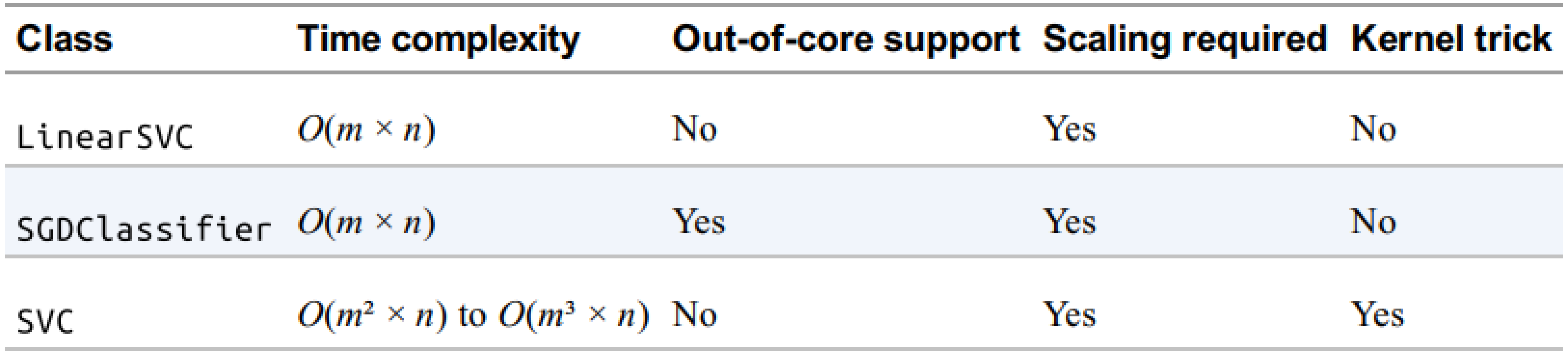
Look at the table that compares Sklearn classes for SVM classification.

- *m* denotes the number of training examples
- *n* denotes the number of features

O(m×n) means that the training time scales linearly with *m* and *n.*

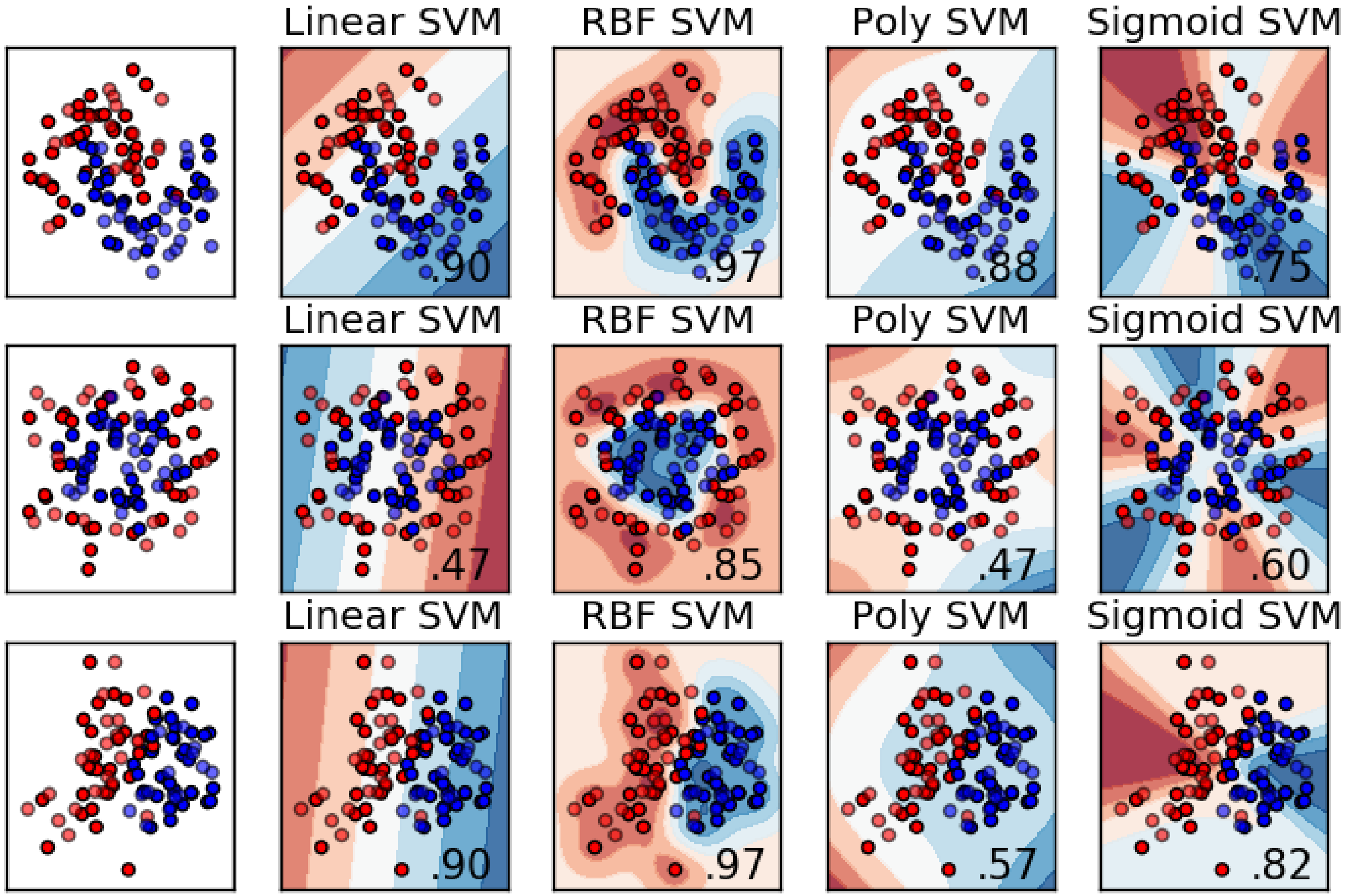
For example, if you double the number of training examples, it takes the first 2 classes twice the time to train the model and 4-8 times slower for the third algorithm.

