SAIF

Shanghai Advanced Institute of Finance

上海高级金融学院

量化俱乐部-机器学习-SVM

2019-08-31

Menu

1. Linear SVM Classification
2. Nonlinear SVM Classification
3. SVM Regression
4. Q&A

自我介绍

张骁喆，2010年大连理工大学软件学院本科毕业，高金FMBA 2017PTE。9年计算机工作经验，熟悉开发，测试，运维，项目管理，产品设计各环节。
曾就职eBay，唯品会，2016加入SAP，现担任SAP Cloud部门开发团队主管。
2017开始接触量化投资和python，目前毕业论文研究方向使用机器学习在A股进行量化投资。
量化投资/机器学习咨询培训/项目管理产品管理。

Intro

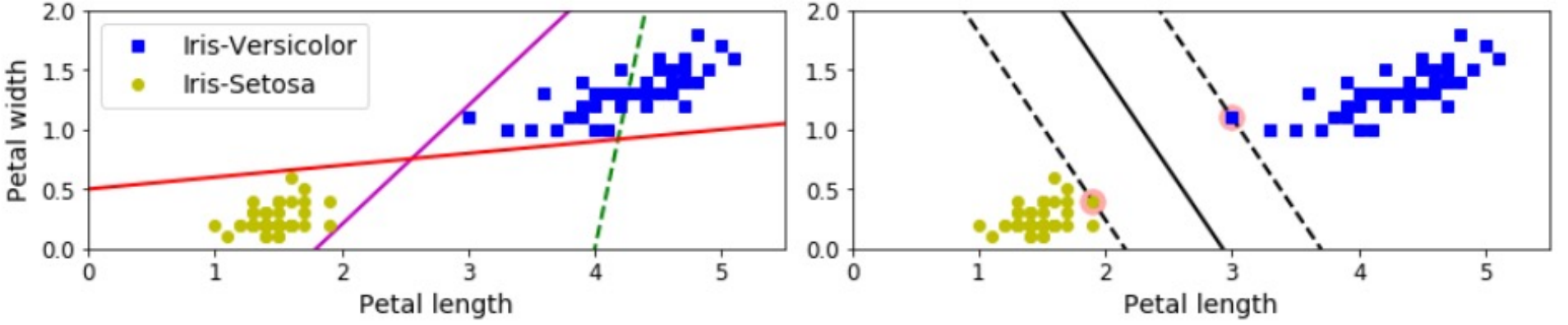
A Support Vector Machine (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox.

SVMs are particularly well suited for classification of complex but small- or medium-sized datasets

Linear Classification

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.

adding more training instances no affected, it is fully determined (or "supported") by the
instances located on the edge of the street. These instances are called the support vectors.



Linear Classification

Decision Function and Predictions :

The linear SVM classifier model predicts the class of a new instance x by simply computing the decision function wT· x + b = w1 x1 + ⋯ + wn xn + b:

ifthe result is positive, the predicted class ŷ is the positive class (1), or else it is the negative class (0);

Equation 5-2. Linear SVM classifier prediction

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

Linear Classification

In [41]:
```
from sklearn.svm import SVC
from sklearn import datasets

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

setosa_or_versicolor = (y == 0) | (y == 1)
X = X[setosa_or_versicolor]
y = y[setosa_or_versicolor]

# SVM Classifier model
svm_clf = SVC(kernel="linear", C=float("inf"))
svm_clf.fit(X, y)

# At the decision boundary, w0*x0 + w1*x1 + b = 0
# => x1 = -w0/w1 * x0 - b/w1
#w = svm_clf.coef_[0]
#w
#b = svm_clf.intercept_[0]
#b
#decision_boundary = -w[0]/w[1] * x0 - b/w[1]

#https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
svm_clf.support_vectors_
```

Out[41]:
```
array([[1.9, 0.4],
       [3. , 1.1]])
```

Linear Classification

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 w and b that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

