Kernel Methods

Lecture 24

Kernel Machine

Stores a subset of its **training examples** (instance-based learning)

Can learn implicitly **alternative feature spaces** without explicitly transforming the data into that space

Relies on a similarity measure, the **kernel function**, to compare test points to the training data

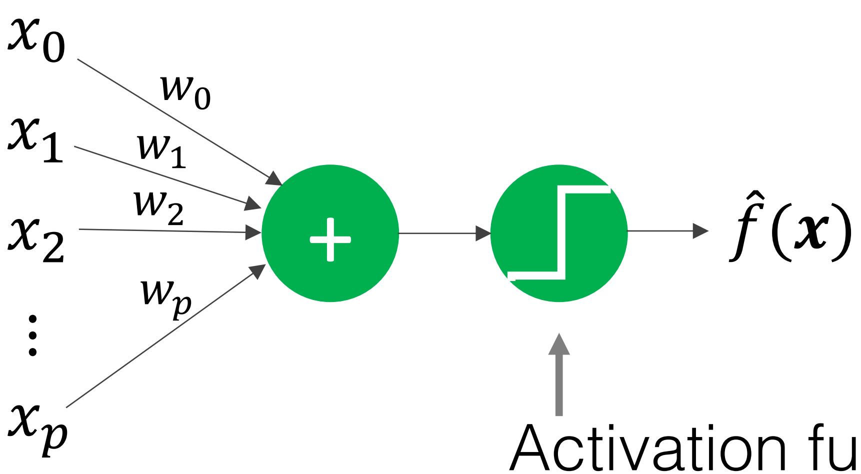
1. Perceptron → kernel perceptron
   (the kernel trick)
2. Kernel functions
   (making features space transforms easy)
3. Maximum margin classifier
   (explicit feature space, linearly separable data)
4. Support vector classifier
   (explicit feature space, non-linearly separable data)
5. Support vector machine
   (kernel-transformed implicit feature space, not linearly separable)

Recall linear models and the perceptron

Linear Classification

(perceptron)

$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_{i=0}^{p} w_i x_i\right) = sign(\boldsymbol{w}^\top \boldsymbol{x})$$



Activation function

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} \qquad x_0 \rightarrow 1$$

$$\boldsymbol{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_p \end{bmatrix} \qquad w_0 \rightarrow b \text{ (intercept)}$$

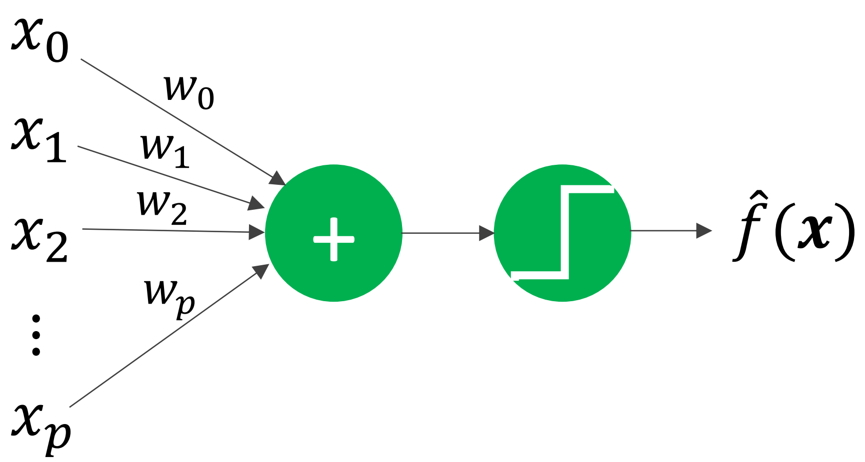
Source: Abu-Mostafa, Learning from Data, Caltech

Perceptron classifier

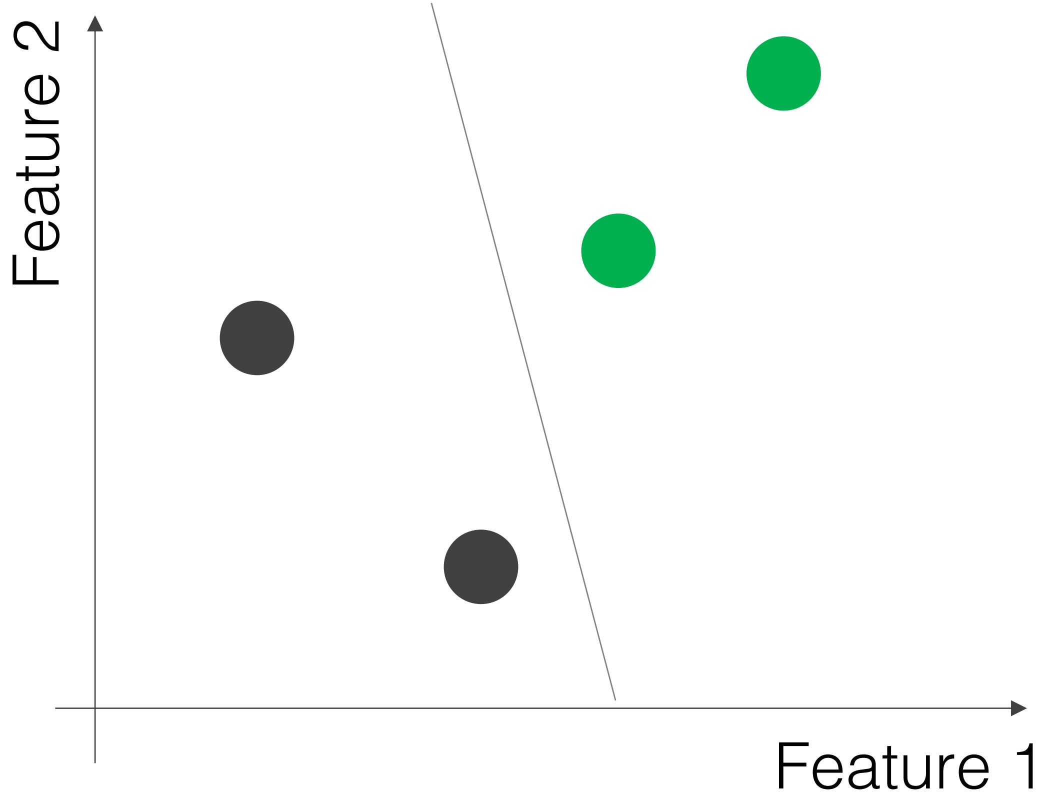
Linear Classification
(perceptron)

$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_{i=0}^{p} w_i x_i\right)$$

$$= sign(\boldsymbol{w}^\top \boldsymbol{x})$$



Idea: draw a line that separates the classes



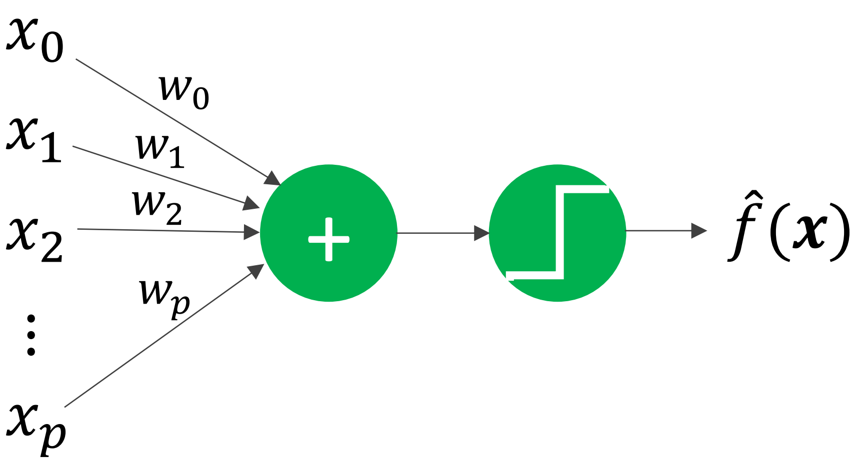
Source: Abu-Mostafa, Learning from Data, Caltech

Perceptron classifier

Linear Classification
(perceptron)

$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_{i=0}^{p} w_i x_i\right)$$

$$= sign(\boldsymbol{w}^\top \boldsymbol{x})$$



Training data: $(\boldsymbol{x}_n, y_n), \quad n = 1, \ldots, N$
with binary $y_n = \{-1,1\}$

Decision rule based on $sign(\boldsymbol{w}^T \boldsymbol{x})$:
if $\boldsymbol{w}^\top \boldsymbol{x}_n > 0$, then $\hat{y}_n = +1$
if $\boldsymbol{w}^\top \boldsymbol{x}_n < 0$, then $\hat{y}_n = -1$

For correctly classified points: $y_n \boldsymbol{w}^\top \boldsymbol{x}_n > 0$
(and no error is assigned if correctly classified)

Source: Abu-Mostafa, Learning from Data, Caltech

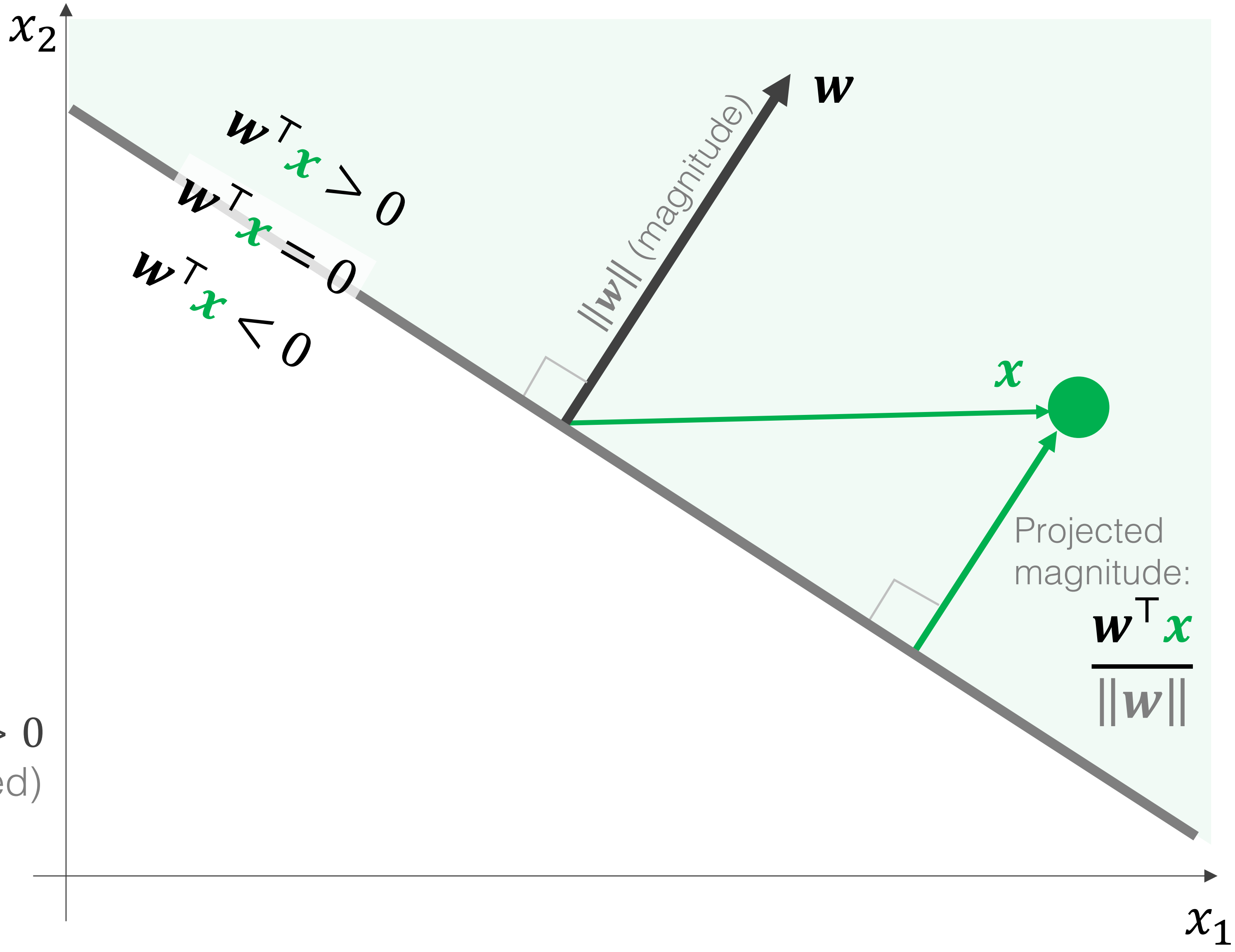
The perceptron classifier

$$\hat{f}(\boldsymbol{x}) = sign(\boldsymbol{w}^\top \boldsymbol{x})$$

Decision rule based on $sign(\boldsymbol{w}^T\boldsymbol{x})$:

$$\text{if } \boldsymbol{w}^\top \boldsymbol{x}_n > 0, \text{ then } \hat{y}_n = +1$$
$$\text{if } \boldsymbol{w}^\top \boldsymbol{x}_n < 0, \text{ then } \hat{y}_n = -1$$

For correctly classified points: $y_n \boldsymbol{w}^\top \boldsymbol{x}_n > 0$
(and no error is assigned if correctly classified)

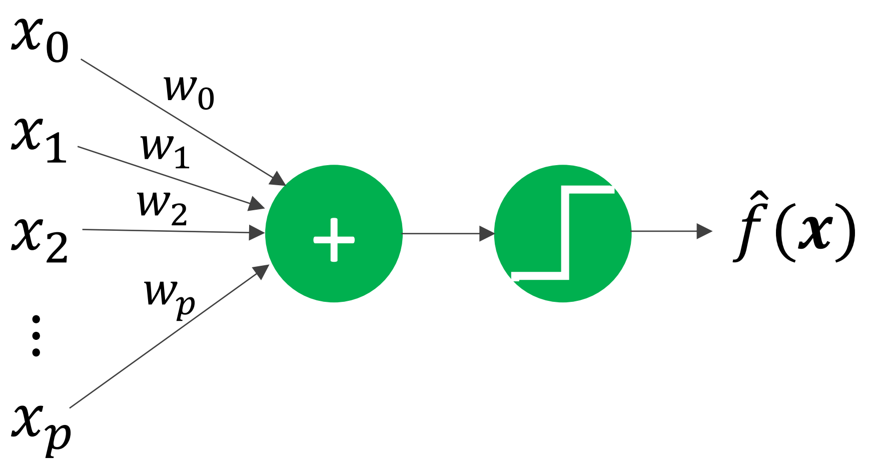


Perceptron classifier

Linear Classification
(perceptron)

$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_{i=0}^{p} w_i x_i\right)$$

$$= sign(\boldsymbol{w}^\top \boldsymbol{x})$$



Training data: $(\boldsymbol{x}_n, y_n)$, $\quad n = 1, \ldots, N$
with binary $y_n = \{-1,1\}$

Decision rule based on $sign(\boldsymbol{w}^\top \boldsymbol{x})$:
if $\boldsymbol{w}^\top \boldsymbol{x}_n > 0$, then $\hat{y}_n = +1$
if $\boldsymbol{w}^\top \boldsymbol{x}_n < 0$, then $\hat{y}_n = -1$

For correctly classified points: $y_n \boldsymbol{w}^\top \boldsymbol{x}_n > 0$
(and no error is assigned if correctly classified)

Our cost (error) function to minimize:

$$C = -\sum_{\substack{n \in \{\text{mistakes}\} \\ \hat{y}_n \neq y_n}} y_n \boldsymbol{w}^\top \boldsymbol{x}_n$$

Source: Abu-Mostafa, Learning from Data, Caltech

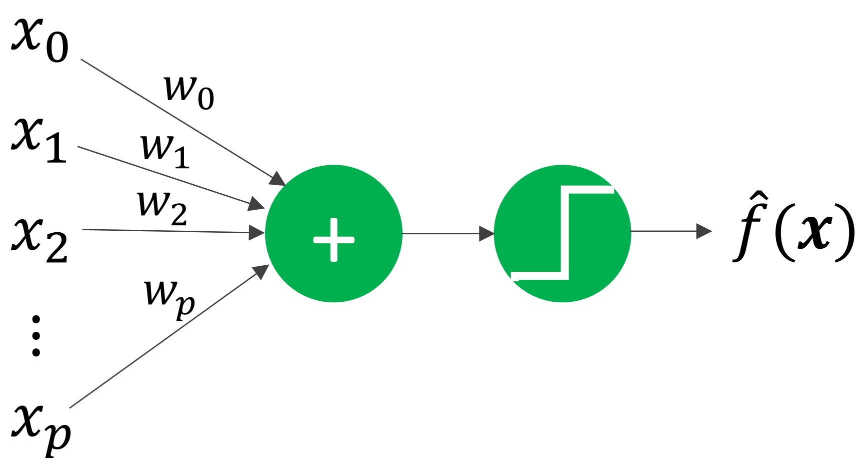
Perceptron classifier

Linear Classification

(perceptron)

$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_{i=0}^{p} w_i x_i\right)$$

$$= sign(\boldsymbol{w}^\top \boldsymbol{x})$$



Our cost (error) function to minimize:

$$C = -\sum_{n \in \{\text{mistakes}\}} y_n \boldsymbol{w}^\top \boldsymbol{x}_n$$

The gradient with respect to $\boldsymbol{w}$:

$$\frac{\partial E}{\partial \boldsymbol{w}} = -\sum_{n \in \{\text{mistakes}\}} y_n \boldsymbol{x}_n$$

Applying stochastic gradient:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{\partial E}{\partial \boldsymbol{w}}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n \boldsymbol{x}_n$$

process one mistake at a time and assume a learning rate of 1

Source: Abu-Mostafa, Learning from Data, Caltech

Perceptron Learning Algorithm

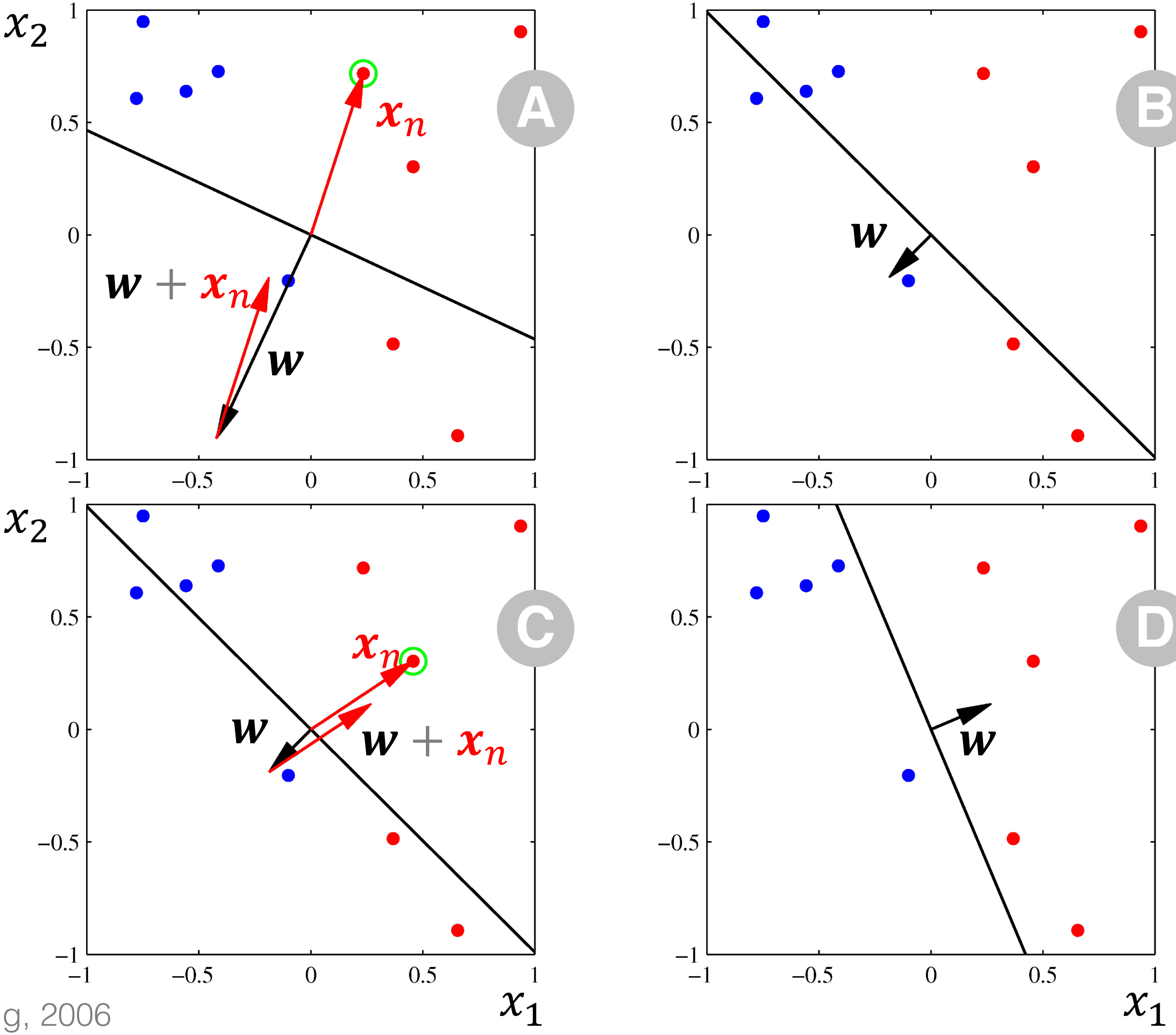
1. Pick a misclassified point and use it to update the weights:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n \boldsymbol{x}_n$$

2. Reclassify all the data:

$$\hat{y}_n = sign(\boldsymbol{w}^\top \boldsymbol{x}_n)$$

3. Repeat until no mistakes



Bishop, Pattern Recognition and Machine Learning, 2006

Perceptron Learning Algorithm (towards kernels)

1. Pick a misclassified point and use it to update the weights:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n \boldsymbol{x}_n$$