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
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM Kernel Selection



SVM for Regression

SVM also supports linear and nonlinear regression. 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 hyper parameter ϵ.

large margin (ϵ = 1.5) and a small margin (ϵ = 0.5)

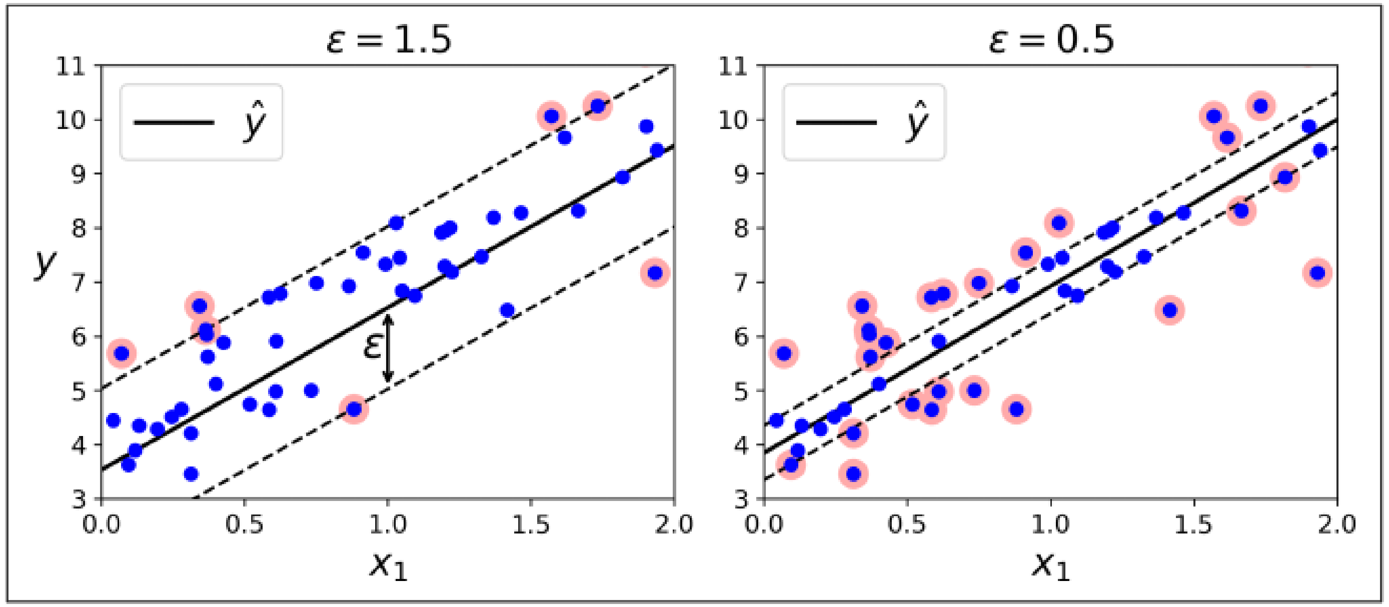