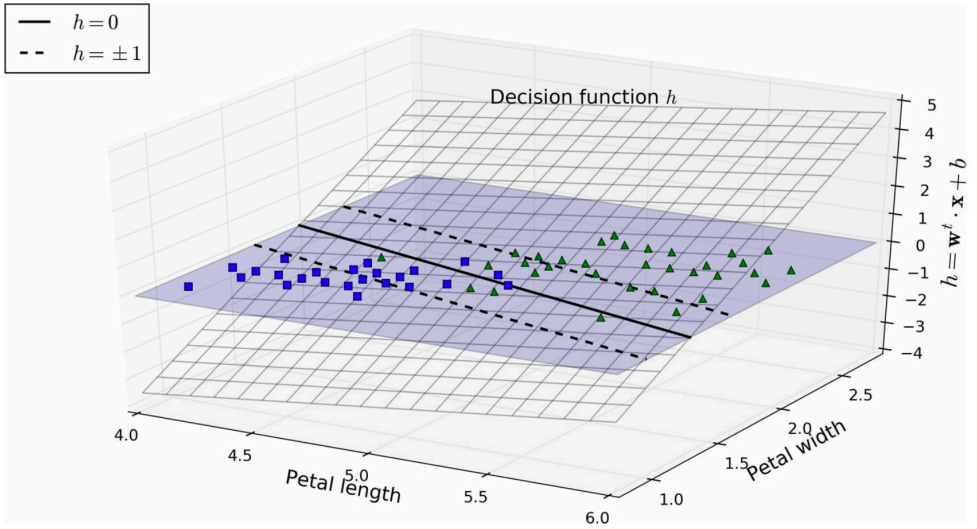
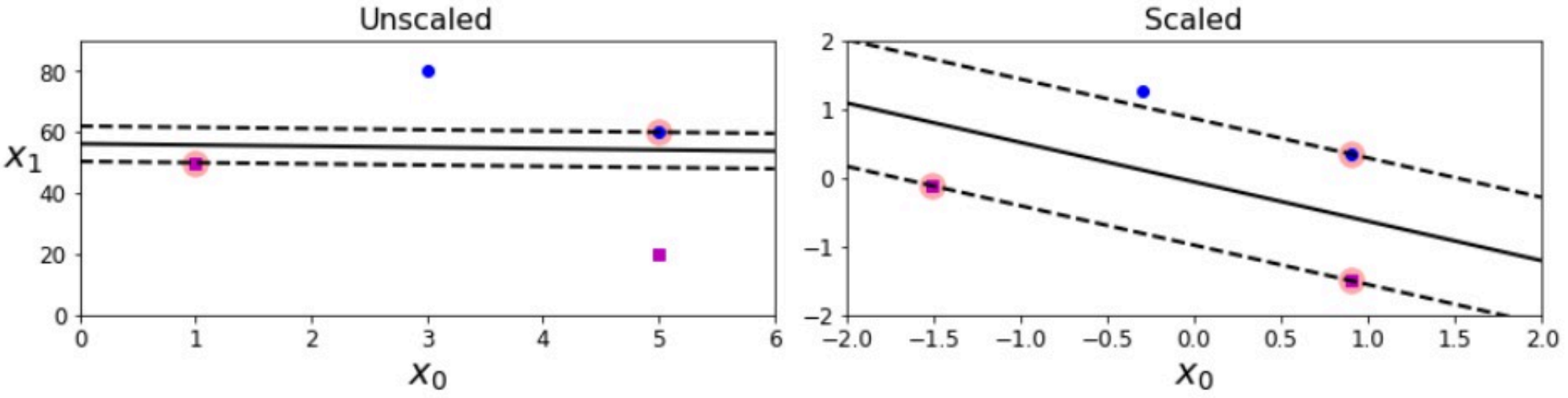


Figure 5-12. Decision function for the iris dataset

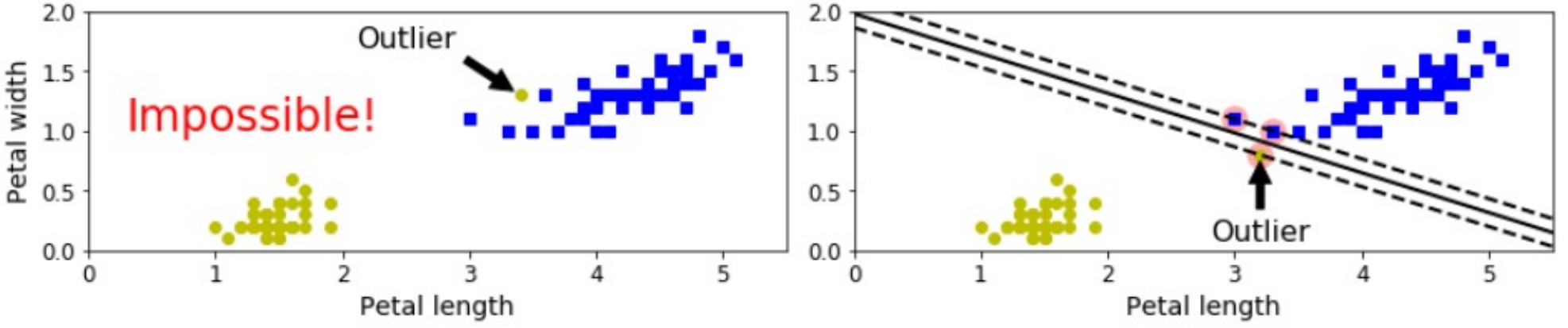
Linear Classification

Feature Scale: 左边垂直方向的scale比水平大很多，所以训练出来的street接近水平。After feature scaling (e.g., using Scikit-Learn’s StandardScaler), 新训练出来的decision boundary看起来好很多。



Linear Classification-Soft Margin

hard margin classification：所有的点都在街道两边，并且没有错误分类。主要两个challenge:
1.左图那种画不出来.
2. 右图那种outlier导致街道很窄



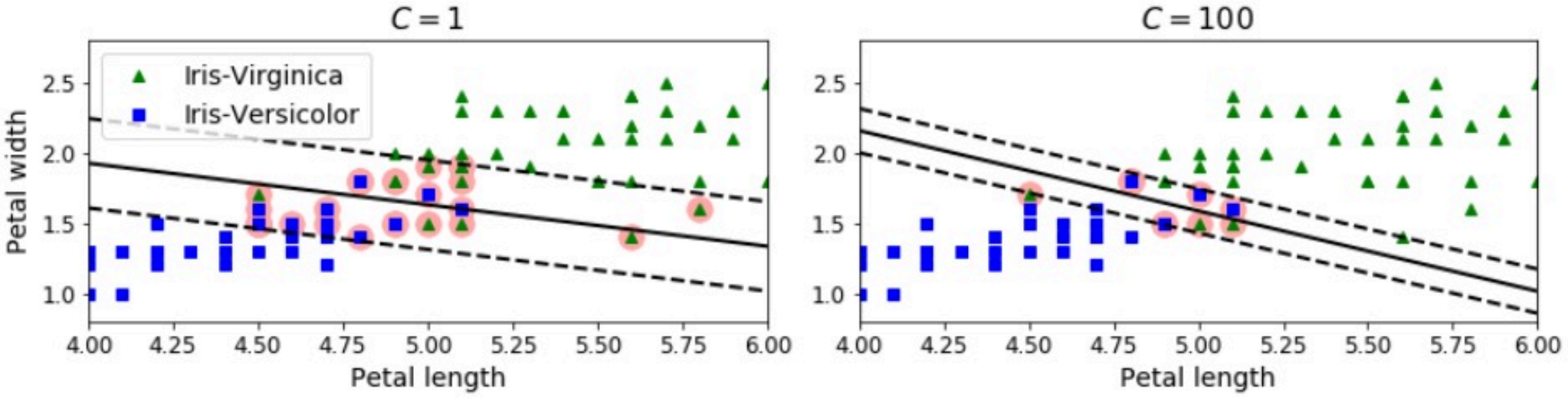
Linear Classification-Soft Margin

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**

Linear Classification-Soft Margin

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 Classification-Soft Margin

Shanghai Advanced Institute of Finance 上海高级金融学院

Demo with LinearSVC :

