Northeastern University

Support Vector Machines

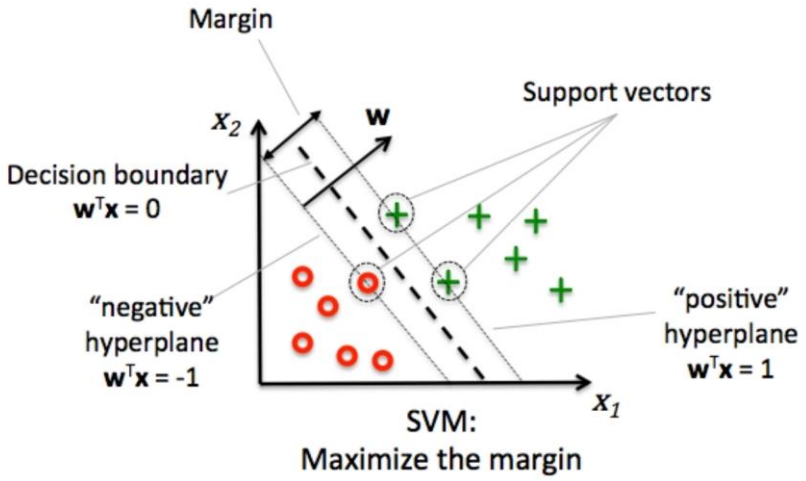
Roi Yehoshua

Agenda

- Support vector machines (SVM)
- Linear SVM
- Soft-margin classification
- Nonlinear SVM
- Kernels
- Support vector regression (SVR)

Support Vector Machines (SVMs)

- SVM is a very powerful and versatile Machine Learning model
- Can perform linear or nonlinear classification, regression, and even outlier detection
- Particularly suited for classification of complex but small- to medium-sized data sets
- Developed at AT&T Bell laboratories by Vladimir Vapnik and colleagues in 1992
- Main idea: find a hyperplane that maximizes the **margin** between the classes



Settings

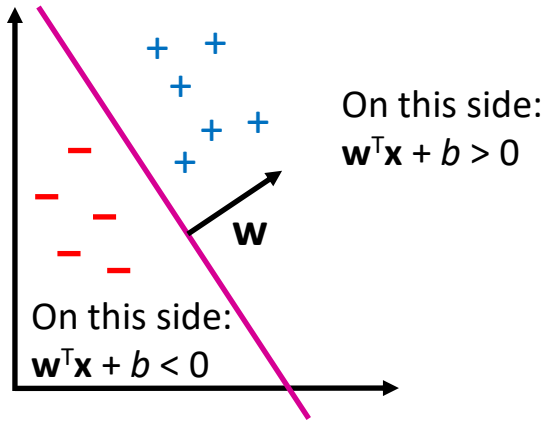
- We consider a binary classification problem with class labels $y \in \{-1, +1\}$
- Our hypothesis is a linear classifier (as in logistic regression):

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$$

  - We use a separate parameter *b* for the bias (instead of w_0)
  - Our classifier directly predicts either +1 or -1 (no probabilities)
- The equation of the separating hyperplane is $\mathbf{w}^T\mathbf{x} + b = 0$
  - The vector **w** is orthogonal to the hyperplane
  - *b* is the distance of the hyperplane from the origin

$$\begin{aligned} \mathbf{w}^T\mathbf{x}_i + b > 0 \quad &\text{if } y_i = 1 \\ \mathbf{w}^T\mathbf{x}_i + b < 0 \quad &\text{if } y_i = -1 \end{aligned} \quad \Rightarrow \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) > 0$$

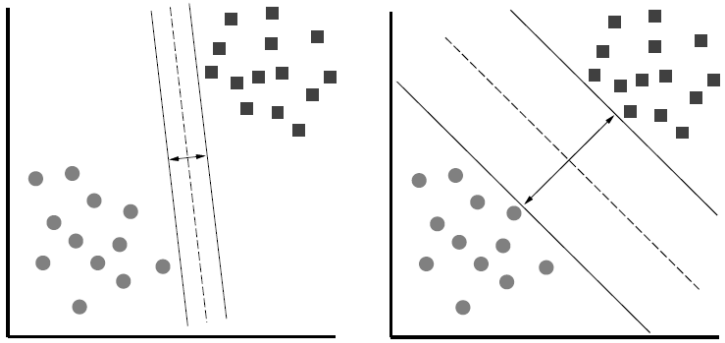
- The data set is **linearly separable** if a hyperplane exists that perfectly separates the classes



Hyperplane perpendicular to **w**
$H = \{\mathbf{x}: \mathbf{w}^T\mathbf{x} + b = 0\}$

Separating Hyperplane

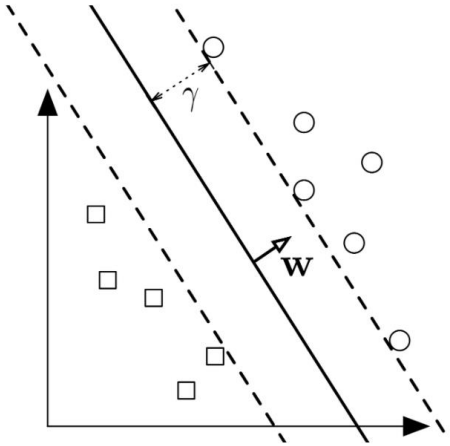
- There are infinitely many separating hyperplanes



- **What is the best separating hyperplane?**
- **SVM Answer:** The one that maximizes the **margin**
  - i.e., maximizes the distance to the closest data points from either class
- Hyperplanes with large margins tend to have better generalization performance
- Hyperplanes with small margins are more sensitive to noise in the training set

Margin

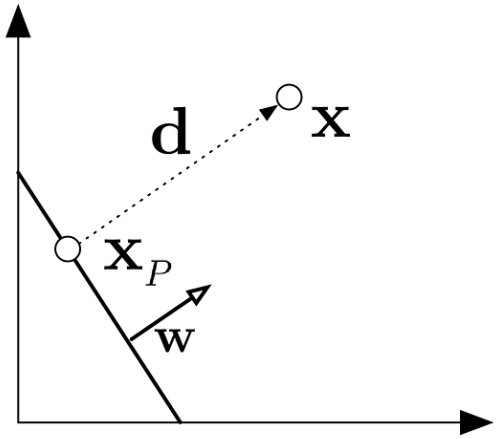
- The **margin** γ is the distance from the separating hyperplane to the closest points from both classes
- A maximum-margin hyperplane must lie exactly in the middle between the classes



- We would like to express γ as a function of our data points

Margin

- What is the distance of a point **x** to the separating hyperplane *H*?
- Let **d** be the vector from *H* to **x** of minimum length
- Let $\mathbf{x}_P$ be the projection of **x** onto *H* $\quad \mathbf{x}_P = \mathbf{x} - \mathbf{d}$
- **d** is parallel to **w**, so **d** = α**w** for some $\alpha \in R$
- $\mathbf{x}_P$ is a vector on *H*, therefore $\quad \mathbf{w}^T\mathbf{x}_P + b = 0$



$$\mathbf{w}^T\mathbf{x}_P + b = \mathbf{w}^T(\mathbf{x} - \mathbf{d}) + b = \mathbf{w}^T(\mathbf{x} - \alpha\mathbf{w}) + b = 0$$

$$\alpha = \frac{\mathbf{w}^T\mathbf{x} + b}{\mathbf{w}^T\mathbf{w}}$$

$$||\mathbf{d}|| = \sqrt{\mathbf{d}^T\mathbf{d}} = \sqrt{\alpha^2\mathbf{w}^T\mathbf{w}} = \frac{|\mathbf{w}^T\mathbf{x} + b|}{\sqrt{\mathbf{w}^T\mathbf{w}}} = \frac{|\mathbf{w}^T\mathbf{x} + b|}{||\mathbf{w}||}$$

- Therefore, the margin of *H* is:

$$\gamma(\mathbf{w}, b) = \min_i \frac{|\mathbf{w}^T\mathbf{x}_i + b|}{||\mathbf{w}||}$$

Maximum Margin Classifier

- We now formulate our search for the maximum margin hyperplane as a constrained optimization problem:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \gamma(\mathbf{w}, b) \\ \text{subject to} \quad & y_i(\mathbf{w}^T\mathbf{x}_i + b) > 0, \quad i = 1, ..., n \end{aligned}$$

All the data points must lie on the correct side of the hyperplane

- If we plug in the definition of γ we obtain:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \frac{1}{||\mathbf{w}||} \min_i |\mathbf{w}^T\mathbf{x}_i + b| \\ \text{subject to} \quad & y_i(\mathbf{w}^T\mathbf{x}_i + b) > 0, \quad i = 1, ..., n \end{aligned}$$

- This optimization problem is non-convex and very hard to solve

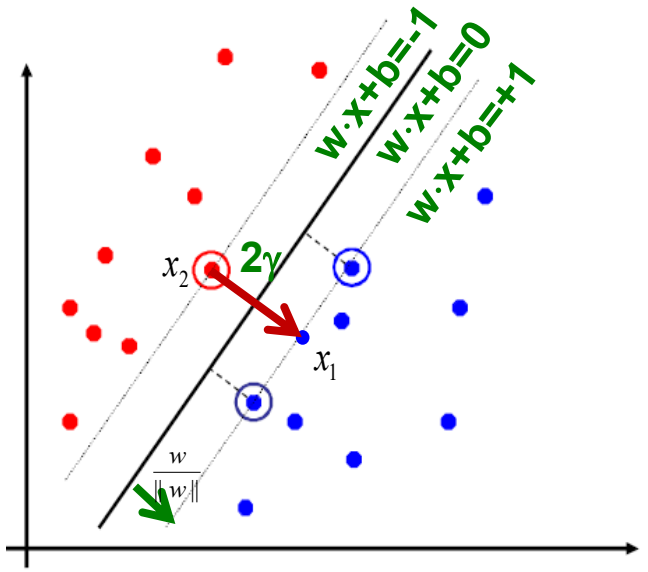
Maximum Margin Classifier

- By definition, the hyperplane and the margin are **scale invariant**:

$$\gamma(\beta\mathbf{w}, \beta b) = \gamma(\mathbf{w}, b), \quad \forall \beta \neq 0$$

- Thus, we can set $y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$ for the points that are closest to the hyperplane
- In this case the margin becomes:

$$\gamma = \min_i \frac{|\mathbf{w}^T\mathbf{x}_i + b|}{||\mathbf{w}||} = \frac{1}{||\mathbf{w}||}$$



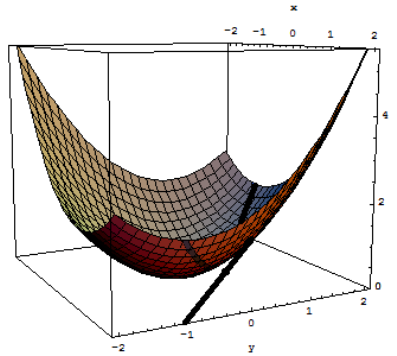
- Thus in order to maximize the margin, we only need to minimize ||**w**||

