6 Support Vector Machines

The core idea of Support Vector Machines becomes clear when viewed geometrically. Figure 6.1 shows a slice of the iris dataset containing two linearly separable species. In the left panel two conventional linear classifiers are drawn; although both label every training point correctly, their separating lines sit very close to several observations, so even a small perturbation could lead to misclassifications on new data. The right panel plots the boundary produced by an SVM. The solid line still splits the classes cleanly, but it is positioned to maximise the distance to the nearest points, leaving the widest possible "street" (bounded by the dashed parallels). The samples that lie on these margins are the *support vectors*. This strategy, called large margin classification, usually delivers models that generalise better than those that merely separate the training set.

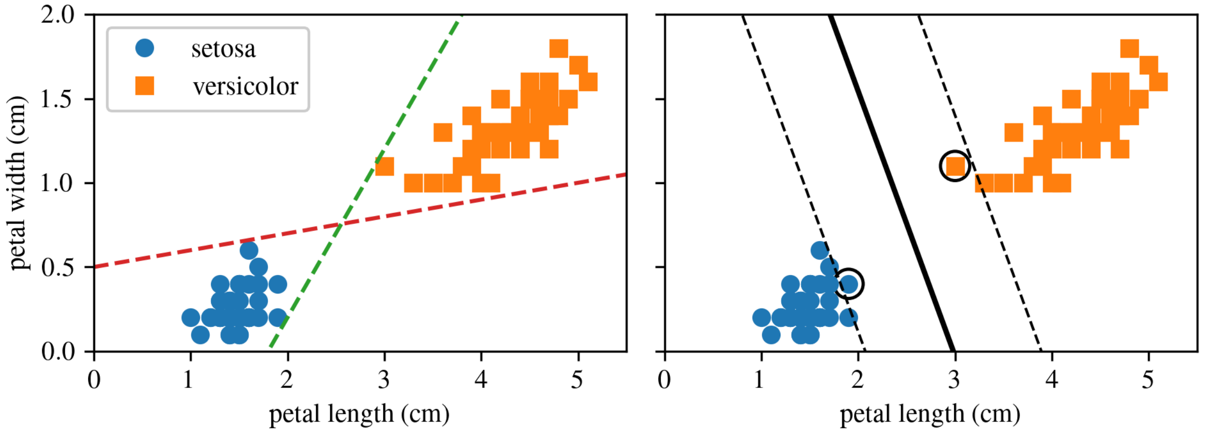

Figure 6.1: Large margin classification.

Observe that the addition of further training instances outside the "street" does not influence the decision boundary; it is entirely shaped by the instances situated on the boundary's edge. These pivotal instances are termed support vectors and are highlighted with circles in Figure 6.1.

NOTE

The sensitivity of SVMs to feature scales is evident in Figure 6.2. In the left plot, the vertical dimension greatly outweighs the horizontal dimension, resulting in a nearly horizontal "street." However, after applying feature scaling such as using Scikit-Learn's `StandardScaler` the decision boundary becomes more appropriate, as illustrated in the right plot.

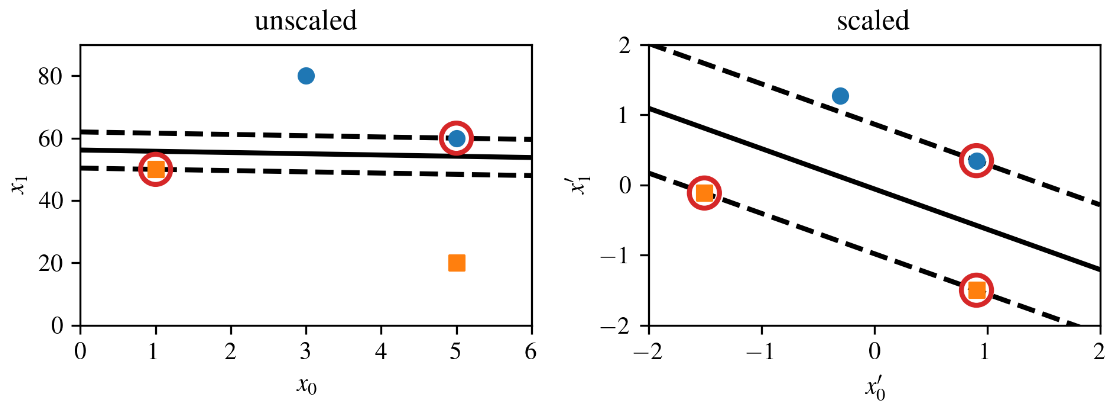

Figure 6.2: Sensitivity of support vectors to feature scaling.

6.1 Linear SVM Classification

Hard margin classification demands that all instances be correctly classified without any margin violations. This strict approach faces two significant challenges:

- It is only feasible when the data is linearly separable.
- It is highly sensitive to outliers.

Figure 6.3 illustrates these challenges using the iris dataset with an added outlier. On the left, achieving a hard margin is impossible due to the outlier. On the right, although a decision boundary is found, it deviates substantially from the optimal boundary shown in Figure 6.1 and is less likely to perform well on new data.