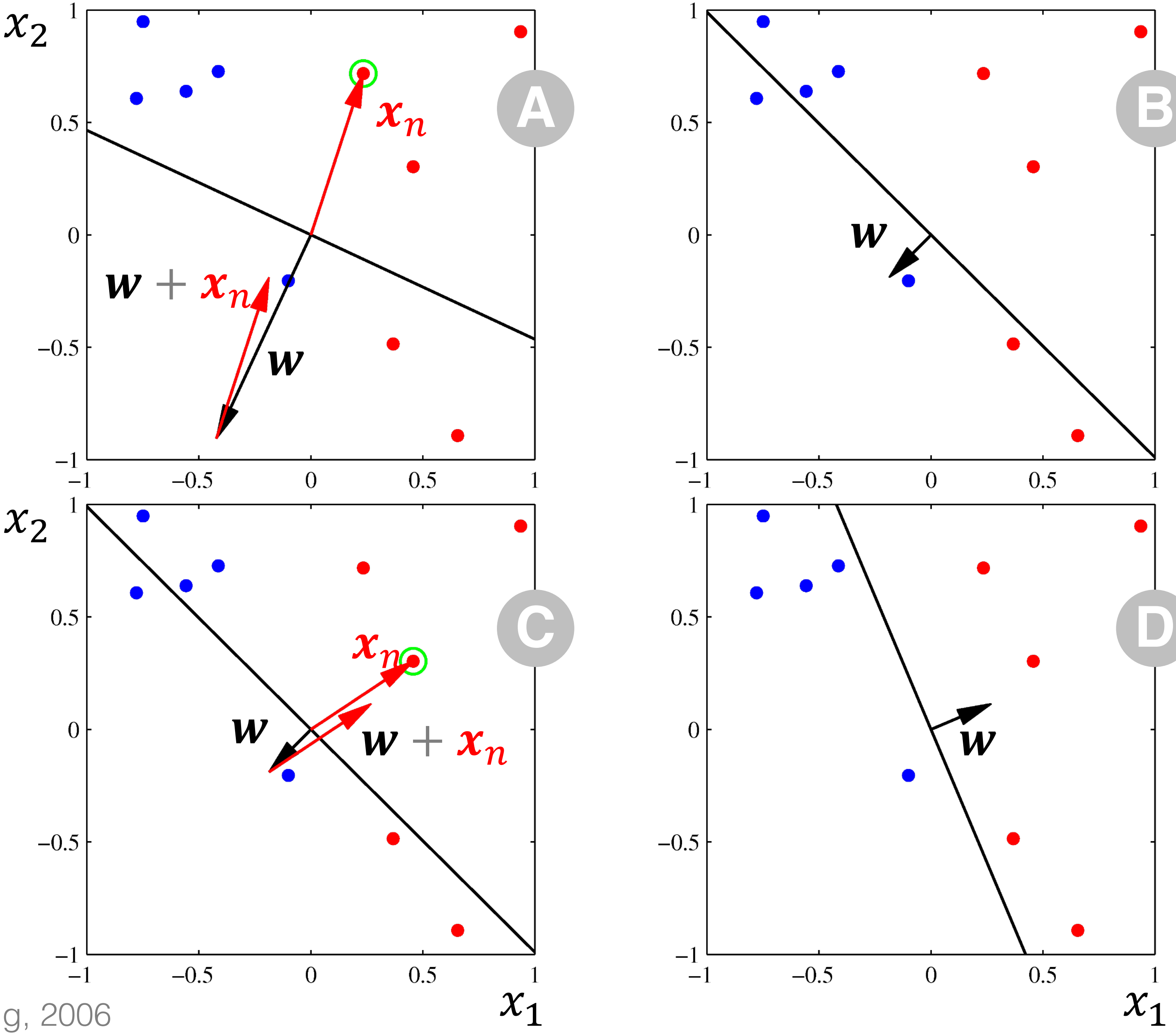
$$a_n \leftarrow a_n + 1$$

(mistake counter)

2. Reclassify all the data:

$$\hat{y}_n = sign(\boldsymbol{w}^\top \boldsymbol{x}_n)$$

3. Repeat until no mistakes



Bishop, Pattern Recognition and Machine Learning, 2006

Perceptron Learning Algorithm (towards kernels)

Update weights

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n \boldsymbol{x}_n$$

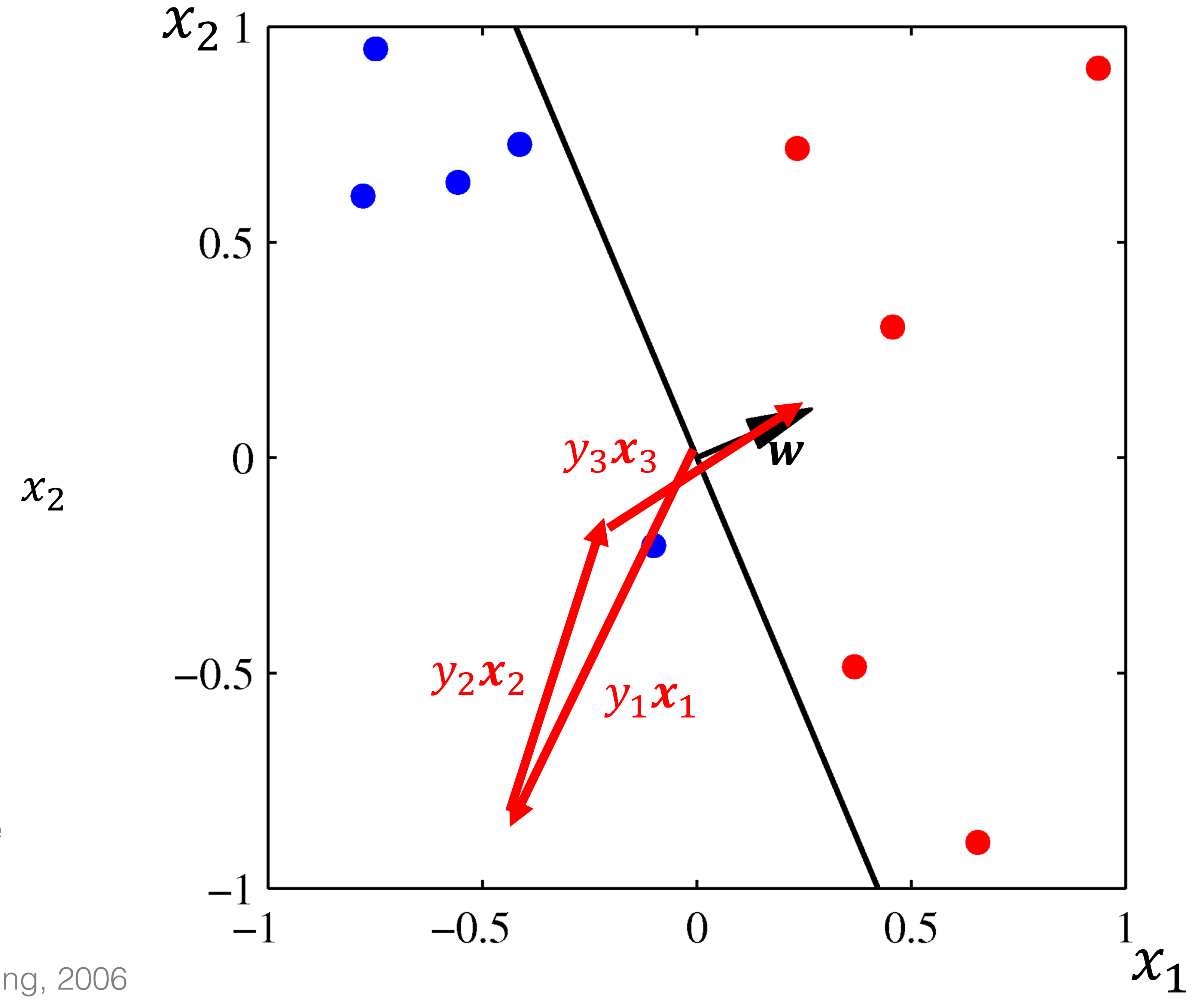
$$a_n \leftarrow a_n + 1$$

(mistake counter)

We can rewrite an expression for our weights:

$$\boldsymbol{w} = \sum_n a_n y_n \boldsymbol{x}_n$$

If we store our mistake counter, we can update our weights as a sum over all observations, but only the mistakes that were considered will have a nonzero value for a_n



Bishop, Pattern Recognition and Machine Learning, 2006

Perceptron Learning Algorithm (towards kernels)

Update weights

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n \boldsymbol{x}_n$$

$$a_n \leftarrow a_n + 1$$

(mistake counter)

We can rewrite an expression for our weights:

$$\boldsymbol{w} = \sum_n a_n y_n \boldsymbol{x}_n$$

If we store our mistake counter, we can update our weights as a sum over all observations, but only the mistakes that were considered will have a nonzero value for a_n

Let's plug this new expression into our classifier:

$$\hat{y} = \hat{f}(\boldsymbol{x}) = sign(\boldsymbol{w}^\top \boldsymbol{x})$$

$$= sign\left(\left(\sum_n a_n y_n \boldsymbol{x}_n\right)^\top \boldsymbol{x}\right)$$

$$= sign\left(\sum_n \underline{a_n y_n} \, \underline{\boldsymbol{x}_n^\top \boldsymbol{x}}\right)$$

new model parameters inner product

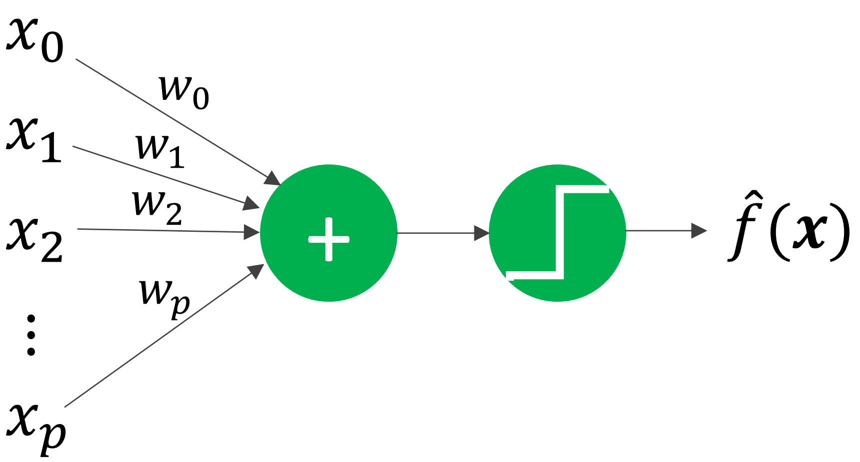
Our classifier **stores training data**, but it only depends on **inner products**

Kernel perceptron classifier

Linear Classification

(perceptron)

$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_n a_n y_n \boldsymbol{x}_n^\top \boldsymbol{x}\right)$$



Our classifier **stores training data**, but it only depends on an **inner product**

$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_n a_n y_n \boldsymbol{x}_n^\top \boldsymbol{x}\right)$$

We can write this inner product as a **kernel function**, $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{x}'$

$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_n a_n y_n K(\boldsymbol{x}_n, \boldsymbol{x})\right)$$