Maximum Margin Classifier

- The new optimization problem becomes:

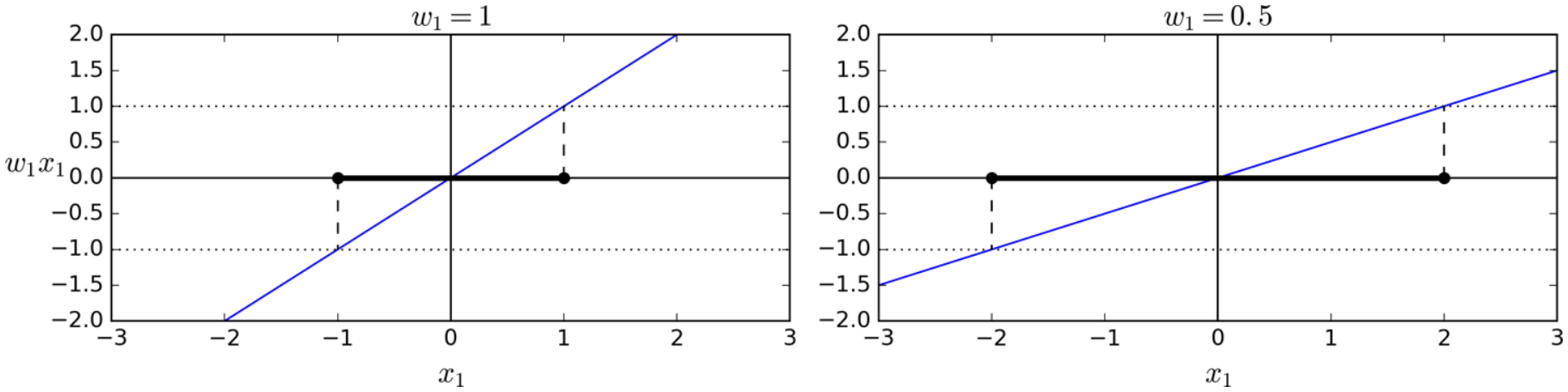
$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \text{subject to} \quad & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad i = 1, ..., n \end{aligned}$$

  - i.e., find the hyperplane with the smallest **w** such that all the points lie at least 1 unit away from it on the correct side
- The new formulation is a quadratic optimization problem with linear constraints
- It has a unique solution whenever a separating hyperplane exists
- We can solve it with **quadratic programming** (QP) software
  - Time complexity is $O(n^3)$, where n is the number of variables



Maximum Margin Classifier

- The smaller the weight vector **w**, the larger the margin:



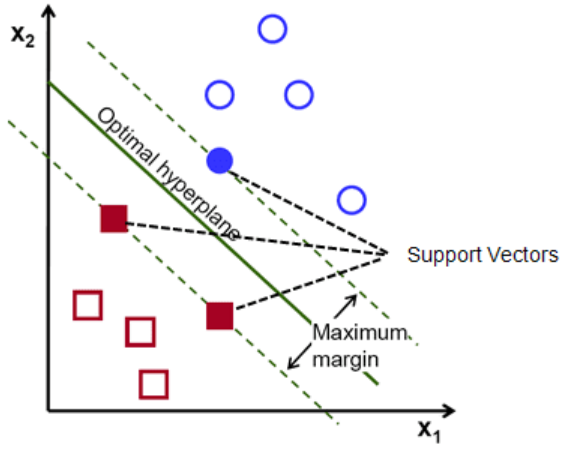
  - The slope of the separating line in this case is $\|\mathbf{w}\| = w_1$
  - If we divide this slope by 2, the points where the decision function is equal to ±1 are going to be twice as far away from the decision boundary

Support Vectors

- For the optimal **w**, *b* pair, some training points will have tight constraints, i.e.

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$$

- We call these points **support vectors**
- Support vectors are important because they define the maximum margin hyperplane
  - If we change one of them, the resulting hyperplane would change



Exercise

- We trained a linear SVM on a binary classification problem and got a weight vector **w** = (1, 2, 3)
- We also know that **x** = (4, 2, 1) is a support vector and is classified by the SVM as -1
- What is the value of *b* in the classification equation of the SVM?

SVM Dual Problem

- The SVM optimization problem has an alternative form called the **dual problem**
  - The dual problem has the same solutions but is often faster to solve than the primal problem
- We first rewrite the objective function using Lagrange multipliers:

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i \left[y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 \right]$$

- We set the partial derivatives of *L* with respect to **w** and *b* to be equal to 0:

$$\frac{\partial L(\mathbf{w}, b, \alpha)}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

$$\frac{\partial L(\mathbf{w}, b, \alpha)}{\partial b} = 0 \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0$$

- Using these conditions we can eliminate **w** and b from $L(\mathbf{w}, b, \alpha)$

SVM Dual Problem

- This gives the **dual representation** of the maximum margin problem:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{subject to} \quad & \alpha_i \geq 0, \qquad i = 1, ..., n \\ & \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned}$$

- Maximizing the dual problem with respect to α_i is equivalent to minimizing the primal problem with respect to **w** and *b*
- We can solve the dual optimization problem using standard QP solvers

SVM Dual Problem

- Once we find an optimal solution for α_i, we can use the following equation to find **w**:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

- It can be shown that for every data point the following must hold:

$$\alpha_i \left[y_i (\mathbf{w}^T \mathbf{x}_i + b) - 1 \right] = 0$$

  - i.e., either $\alpha_i = 0$ or $y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$
- Thus, $\alpha_i > 0$ only when $\mathbf{x}_i$ lies on a margin hyperplane, i.e., when $\mathbf{x}_i$ is a **support vector**
- The parameters of the separating hyperplane depend only on the support vectors!
  - Thus after training, a significant proportion of the data points can be discarded

SVM Dual Problem

- We can determine the value of *b* by noting that any support vector $\mathbf{x}_i$ satisfies

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$$

- Although we can use any support vector to find *b*, to get a more numerically stable solution we average this equation over all the support vectors:

$$b = \frac{1 - y_i\mathbf{w}^T\mathbf{x}_i}{y_i} = \frac{y_i - y_i^2\mathbf{w}^T\mathbf{x}_i}{y_i^2} = y_i - \mathbf{w}^T\mathbf{x}_i$$

Using the fact that $y_i^2 = 1$

$$b = \frac{1}{n_S}\sum_{i \in S}\left(y_i - \mathbf{w}^T\mathbf{x}_i\right)$$

S is the set of support vectors

SVM Dual Problem

- To classify a new data point **x**, we compute:

$$\mathbf{w}^T\mathbf{x} + b = \left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\right)^T \mathbf{x} + b = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b$$

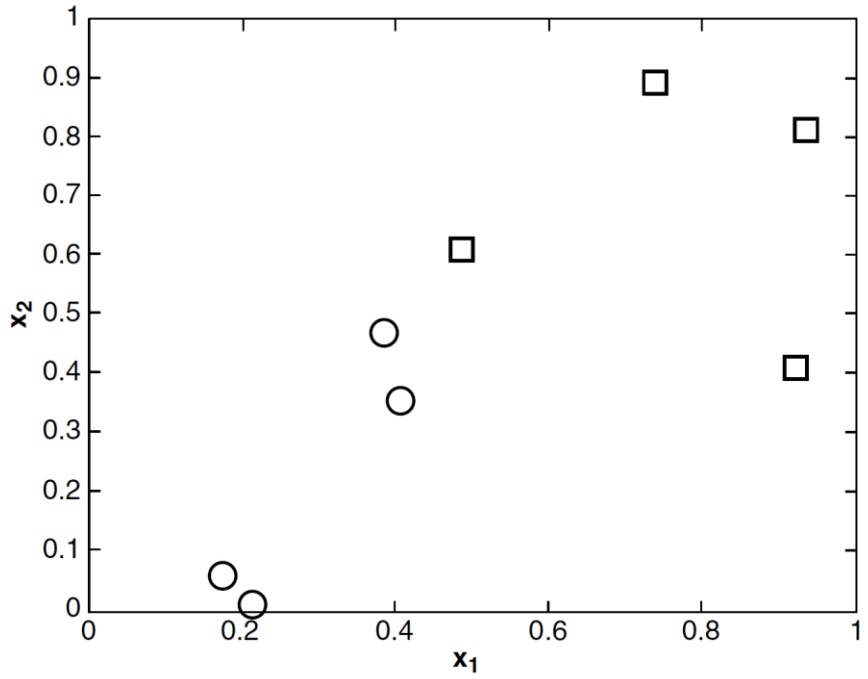
- $\alpha_i \neq 0$ only for the support vectors, thus making predictions involves computing the dot product of the new input vector only with the support vectors

SVM Dual Problem

- Differences between the dual and primal problems:
  - The solution of the dual problem is influenced only by the support vectors
    - Which are easier to manage computationally
  - The dual problem is faster to solve than the primal when the number of training samples *n* is smaller than the number of features *d*
    - The dual problem involves *n* variables while the primal problem involves *d* variables
  - The objective of the dual problem includes only inner products of the features $\mathbf{x}_i^T\mathbf{x}_j$
    - This will be very useful later on when we discuss non-linear SVM

Numerical Example

- Consider the 2D data set shown in the following figure, which contains 8 samples



Numerical Example

- Using quadratic programming, we can find the Lagrange multiplier for each sample:

x_1	x_2	y	Lagrange Multiplier
0.3858	0.4687	1	65.5261
0.4871	0.611	–1	65.5261
0.9218	0.4103	–1	0
0.7382	0.8936	–1	0
0.1763	0.0579	1	0
0.4057	0.3529	1	0
0.9355	0.8132	–1	0
0.2146	0.0099	1	0

These are the support vectors!