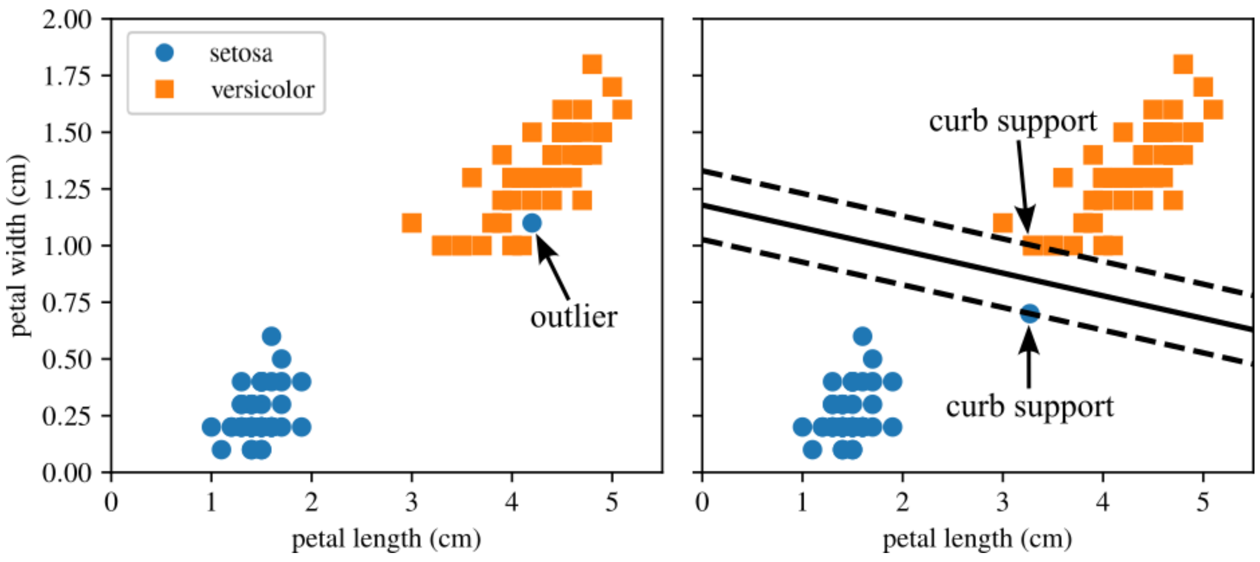

Figure 6.3: Support vector machines showing (left) an un-separable case, and (right) a separable case with two data points supporting the curbs of the support vector machine.

To mitigate the limitations of hard margin classification, a more adaptable model, known as soft margin classification, is often employed. The goal here is to achieve an optimal balance between maximizing the margin width and minimizing margin violations, where instances might fall into the margin or on the incorrect side.

Scikit-Learn's SVM implementations facilitate this balance through the hyperparameter C. A smaller value of C results in a wider margin but allows more margin violations, which is beneficial for model flexibility. Conversely, a larger C value tightens the margin, reducing margin violations but at the risk of a less flexible model. Figure 6.4 demonstrates this trade-off: the left plot with a low C value

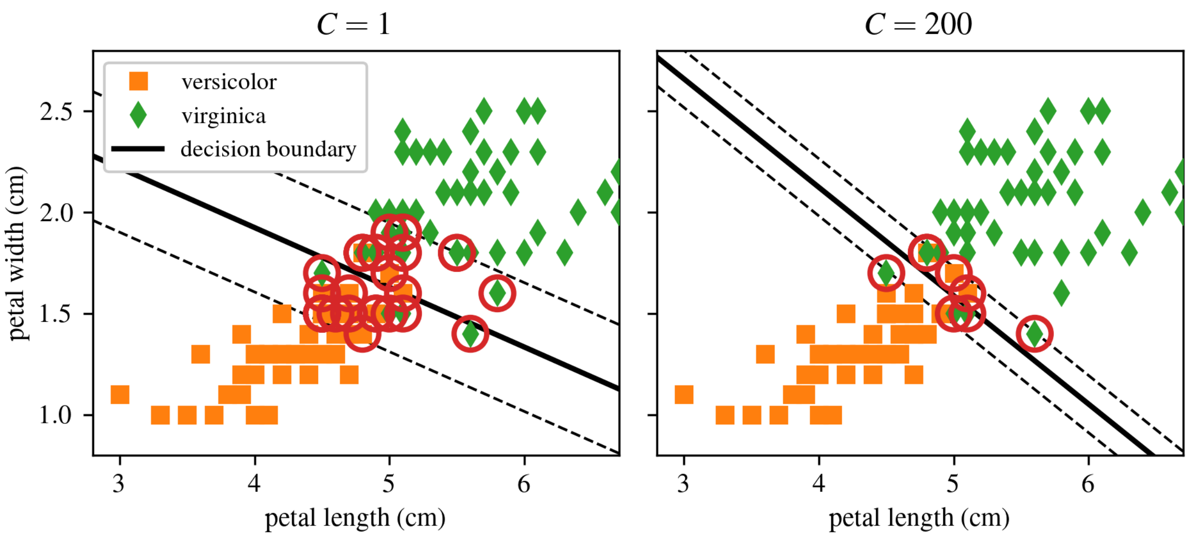

Figure 6.4: SVM margin sizes for different C values.

NOTE

Overfitting in an SVM model can often be addressed by reducing the C value, which increases regularization.

In earlier chapters we placed every model parameter in a single vector θ: the first entry θ_0 acted as the bias, while $\theta_1, \dots, \theta_n$ were the feature weights, and each input was augmented with a constant bias feature $x_0 = 1$. In this chapter we adopt the notation most common for SVMs. The bias is written as b, the weight vector as $\mathbf{w}$, and no extra bias feature is appended to the input vectors.

6.1.1 Decision Function and Predictions

A linear SVM classifier determines the class of a new instance x by calculating the decision function

$$\mathbf{w}^\mathrm{T}\mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b. \tag{6.1}$$

If the outcome is positive, the predicted class $\hat{y}$ is the positive class (1); otherwise, it is the negative class (0). This is written as

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^\mathrm{T}\mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^\mathrm{T}\mathbf{x} + b \geq 0. \end{cases} \tag{6.2}$$

Figure 6.5 plots the decision function for a model with two features, so the surface is a plane in $\mathbb{R}^2$. The thick solid line marks the decision boundary where the function equals zero. The dashed lines show the loci where the function equals 1 and -1; they run parallel to the boundary and sit at equal distance from it, outlining the margin. Training a linear SVM adjusts $\mathbf{w}$ and b to make this margin as wide as possible while either prohibiting margin violations (hard margin) or keeping them small through a penalty term (soft margin).