Figure 5-10. SVM Regression

SVM Regression Code

You can use Scikit-Learn's **LinearSVR** class to perform linear SVM Regression. The following code produces the model represented on the left in (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)
```

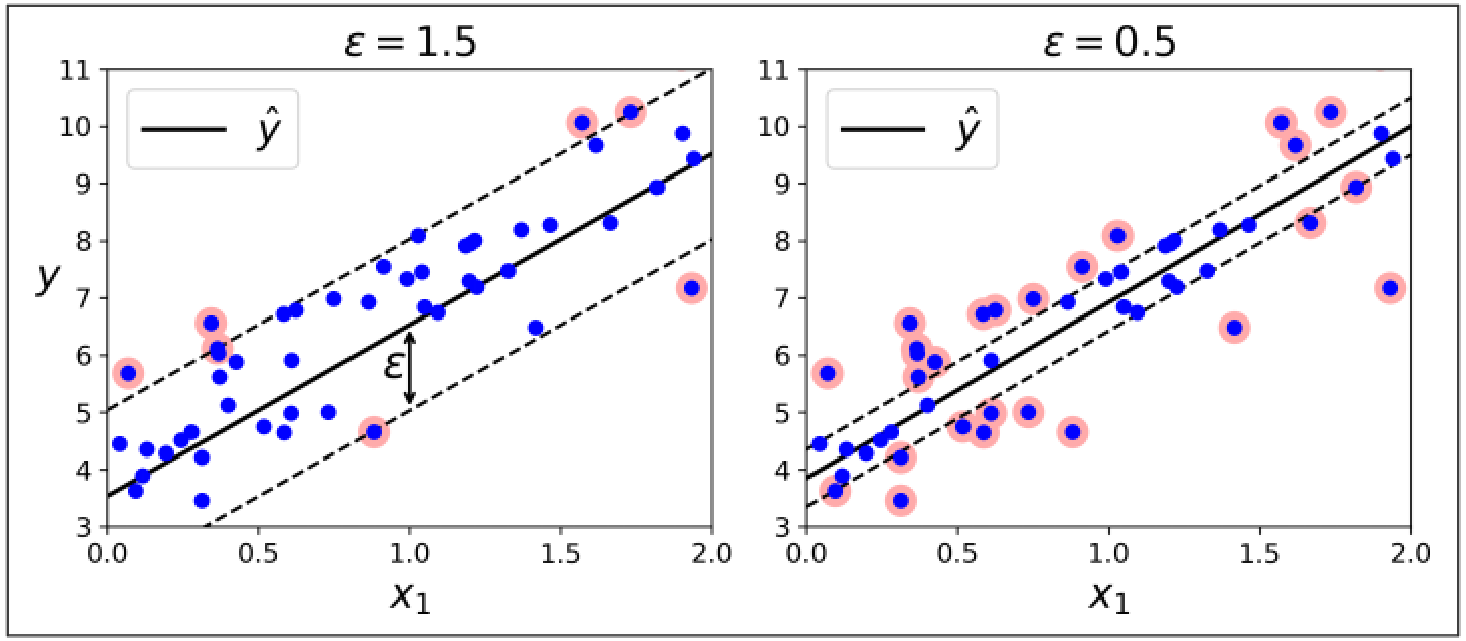


Figure 5-10. SVM Regression

Kernelized SVM Regression