Numerical Example

- Let **w** = (w_1, w_2) and b denote the parameters of the decision boundary
- We can solve for w_1 and w_2 as follows:

$$w_1 = \sum_i \alpha_i y_i x_{i1} = 65.5621 \cdot 1 \cdot 0.3858 + 65.5621 \cdot (-1) \cdot 0.4871 = -6.64$$

$$w_2 = \sum_i \alpha_i y_i x_{i2} = 65.5621 \cdot 1 \cdot 0.4687 + 65.5621 \cdot (-1) \cdot 0.611 = -9.32$$

- Next, we compute the bias term for each support vector:

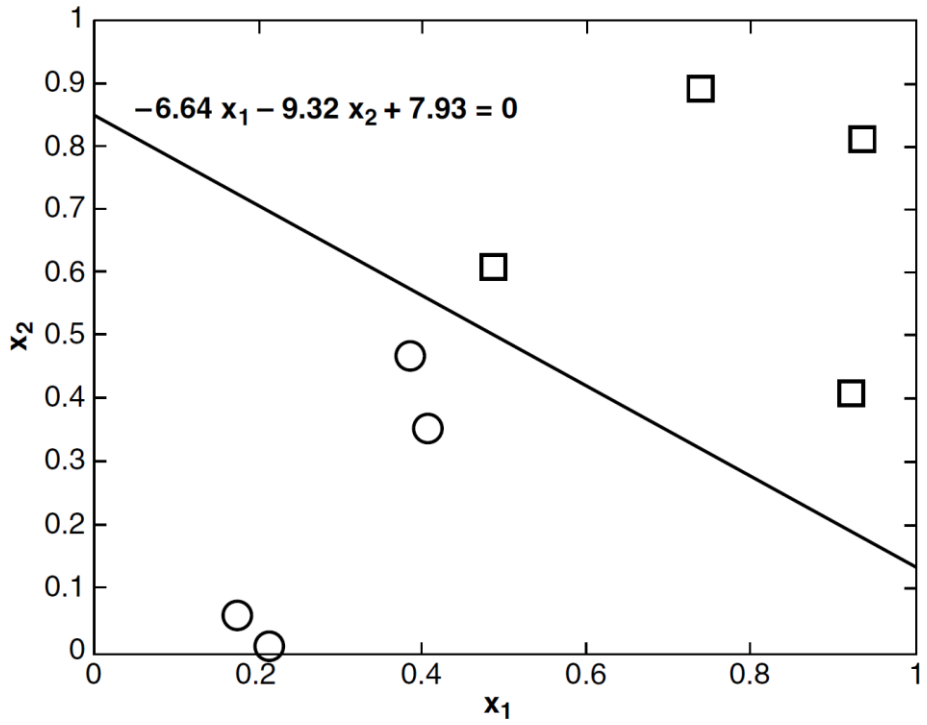
$$b^{(1)} = 1 - \mathbf{w}^T \mathbf{x}_1 = 1 - (-6.64) \cdot 0.3858 - (-9.32) \cdot 0.4687 = 7.93$$

$$b^{(2)} = -1 - \mathbf{w}^T \mathbf{x}_2 = -1 - (-6.64) \cdot 0.4871 - (-9.32) \cdot 0.611 = 7.9289$$

- Averaging these values, we obtain b = 7.93

Numerical Example

- The decision boundary corresponding to these parameters is:



SVM in Scikit-Learn

- Scikit-Learn provides two classes that implement SVM classification:
  - SVC (Support Vector Classifier)
  - LinearSVC
- LinearSVC is faster but can do only linear classification

The LinearSVC Class

class `sklearn.svm.` **`LinearSVC`**(*penalty='l2', loss='squared_hinge', *, dual=True, tol=0.0001, C=1.0, multi_class='ovr', fit_intercept=True, intercept_scaling=1, class_weight=None, verbose=0, random_state=None, max_iter=1000*) [source]

Argument	Description
penalty	Specifies the norm used in the penalization. The ‘l2’ penalty is the standard used in SVC.
loss	Specifies the loss function. ‘hinge’ is the standard SVM loss, while ‘squared_hinge’ is the square of the hinge loss. The combination of penalty='l1' and loss='hinge' is not supported.
dual	Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.
C	Regularization parameter. The strength of the regularization is inversely proportional to C.
multi_class	Determines the multi-class strategy if y contains more than two classes. "ovr" trains n_classes one-vs-rest classifiers, while "crammer_singer" optimizes a joint objective over all classes.
class_weight	Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one.
max_iter	The maximum number of iterations to be run (default is 1000).

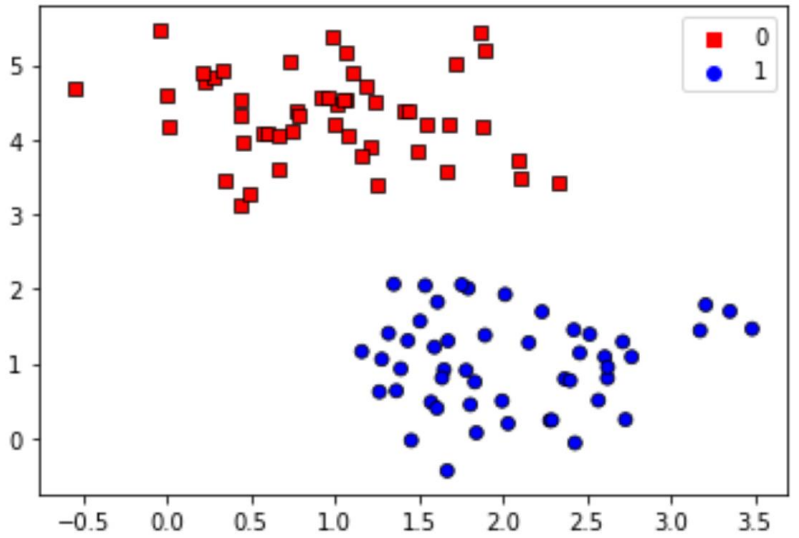
Linear SVM Example

- We start with a simple case of a classification task, in which the two classes of points are well separated:

```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=100, centers=2, cluster_std=0.6,
                  random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
sns.scatterplot(X[:, 0], X[:, 1], hue=y, style=y, markers=('s', 'o'),
                palette=('r', 'b'), edgecolor='black');
```



Linear SVM Example

- We now use the LinearSVC class to train a linear SVM
- To force a hard-margin classification, we set the *C* parameter to a very large number

```
from sklearn.svm import LinearSVC

svc = LinearSVC(C=1E10)
svc.fit(X_train, y_train)
```

```
LinearSVC(C=10000000000.0)
```

```
svc.score(X_train, y_train)
```

```
1.0
```

```
svc.score(X_test, y_test)
```

```
1.0
```

Linear SVM Example

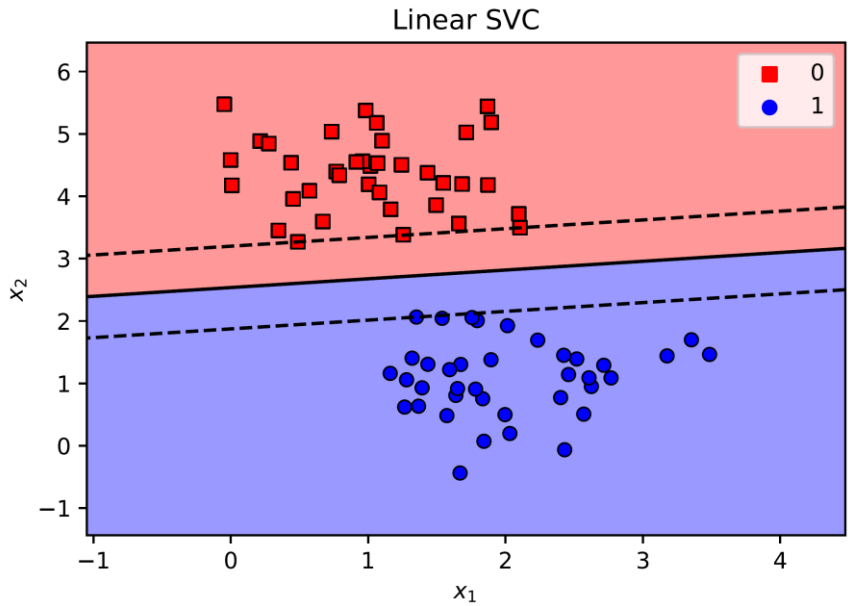
- We can visualize the SVM decision boundary and the margins:

```python
def plot_decision_boundaries(clf, X, y, feature_names, ax, title, h=0.02):
    colors = ['r', 'b']
    cmap = ListedColormap(colors)

    # Fill each side of the decision boundary with the appropriate color
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=0.4, cmap=cmap)

    # Plot the decision boundary and the margins
    xy = np.column_stack([xx.ravel(), yy.ravel()])
    P = clf.decision_function(xy).reshape(xx.shape)
    ax.contour(xx, yy, P, colors='k', levels=[-1, 0, 1],
               linestyles=['--', '-', '--'])

    # Plot also the sample points
    sns.scatterplot(X[:, 0], X[:, 1], hue=y, style=y, ax=ax,
                    palette=colors, markers=('s', 'o'), edgecolor='black')
    ax.set_xlabel(feature_names[0])
    ax.set_ylabel(feature_names[1])
    ax.set_title(title)
    ax.legend()
```



Linear SVM Example

- As usual, after fitting the model, you can use it to make predictions:

```
svc.predict([[1.5, 2]])
```

```
array([1])
```

```
svc.predict([[1.5, 3]])
```

```
array([0])
```

- You can retrieve the coefficients of the separating hyperplane using the coef_ (**w**) and intercept_ (*b*) attributes:

```
svc.coef_
```

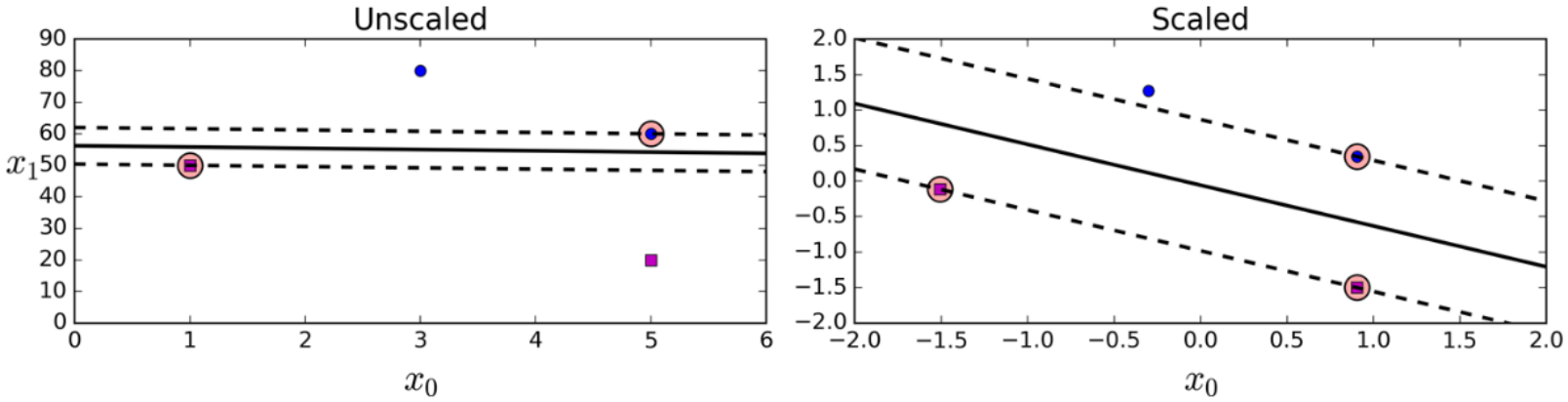
```
array([[ 0.21189504, -1.50897945]])
```

```
svc.intercept_
```

```
array([3.8279331])
```