In [6]:
```
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", random_state=42)),
    ])

svm_clf.fit(X, y)
```

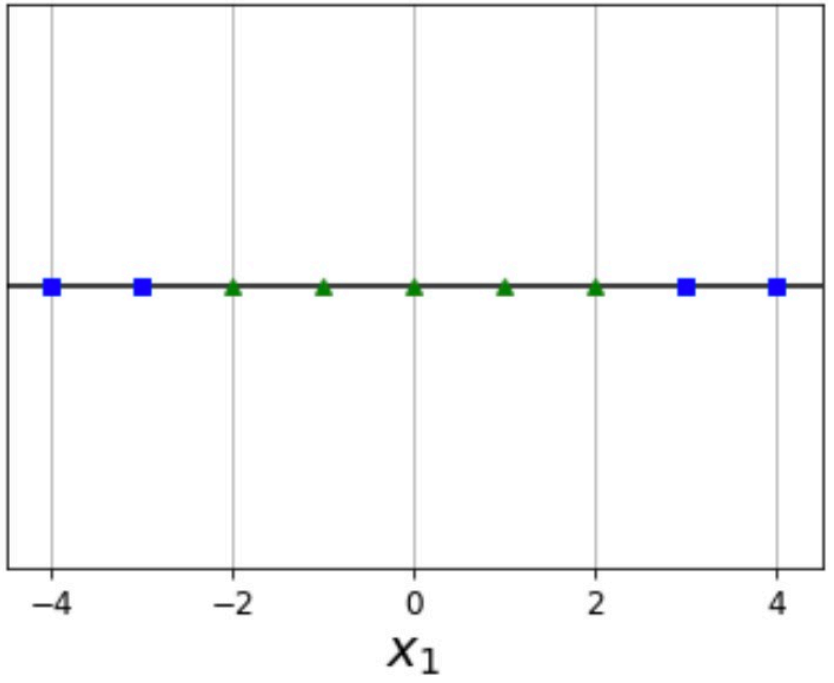
Out[6]:
```
Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('linear_svc',
                 LinearSVC(C=1, class_weight=None, dual=True,
                           fit_intercept=True, intercept_scaling=1,
                           loss='hinge', max_iter=1000, multi_class='ovr',
                           penalty='l2', random_state=42, tol=0.0001,
                           verbose=0))],
         verbose=False)
```

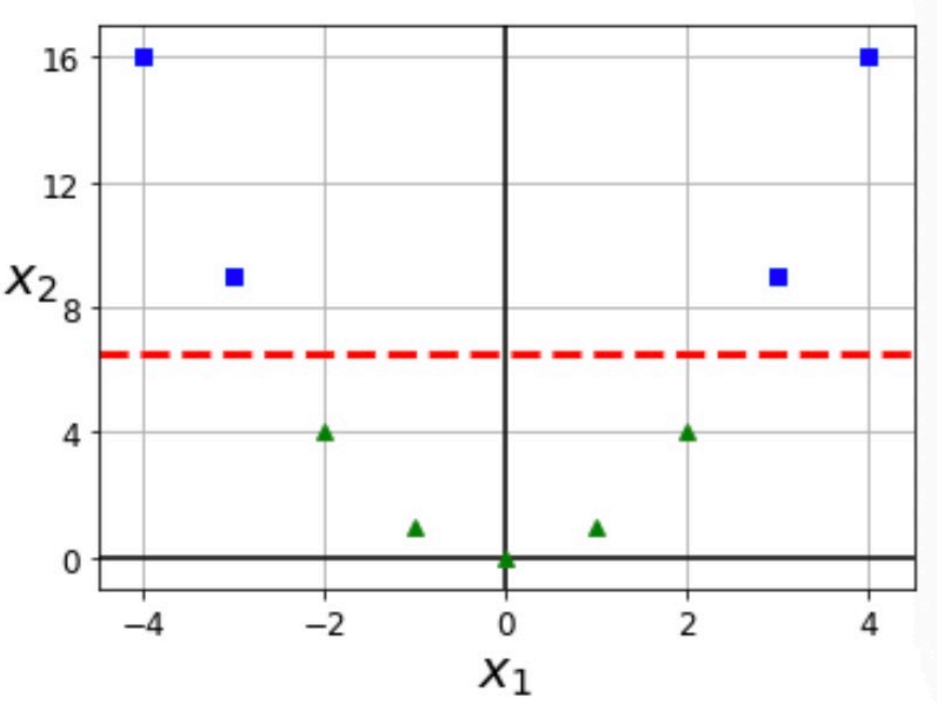
In [7]:
```
svm_clf.predict([[5.5, 1.7]])
```