To tackle nonlinear regression tasks, you can use a kernelized SVM model. For example, below Figure 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).

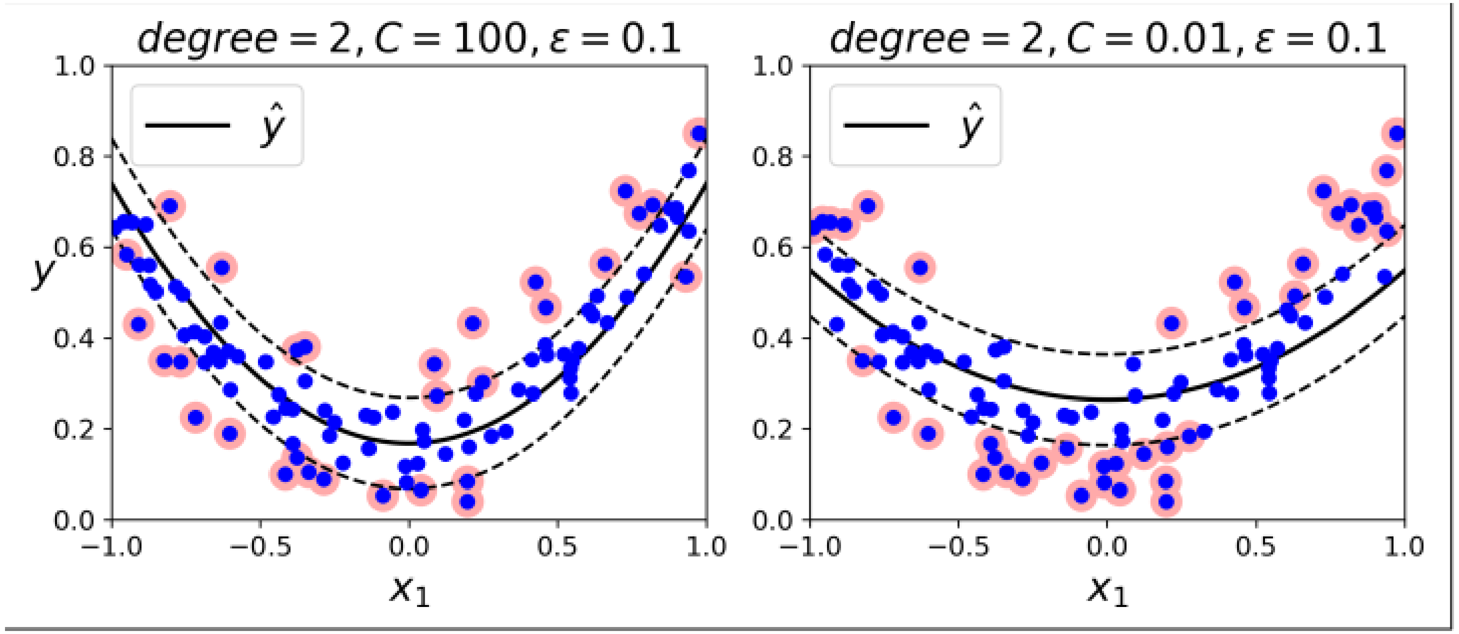
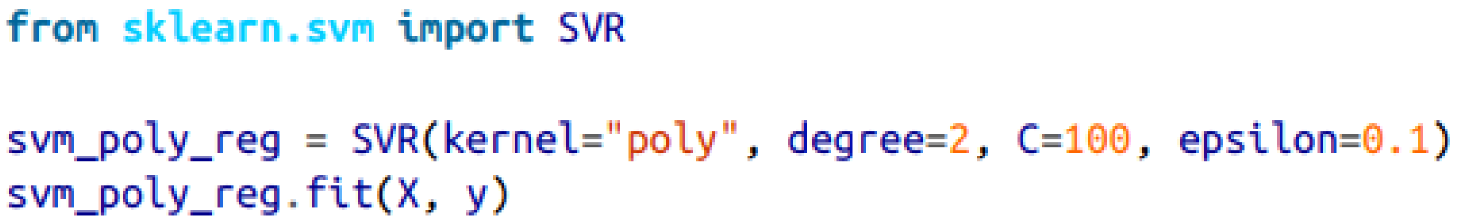


Figure 5-11. SVM regression using a 2nd-degree polynomial kernel

SVM Regression Code



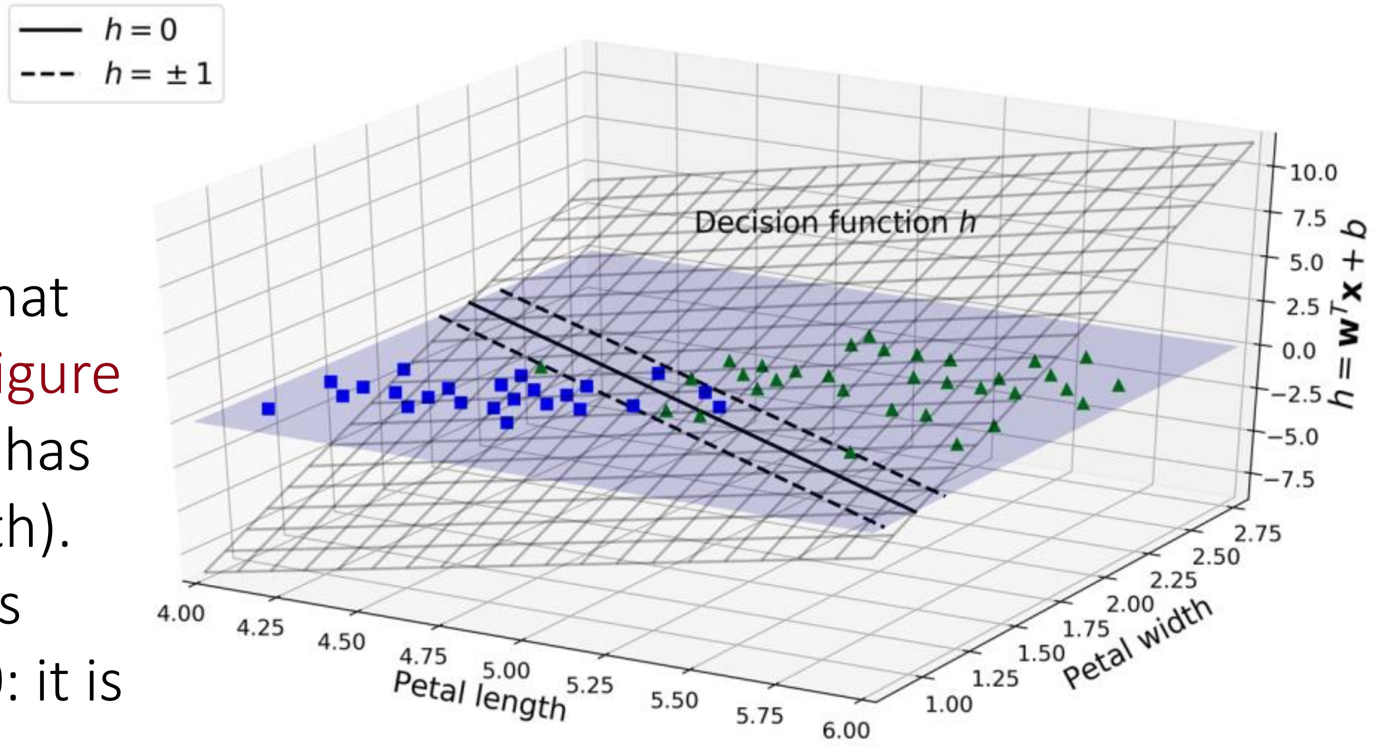
```
from sklearn.svm import SVR