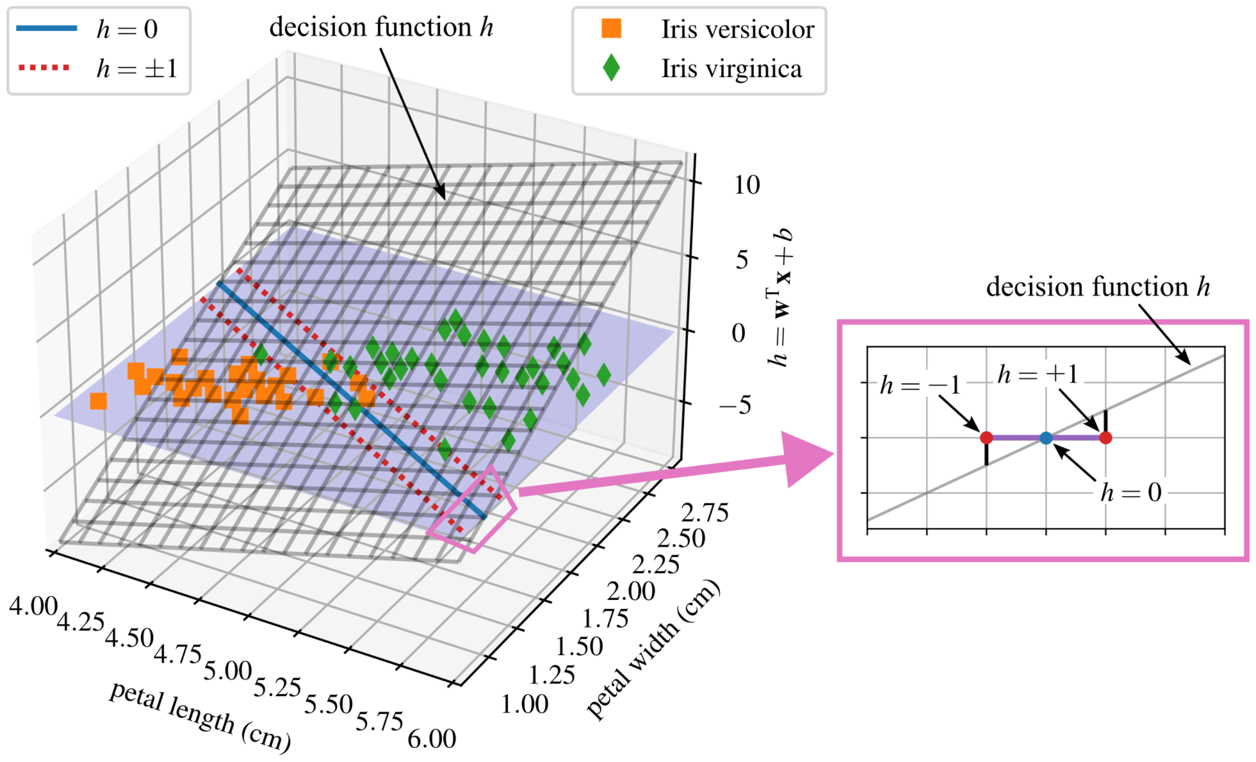

Figure 6.5: Decision function for the Iris Dataset showing how the decision function h cuts through the feature space.

6.1.2 Training Objective

The slope of the decision function corresponds to the norm of the weight vector, $\|\mathbf{w}\|$. Halving this slope causes the decision boundary margins, where the decision function equals ± 1, to double in distance from the decision boundary. Effectively, reducing the norm of $\mathbf{w}$ by half doubles the margin. This geometric interpretation is perhaps simpler to visualize in two dimensions, as shown in Figure 6.6. Thus, minimizing $\|\mathbf{w}\|$ maximizes the margin.

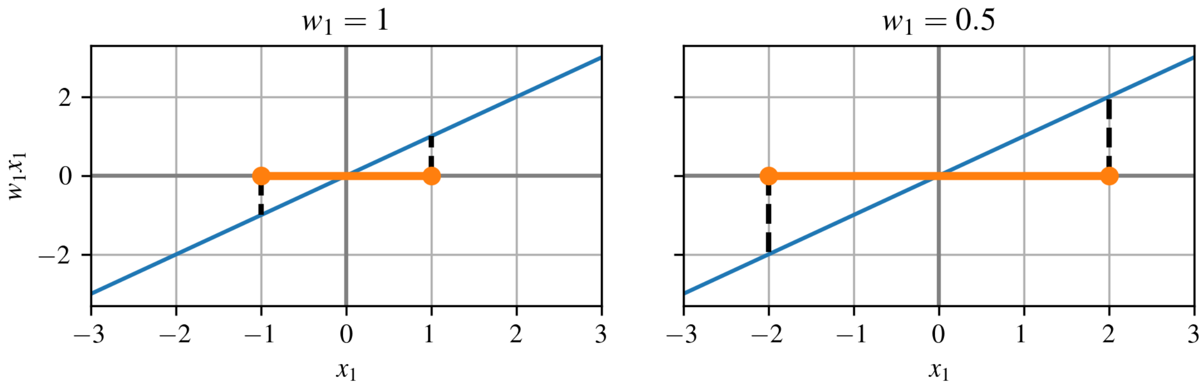

Figure 6.6: The margin is dependent on the value of the weight vector where a smaller weight vector results in a larger margin and vise versa.

To achieve a large margin while enforcing that no data points fall within the margin (hard margin), we ensure the decision function exceeds +1 for all positive training instances and is less than -1 for all negative instances. Let $t^{(i)}$ equal -1 for negative instances ($y^{(i)} = 0$) and +1 for positive ones ($y^{(i)} = 1$). The constraints then require

$$t^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq 1 \tag{6.3}$$

for all training instances. This forms the basis of the hard margin linear SVM classifier optimization problem:

$$\begin{aligned} &\underset{\mathbf{w},b}{\text{minimize}} && \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \\ &\text{subject to} && t^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq 1 \text{ for } i = 1, 2, \ldots, m \end{aligned} \tag{6.4}$$

NOTE

The objective function minimized is $\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$, equivalent to $\frac{1}{2}\|\mathbf{w}\|^2$. This formulation is chosen over minimizing $\|\mathbf{w}\|$ directly because $\frac{1}{2}\|\mathbf{w}\|^2$ offers a straightforward derivative, simply $\mathbf{w}$, facilitating gradient calculations. In contrast, $\|\mathbf{w}\|$ lacks differentiability at $\mathbf{w} = 0$, posing challenges for optimization algorithms, which typically require smooth, differentiable functions to ensure effective optimization.

To formulate the soft margin objective, it is necessary to introduce a slack variable $\zeta^{(i)} \geq 0$ for each instance. This variable, $\zeta^{(i)}$, quantifies the permissible margin violation for the i$^{\text{th}}$ instance. Consequently, we face dual objectives: minimizing the slack variables to reduce margin violations and minimizing $\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$ to maximize the margin. The hyperparameter C plays a crucial role here, enabling a balance between these competing objectives. The introduction of C transforms our task into a constrained optimization problem.

$$\begin{aligned} &\underset{\mathbf{w},b,\zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)} \\ &\text{subject to} \quad t^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \text{ and } \zeta^{(i)} \geq 0 \text{ for } i = 1,\, 2,\, \cdots,\, m \end{aligned} \tag{6.5}$$

6.1.3 Quadratic Programming

Both hard margin and soft margin problems are examples of convex quadratic optimization problems with linear constraints, commonly referred to as Quadratic Programming (QP) problems. QP involves solving optimization problems where the objective is a quadratic function and the constraints are linear. This form of programming, established in the 1940s, predates and is distinct from "computer programming," and is sometimes more descriptively termed "quadratic optimization" to avoid confusion.