We can replace this with **any valid kernel**

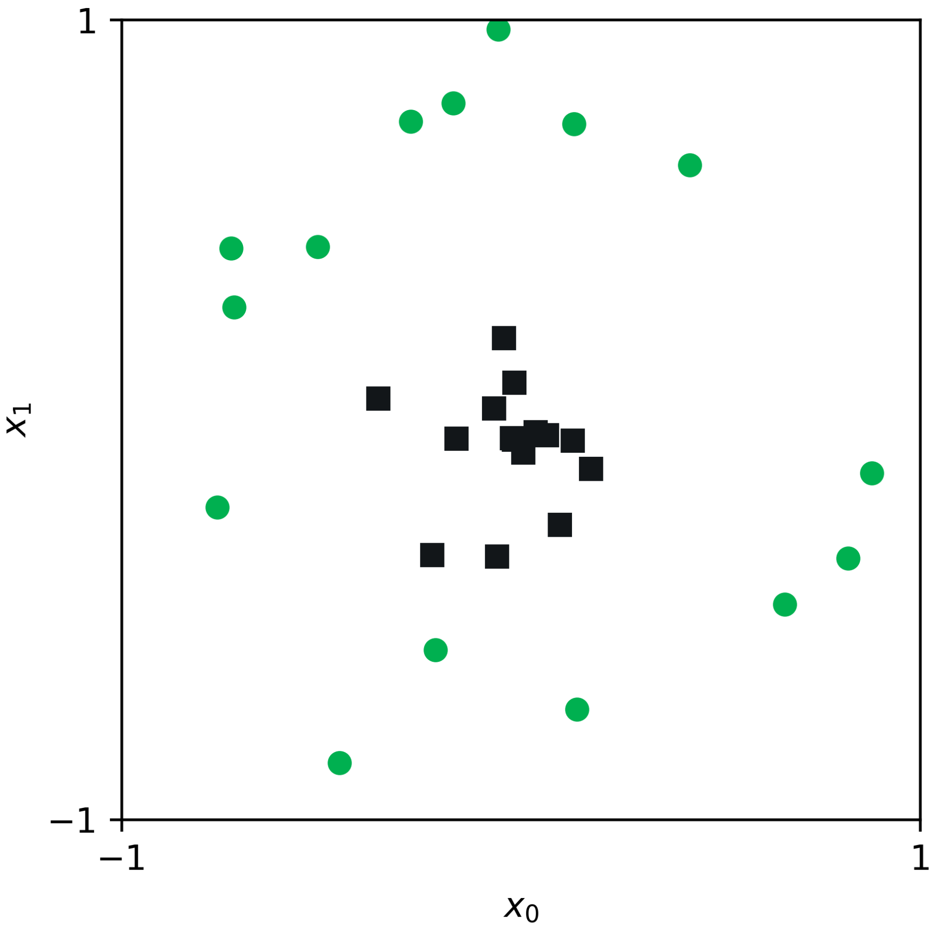
Source: Abu-Mostafa, Learning from Data, Caltech

What are **kernels** and why are they useful?

Limitations of linear decision boundaries

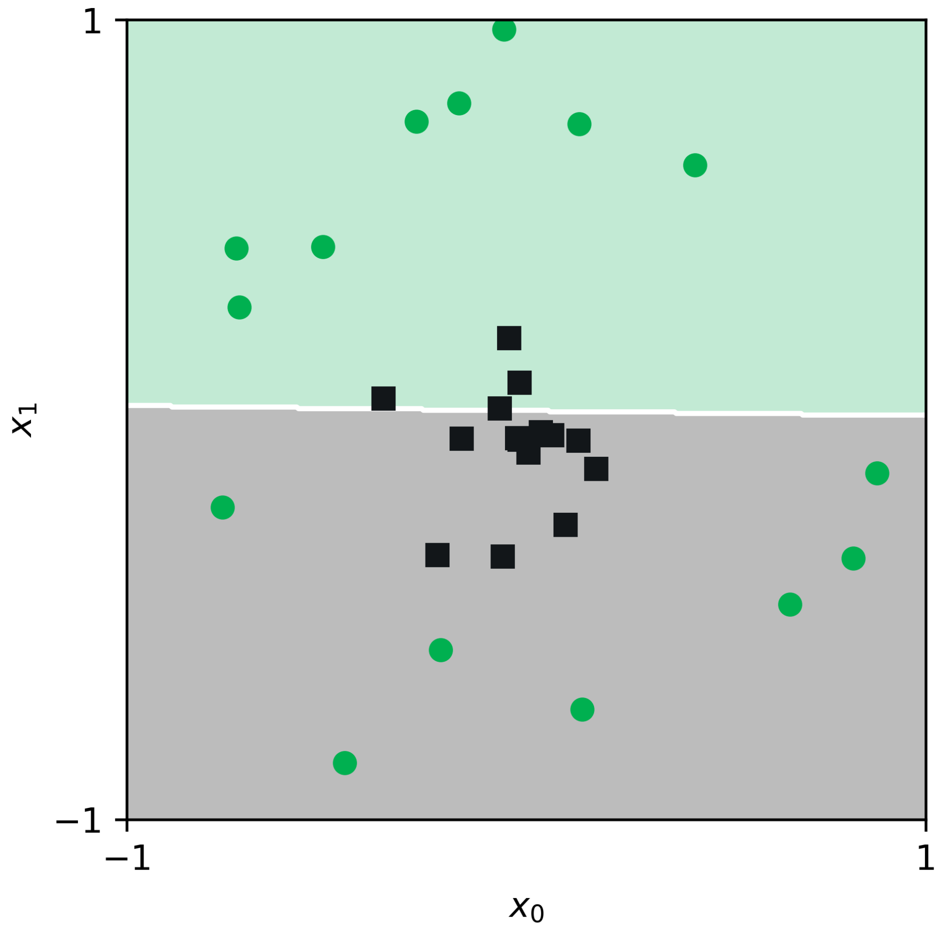
Original data

$\boldsymbol{x}$



Classify the features in this X-space

$\hat{f}_{\boldsymbol{x}}(\boldsymbol{x}) = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x})$



Transformations of features

Recall our digits example…

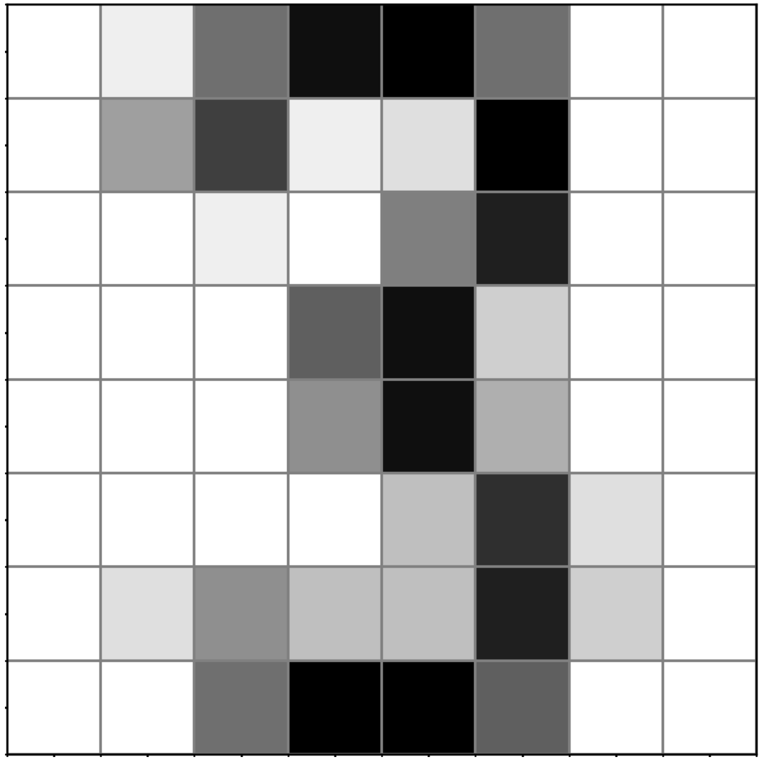
$$\boldsymbol{x} = [x_1, x_2, x_3, \dots, x_{64}]$$

We could create features based on the raw features. For example:

$$\boldsymbol{z} = [x_1 x_2, x_3^2, \frac{x_{64}}{x_{42}}]$$

Which can be written simply as variables in a new feature space:

$$\boldsymbol{z} = [z_1, z_2, z_3]$$

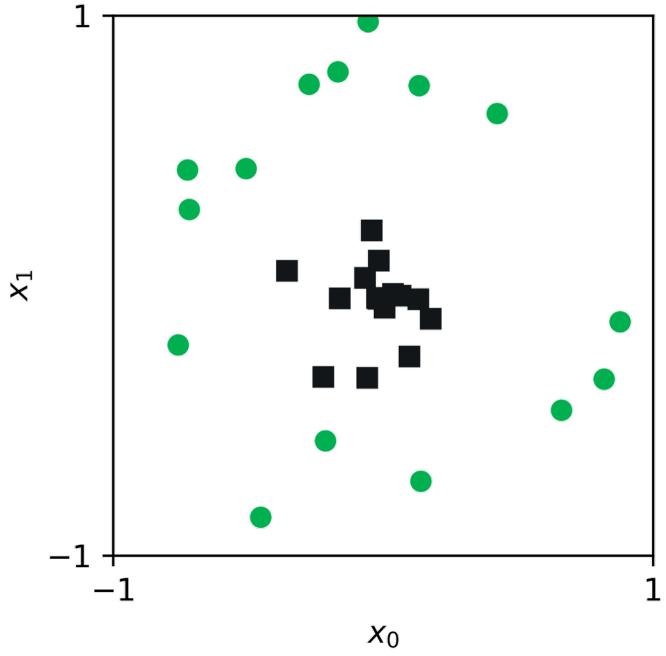


Source: Abu-Mostafa, Learning from Data, Caltech

1

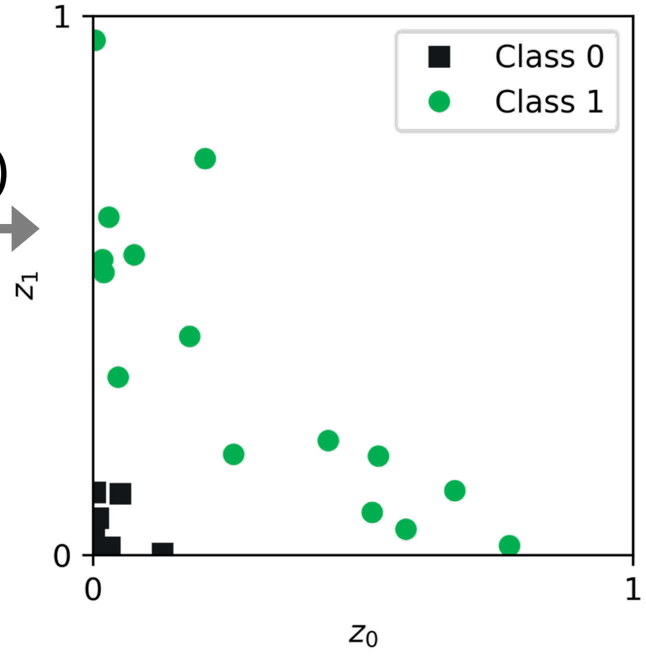
Original data

$\boldsymbol{x}$



transform the data

$\boldsymbol{z} = \Phi(\boldsymbol{x})$



2

This example transform is quadratic

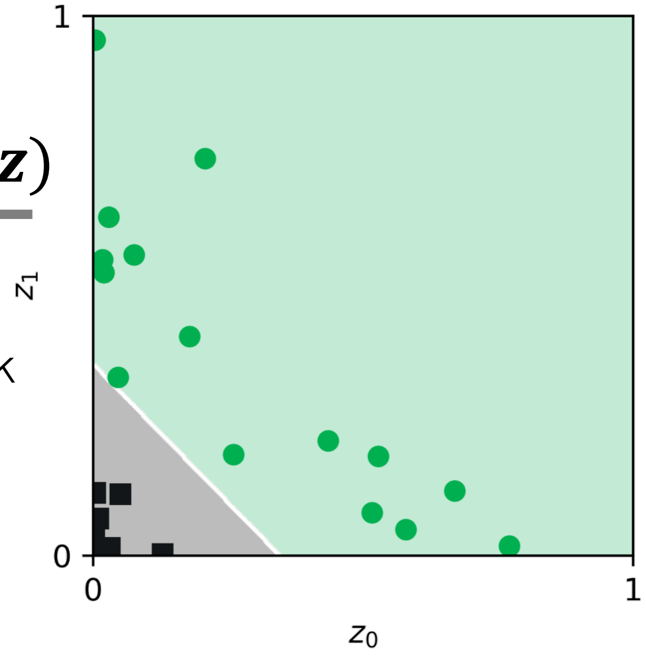
$z_i = \Phi(x_i) = x_i^2$

$z_0 = x_0^2$

$z_1 = x_1^2$

Classify the features in this Z-space

$\hat{f}_z(\boldsymbol{z}) = \text{sign}(\boldsymbol{w}^\top \boldsymbol{z})$



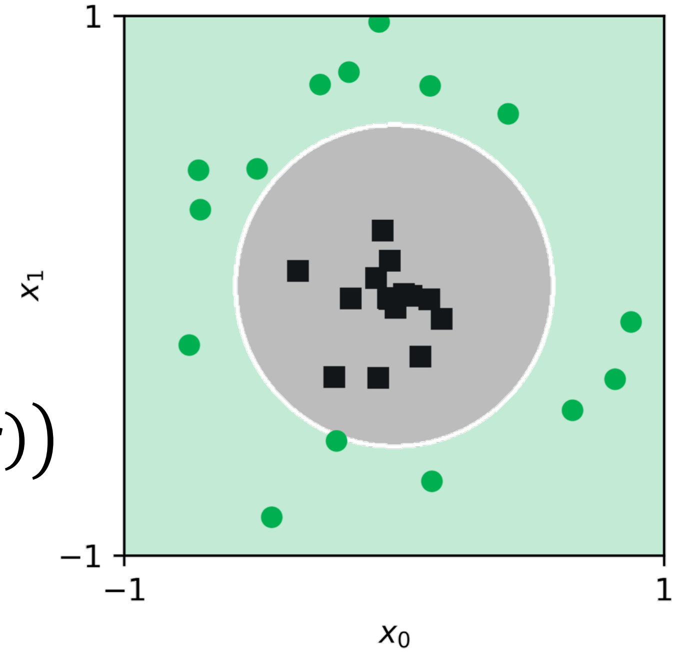
3

$\boldsymbol{x} = \Phi^{-1}(\boldsymbol{z})$

transform the data back

$x_0 = z_0^{1/2}$

$x_1 = z_1^{1/2}$



Predictions in the original X-space

$\hat{f}(\boldsymbol{x}) = \hat{f}_z\big(\Phi(\boldsymbol{x})\big)$

4

We can transform the feature space

Transform the feature space

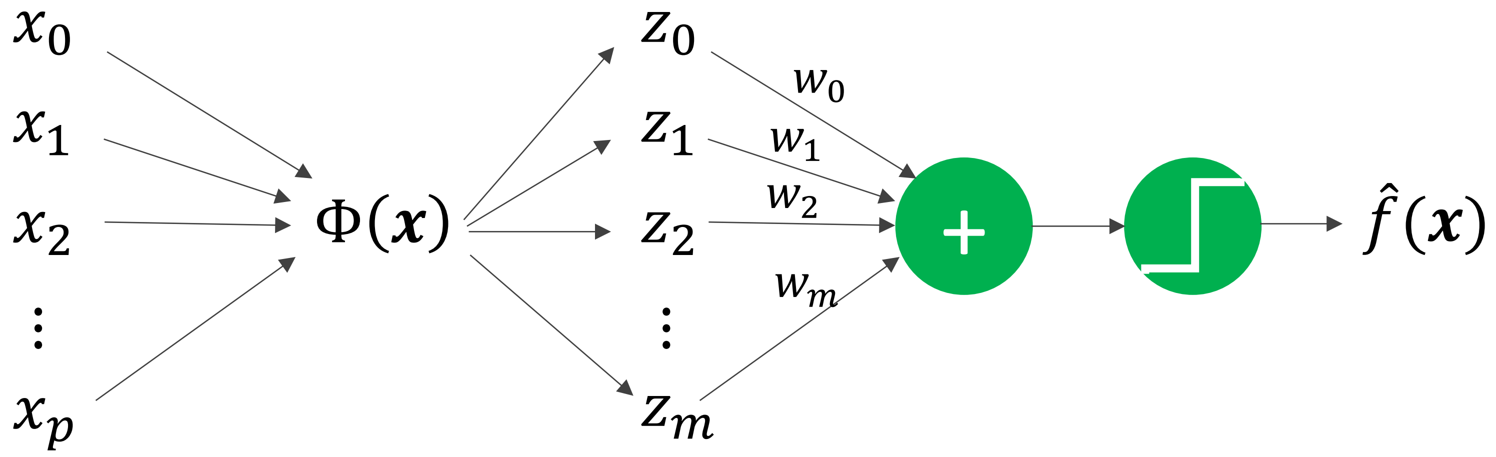
$$\boldsymbol{z} = \Phi(\boldsymbol{x})$$

Perceptron Classifier

$$\hat{y} = \hat{f}(\boldsymbol{x}) = sign(\boldsymbol{w}^\top \boldsymbol{z})$$

Perceptron Learning Algorithm still applies

1. $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n \boldsymbol{z}_n$
2. $\hat{y}_n = sign(\boldsymbol{w}^\top \boldsymbol{z}_n)$



For example, a polynomial feature space

$$\boldsymbol{x} = [x_1 \quad x_2]^{\mathsf{T}}$$

$$\boldsymbol{z} = \Phi(\boldsymbol{x}) = [1 \quad x_1 \quad x_2 \quad x_1^2 \quad x_2^2 \quad x_1 x_2]^{\mathsf{T}}$$

Transform into a 2^{nd}-order polynomial feature space

This second order polynomial space with 2 features is simple enough

What about a 100^{th} order polynomial space with 25 features?

That would be more than $\mathbf{10^{26}}$ terms!

Transformations into alternative feature spaces may make the prediction problem easier

Can be **computationally challenging** to complete the transformation into those feature spaces explicitly…

Solution: **kernel functions / the kernel trick**

Perform learning in the feature space without explicitly transforming features into it

Kernel function

Definition for kernel methods

Similarity measure between two points $\boldsymbol{x}$ and $\boldsymbol{x}'$

A **kernel function**, $K(\boldsymbol{x}, \boldsymbol{x}')$, represents an **inner product in some feature space**

$$\langle \boldsymbol{z}, \boldsymbol{z}' \rangle = \boldsymbol{z} \cdot \boldsymbol{z}' = \boldsymbol{z}^T \boldsymbol{z}' \qquad \boldsymbol{z} = \Phi(\boldsymbol{x})$$

for Euclidean spaces

For a valid kernel, there is some feature transformation, $\boldsymbol{z} = \Phi(\boldsymbol{x})$, where:

$$K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{z}^\top \boldsymbol{z}$$

Simplest example: the linear kernel $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{x}'$

Kernel function example

$$\boldsymbol{x} = [x_1 \quad x_2]^\top$$

$$\boldsymbol{z} = \Phi(\boldsymbol{x}) = [1 \quad x_1 \quad x_2 \quad x_1^2 \quad x_2^2 \quad x_1 x_2]^\top$$

Transform into a 2nd-order polynomial feature space

The kernel function is:

$$K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{z}^\top \boldsymbol{z}' = 1 + x_1 x_1' + x_2 x_2' + x_1^2 {x_1'}^2 + x_2^2 {x_2'}^2 + x_1 x_1' x_2 x_2'$$

Compute $K(\boldsymbol{x}, \boldsymbol{x}')$ without the explicit $\boldsymbol{z} = \Phi(\boldsymbol{x})$ feature space transformation:

Kernel Trick

Source: Abu-Mostafa, Learning from Data, Caltech

Kernel trick

$\boldsymbol{x} = [x_1 \quad x_2]^{\mathsf{T}}$

Compute $K(\boldsymbol{x}, \boldsymbol{x}')$ without the $\boldsymbol{z} = \Phi(\boldsymbol{x})$ feature space transformation

Example:

$K(\boldsymbol{x}, \boldsymbol{x}') = (1 + \boldsymbol{x}^T \boldsymbol{x}')^2$ This is not an inner product in X-space

$= (1 + x_1 x_1' + x_2 x_2')^2$

$= 1 + x_1 x_1' + x_2 x_2' + 2 x_1^2 {x_1'}^2 + 2 x_2^2 {x_2'}^2 + 2 x_1 x_1' x_2 x_2'$

Similar to the inner product for: $\boldsymbol{z} = \Phi(\boldsymbol{x}) = [1 \quad x_1 \quad x_2 \quad x_1^2 \quad x_2^2 \quad x_1 x_2]^{\mathsf{T}}$

It **IS an inner product** in a **different** Z-space:

$$\boldsymbol{z} = \Phi(\boldsymbol{x}) = \left[1 \quad x_1 \quad x_2 \quad \sqrt{2}x_1^2 \quad \sqrt{2}x_2^2 \quad \sqrt{2}x_1 x_2\right]^{\mathsf{T}}$$

$K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{z}^T \boldsymbol{z}'$

Computing $K(\boldsymbol{x}, \boldsymbol{x}') = (1 + \boldsymbol{x}^T \boldsymbol{x}')^2$ Is much easier than the full Z-space transform. Imagine if this was $(1 + \boldsymbol{x}^T \boldsymbol{x}')^{100}$!