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

Note: 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).

Under the Hood

The linear SVM classifier model predicts the class of a new instance **x** by simply computing the decision function $\mathbf{w}^T \cdot \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b$. If the result is positive, the predicted class $\hat{y}$ is the positive class (1), and otherwise it is the negative class (0)

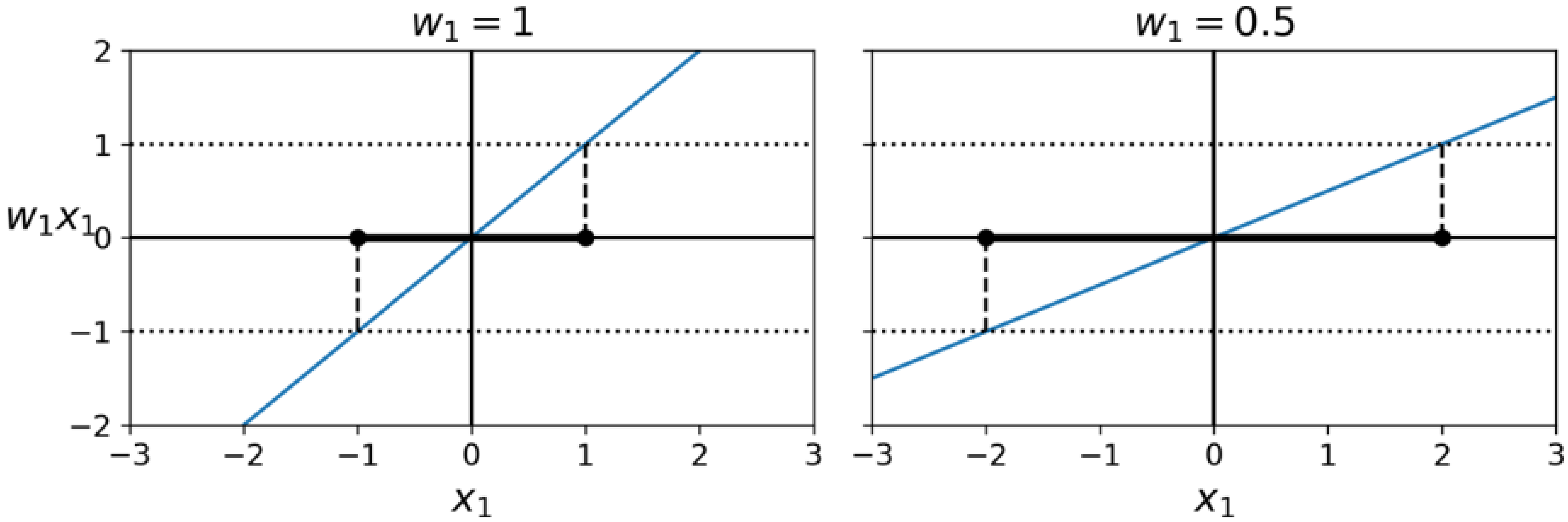
$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^\intercal \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^\intercal \mathbf{x} + b \geq 0 \end{cases}$$