SVM and Feature Scaling

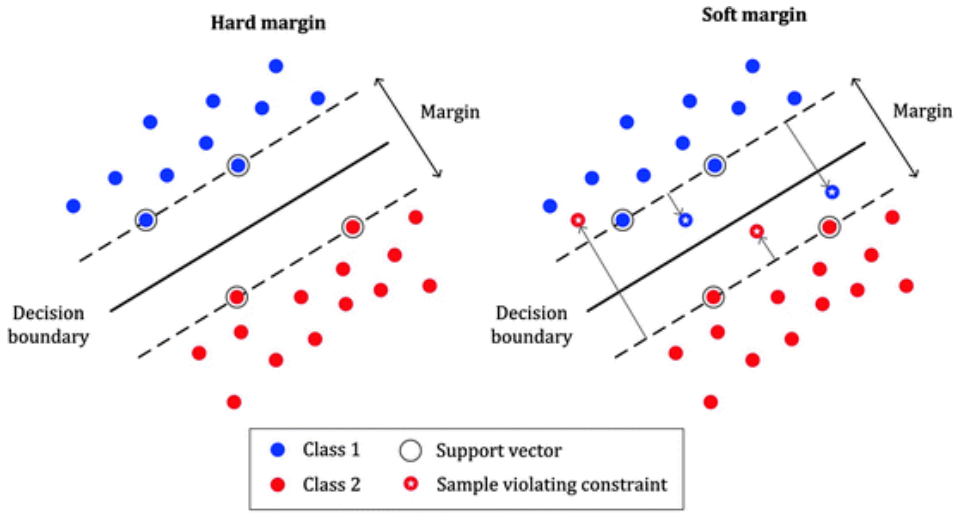
- SVMs are sensitive to the feature scales
- On the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible margin is close to horizontal
- After feature scaling, the decision boundary looks much better (the right plot)



- So **it is highly recommended to scale your data**

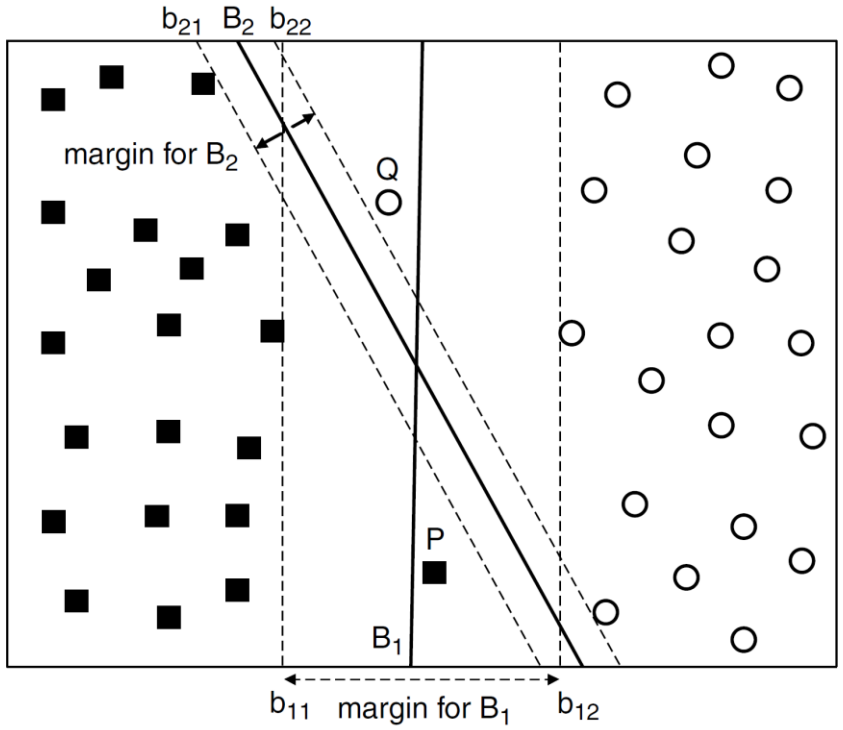
Soft-Margin SVM

- Up until now we assumed that all the data points lie outside the margin
  - This is called **hard-margin SVM**
- Issues in hard-margin SVM:
  - Only works if the data is linearly separable
  - Sensitive to outliers
- In **soft-margin SVM** we allow some of the samples to violate this constraint



Soft-Margin SVM

- For example, in the following case we would prefer a decision boundary (B_1) that misclassifies some of the examples but has a wider margin than B_2

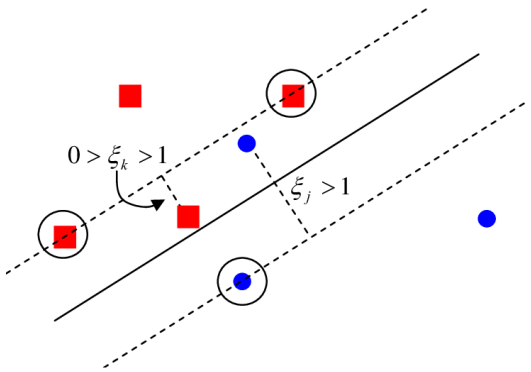


Soft-Margin SVM

- Idea: data points are allowed to be on the 'wrong side' of the margin, but with a cost that increases with the distance from that boundary
- We introduce a **slack variable** $\xi_i \geq 0$ for each training sample $\mathbf{x}_i$ such that:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i$$

  - Points for which $\xi_i = 0$ are correctly classified and are either on the margin or on its correct side
  - Points for which $0 < \xi_i \leq 1$ lie inside the margin, but on the correct side of the decision boundary
  - Points for which $\xi_i > 1$ lie on the wrong side of the decision boundary and are misclassified



Soft-Margin SVM

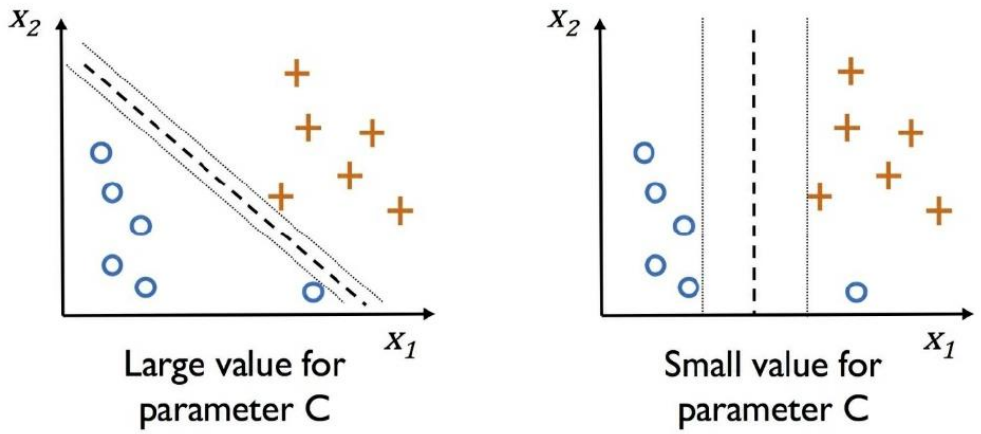
- Our goal is now to **maximize the margin** (ensuring good generalization) while **minimizing the values of the slack variables** (ensuring low training error)
- Therefore, we modify the optimization problem of SVM as follows:

$$\begin{aligned} \min_{\mathbf{w},b,\xi_i} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad i = 1, ..., n \\ & \xi_i \geq 0, \quad i = 1, ..., n \end{aligned}$$

- $C > 0$ is a hyperparameter that controls the tradeoff between maximizing the margin and minimizing the training error

Soft-Margin SVM

- A larger *C* value leads to a narrower margin but less margin violations
  - If C is very large, SVM tries to get all the points to be on the right side of the hyperplane
  - If C is very small, SVM may "sacrifice" some points to obtain a simpler (i.e., lower $\|\mathbf{w}\|$) solution



Soft-Margin SVM

- As before, we can form the dual problem:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_j \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \qquad i = 1, ..., n \\ & \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned}$$

  - for $C = \infty$ we get the same dual problem as in the linearly separable case

- As before, we can use a QP solver to solve the dual soft-margin problem
- And the optimal value of **w** and *b* can be similarly obtained:

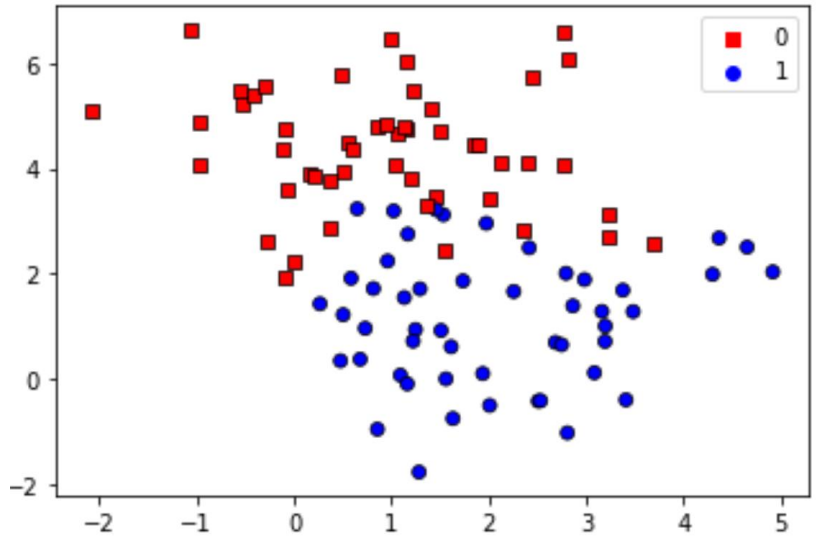
$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \qquad b = \frac{1}{n_S} \sum_{i \in S} \left(y_i - \mathbf{w}^T \mathbf{x}_i \right)$$

Soft-Margin Classification

- Let’s now take a look at the following data, which has some amount of overlap between the classes:

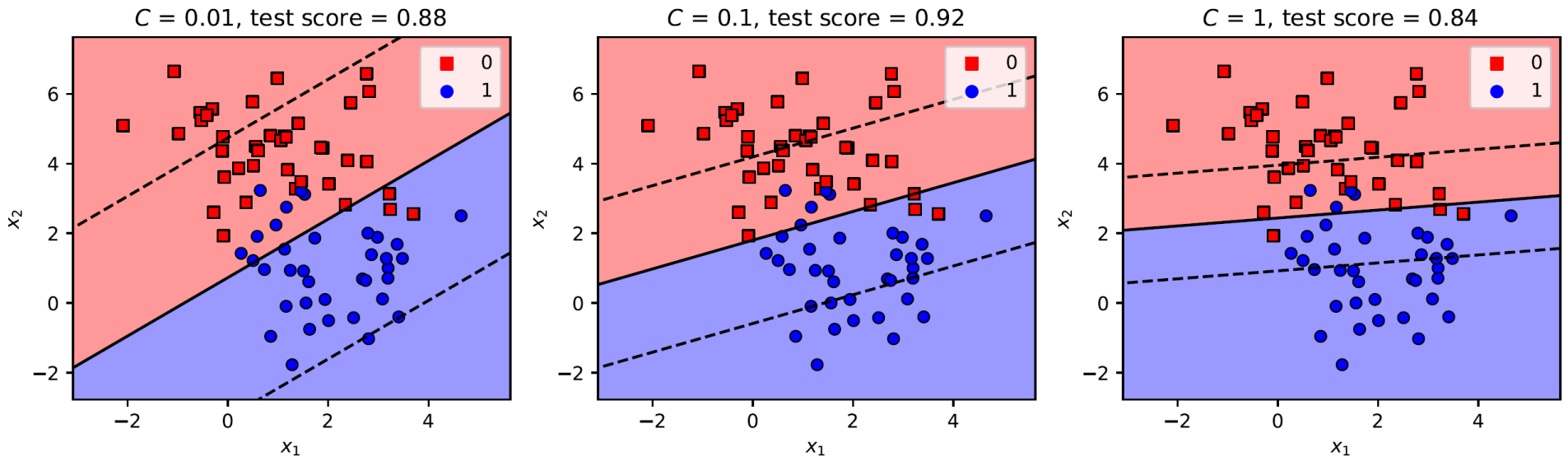
```
X, y = make_blobs(n_samples=100, centers=2, cluster_std=1.2,
                  random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
sns.scatterplot(X[:, 0], X[:, 1], hue=y, style=y, markers=('s', 'o'),
                palette=('r', 'b'), edgecolor='black');
```



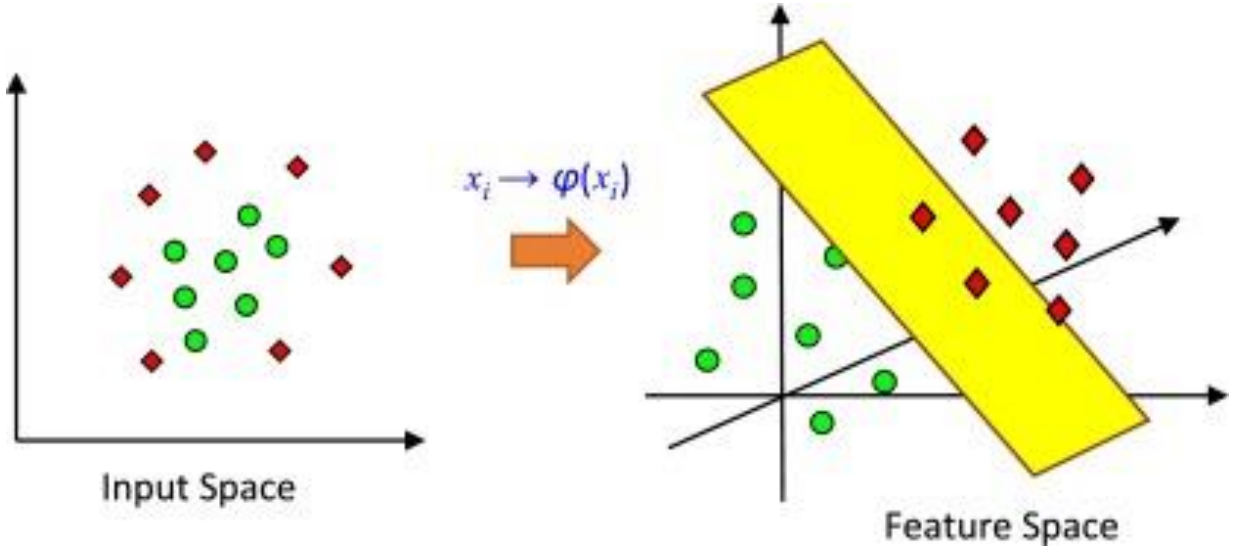
Soft-Margin Classification

- The following plot shows how changing the *C* parameter affects the final fit, via the softening of the margin:



Nonlinear SVM

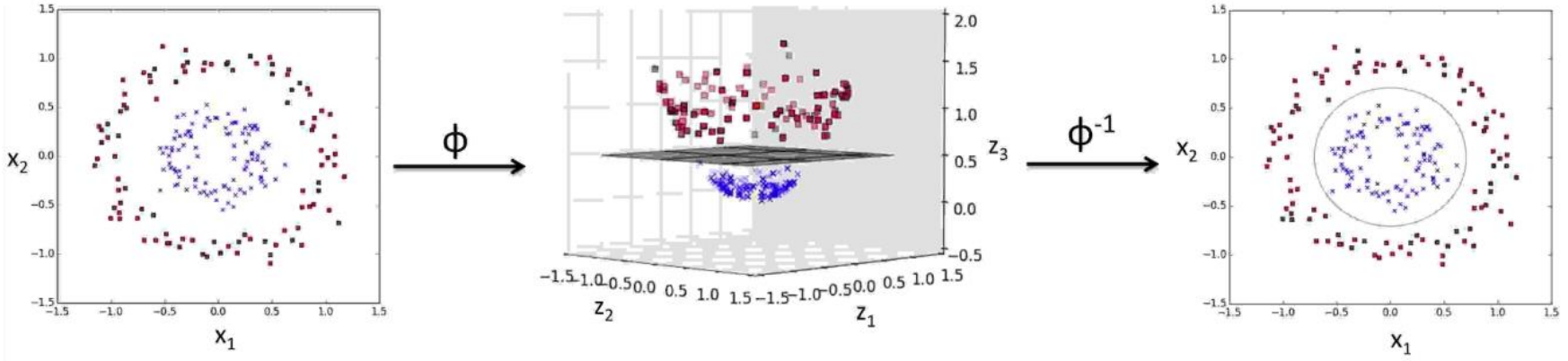
- Idea: transform the data from its original input space **x** into a new space $\phi(\mathbf{x})$ where the data set is linearly separable
- The learned hyperplane can then be projected back to the original input space, resulting in a nonlinear decision boundary



Nonlinear Transformations

- For example, we can transform the following non-linearly separable 2D dataset onto a 3D feature space where the classes become separable via the following projection:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, {x_1}^2 + {x_2}^2)$$



- Adding polynomial features is simple to implement and can work great
- However, at a low polynomial degree it cannot deal with complex datasets, and with a high polynomial degree it creates a huge number of features

The Kernel Trick

- Define a **kernel function** $k(\mathbf{x}, \mathbf{x}')$ between two vectors **x** and **x'**:

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$$

  - A kernel function allows you to compute inner products in the feature space without explicitly computing the feature vectors $\phi(\mathbf{x})$
- The **kernel trick**
  - Formulate your algorithm such that the input vectors **x** are used only inside dot products
  - Use the kernel function to do all the computations in the feature space in $O(d)$ time instead of $O(m)$, where $m >> d$

Constructing Kernels

- In order to exploit the kernel trick, we need to be able to construct valid kernels
- Approach 1: choose a feature space mapping and then use it to find the kernel
- For example, for one-dimensional input space we can write:

$$k(x, x') = \phi(x)^T \phi(x') = \sum_{i=1}^{m} \phi_i(x)\phi_i(x')$$

  - where $\phi_i(x)$ are the basis functions
- Examples for basis functions:
  - Polynomial $\phi_i(x) = x^i$
  - Gaussian $\phi_i(x) = \exp\left[-\frac{(x-\mu_i)^2}{2\sigma^2}\right]$
  - Sigmoid $\phi_i(x) = \sigma\left(\frac{x-\mu_i}{s}\right)$

Constructing Kernels

- Approach 2: construct kernel functions directly such that it corresponds to a dot product in some feature space
- For example, consider the following kernel function: $k(\mathbf{x}, \mathbf{y}) = \left(\mathbf{x}^T \mathbf{y}\right)^2$
- In the case of a 2D input space:

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{y}) &= \left(\mathbf{x}^T \mathbf{y}\right)^2 \\
&= (x_1 y_1 + x_2 y_2)^2 \\
&= x_1^2 y_1^2 + 2 x_1 y_1 x_2 y_2 + x_2^2 y_2^2 \\
&= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)(y_1^2, \sqrt{2} y_1 y_2, y_2^2)^T \\
&= \phi(\mathbf{x})^T \phi(\mathbf{y})
\end{aligned}
$$

- We see that the feature mapping takes the form: $\phi(\mathbf{x}) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2)^T$

Constructing Kernels

- New kernels can be constructed by using simpler kernels as building blocks

Techniques for Constructing New Kernels.

Given valid kernels $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$, the following new kernels will also be valid:

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= ck_1(\mathbf{x}, \mathbf{x}') && (6.13)\\
k(\mathbf{x}, \mathbf{x}') &= f(\mathbf{x})k_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}') && (6.14)\\
k(\mathbf{x}, \mathbf{x}') &= q\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.15)\\
k(\mathbf{x}, \mathbf{x}') &= \exp\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.16)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') && (6.17)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}') && (6.18)\\
k(\mathbf{x}, \mathbf{x}') &= k_3\left(\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}')\right) && (6.19)\\
k(\mathbf{x}, \mathbf{x}') &= \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}' && (6.20)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}'_a) + k_b(\mathbf{x}_b, \mathbf{x}'_b) && (6.21)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}'_a)k_b(\mathbf{x}_b, \mathbf{x}'_b) && (6.22)
\end{aligned}
$$

where $c > 0$ is a constant, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial with nonnegative coefficients, $\boldsymbol{\phi}(\mathbf{x})$ is a function from $\mathbf{x}$ to $\mathbb{R}^M$, $k_3(\cdot, \cdot)$ is a valid kernel in $\mathbb{R}^M$, $\mathbf{A}$ is a symmetric positive semidefinite matrix, $\mathbf{x}_a$ and $\mathbf{x}_b$ are variables (not necessarily disjoint) with $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$, and k_a and k_b are valid kernel functions over their respective spaces.