Out[7]:
```
array([1.])
```

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

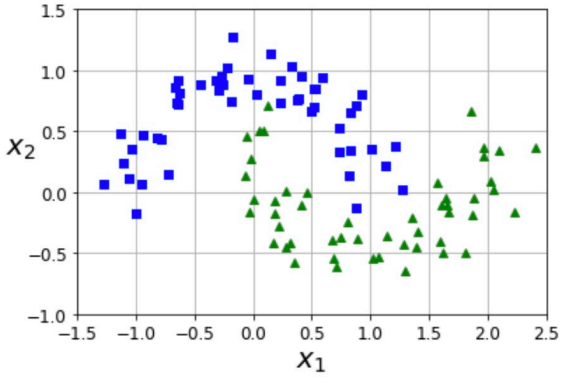




Nonlinear SVM Classification

To implement this idea using Scikit-Learn, you can create a Pipeline containing a PolynomialFeatures transformer (discussed in "Polynomial Regression"), followed by a StandardScaler and a LinearSVC. Let's test this on the moons dataset

In [18]:
```
# user guide for make_moons   https://scikit-learn.org/stable/da
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()
```



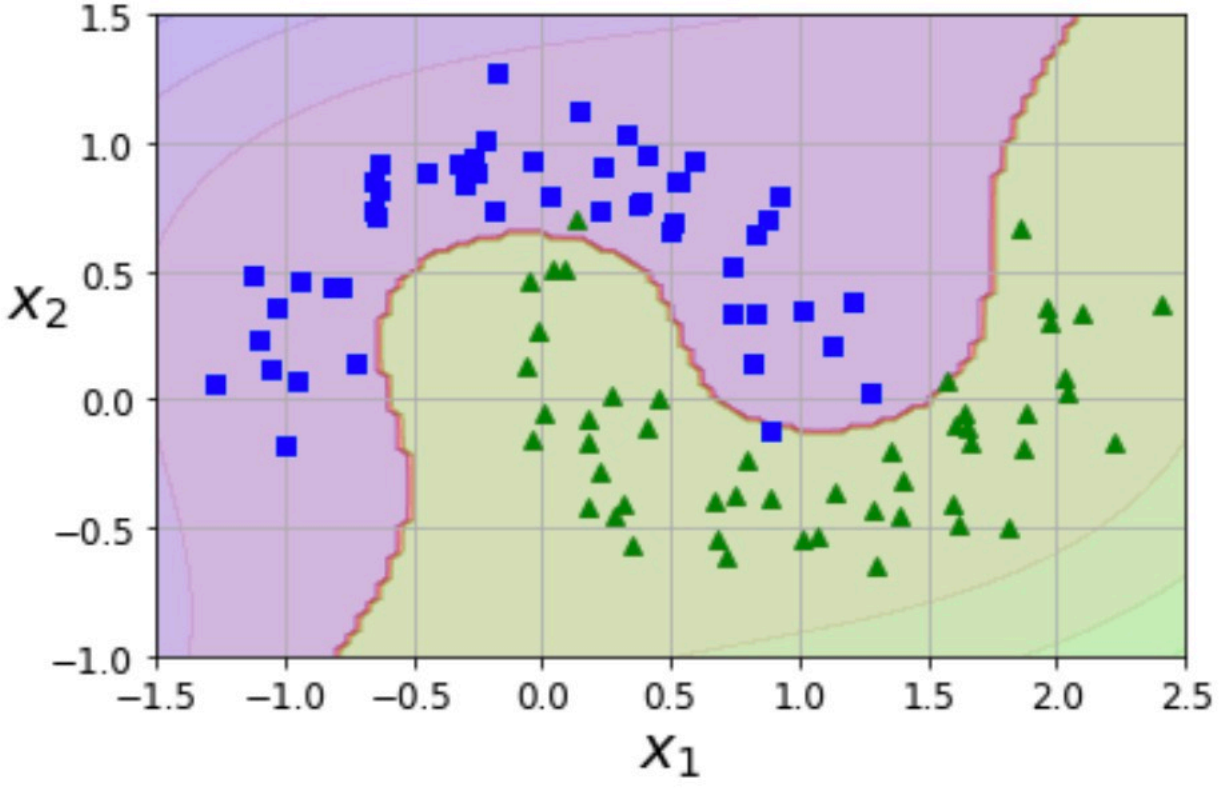
Nonlinear SVM Classification

To implement this idea using Scikit-Learn, you can create a Pipeline containing a PolynomialFeatures transformer (discussed in "Polynomial Regression"), followed by a StandardScaler and a LinearSVC. Let's test this on the moons dataset

In [25]:
```
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", random_state=42))
    ])

polynomial_svm_clf.fit(X, y)
```

Nonlinear SVM Classification

To implement this idea using Scikit-Learn, you can create a Pipeline containing a PolynomialFeatures transformer (discussed in "Polynomial Regression"), followed by a StandardScaler and a LinearSVC. Let's test this on the moons dataset



Nonlinear – Polynomial Kernal

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. So there is no combinatorial explosion of the number of features since you don’ t actually add any features.

