Neural Networks and Learning Systems
TBMI26 / 732A55
2023

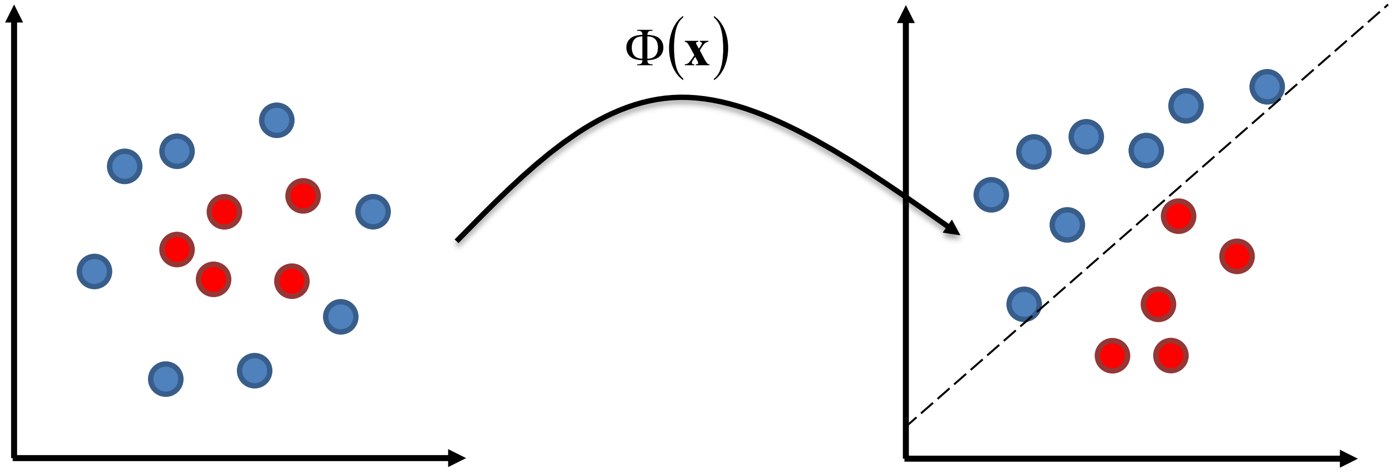
Lecture 9

Kernel methods

Magnus Borga
magnus.borga@liu.se

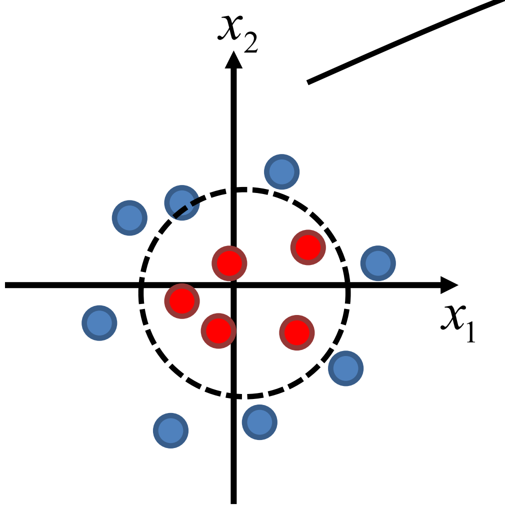
Introduction

- We have seen nonlinear mappings of input features to a new feature space:
  - Hidden layers in a neural network
  - Base classifiers in ensemble learning