Common Kernels

- Linear kernel $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$
- Polynomial kernels $k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}' + c)^m$
- Gaussian kernel $k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\sigma^2}\right)$
  - The feature vector that corresponds to this kernel has infinite dimensionality
- Sigmoidal kernel $k(\mathbf{x}, \mathbf{x}') = \tanh\left(a\mathbf{x}^T \mathbf{x}' + b\right)$
  - This function is not a valid kernel, yet it still generally works well in practice

Mercer's Theorem

- Mercer's theorem provides a simply way to test whether a function is a valid kernel without having to construct the function $\phi(\mathbf{x})$ explicitly
- For a set of points $\{\mathbf{x}_i\}$, the **kernel matrix** K is the symmetric $n \times n$ matrix

$$K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$$

  - also known as the Gram matrix
- **Mercer's theorem** states that $k(\mathbf{x}, \mathbf{x}')$ is a valid kernel if only if K is a positive-semidefinite for all possible choices of the set $\{\mathbf{x}_i\}$
  - A matrix A is positive-semidefinite if $\mathbf{w}^T A\mathbf{w} \geq 0$ for all $\mathbf{w}$
- This theorem ensures us that the mapping ϕ exists, even if we don't know what ϕ is

Nonlinear SVM

- Using a suitable function ϕ, we can transform any data sample **x** to ϕ(**x**)
- The hyperplane in the transformed space can be expressed as

$$\mathbf{w}^T\phi(\mathbf{x}) + b = 0$$

- To learn the optimal separating hyperplane, we can substitute in the formulation of SVM to obtain the following optimization problem:

$$\begin{aligned} \min_{\mathbf{w},b,\xi_i} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \quad i = 1, ..., n \\ & \xi_i \geq 0, \quad i = 1, ..., n \end{aligned}$$

Nonlinear SVM

- Using Lagrange multipliers , this can be converted to the dual optimization problem:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \qquad i = 1, ..., n \\ & \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned}$$

- In order to solve this problem we only need to compute inner products of $\phi(\mathbf{x})$
- Hence, even though $\phi(\mathbf{x})$ may be nonlinear and high-dimensional, it suffices to use a function of the inner products of $\phi(\mathbf{x})$

Nonlinear SVM

- The optimal **w** of the hyperplane in the transformed space is:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \phi(\mathbf{x}_i)$$

- Therefore, the equation of the hyperplane is:

$$\mathbf{w}^T \phi(\mathbf{x}) + b = 0$$

$$\sum_{i=1}^{n} \alpha_i y_i \phi(\mathbf{x}_i)^T \phi(\mathbf{x}) + b = 0$$

- where *b* is given by

$$b = \frac{1}{n_S} \sum_{i \in S} \left(y_i - \mathbf{w}^T \phi(\mathbf{x}_i) \right) = \frac{1}{n_S} \left(\sum_{i \in S} y_i - \sum_{i \in S} \sum_{j=1}^{n} \alpha_j y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \right)$$

- Again we only need the inner products of $\phi(\mathbf{x})$ to make predictions for new points

NonLinear SVM

- Example for using a polynomial kernel to find a nonlinear decision boundary:

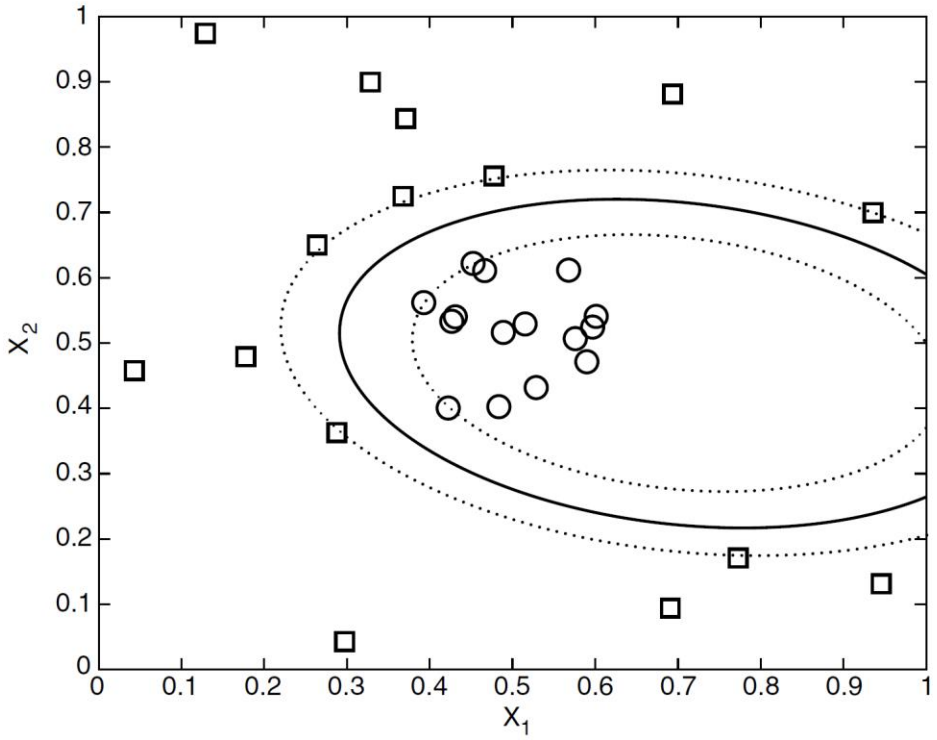


Figure 5.29. Decision boundary produced by a nonlinear SVM with polynomial kernel.

The SVC Class

- Implements support vector classification with the kernel trick

class `sklearn.svm.` **SVC**(**, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=- 1, decision_function_shape='ovr', break_ties=False, random_state=None*) [source]

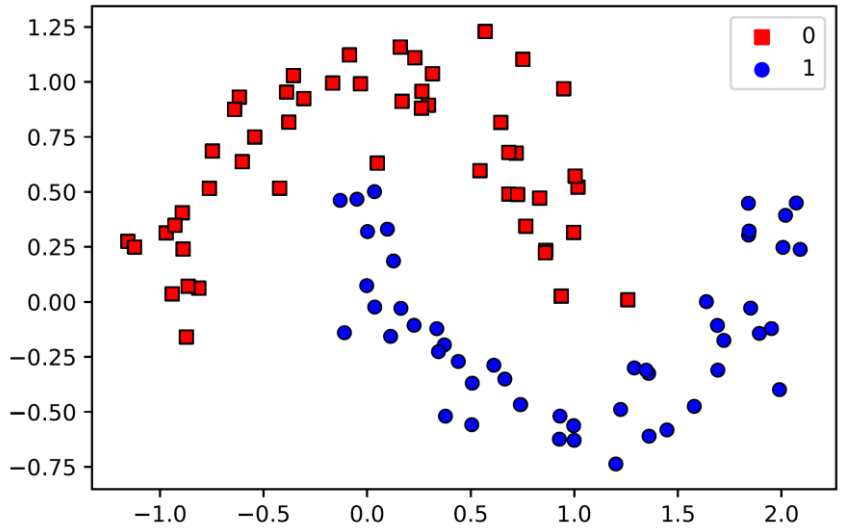
Argument	Description
C	Regularization parameter. The strength of the regularization is inversely proportional to C
kernel	Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable. If none is given, ‘rbf’ will be used.
degree	Degree of the polynomial kernel function (‘poly’). Ignored by all other kernels.
gamma	Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’. If gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma. If ‘auto’, uses 1 / n_features.
coef0	Independent term in kernel function. It is only significant in ‘poly’ and ‘sigmoid’.
probability	Whether to enable probability estimates. Internally uses uses 5-fold cross-validation.
max_iter	Hard limit on iterations within solver, or -1 for no limit.

Kernel SVM Example

- Let's train a SVM with different kernels on the moons dataset:

```
X, y = make_moons(n_samples=100, noise=0.15, random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
sns.scatterplot(X[:, 0], X[:, 1], hue=y, style=y, markers=('s', 'o'),
                palette=('r', 'b'), edgecolor='black')
plt.savefig('figures/svc_moons.pdf')
```



Linear Kernel

- We'll start with a linear kernel:

```
from sklearn.svm import SVC

svc = SVC(kernel='linear')
svc.fit(X_train, y_train)
```

```
SVC(kernel='linear')
```

```
svc.score(X_train, y_train)
```

```
0.8666666666666667
```

```
svc.score(X_test, y_test)
```

```
0.84
```

Linear Kernel

- In contrast to LinearSVC, the SVC class allows you to retrieve the support vectors using the following attributes:

```
# Get support vectors
svc.support_vectors_
```

```
array([[-4.22571849e-01,  5.16716435e-01],
       [-8.72287428e-01, -1.59446492e-01],
       [ 7.18965237e-01,  6.76279391e-01],
       [-9.41153792e-01,  3.62452210e-02],
       [ 8.61804797e-01,  2.34433125e-01],
       [ 8.60776122e-01,  2.23608288e-01],
       [ 1.25898341e+00,  9.45720361e-03],
```

```
# Get number of support vectors for each class
svc.n_support_
```

```
array([14, 14])
```

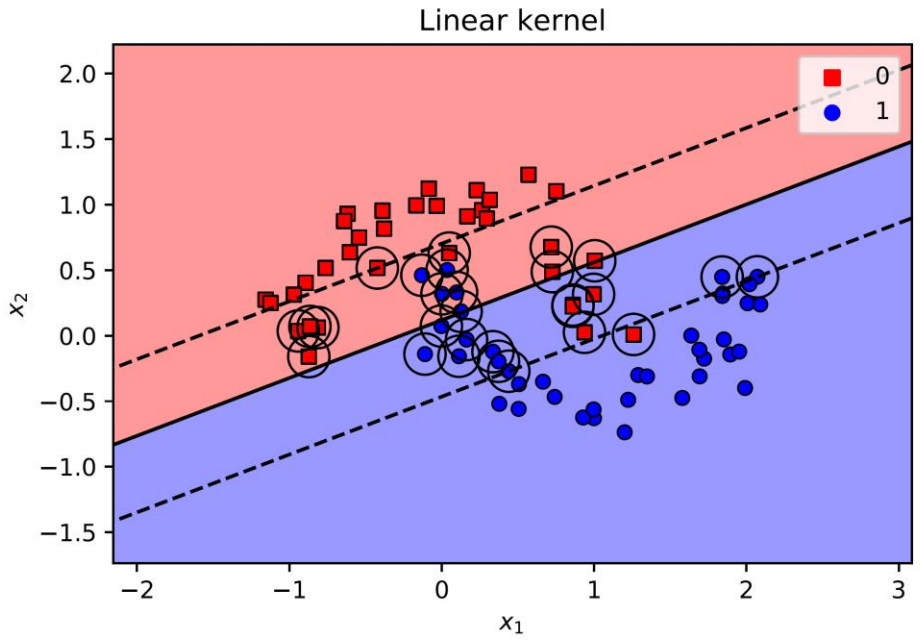
Linear Kernel

- We can visualize the SVM decision boundaries and the support vectors:

```
def plot_support_vectors(clf, ax):
    ax.scatter(clf.support_vectors_[:, 0],
               clf.support_vectors_[:, 1],
               s=300, edgecolors='black', color='none')
```

```
plot_decision_boundaries(svc, X_train, y_train, feature_names,
                         plt.gca(), title='Linear kernel')
plot_support_vectors(svc, plt.gca())

plt.savefig('figures/svc_linear_kernel.pdf')
```