A variety of techniques available through off-the-shelf solvers can address these QP problems, though they extend beyond the scope of this text. The general formulation of a QP problem is as follows:

$$\begin{aligned} &\underset{\mathbf{p}}{\text{minimize}} \quad \frac{1}{2}\mathbf{p}^{\mathrm{T}}\mathbf{H}\mathbf{p} + \mathbf{f}^{\mathrm{T}}\mathbf{p} \\ &\text{subject to} \quad \mathbf{A}\mathbf{p} \leq \mathbf{b} \end{aligned} \tag{6.6}$$

Here, $\mathbf{p}$ is an n_p-dimensional vector (where n_p is the number of parameters), $\mathbf{H}$ is an $n_p \times n_p$ matrix, $\mathbf{f}$ is an n_p-dimensional vector, $\mathbf{A}$ is an $n_c \times n_p$ matrix (with n_c being the number of constraints), and $\mathbf{b}$ is an n_c-dimensional vector.

Equation 6.6 defines a standard quadratic program with constraints of the form $\mathbf{A}\mathbf{p} \leq \mathbf{b}$. For training a hard margin linear SVM, this setup can be used by choosing parameters that encode the SVM objective and constraints. The solution vector $\mathbf{p}$ contains both the bias term and the feature weights. Using this knowledge, a standard QP solver can be applied directly to find the optimal SVM classifier.

Example 6.1 Support Vector Machine Classification

This example applies a linear Support Vector Machine (SVM) classifier to distinguish Iris-Virginity flowers based on petal length and width. The decision boundary is derived after scaling the data, and margins are visualized. Misclassified instances within the margin are identified and marked. The model's performance is assessed using a confusion matrix and F1 score.

6.2 Nonlinear SVM Classification

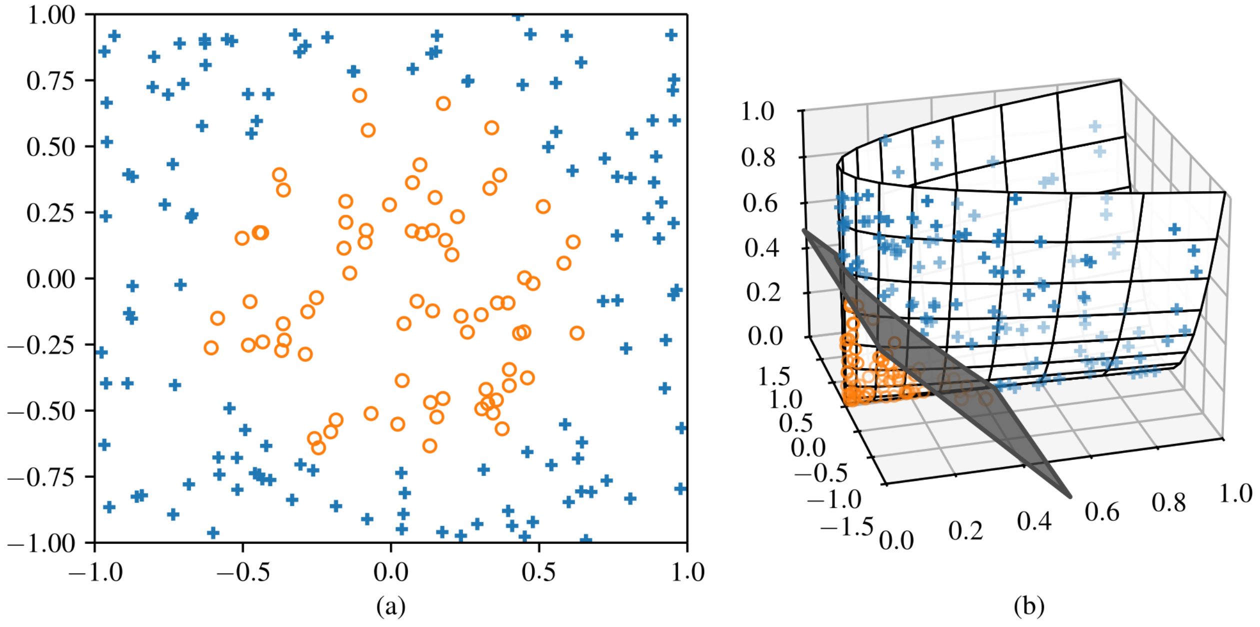

Figure 6.7: Nonlinear SVM example and illustration that shows: (a) not linearly separable 2D Data, and (b) the same data plotted in a transformed feature space such that is is now linearly separable.

While linear SVM classifiers are quite effective and perform exceptionally well in various scenarios, many datasets are far from being linearly separable. One strategy to address non-linear datasets is to introduce additional features, such as polynomial feature. Adding features can sometimes transform the dataset into one that is linearly separable. A representation of this technique is shown in 6.7.

A simple example of converting non-linearly separable variables is shown in figure 6.8 where the left plot displays a simple dataset with a single feature x_1. Clearly, this dataset is not linearly separable. However, by adding another feature $x_2 = (x_1)^2$, the dataset becomes perfectly linearly separable in two dimensions.