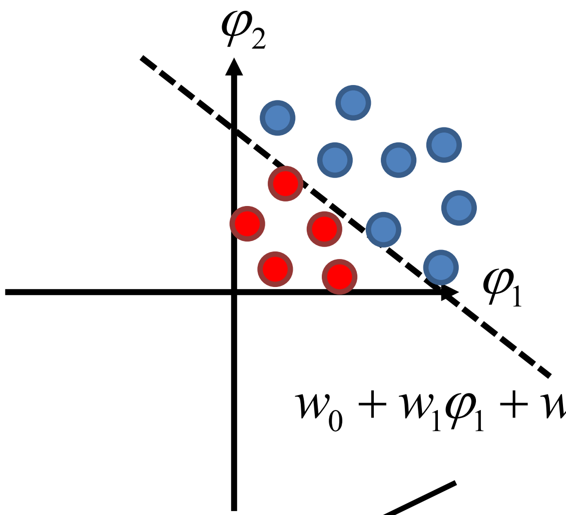


Cover's theorem: The probability that classes are linearly separable increases when the features are nonlinearly mapped to a higher dimensional feature space.
(*An extreme example: Put each sample in a dimension of its own!*)

Nonlinear mapping example



$$\Phi(\mathbf{x}) = \begin{bmatrix} \varphi_1 \\ \varphi_2 \end{bmatrix} = \begin{bmatrix} x_1^2 \\ x_2^2 \end{bmatrix}$$



$$w_0 + w_1\varphi_1 + w_2\varphi_2 = 0$$

$$w_0 + w_1 x_1^2 + w_2 x_2^2 = 0$$

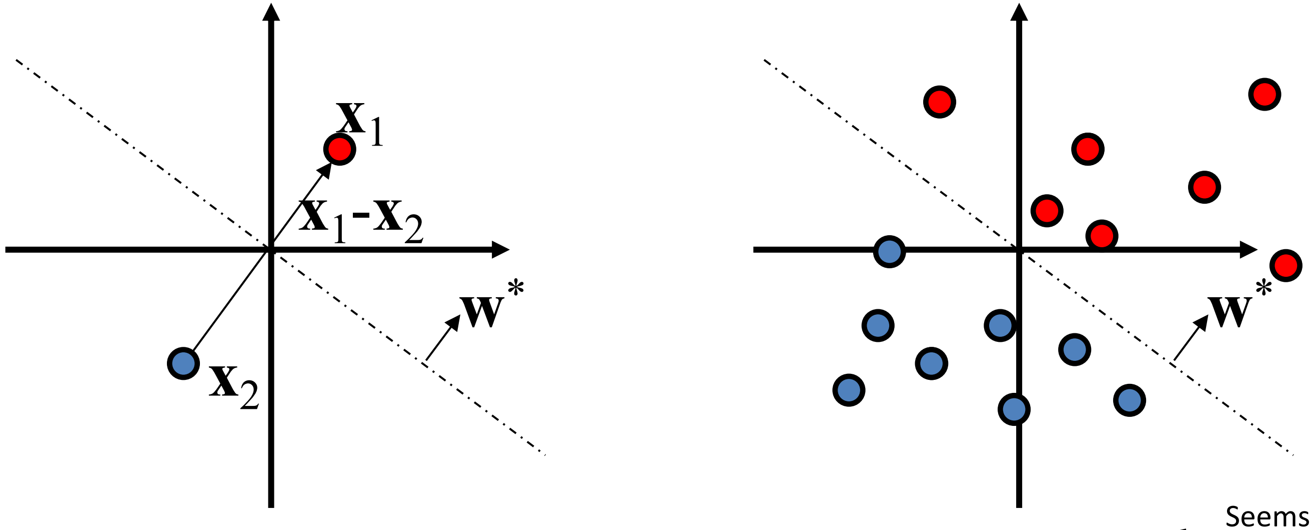
Kernel methods

A general approach to making linear methods non-linear.

The name *kernel* refers to positive definite kernels in operator theory mathematics.

Consider a linear classifier

$$f(\mathbf{x};\mathbf{w}) = \mathbf{w}^T\mathbf{x} + w_0$$



$$\mathbf{w}^* = \mathbf{x}_1 - \mathbf{x}_2$$

$$\mathbf{w}^* = \sum_{n=1}^{N} \alpha_n \mathbf{x}_n$$

Seems plausible that the optimal direction can be expressed as a linear combination of the training data!