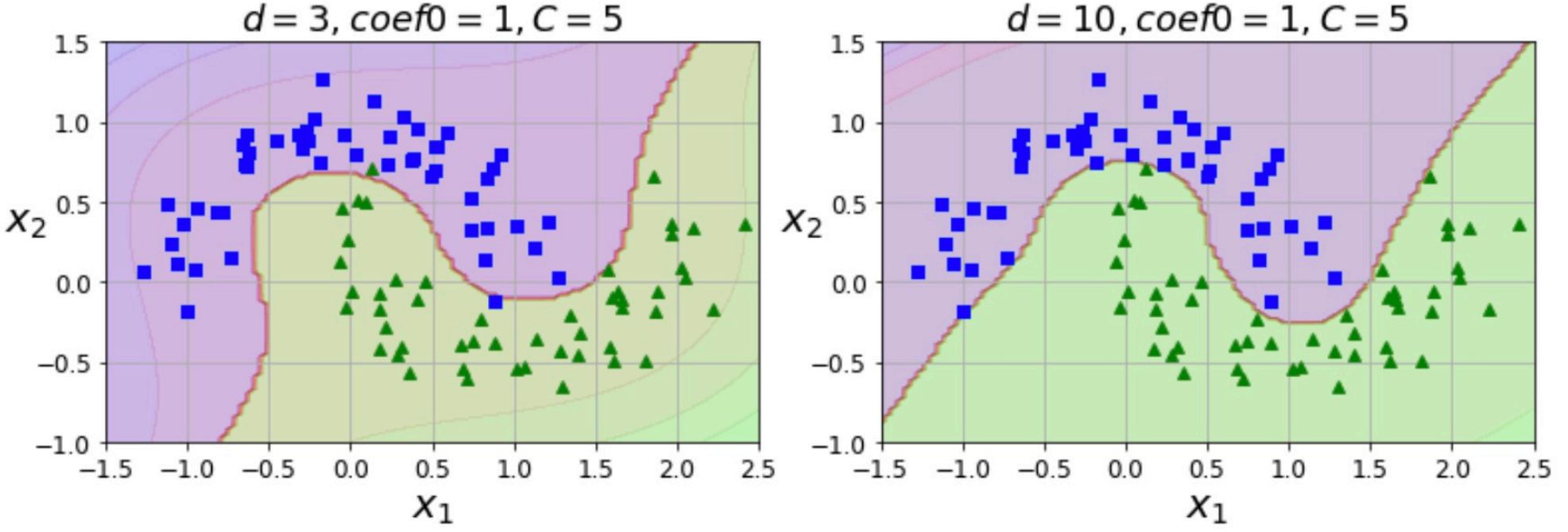
Nonlinear – Polynomial Kernal

This trick is implemented by the SVC class. Let’ s test it on the moons dataset

In [32]:
```
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)
```

Nonlinear – Polynomial Kernal

This trick is implemented by the SVC class. Let’ s test it on the moons dataset



Nonlinear – 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.

Next, let’ s define the similarity function to be the Gaussian Radial Basis Function (RBF)

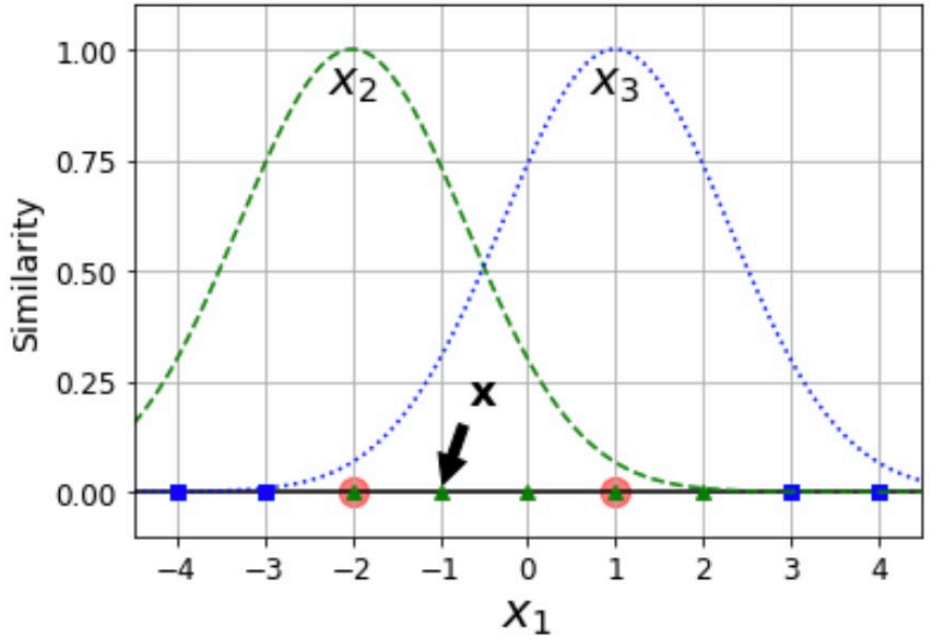
Equation 5-1. Gaussian RBF

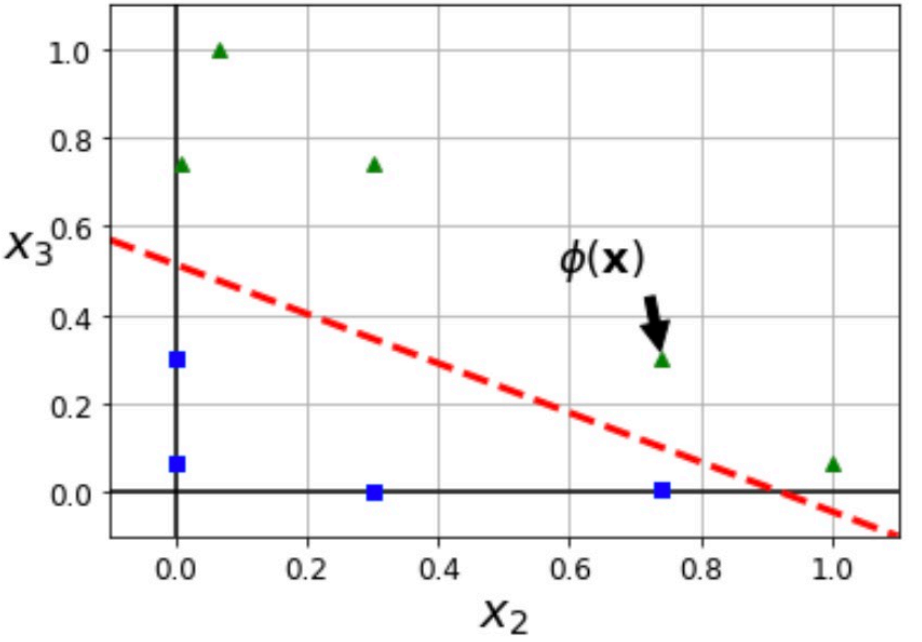
$$\phi\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

Nonlinear – Adding Similarity Features

For example, let’ s take the one-dimensional dataset discussed earlier and add two landmarks to it at x1 = –2 and x1 = 1

let’ s look at the instance x1 = –1: it is located at a distance of 1 from the first landmark, and 2 from the second landmark. Therefore its new features are x2 = exp (–0.3 × 12) ≈ 0.74 and x3 = exp (–0.3 × 22) ≈ 0.30.