[a]Machine Learner, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons

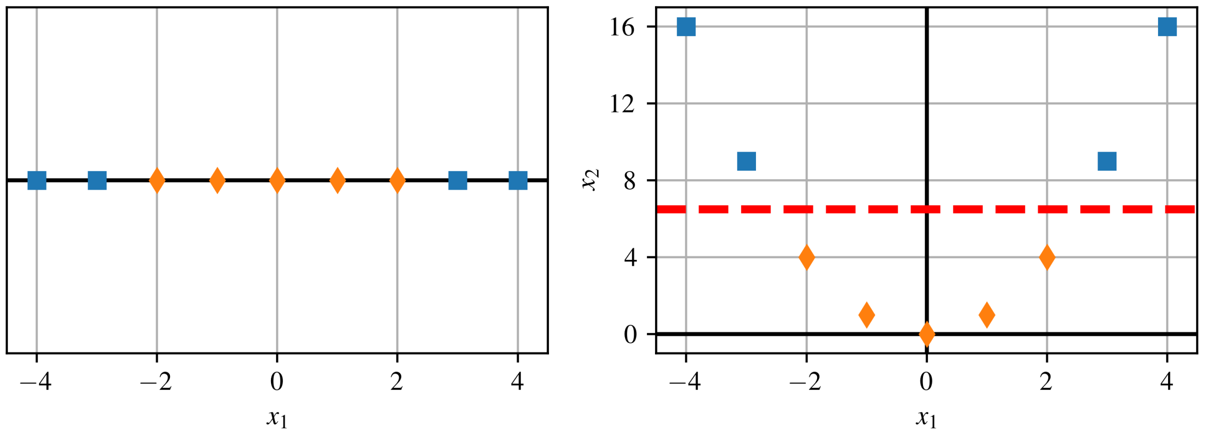

Figure 6.8: Illustration of SVM in higher dimensions

You can implement this idea in Scikit-Learn by creating a pipeline that applies a Polynomial Features transformer (introduced in the Regression Chapter), followed by a `StandardScaler` and a `LinearSVC`. The approach works nicely on the moons dataset, a toy binary-classification problem in which the samples trace two interleaving half-circles, as illustrated in Figure 6.9. You can generate this dataset with the function `make_moons()`.

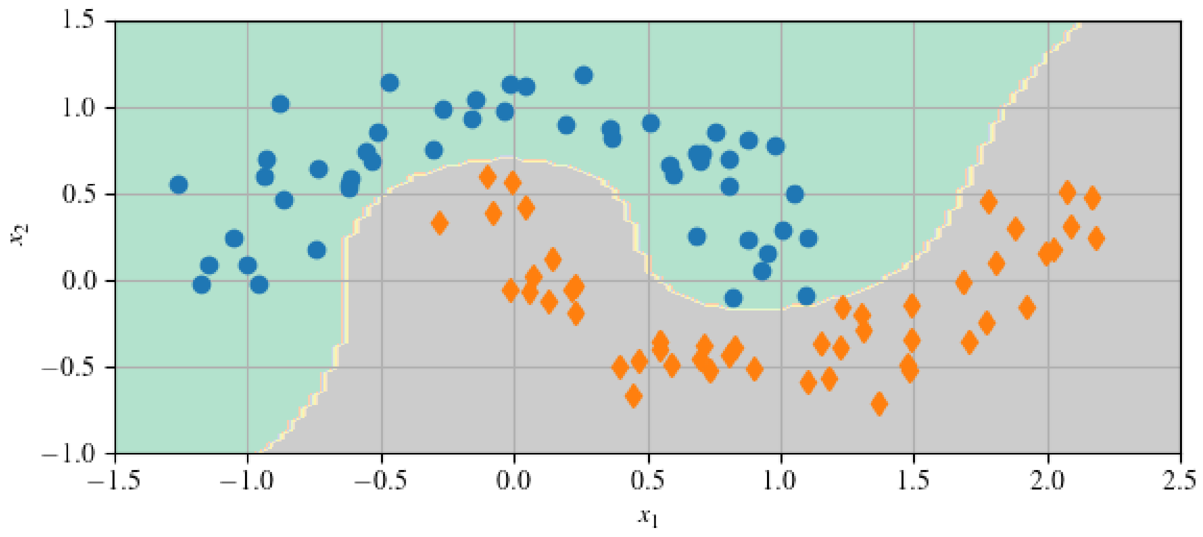

Figure 6.9: Demonstration of a SVM classifier with polynomial features.

Example 6.2 Nonlinear Classification
This example demonstrates nonlinear classification leveraging `LinearSVC` in a pipeline with `preprocessing.PolynomialFeatures` using the `make_moons` dataset. A polynomial fea-

ture transformation is combined with a linear SVM to classify the data, and the resulting decision boundaries are visualized.

6.2.1 Kernel trick

While adding polynomial features is straightforward and enhances the performance of various Machine Learning algorithms (not limited to SVMs), it presents limitations. Specifically, lower-degree polynomials may not adequately handle complex datasets, and higher-degree polynomials significantly increase the feature count, slowing down the model.

SVMs offer a unique solution through a remarkable mathematical technique known as the kernel trick, which allows for the benefits of high-degree polynomial features without actually expanding the feature space, thereby avoiding a rapid increase in computation. This kernel trick is incorporated within the SVC class.

In the dual form of an SVM the explicit dot product of two input vectors

$$\mathbf{x}^{\mathrm{T}}\mathbf{z} \tag{6.7}$$

is replaced by a kernel function

$$k(\mathbf{x},\mathbf{z}), \tag{6.8}$$

where $\mathbf{x}$ and $\mathbf{z}$ are n-dimensional feature vectors representing any two data points in the training set. This substitution lets the algorithm construct a linear separator in a (possibly infinite-dimensional) feature space while all calculations still occur in the original input coordinates.

Figure 6.10 shows configured SVM classifiers using a 3rd and 3th degree polynomial kernel; on the left and right respectively. Adjusting the polynomial degree can help manage model fit: reducing the degree may prevent overfitting, whereas increasing it may be necessary for underfitting scenarios. The ‘coef0‘ hyperparameter is crucial as it determines the influence of high versus low-degree polynomials in the model.

NOTE

A typical method for determining optimal hyperparameter settings involves utilizing grid search techniques. Starting with a broad, coarse grid search to quickly narrow down potential candidates, followed by a more detailed, finer grid search centered on these promising values often yields faster results. Additionally, understanding the function and influence of each hyperparameter aids in efficiently targeting the most relevant areas of the hyperparameter space.