Source: Abu-Mostafa, Learning from Data, Caltech

Common kernel functions

Linear kernel: $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$

Polynomial kernels: $K(\boldsymbol{x}, \boldsymbol{x}') = (1 + \boldsymbol{x}^T \boldsymbol{x}')^d$

(all polynomials up to degree d)

Radial basis function kernel: $K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|^2}{2\sigma^2}\right)$

(infinite dimensional)

For an excellent explanation of how this is infinite dimensional, see Yaser Abu-Mostafa's explanation

Kernel function properties

Symmetric: $K(\boldsymbol{x}, \boldsymbol{x}') = K(\boldsymbol{x}', \boldsymbol{x})$

All kernels are symmetric

Stationary kernels: $K(\boldsymbol{x}, \boldsymbol{x}') = K(\boldsymbol{x} - \boldsymbol{x}')$

Invariant to translation in the input space

Only a function of the difference between arguments

Homogeneous kernels: $K(\boldsymbol{x}, \boldsymbol{x}') = \mathrm{K}(\|\boldsymbol{x} - \boldsymbol{x}'\|)$

Depend only on the magnitude of the distance between arguments

Kernel perceptron classifier

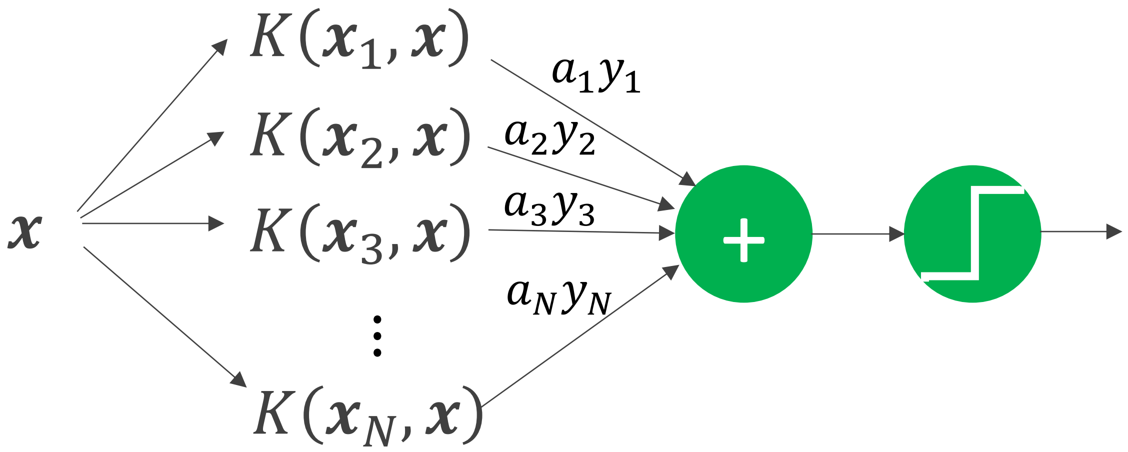
No need to explicitly transform the feature space

$$\cancel{\boldsymbol{z} = \Phi(\boldsymbol{x})}$$

We only need the kernel function

Now we need to store our training data

We have to use all the training data in each prediction



$$\hat{f}(\boldsymbol{x}) = sign\left(\sum_n a_n y_n K(\boldsymbol{x}_n, \boldsymbol{x})\right)$$

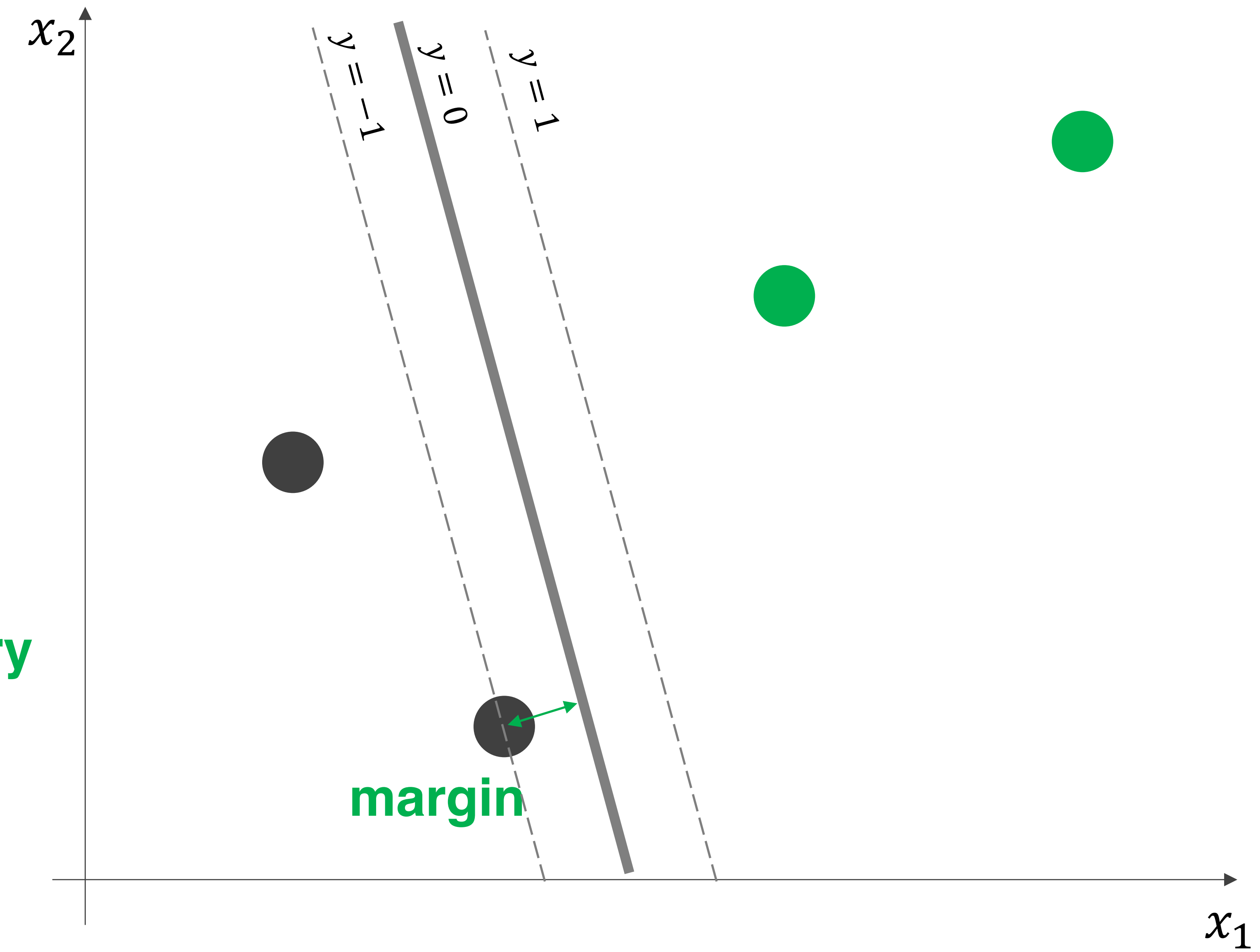
How can we improve on the perceptron

Assume our data are linearly separable

How do we pick the "best" separating line (hyperplane)?

Maximize the **margin**

Margin = the smallest distance between the **decision boundary** and **any** of the samples



Bishop, Pattern Recognition and Machine Learning, 2006

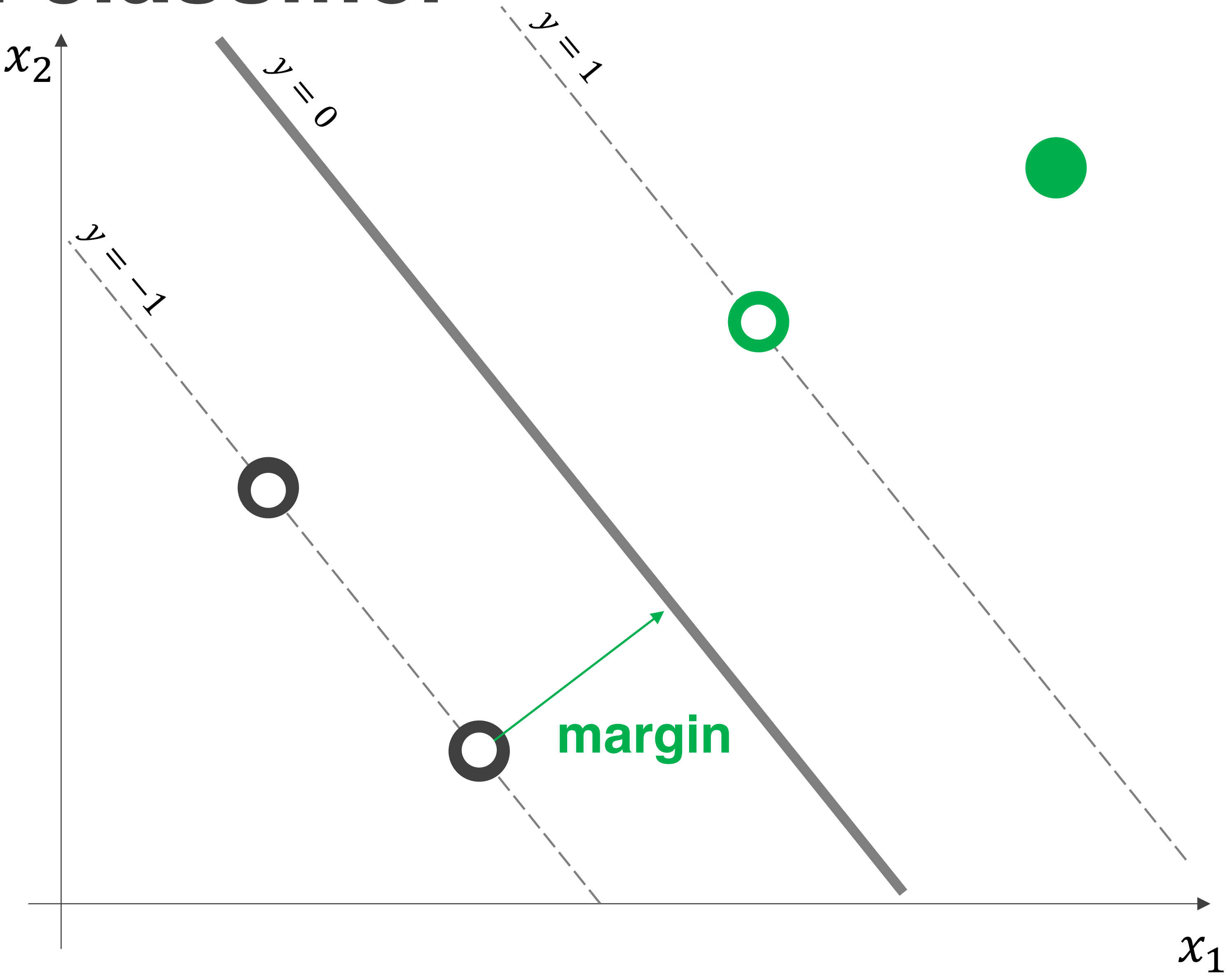
Maximum margin classifier

The decision boundary is determined by the weight, $\boldsymbol{w}$, as with the perceptron

Pick $\boldsymbol{w}$ to maximize the margin

Assumes linear separability

Hard margin classifier



Bishop, Pattern Recognition and Machine Learning, 2006

Support vector classifier

Penalty term for violating the margin:
$\xi_n = |y_n - \boldsymbol{w}^\top \boldsymbol{x}|$
where $y = \boldsymbol{w}^\top \boldsymbol{x}$
and $y_n \in \{-1,1\}$

The decision boundary is determined by the weight, $\boldsymbol{w}$, as with the perceptron