Nonlinear – Adding Similarity Features

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.

This 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.

Nonlinear – 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. However, once again the kernel trick does its SVM magic: it makes it possible to obtain a similar result as if you had added many similarity features, without actually having to add them

Nonlinear – Gaussian RBF Kernel

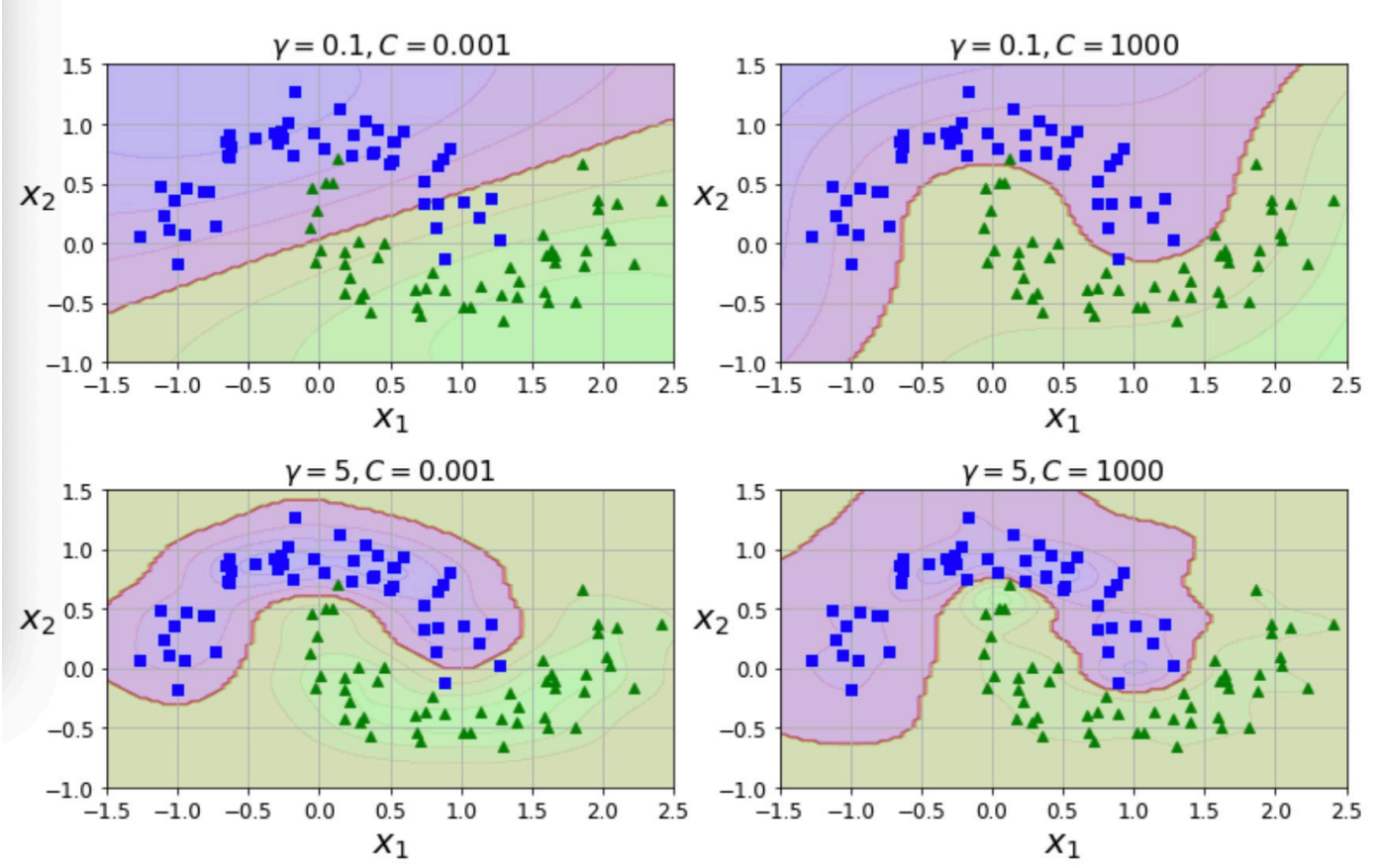
Let’ s try the Gaussian RBF kernel using the SVC class

```
rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)
```

```
Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('svm_clf',
                 SVC(C=0.001, cache_size=200, class_weight=None, coef0=0.0,
                     decision_function_shape='ovr', degree=3, gamma=5,
                     kernel='rbf', max_iter=-1, probability=False,
                     random_state=None, shrinking=True, tol=0.001,
                     verbose=False))],
         verbose=False)
```

Nonlinear – Gaussian RBF Kernel

The other plots show models trained with different values of hyperparameters gamma (γ) and C.



Nonlinear – Gaussian RBF Kernel

TIP

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 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.