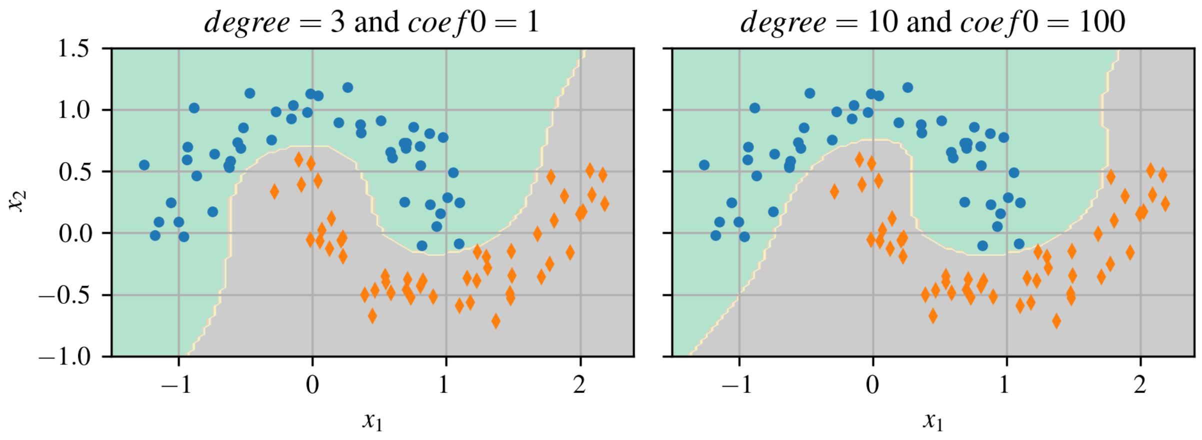

Figure 6.10: SVM polynomial kernel

6.2.2 Standard Kernels

Three kernels cover the vast majority of practical cases.

- The polynomial kernel captures interactions between input features up to a chosen degree d:

$$k_{\text{poly}}(\mathbf{x},\mathbf{z}) = \left(\mathbf{x}^{\mathrm{T}}\mathbf{z}+c\right)^{d} \tag{6.9}$$

  where $c \geq 0$ is a constant offset. It is effective when domain knowledge suggests that a low-order combination of variables explains the target. Keep d modest (typically $d \leq 5$) and standardise inputs to avoid exploding feature dimensions and overfitting.

- The radial basis function (RBF) kernel builds smooth, highly flexible decision boundaries:

$$k_{\text{rbf}}(\mathbf{x},\mathbf{z}) = \exp\left(-\gamma\|\mathbf{x}-\mathbf{z}\|^{2}\right) \tag{6.10}$$

  with width parameter $\gamma > 0$. A large γ makes the surface too flat (underfitting), while a small γ lets every point carve its own pocket (overfitting). Tune C and γ jointly, typically on a logarithmic grid, after x-score standardising the features.

- The sigmoid kernel adds an S-shaped non-linearity to the inner product:

$$k_{\text{sig}}(\mathbf{x},\mathbf{z}) = \tanh\left(\kappa\mathbf{x}^{\mathrm{T}}\mathbf{z}+\theta\right) \tag{6.11}$$

  where κ controls the slope and θ the offset. Although useful for some sparse or text data, this kernel is not always positive-semidefinite, so ensure your software handles the resulting optimisation safely.

Example 6.3 Polynomial Kernel Trick
This example uses the kernel trick to enable an SVM to classify non-linearly separable data using `SVC` (not `LinearSVC`). A third-degree polynomial kernel is applied to the moons dataset

using Scikit-Learn's `Pipeline` and `SVC` tools, producing a curved decision boundary that cleanly separates the classes.

NOTE

Begin with the RBF kernel as a strong default, explore polynomial kernels when you expect specific interaction orders, and treat the sigmoid option as experimental unless you have evidence it helps.

6.3 Computational Complexity

Scikit-Learn provides two main SVM classifiers that trade accuracy for speed in different ways. `LinearSVC` is optimized for large, high-dimensional data when a linear decision boundary is adequate; `SVC` sacrifices runtime for the expressive power of kernels and thus handles complex, nonlinear patterns. Table 1 highlights how these priorities translate into computational costs and practical constraints and compares them to `SGDClassifier` for reference.

Table 1: Key characteristics of three Scikit-Learn SVM classifier implementations.

Trule 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

`LinearSVC` builds on the `liblinear` solver and is limited to *linear* decision boundaries. Because it does not apply the kernel trick, its training cost grows almost linearly with both the number of samples m and features n, i.e. $O(mn)$. Convergence is controlled by the tolerance parameter `tol` (denoted ε in the literature); the default value is usually sufficient, but smaller tolerances can be specified when higher accuracy is critical.

`SVC`, in contrast, relies on `libsvm` and *does* support kernel functions. Its computational burden is markedly heavier, between $O(m^2n)$ and $O(m^3n)$ in practice, so training becomes prohibitive once the dataset reaches the hundreds-of-thousands range. Nevertheless, `SVC` excels on smaller or medium-sized problems that demand nonlinear decision surfaces. Runtime also scales with the average count of non-zero features per instance, meaning sparse high-dimensional inputs remain tractable.

6.4 SVM Regression

Support Vector Regression (SVR) adapts the SVM idea to regression by surrounding the prediction curve with a tube of width ε; points outside this tube incur a penalty and the optimizer keeps the tube as flat as possible while allowing a limited number of violations governed by the regularization constant C. Conceptually this reverses the classification goal: instead of widening a margin to separate two classes, SVR encourages the data to lie inside the margin, making ε the principal knob that controls how loose the fit may be for both linear and kernel-based models.

6.4.1 Linear SVR

The presence of additional training instances within the margin does not influence the predictions of the model, rendering it ε-insensitive. For linear SVM Regression tasks, the LinearSVR class from Scikit-Learn can be utilized. For instance, Figure 6.11 demonstrates two linear SVM Regression models trained on random linear data; one features a wide margin ($\varepsilon = 1.5$), and the other a narrower margin ($\varepsilon = 0.5$).