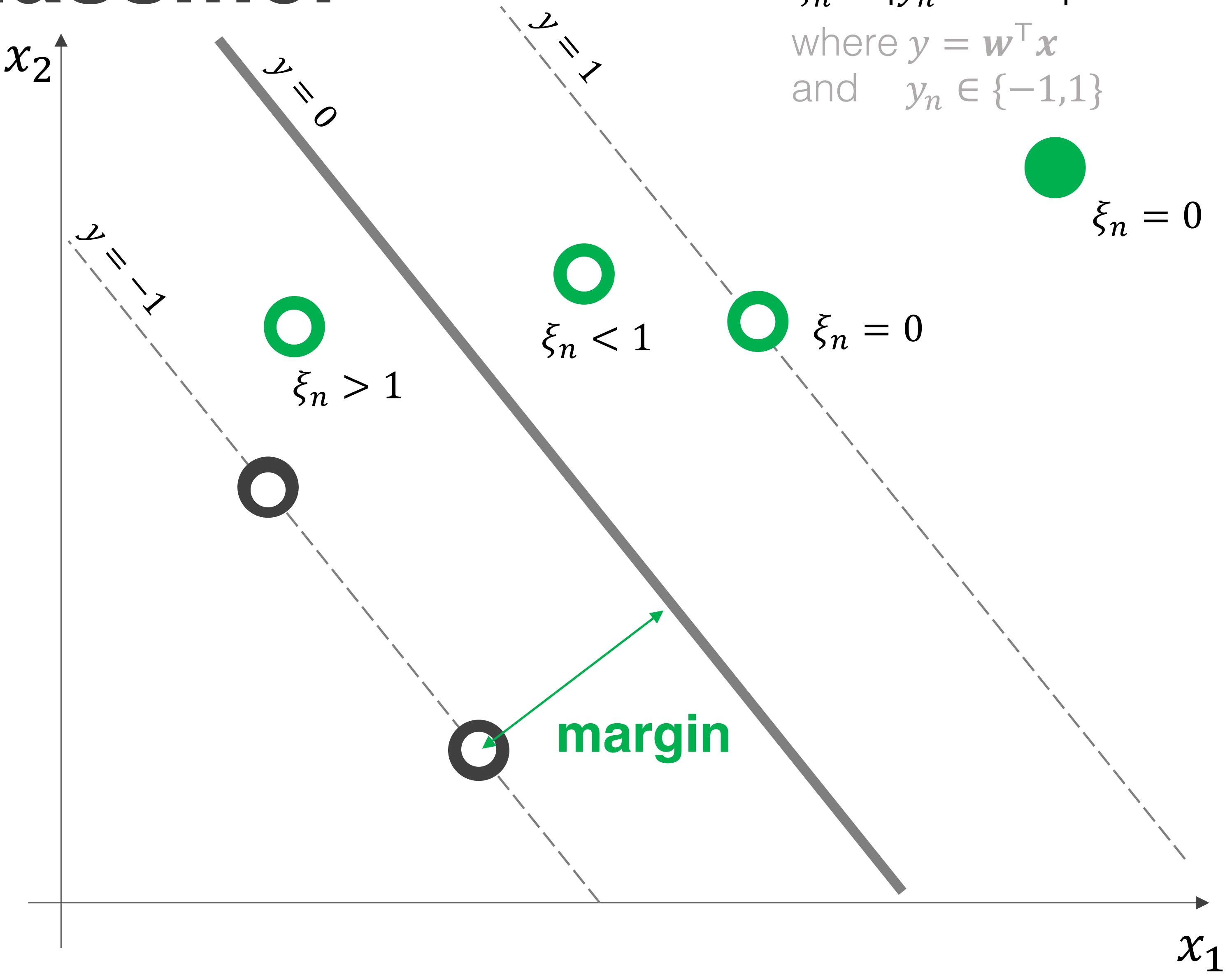
Pick $\boldsymbol{w}$ to maximize the margin

Does not assume linear separability

Soft margin classifier

Minimize: $L(x) = \sum_{n=1}^{N} \xi_n + C\|\boldsymbol{w}\|^2$

$\xi_n \geq 0$



Bishop, Pattern Recognition and Machine Learning, 2006

Support vector machine

Penalty term for violating the margin:
$\xi_n = |y_n - \boldsymbol{w}^\top \boldsymbol{x}|$
where $y = \boldsymbol{w}^\top \boldsymbol{x}$
and $y_n \in \{-1,1\}$

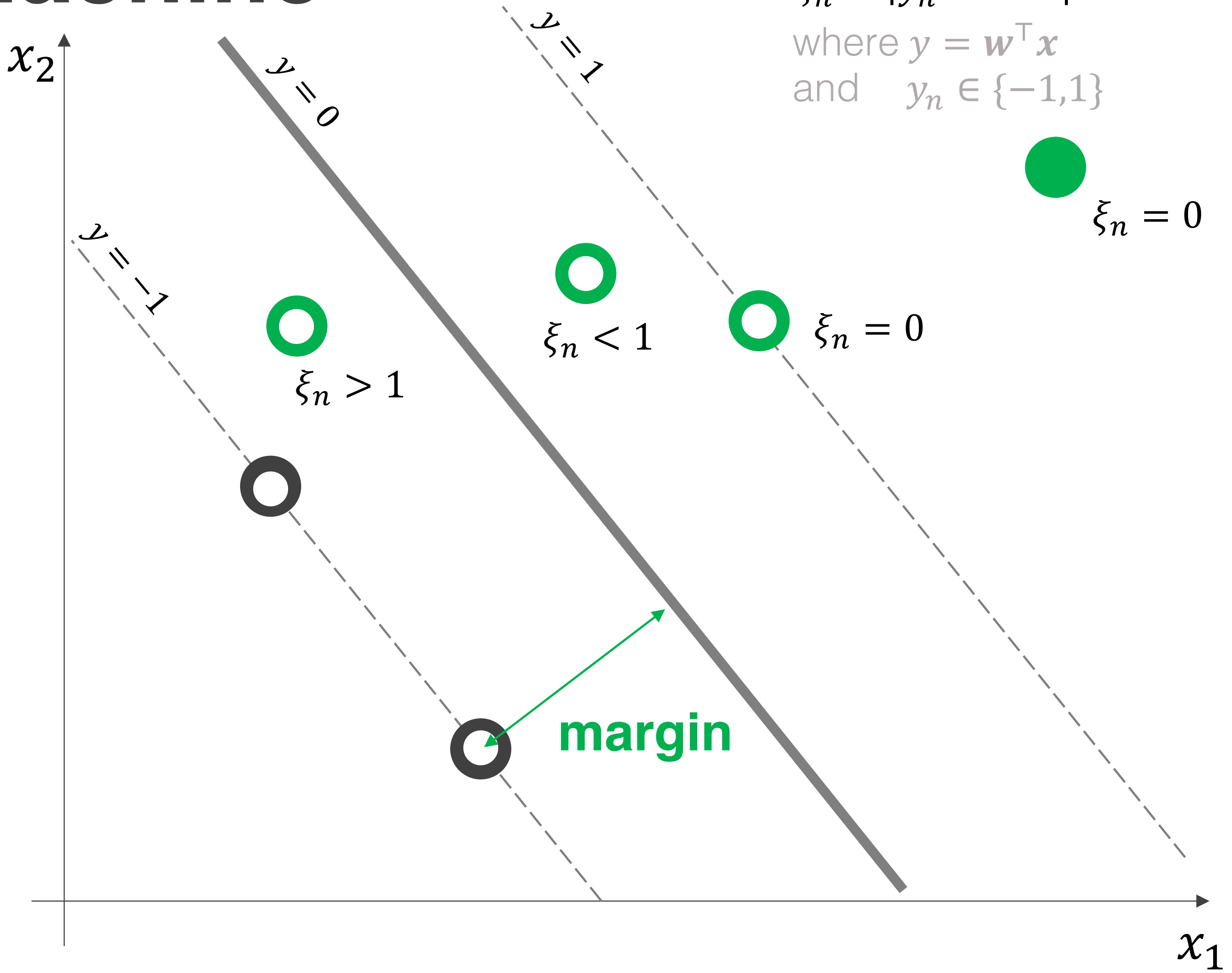
The decision boundary is determined by the weight, $\boldsymbol{w}$, as with the perceptron

Pick $\boldsymbol{w}$ to maximize the margin

Does not assume linear separability

Soft margin classifier

Use the **kernel trick** to classify in other feature spaces



Bishop, Pattern Recognition and Machine Learning, 2006

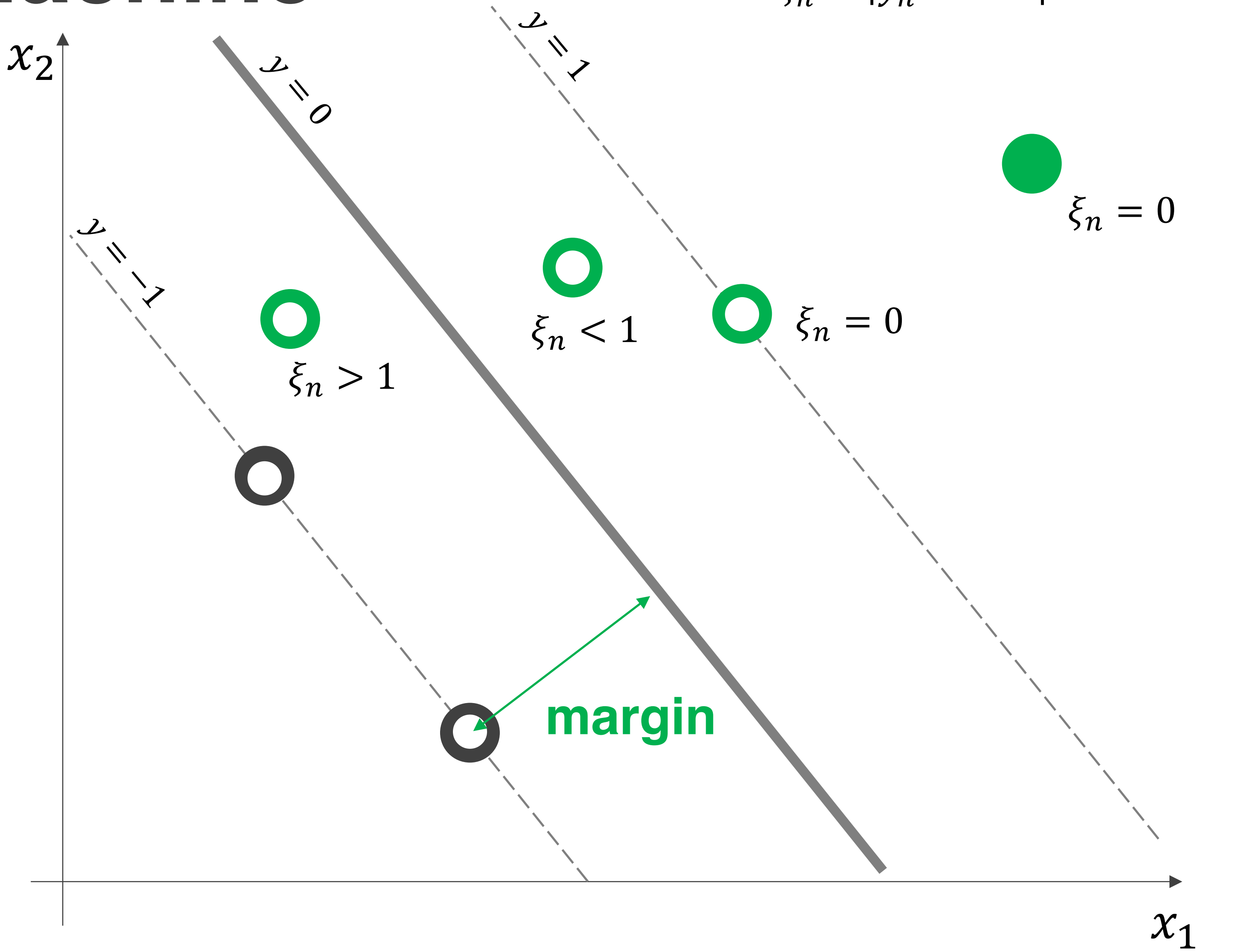
Support vector machine

Penalty term for violating the margin:
$\xi_n = |y_n - \boldsymbol{w}^\top \boldsymbol{x}|$

Use the **kernel trick** to classify in other feature spaces

Sparse kernel machine

Prediction: kernel comparisons with weighted support vectors (very similar to the perceptron)