Linear classifier in scalar product form

$$f(\mathbf{x};\mathbf{w}) = \mathbf{w}^T\mathbf{x} + w_0$$

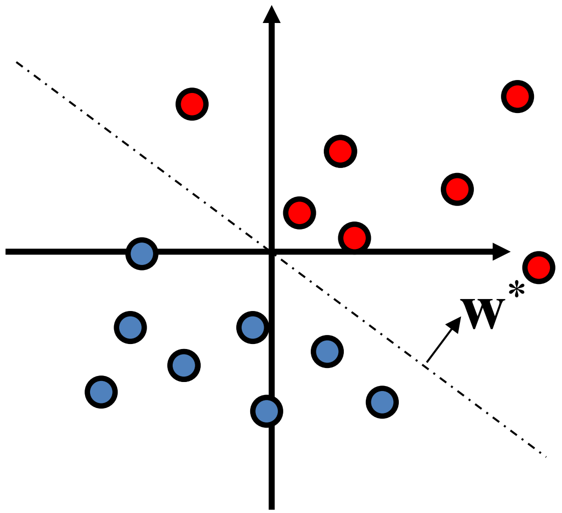
$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n \mathbf{x}_n$$

$\mathbf{w}$ is expressed as a linear combination of the training data

$$f(\mathbf{x};\boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n \mathbf{x}_n^T \mathbf{x} + \alpha_0$$

$\alpha_0 = w_0$

Scalar products between training data and the new sample



(i) $f(\mathbf{x};\mathbf{w}) = \mathbf{w}^T\mathbf{x}$

(ii) $f(\mathbf{x};\boldsymbol{\alpha}) = \sum_{n=0}^{N} \alpha_n \mathbf{x}_n^T \mathbf{x}$

For the bias weight

$$\mathbf{x} \leftarrow \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}$$

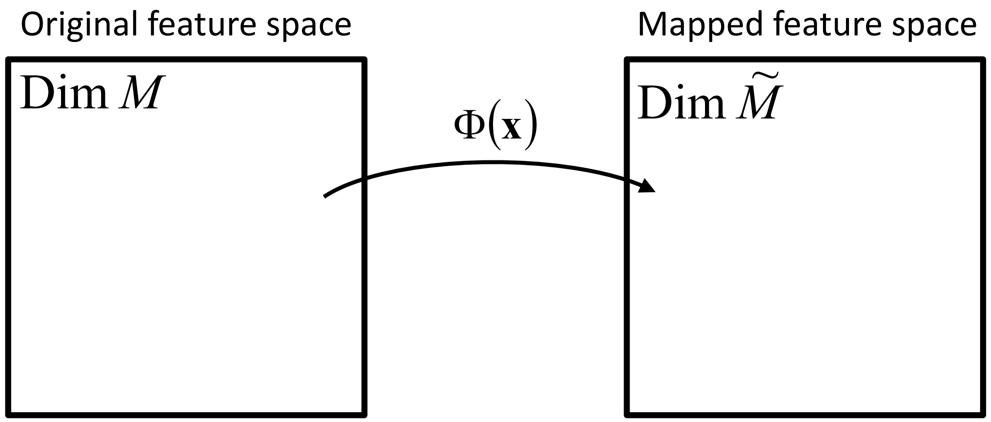
Add a dummy training example $\mathbf{x}_0 = \begin{bmatrix} 1 \\ \mathbf{0} \end{bmatrix}$ for α_0.

NOTE: Classifier form (ii) must store all training examples for the classification, whereas form (i) must not.

Why would we want to use form (ii)?