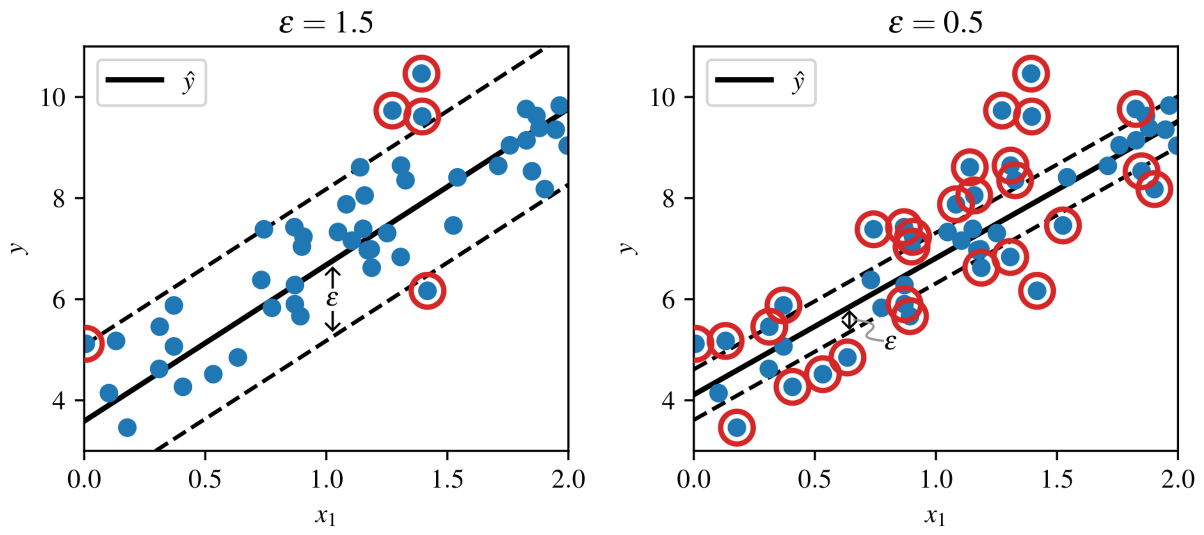

Figure 6.11: SVM Regression models with different ε values.

For data that follow an approximately linear trend, the `LinearSVR` class in Scikit-Learn solves the primal problem directly. It scales in $O(mn)$ time with m samples and n features, making it a practical choice for large data sets where a simple linear fit is adequate.

6.4.2 Kernel SVR

When the relationship is non-linear, the `SVR` class employs the kernel trick:

$$f(x) = \sum_{i=1}^{m} (\alpha_i - \alpha_i^*)\, k(x_i, x) + b, \tag{6.12}$$

where $k(\cdot,\cdot)$ is typically RBF or polynomial. Figure 6.12 shows a 2^{nd}-degree polynomial kernel capturing quadratic structure under various regularisation levels.

The Scikit-lern SVR class, supporting the kernel trick and acting as the regression counterpart to the SVC class, performs well with small to medium-sized datasets but slows considerably as dataset size increases. In contrast, the LinearSVR class, akin to the LinearSVC class, scales linearly with the size of the training set.

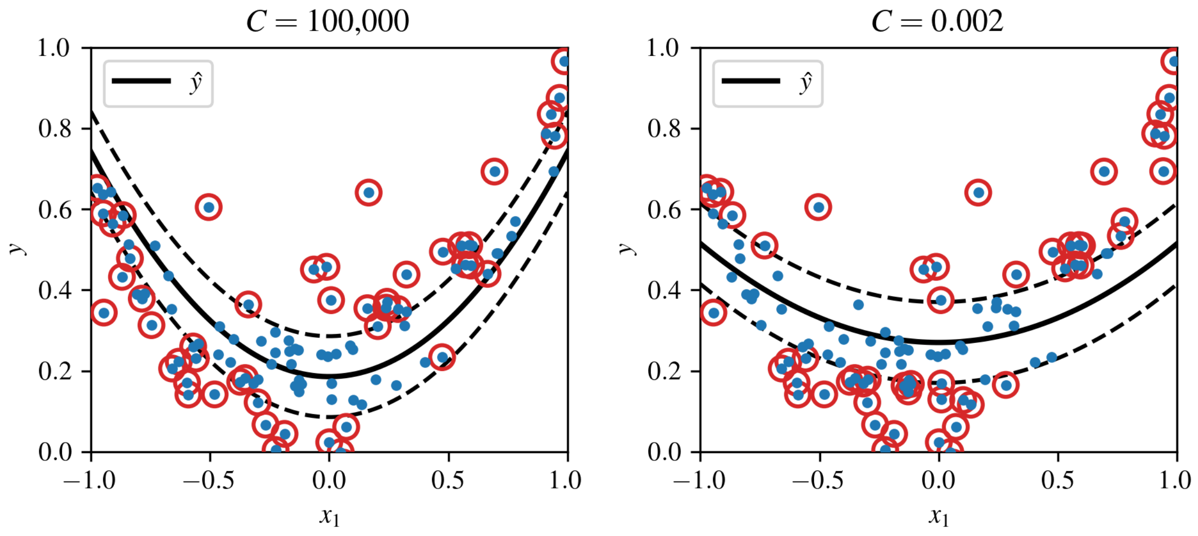

Figure 6.12: SVM regression with a 2nd-degree polynomial kernel, showcasing different regularization levels.

Example 6.4 SVM Polynomial Regression

This example fits a non-linear regression curve to noisy quadratic data using Support Vector Regression (SVR) with a polynomial kernel. It highlights how SVR models can handle non-linear relationships by introducing a margin of tolerance.

6.5 Examples

Example 6.1

```
"""
Example 6.1 Support Vector Machine Classification
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import datasets
from sklearn import svm
from sklearn import pipeline

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


# This code will build and train a support vector machine classifier with soft
# for the iris flower data set.

#%% Load your data

# We will use the Iris data set.  This dataset was created by biologist Ronald
# Fisher in his 1936 paper "The use of multiple measurements in taxonomic
# problems" as an example of linear discriminant analysis
iris = sk.datasets.load_iris()

# for simplicity, extract some of the data sets
X = iris['data'] # this contains the length of the petals and sepals
Y = iris['target'] # contains what type of flower it is
Y_names = iris['target_names'] # contains the name that aligns with the type of the
flower
feature_names = iris['feature_names'] # the names of the features

# plot the Sepal data
plt.figure(figsize=(6.5,3))
plt.subplot(121)
plt.grid(True)
plt.scatter(X[Y==0,0],X[Y==0,1],marker='o',zorder=10)
plt.scatter(X[Y==1,0],X[Y==1,1],marker='s',zorder=10)
plt.scatter(X[Y==2,0],X[Y==2,1],marker='d',zorder=10)
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])

plt.subplot(122)
plt.grid(True)
plt.scatter(X[Y==0,2],X[Y==0,3],marker='o',label=Y_names[0],zorder=10)
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2],zorder=10)
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend(framealpha=1)
plt.tight_layout()

#%% Extract just the petal space of the code

X_petal = X[50:, (2, 3)]  # petal length, petal width
y_petal = Y[50:] == 2

#%% Build and train the SVM classifier

# build handles to regularize the model data and a Linear Support Vector Classification.
scaler = sk.preprocessing.StandardScaler()
svm_clf = sk.svm.LinearSVC(C=1000000,max_iter=10000)