The figure shows the decision function that corresponds to the model in the left in Figure 5-4: it is a 2D plane because 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)

Training Objective

Consider the slope of the decision function: it is equal to the norm of the weight vector, ‖ **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. This may be easier to visualize in 2D, as shown in the figure. The smaller the weight vector **w**, the larger the margin.



Training Objective

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 $t = -1$ for negative instances (if $y = 0$) and $t = 1$ for positive instances (if $y = 1$), then we can express this constraint as $\boldsymbol{t^{(i)}} \mathbf{(w^\top x^{(i)} + \boldsymbol{b})} \geq 1$ for all instances.

We can therefore express the **hard margin linear SVM** classifier objective as the constrained optimization problem:

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

Training Objective

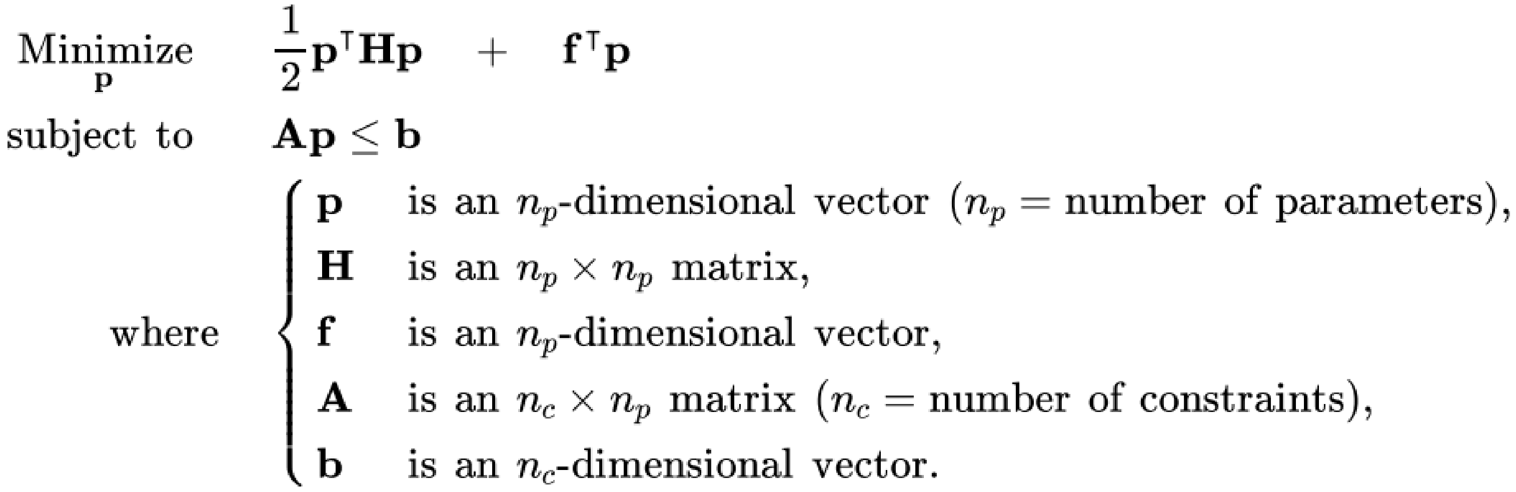
To get the **soft margin objective**, we need to introduce a *slack variable* $\zeta \geq 0$ for each instance: ζ measures how much the *i* instance is allowed to violate the margin. We now have two conflicting objectives: make the slack variables as small as possible to reduce the margin violations, and make $\boldsymbol{w^T w}$ 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:

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

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 by using a variety of techniques that are outside the scope of this book.