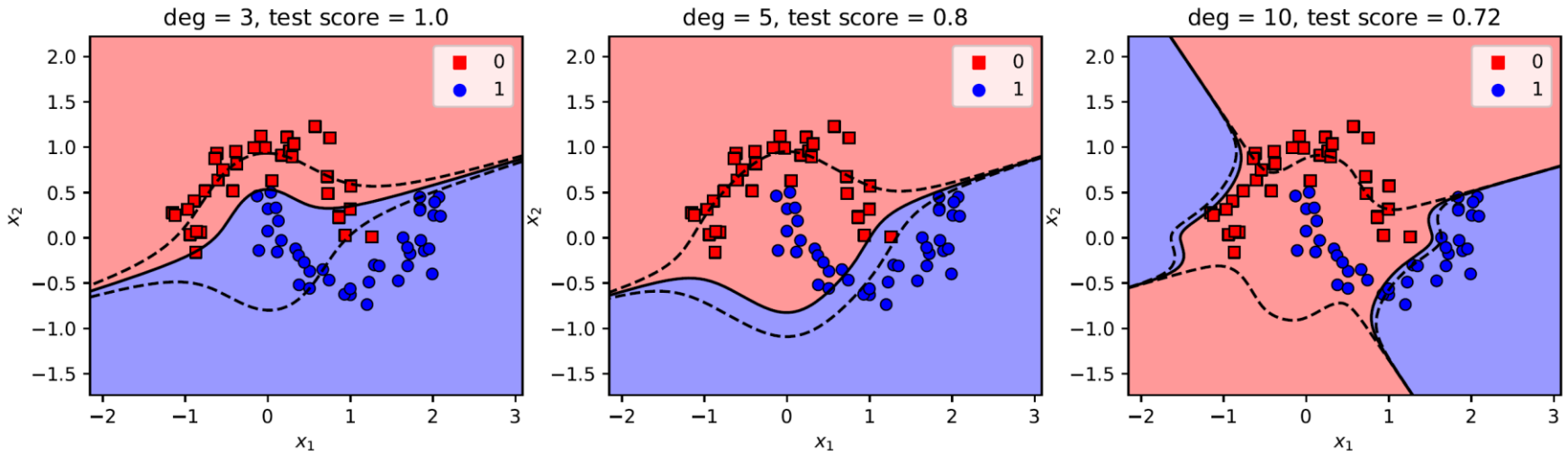


Polynomial Kernel

- We now train 3 SVM classifiers with polynomial kernels of various degrees:

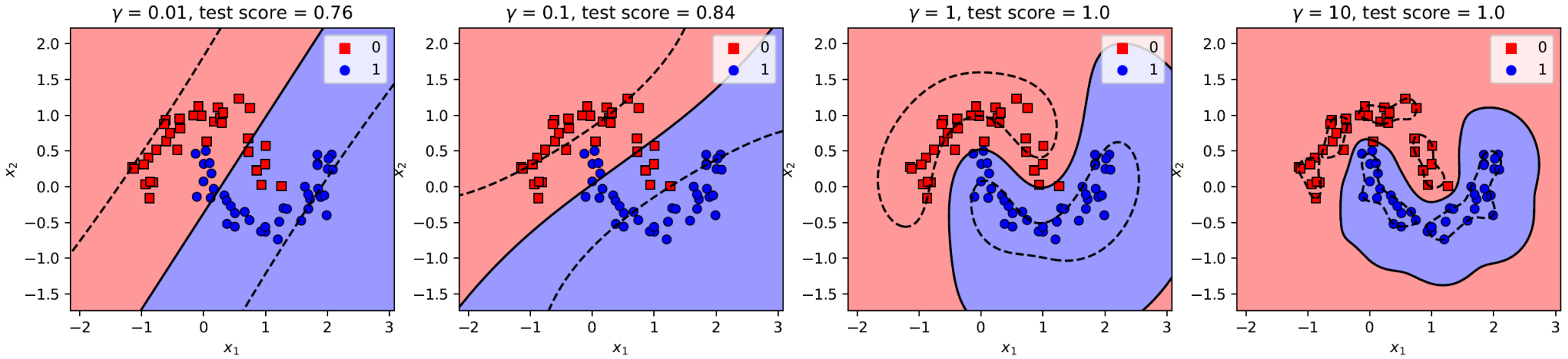
```
fig, axes = plt.subplots(1, 3, figsize=(14, 3.5))

for ax, deg in zip(axes, [3, 5, 10]):
    svc = SVC(kernel='poly', degree=deg).fit(X_train, y_train)
    test_score = svc.score(X_test, y_test)
    plot_decision_boundaries(svc, X_train, y_train, feature_names, ax,
                             title=f'deg = {deg}, test score = {test_score}')
```



Gaussian RBF Kernel

- Now let's use RBF kernels with various γ values:



- Increasing γ makes the bell-shape curve narrower, and as a result the decision boundary ends up being more irregular, wiggling around individual instances
- γ acts like a regularization parameter: if the model is overfitting, you should reduce it
- Proper choice of C and γ is critical to the SVM's performance
  - You should use GridSearchCV with C and γ spaced exponentially far apart to find good values

Other Kernels

- The available built-in kernels in the SVC class are: ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ (the default is ‘rbf’)
- You can also provide your own custom kernel
- As a rule of thumb, you should always try the linear kernel first, 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
- Then if you have spare time and computing power, you can also experiment with a few other kernels using cross-validation and grid search

Scores and Probabilities

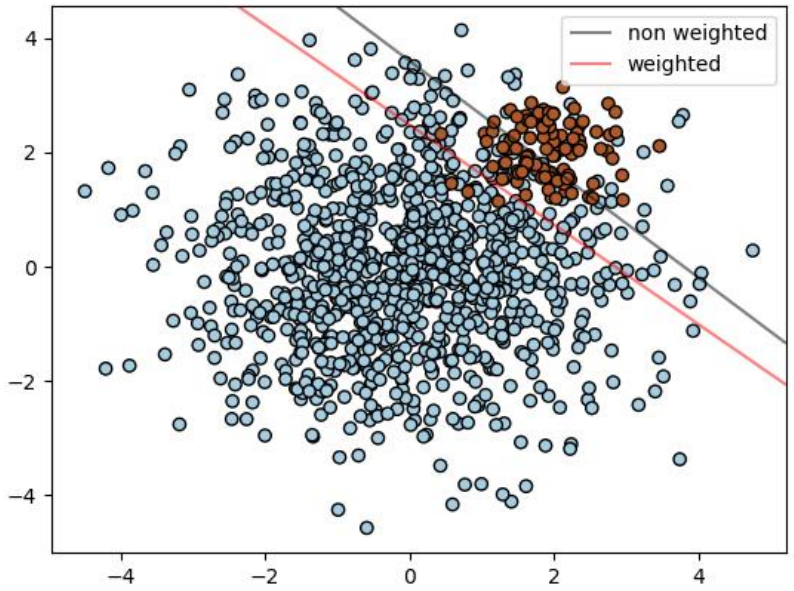
- SVM doesn't provide probabilistic outputs but only decision scores $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$
  - Accessible via the decision_function() method
- To address this issue, Platt (2000) proposed to fit a logistic regression to the outputs of the trained SVM:

$$P(y = 1|\mathbf{x}) = \sigma(ah(\mathbf{x}) + b)$$

- The parameters *a* and *b* are found by using cross-validation on the training data
- The probability estimates may be inconsistent with the SVM scores
  - The argmax of the scores may not be the argmax of the probabilities
- To generate probability estimates set the constructor argument *probability* to True
  - This will make the method predict_proba() available

Unbalanced Problems

- In problems where it is desired to give more importance to certain classes or certain individual samples, the parameters class_weight and sample_weight can be used



Computational Complexity

- The SVC class is based on the *libsvm* library, which uses a QP solver that scales between $O(n^2d)$ and $O(n^3d)$
  - n is the number of samples and d is the number of features
- Thus, it may be impractical beyond tens of thousands of samples
- The LinearSVC class is based on the *liblinear* library, whose implementation is much more efficient and scales linearly with the number of samples and features $O(nd)$
  - However, it doesn't support the kernel trick

Support Vector Regression (SVR)

- SVM can be extended to solve regression problems
- For regression we reverse the objective: we try to fit as many samples as possible *inside the margin* while limiting margin violations (i.e., samples off the margin)
- In simple linear regression, we minimized a regularized error function

$$\sum_{i=1}^{n}(h(\mathbf{x}_i) - y_i)^2 + \lambda||\mathbf{w}||^2$$

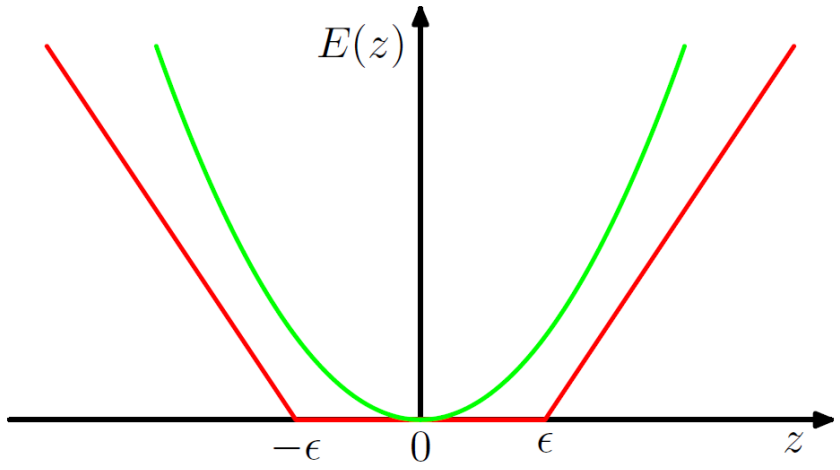
- To get sparse solutions, the quadratic error function is replaced by an ***ε*-insensitive error function**:

$$E_\epsilon(h(\mathbf{x}) - y) = \begin{cases} 0 & \text{if } |h(\mathbf{x}) - y| < \epsilon \\ |h(\mathbf{x}) - y| - \epsilon & \text{otherwise} \end{cases}$$

$$E_\epsilon(h(\mathbf{x}) - y) = \max(0, |h(\mathbf{x}) - y| - \epsilon)$$

Support Vector Regression (SVR)

Plot of an ϵ-insensitive error function (in red) in which the error increases linearly with distance beyond the insensitive region. Also shown for comparison is the quadratic error function (in green).