Nonlinear – Computational Complexity

The LinearSVC class is based on the liblinear library, which implements an optimized algorithm for linear SVMs. It does not support the kernel trick, but it scales almost linearly with the number of training instances and the number of features

The SVC class is based on the libsvm library, which implements an algorithm that supports the kernel trick. The training time complexity is usually between O(m2 × n) and O(m3 × n). Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (e.g., hundreds of thousands of instances). This algorithm is perfect for complex but small or medium training sets

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

SVM Regression

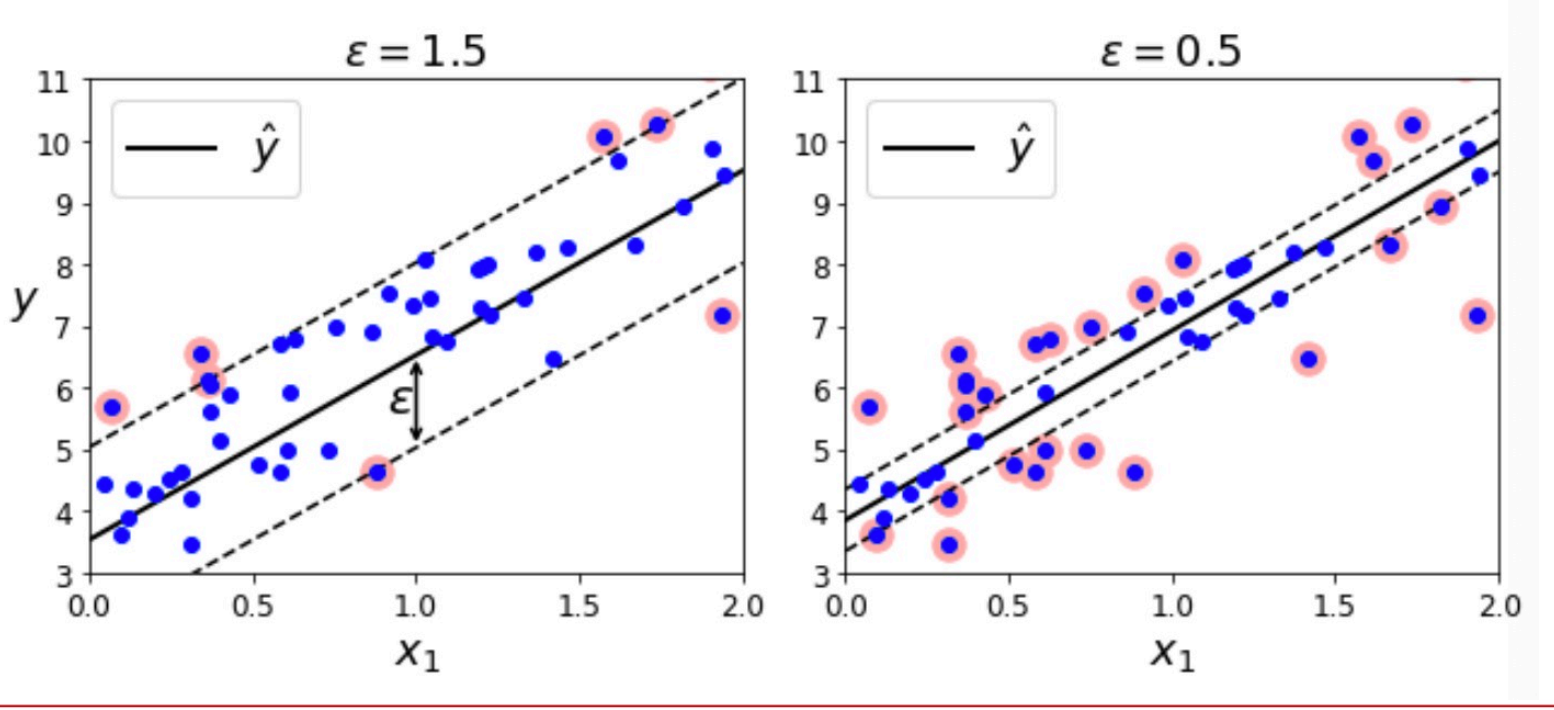
Not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression.

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 of the street).

SVM Regression

The width of the street is controlled by a hyperparameter ϵ. Below shows two linear SVM Regression models trained on some random linear data, one with a large margin (ϵ = 1.5) and the other with a small margin (ϵ = 0.5)

Adding more training instances within the margin does not affect the model’ s predictions; thus, the model is said to be ϵ-insensitive.