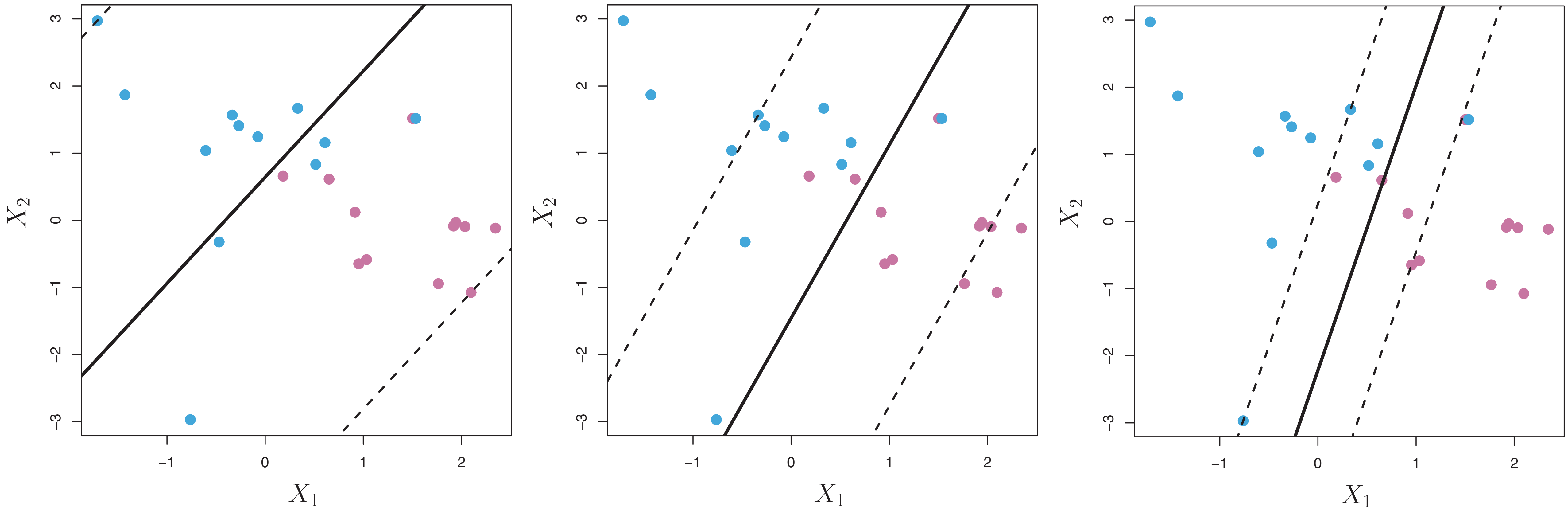


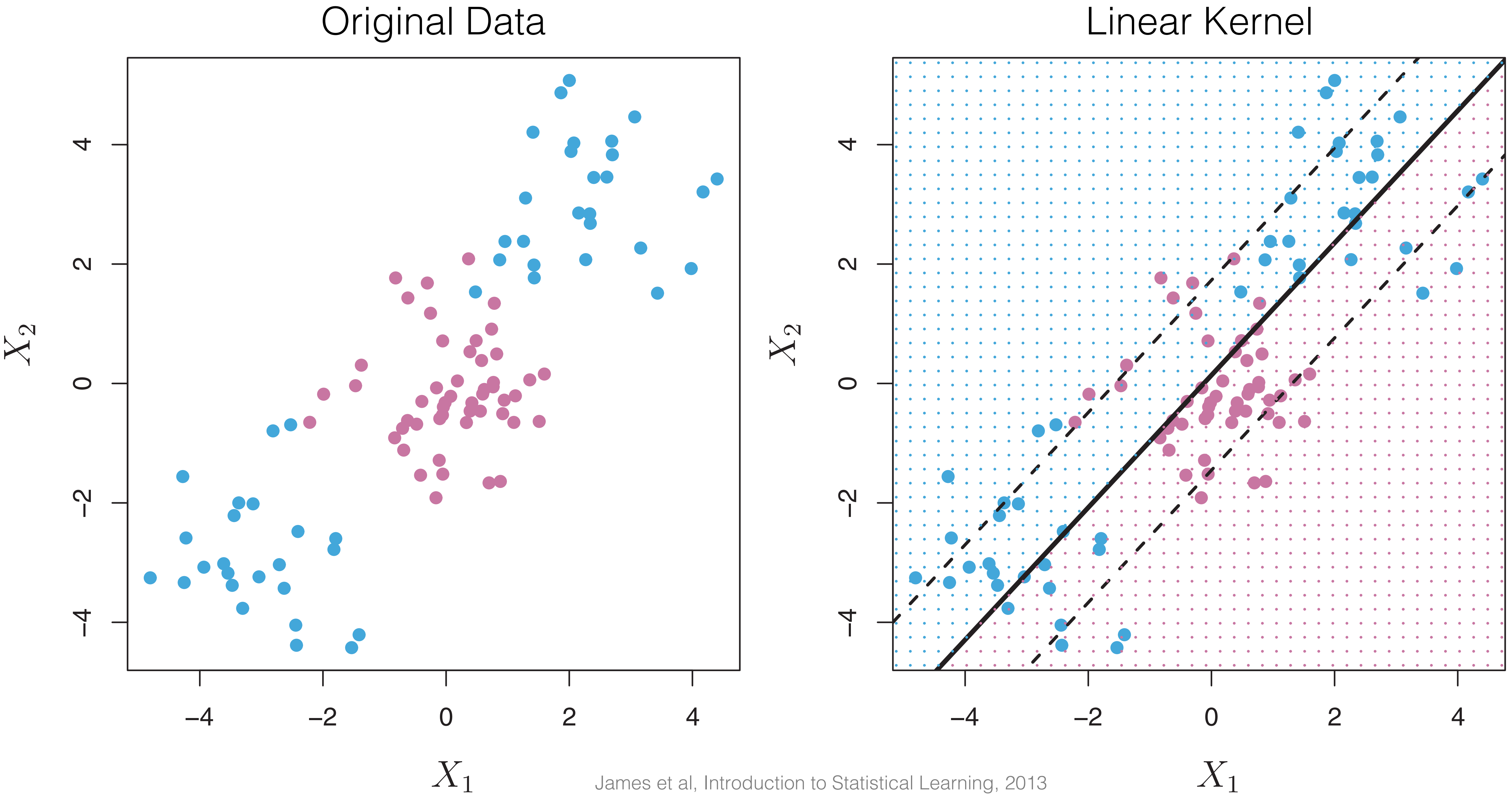
Bishop, Pattern Recognition and Machine Learning, 2006

SVM Margin Violation Penalty

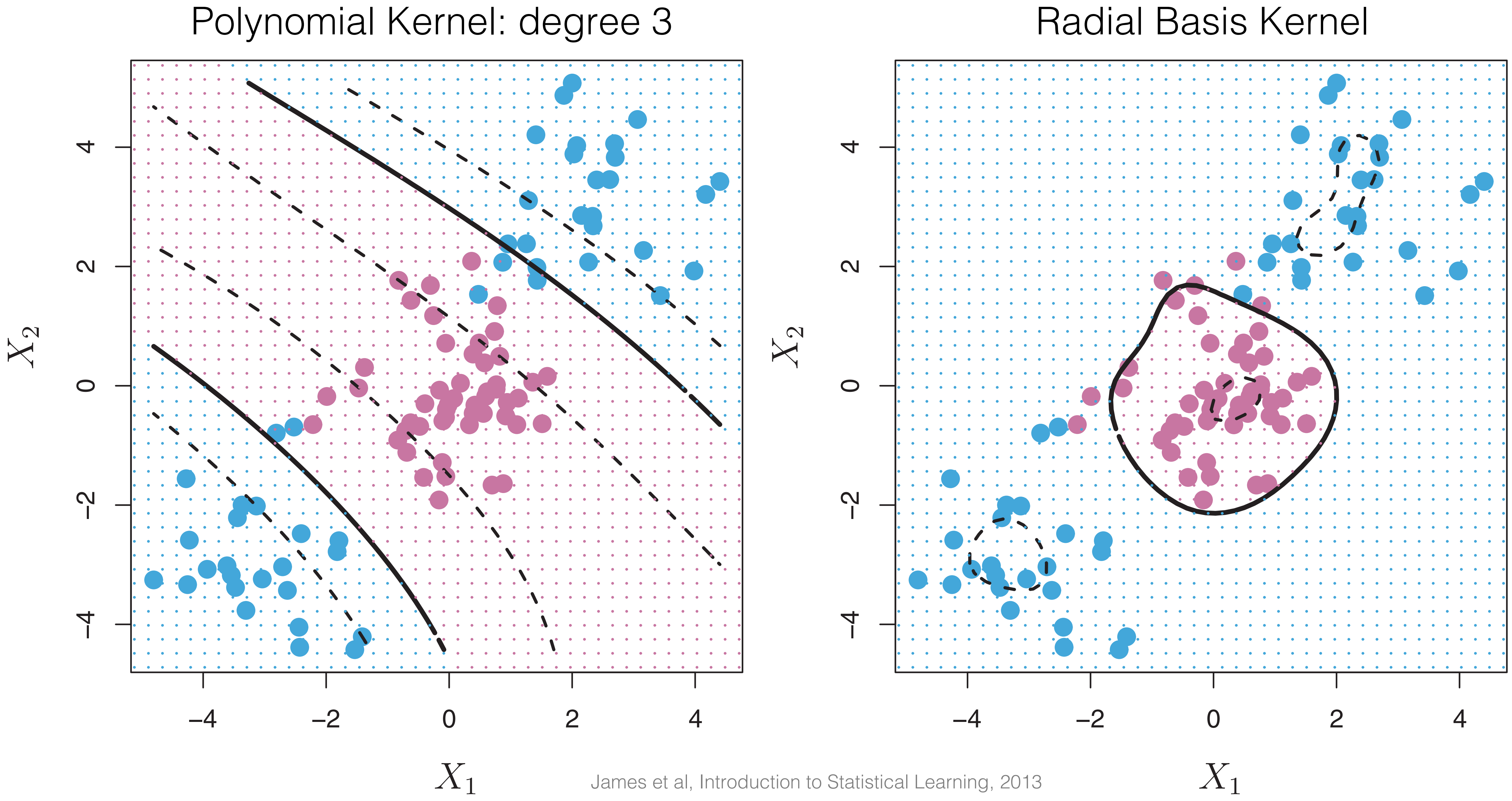
margin violation penalty
(and a regularization term)

small ⟷ large





James et al, Introduction to Statistical Learning, 2013



James et al, Introduction to Statistical Learning, 2013

SVMs can also be extended for use with regression

Relevance Vector Machines (RVMs)

- Bayesian extension of the SVM
- Produces sparser models, faster performance
- Provides probabilistic predictions

Perceptron → kernel perceptron
(the kernel trick)

Kernel functions
(making features space transforms easy)

Maximum margin classifier
(explicit feature space, linearly separable)

Support vector classifier
(explicit feature space, not linearly separable)

Support vector machine
(kernel-transformed implicit feature space, not linearly separable)

Supervised Learning Techniques

- Linear Regression
- K-Nearest Neighbors
- Perceptron
- Logistic Regression
- Fisher's Linear Discriminant
- Linear Discriminant Analysis
- Quadratic Discriminant Analysis
- Naïve Bayes
- Decision Trees and Random Forests
- Ensemble methods (bagging, boosting, stacking)
- Neural Networks
- Support Vector Machines

Appropriate for:
- Classification
- Regression

Can be used with many machine learning techniques