The general problem formulation is given by



$$\begin{array}{ll} \underset{\mathbf{p}}{\text{Minimize}} & \frac{1}{2}\mathbf{p}^{\intercal}\mathbf{H}\mathbf{p} \quad + \quad \mathbf{f}^{\intercal}\mathbf{p} \\ \text{subject to} & \mathbf{A}\mathbf{p} \le \mathbf{b} \end{array}$$

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

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

Quadratic Programming

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

$$\begin{array}{lll} \underset{\mathbf{p}}{\text{Minimize}} & \frac{1}{2}\mathbf{p}^\intercal \mathbf{H} \mathbf{p} & + \quad \mathbf{f}^\intercal \mathbf{p} \\ \text{subject to} & \mathbf{A}\mathbf{p} \le \mathbf{b} & \end{array}$$

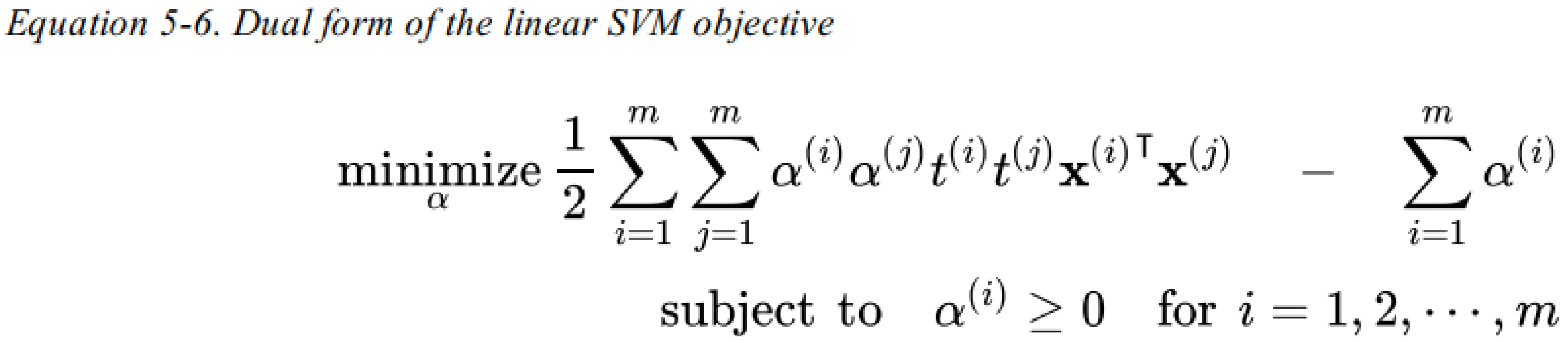
Quadratic Programming

One way to train a hard margin linear SVM classifier is to use an off-the-shelf QP solver and pass it the preceding parameters. The resulting vector $\boldsymbol{p}$ will contain the bias term $b = p_0$ and the feature weights $w_i = p_i$ for $i = 1, 2, \cdots, n$. Similarly, you can use a QP solver to solve the soft margin problem.

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 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; 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 Appendix C)



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

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

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

The Dual Problem

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

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

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

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

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

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

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

The Dual Problem

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. So what is this kernel trick, anyway?

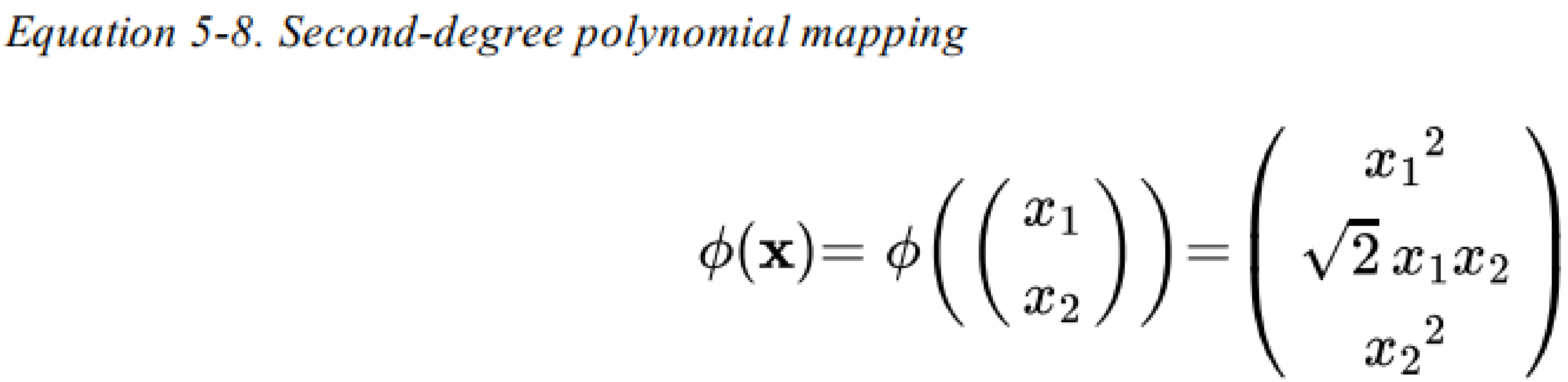
Equation 5-6. Dual form of the linear SVM objective

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

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

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. Notice that the transformed vector is 3D instead of 2D.