Non-linear mappings

$$\Phi(\mathbf{x}): R^{M} \rightarrow R^{\tilde{M}}, \text{ with } \tilde{\mathrm{M}} > \mathrm{M}$$



(i) $f(\mathbf{x};\mathbf{w}) = \mathbf{w}^{T}\mathbf{x}$

(ii) $f(\mathbf{x};\boldsymbol{\alpha}) = \sum_{n=0}^{N} \alpha_n \mathbf{x}_n^{T}\mathbf{x}$

(i) $f(\mathbf{x};\mathbf{w}) = \mathbf{w}^{T}\Phi(\mathbf{x})$

(ii) $f(\mathbf{x};\boldsymbol{\alpha}) = \sum_{n=0}^{N} \alpha_n \Phi(\mathbf{x}_n)^{T}\Phi(\mathbf{x})$

Example: $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

$$\Phi(\mathbf{x}) = \begin{bmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{bmatrix}$$

Explicit and implicit mapping

Classifier form (ii) offers <u>two</u> different ways of defining $\Phi(\mathbf{x})$!

$$f(\mathbf{x};\boldsymbol{\alpha}) = \sum_{n=0}^{N} \alpha_n \Phi(\mathbf{x}_n)^T \Phi(\mathbf{x})$$

Reminder: We only need the scalar product!

Explicit: Do the actual mapping, for example $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ $\quad \Phi(\mathbf{x}) = \begin{bmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{bmatrix}$

But this is only an intermediate vector that we do not really need.

Implicit: Define the new feature space by defining the scalar product in that space, i.e., how distances and angles are measured. For example:

$$\kappa(\mathbf{x}, \mathbf{z}) \triangleq \Phi(\mathbf{x})^T \Phi(\mathbf{z}) = (\mathbf{x}^T \mathbf{z})^2$$

Kernel function

Definition

Explicit and implicit mappings are equivalent

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \qquad \mathbf{z} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

$$\kappa(\mathbf{x},\mathbf{z}) = \left(\mathbf{x}^T\mathbf{z}\right)^2 = \left(x_1 z_1 + x_2 z_2\right)^2 = \left(x_1^2 z_1^2 + 2x_1 x_2 z_1 z_2 + x_2^2 z_2^2\right) = \underbrace{\begin{pmatrix} x_1^2 \\ \sqrt{2}x_1 x_2 \\ x_2^2 \end{pmatrix}^T \begin{pmatrix} z_1^2 \\ \sqrt{2}z_1 z_2 \\ z_2^2 \end{pmatrix}}_{\Phi(\mathbf{x})^T \Phi(\mathbf{z})}$$

Define!

The kernel function $\kappa(\mathbf{x},\mathbf{z}) = \left(\mathbf{x}^T\mathbf{z}\right)^2$ defines the same space as the explicit mapping $\mathbf{x} \to \Phi(\mathbf{x})$.

<u>Only in some special cases can we find the explicit mapping function from the implicit kernel function!</u>

Why not always use explicit mappings?

- Assume we have 20 input features….
- Create all polynomial combinations up to degree 5 (e.g., x_1, x_1^5, $x_2^2x_9^3$,….)

- Generates a new feature space with dimension > 50,000!

- For example, PCA in new space: Eigendecomposition of a 50,000 x 50,000 matrix.

The kernel function

$$\mathbf{x} \cdot \mathbf{z} = \mathbf{x}^T \mathbf{z}$$

<u>Needs to define a valid scalar product in some space</u>

$$\left.\begin{aligned} &\mathbf{x} \cdot \mathbf{z} = \mathbf{z} \cdot \mathbf{x} \\ &a\mathbf{x} \cdot b\mathbf{z} = ab(\mathbf{x} \cdot \mathbf{z}) \\ &\mathbf{x} \cdot (\mathbf{z}_1 + \mathbf{z}_2) = \mathbf{x} \cdot \mathbf{z}_1 + \mathbf{x} \cdot \mathbf{z}_2 \\ &\vdots \end{aligned}\right\}$$

Properties of a scalar product

Polynomial kernels

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i^T \mathbf{x}_j)^d$$

Gaussian kernel

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right)$$

Sigmoid kernel

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\mathbf{x}_i^T \mathbf{x}_j)$$

Many other kernels, see for example:
http://crsouza.com/2010/03/17/kernel-functions-for-machine-learning-applications/

Summary so far and open questions

- We assumed that the optimal solution for a linear classifier can be expressed as: $$\mathbf{w} = \sum_{n=1}^{N} \alpha_n \mathbf{x}_n$$ This must be verified!
- The linear classifier can then be expressed as:

$$f(\mathbf{x};\boldsymbol{\alpha}) = \sum_{n=0}^{N} \alpha_n \mathbf{x}_n^T \mathbf{x}$$ How do we find the α's?