```

```
# build the model pipeline of regularization and a Linear Support Vector Classification.
scaled_svm_clf = sk.pipeline.Pipeline([
        ("scaler", scaler),
        ("linear_svc", svm_clf),
    ])

# train the data
scaled_svm_clf.fit(X_petal, y_petal)


#%% Build and plot the decision boundary along with the curbs

# Convert to unscaled parameters as the SVM is solved in a scaled space.
w = svm_clf.coef_[0] / scaler.scale_
b = svm_clf.decision_function([-scaler.mean_ / scaler.scale_])

# At the decision boundary, w0*x0 + w1*x1 + b = 0
# => x1 = -w0/w1 * x0 - b/w1
x0 = np.linspace(4, 5.9, 200)
decision_boundary = -w[0]/w[1] * x0 - b/w[1]

margin = 1/w[1]
curbs_up = decision_boundary + margin
curbs_down = decision_boundary - margin

#%% Plot the data and the classifier

plt.figure()
plt.grid(True)
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2],zorder=10)
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend(framealpha=1)
plt.tight_layout()

# plot the decision boundy and margins
plt.plot(x0, decision_boundary, "k-", linewidth=2)
plt.plot(x0, curbs_up, "k--", linewidth=2)
plt.plot(x0, curbs_down, "k--", linewidth=2)


#%% Find the misclassified instancances and add a circle to mark them

# Find support vectors (LinearSVC does not do this automatically) and add them
# to the SVM handle
t = y_petal * 2 - 1 # convert 0 and 1 to -1 and 1
support_vectors_idx = (t * (X_petal.dot(w) + b) < 1) # find the locations
# of the miss classifed data points that fall withing the vectors
svs = X_petal[support_vectors_idx]

plt.scatter(svs[:, 0], svs[:, 1], s=180,marker='o', facecolors='none',edgecolors='k')

#%% compute the confusion matirx and F1 score

y_predicted = scaled_svm_clf.predict(X_petal)
confusion_matrix = sk.metrics.confusion_matrix(y_predicted, y_petal)
f1_score = sk.metrics.f1_score(y_predicted, y_petal)

print(f1_score)
```

Example 6.2

```python
"""
Example 6.2 Polynomial Features
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import datasets
from sklearn import pipeline
from sklearn import svm

plt.close('all')

#%% Build and plot the data

# build the data
X, y = sk.datasets.make_moons(n_samples=100, noise=0.25, random_state=2)

plt.figure()
plt.plot(X[:,0][y==0],X[:,1][y==0],'s')
plt.plot(X[:,0][y==1],X[:,1][y==1],'d')
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

#%% SVM polynominal features
svm_clf = sk.pipeline.Pipeline([
        ("poly_features", sk.preprocessing.PolynomialFeatures(degree=3)),
        ("scaler", sk.preprocessing.StandardScaler()),
        ("svm_clf", sk.svm.LinearSVC(C=10))
    ])
svm_clf.fit(X, y)

# make the 2d space for the color
x1 = np.linspace(-2, 3, 200)
x2 = np.linspace(-2, 2, 100)
x1_grid, x2_grid = np.meshgrid(x1, x2)

# calculate the binary decions and predection values
X2 = np.vstack((x1_grid.ravel(), x2_grid.ravel())).T
y_pred = svm_clf.predict(X2).reshape(x1_grid.shape)
y_decision = svm_clf.decision_function(X2).reshape(x1_grid.shape)

con_lines = [-30,-20,-10,-5,-2,-1,0,1,2,5,10,20,30]


# plot the figure
plt.figure()
# provide the solid background color for classification
plt.contourf(x1_grid, x2_grid, y_pred, cmap=plt.cm.brg, alpha=0.2)
# add the contour colors for the threshold
plt.contourf(x1_grid, x2_grid, y_decision, con_lines, cmap=plt.cm.brg, alpha=0.1)
# add the contour lines
contour = plt.contour(x1_grid, x2_grid, y_decision, con_lines, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "s")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "d")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
```

Example 6.3

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 6.3 SVM polynomial kernel

@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import datasets
from sklearn import pipeline
from sklearn import svm

plt.close('all')


#%% Build and plot the data

X, y = sk.datasets.make_moons(n_samples=100, noise=0.25, random_state=2)

plt.figure()
plt.plot(X[:,0][y==0],X[:,1][y==0],'s')
plt.plot(X[:,0][y==1],X[:,1][y==1],'d')
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.grid(True)


#%% SVM polynominal features
svm_clf = sk.pipeline.Pipeline([
        ("scaler", sk.preprocessing.StandardScaler()),
        ("svm_clf", sk.svm.SVC(kernel="poly", degree=3, coef0=1, C=100))
    ])
svm_clf.fit(X, y)


#%% Prepare fro plotting
# make the 2d space for the color
x1 = np.linspace(-2, 3, 200)
x2 = np.linspace(-2, 2, 100)
x1_grid, x2_grid = np.meshgrid(x1, x2)

# calculate the binary decions and prediction values
X2 = np.vstack((x1_grid.ravel(), x2_grid.ravel())).T
y_decision = svm_clf.decision_function(X2).reshape(x1_grid.shape)

# plot the figure (the show up in the figure enviorment defined above
con_lines = [-30,-20,-10,-5,0,5,10,20,30]
contour = plt.contour(x1_grid, x2_grid, y_decision, con_lines, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)


```

Example 6.4

```
"""
Example 6.4 SVM Regression
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import svm


plt.close('all')

#%% build the data sets
np.random.seed(2) # 2 and 6 are pretty good
m = 100
X = 6 * np.random.rand(m,1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m,1)
y = y.ravel()

# plot the data
plt.figure()
plt.grid(True)
plt.plot(X,y,'o')
plt.xlabel('x')
plt.ylabel('y')


#%% SVM regression

svm_reg = sk.svm.SVR(kernel="rbf", degree=3, C=1, epsilon=0.8, gamma="scale")
# Try poly kernal, and different degree, C, and epsilon values
svm_reg.fit(X, y)
x1 = np.linspace(-3, 3, 100).reshape(100, 1)
y_pred = svm_reg.predict(x1)


# plot the SVR model on top of the existing data
plt.plot(x1, y_pred, "-", linewidth=2, label=r"$\hat{y}$")
plt.plot(x1, y_pred + svm_reg.epsilon, "g--",label='curb')
plt.plot(x1, y_pred - svm_reg.epsilon, "g--")
plt.scatter(X[svm_reg.support_], y[svm_reg.support_], s=100,marker='o', facecolor='none',
 edgecolors='gray')
plt.legend(loc="upper left")
```