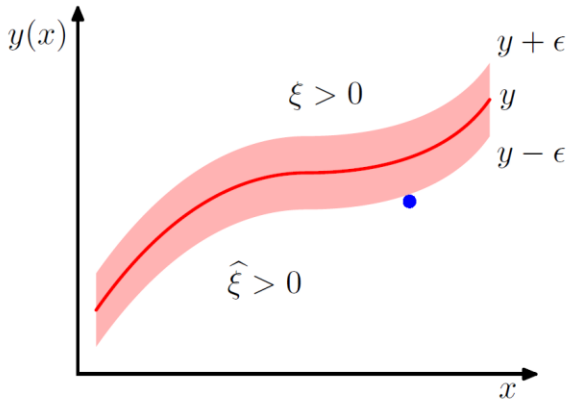
Support Vector Regression (SVR)

- Therefore, the primal problem is formulated as:

$$\min_{\mathbf{w},b} \frac{||\mathbf{w}||^2}{2} + C\sum_{i=1}^{n} \max(0, |\mathbf{w}^T\mathbf{x}_i + b - y| - \epsilon)$$

- As before, we can re-express the optimization problem by introducing slack variables
- For each data point $\mathbf{x}_i$, we define two slack variables:
  - $\xi_i > 0$ corresponds to a point for which $y_i > h(\mathbf{x}_i) + \varepsilon$
  - $\hat{\xi}_i > 0$ corresponds to a point for which $y_i < h(\mathbf{x}_i) + \varepsilon$

Illustration of SVM regression, showing the regression curve together with the ϵ-insensitive 'tube'. Also shown are examples of the slack variables ξ and $\hat{\xi}$. Points above the ϵ-tube have $\xi > 0$ and $\hat{\xi} = 0$, points below the ϵ-tube have $\xi = 0$ and $\hat{\xi} > 0$, and points inside the ϵ-tube have $\xi = \hat{\xi} = 0$.



Support Vector Regression (SVR)

- Thus, we can re-write the optimization problem as:

$$\begin{aligned} \min_{\mathbf{w},b,\xi_i,\hat{\xi}_i} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}(\xi_i + \hat{\xi}_i) \\ \text{subject to} \quad & y_i - \mathbf{w}^T\mathbf{x}_i - b \leq \epsilon + \xi_i, \quad i = 1, ..., n \\ & \mathbf{w}^T\mathbf{x}_i + b - y_i \leq \epsilon + \hat{\xi}_i, \quad i = 1, ..., n \\ & \xi_i, \hat{\xi}_i \geq 0, \quad i = 1, ..., n \end{aligned}$$

- We are penalizing samples whose prediction is at least ε away from their true target
- Similar to SVC, we can solve this problem by introducing Lagrangian multipliers and optimizing the dual Lagrangian form
- This allows us to use the kernel trick

SVR in Scikit-Learn

- As for classification, there are three different implementations of support vector regression: LinearSVR, SVR and NuSVR

class `sklearn.svm.` **`LinearSVR`**(**, epsilon=0.0, tol=0.0001, C=1.0, loss='epsilon_insensitive', fit_intercept=True, intercept_scaling=1.0, dual=True, verbose=0, random_state=None, max_iter=1000*) [source]

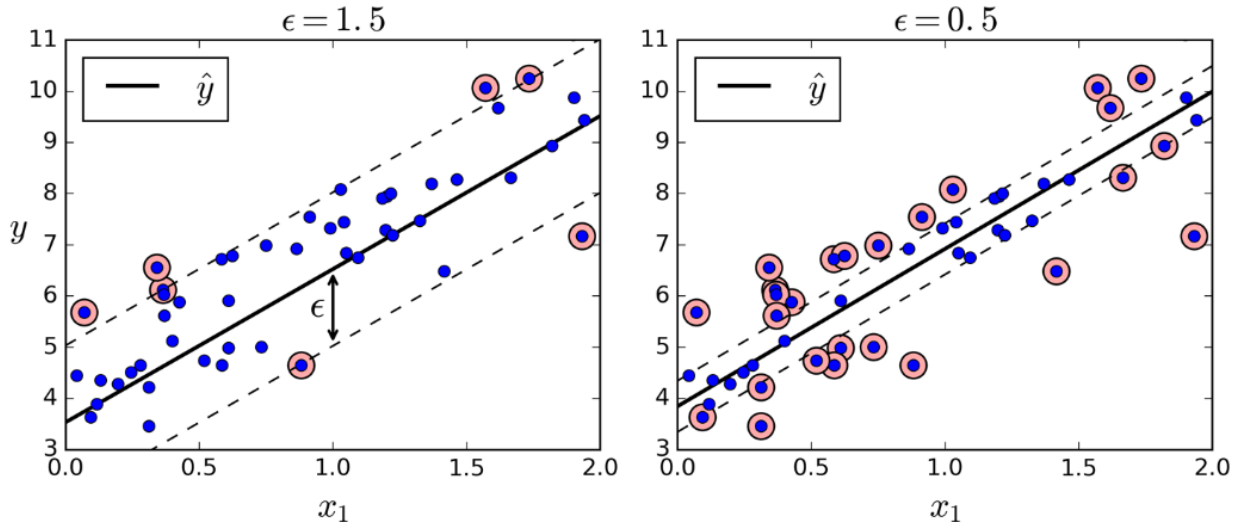
class `sklearn.svm.` **`SVR`**(**, kernel='rbf', degree=3, gamma='scale', coef0=0.0, tol=0.001, C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=- 1*) [source]

class `sklearn.svm.` **`NuSVR`**(**, nu=0.5, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, tol=0.001, cache_size=200, verbose=False, max_iter=- 1*) [source]

- LinearSVR provides a faster implementation than SVR but only supports the linear kernel

SVR Example

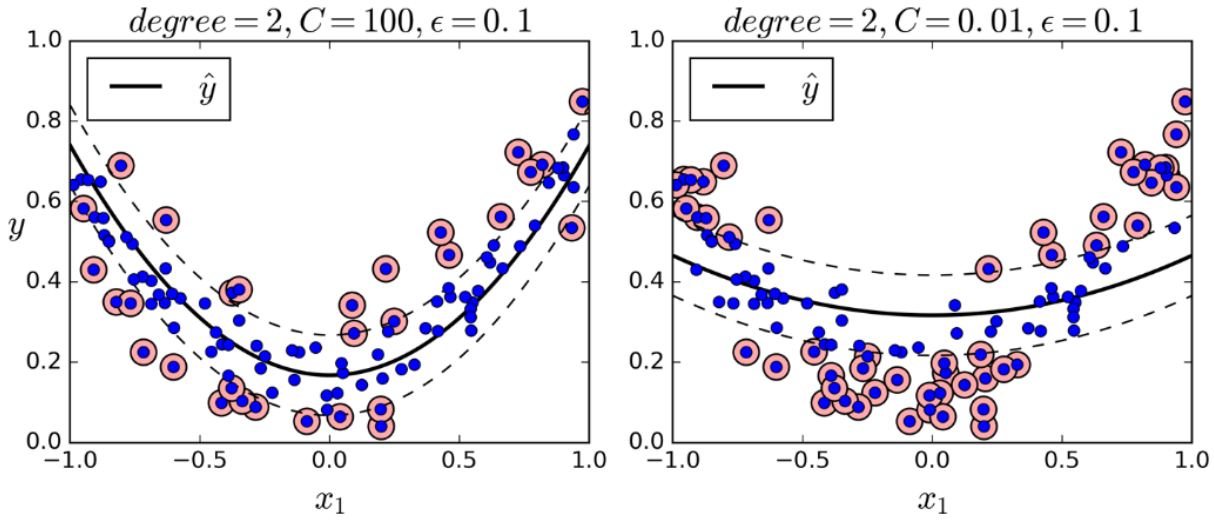
- The following figure shows two linear SVM regression models trained on some random linear data, one with a large margin and the other with a small margin



- As ε increases, the prediction becomes less sensitive to outliers

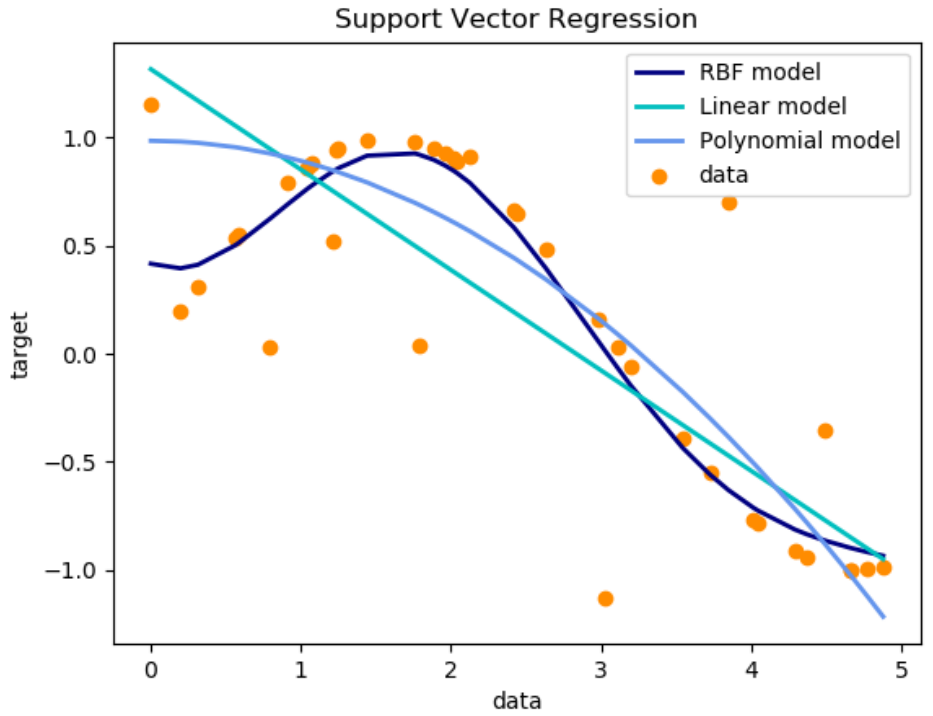
SVR Example

- To tackle nonlinear regression tasks, you can use a kernelized SVM model
- For example, the following figure shows SVM Regression on a random quadratic data set, using a 2nd-degree polynomial kernel with two different *C* values



SVR Example

- Using different SVR kernels on a 1D data:



SVM Summary

Pros

- Can be formulated as a convex optimization problem with a global minimum
- Improves generalization by maximizing the margin of the decision boundary
- Allows to control the balance between model complexity and training errors
- Works well with high-dimensional data
- Requires storage of only the support vectors
- Prediction of new samples is quite fast
- Can handle irrelevant and redundant attributes
- Can be used for outlier detection

Cons

- Training can be slow for large data sets
- Requires tuning of several key parameters
  - e.g., the parameter C and the kernel type
- Supports only binary classification
  - But can be extended to multi-class
- Doesn’t directly provide probability estimates