SVM Regression

You can use Scikit-Learn’ s LinearSVR class to perform linear SVM Regression

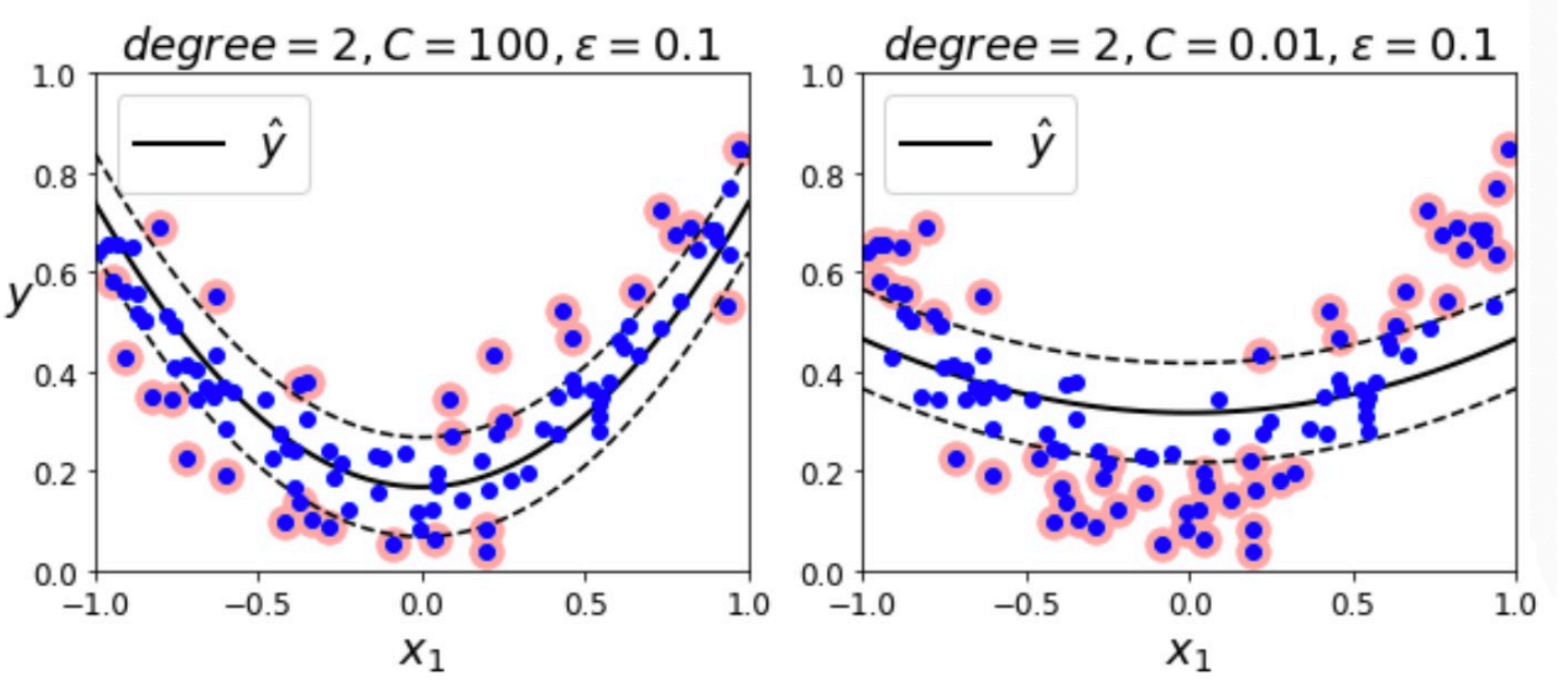
```
In [41]: from sklearn.svm import LinearSVR

         svm_reg = LinearSVR(epsilon=1.5, random_state=42)
         svm_reg.fit(X, y)
```

```
Out[41]: LinearSVR(C=1.0, dual=True, epsilon=1.5, fit_intercept=True,
                   intercept_scaling=1.0, loss='epsilon_insensitive', max_iter=1000,
                   random_state=42, tol=0.0001, verbose=0)
```

SVM Regression

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



SVM Regression

The following code produces the model represented on the left of Figure above 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).
https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR

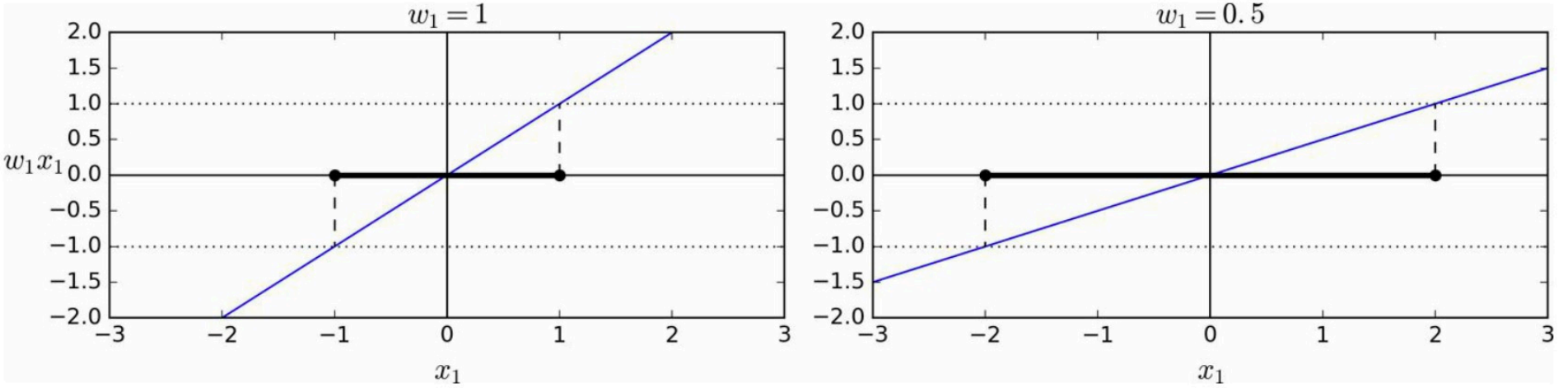
In [44]:
```
np.random.seed(42)
m = 100
X = 2 * np.random.rand(m, 1) - 1
y = (0.2 + 0.1 * X + 0.5 * X**2 + np.random.randn(m, 1)/10).ravel()
```

In [45]:
```
from sklearn.svm import SVR

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

Out[45]:
```
SVR(C=100, cache_size=200, coef0=0.0, degree=2, epsilon=0.1, gamma='auto',
    kernel='poly', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

Q&A – 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. Perhaps this is easier to visualize in 2D in Figure below. The smaller the weight vector w, the larger the margin.
So we want to minimize ‖ w ‖ to get a large margin



Q&A – Training Objective

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

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

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

Q&A – Training Objective

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 ith 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 $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$ 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

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

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

Q&A – Kernelized SVM

https://blog.csdn.net/zhangjun2915/article/details/79261368

SAIF

Shanghai Advanced
Institute of Finance
上海高级金融学院