Equation 5-8. Second-degree polynomial mapping

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

Kernelized SVMs

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 of the transformed vectors (See Equation 5-9).

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

$$\begin{aligned}\phi(\mathbf{a})^{\intercal}\phi\left(\mathbf{b}\right) &= \begin{pmatrix} {a_1}^2 \\ \sqrt{2}\,a_1 a_2 \\ {a_2}^2 \end{pmatrix}^{\intercal} \begin{pmatrix} {b_1}^2 \\ \sqrt{2}\,b_1 b_2 \\ {b_2}^2 \end{pmatrix} = {a_1}^2{b_1}^2 + 2a_1b_1a_2b_2 + {a_2}^2{b_2}^2 \\ &= \left(a_1b_1 + a_2b_2\right)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^{\intercal}\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)^2 = \left(\mathbf{a}^{\intercal}\mathbf{b}\right)^2\end{aligned}$$

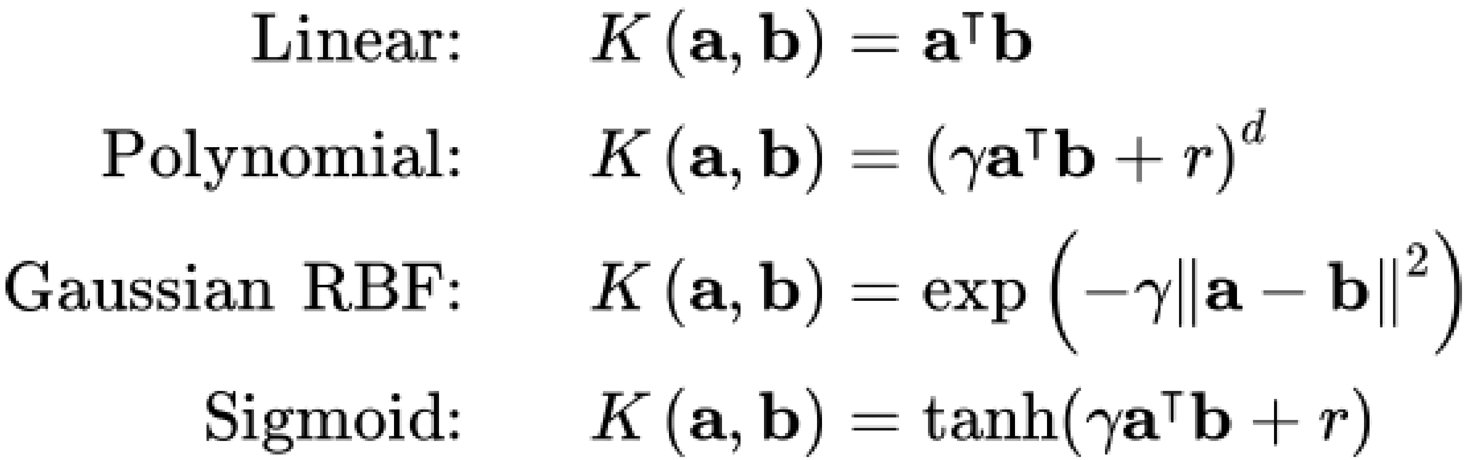
How about that? The dot product of the transformed vectors is equal to the square of the dot product of the original vectors: $\boldsymbol{\phi(a)^T\phi(b)} = \boldsymbol{(a^Tb)}^2$.

Kernelized SVMs

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 $\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\ \phi(\boldsymbol{x}^{(j)})$. 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 $\left(\boldsymbol{x}^{(i)^{T}}\boldsymbol{x}^{(j)}\right)^{2}$. So, you don't need to transform the training instances at all; just replace the dot product by its square in Equation 5-6. 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.

Kernelized SVMs

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



Linear:	$K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\intercal\mathbf{b}$
Polynomial:	$K(\mathbf{a}, \mathbf{b}) = \left(\gamma\mathbf{a}^\intercal\mathbf{b} + r\right)^d$
Gaussian RBF:	$K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma\lVert\mathbf{a} - \mathbf{b}\rVert^2\right)$
Sigmoid:	$K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma\mathbf{a}^\intercal\mathbf{b} + r\right)$

References

- *Gel̀ron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems (2nd ed.). O’Reilly.*
- 1.4. Support Vector Machines — scikit-learn 1.2.2 documentation