- Apply the linear classifier in a higher-dimensional space by defining its scalar product via the kernel function

$$\kappa(\mathbf{x},\mathbf{z}) = \Phi(\mathbf{x})^T \Phi(\mathbf{z})$$

$$f(\mathbf{x};\boldsymbol{\alpha}) = \sum_{n=0}^{N} \alpha_n \kappa(\mathbf{x}_n, \mathbf{x})$$ How do we select the kernel function?

Example: Linear perceptron with square error loss

From lecture 2!

Minimize the following loss function

$$\varepsilon\left(\mathbf{w}\right) = \sum_{i=1}^{N}\left(\mathbf{w}^T\mathbf{x}_i - y_i\right)^2$$

N = # training samples

$y_i \in \{-1,1\}$ depending on the class of training sample i

Example: Linear perceptron algorithm

From lecture 2!

$$\varepsilon(\mathbf{w}) = \sum_{i=1}^{N} \left(\mathbf{w}^T \mathbf{x}_i - y_i\right)^2$$

$$\frac{\partial \varepsilon}{\partial \mathbf{w}} = 2 \sum_{i=1}^{N} \left(\mathbf{w}^T \mathbf{x}_i - y_i\right) \mathbf{x}_i$$

Gradient descent:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\partial \varepsilon}{\partial \mathbf{w}} = \mathbf{w}_t - \eta \overbrace{\sum_{i=1}^{N} \left(\mathbf{w}_t{}^T \mathbf{x}_i - y_i\right) \mathbf{x}_i}^{\text{Weighted sum of } \mathbf{x}_i\text{ !}} \quad (\mathrm{Eq.}\,1)$$

Algorithm:

1. Start with a small random **w**
2. Iterate Eq. 1 until convergence

$$\mathbf{w}^* = \sum_{i=1}^{N} \alpha_i \mathbf{x}_i \text{ as } t \rightarrow \infty$$

Example: Kernel perceptron algorithm

Gradient descent:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i=1}^{N} \left(\mathbf{w}_t{}^T \mathbf{x}_i - y_i \right) \mathbf{x}_i$$ Original space

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i=1}^{N} \underbrace{\left(\mathbf{w}_t^T \Phi(\mathbf{x}_i) - y_i \right)}_{\beta_{t,i}} \Phi(\mathbf{x}_i)$$ Mapped space

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i=1}^{N} \beta_{t,i} \Phi(\mathbf{x}_i)$$

$$\boxed{\mathbf{w}^* = \sum_{i=1}^{N} \alpha_i \Phi(\mathbf{x}_i) \, \text{as} \, t \to \infty}$$

Example: Kernel perceptron algorithm

$$\varepsilon(\mathbf{w}) = \sum_{i=1}^{N} \left(y_i - \mathbf{w}^T \Phi(\mathbf{x}_i)\right)^2$$

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i \Phi(\mathbf{x}_i)$$

$$\varepsilon(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \left(y_i - \sum_{j=1}^{N} \alpha_j \underline{\Phi(\mathbf{x}_j)^T \Phi(\mathbf{x}_i)} \right)^2 = \sum_{i=1}^{N} \left(y_i - \sum_{j=1}^{N} \alpha_j \underline{\kappa(\mathbf{x}_j, \mathbf{x}_i)} \right)^2$$

Kernel trick!

Gradient:

$$\frac{\partial \varepsilon}{\partial \alpha_k} = -2 \sum_{i=1}^{N} \left(y_i - \sum_{j=1}^{N} \alpha_j \kappa(\mathbf{x}_j, \mathbf{x}_i) \right) \kappa(\mathbf{x}_k, \mathbf{x}_i)$$

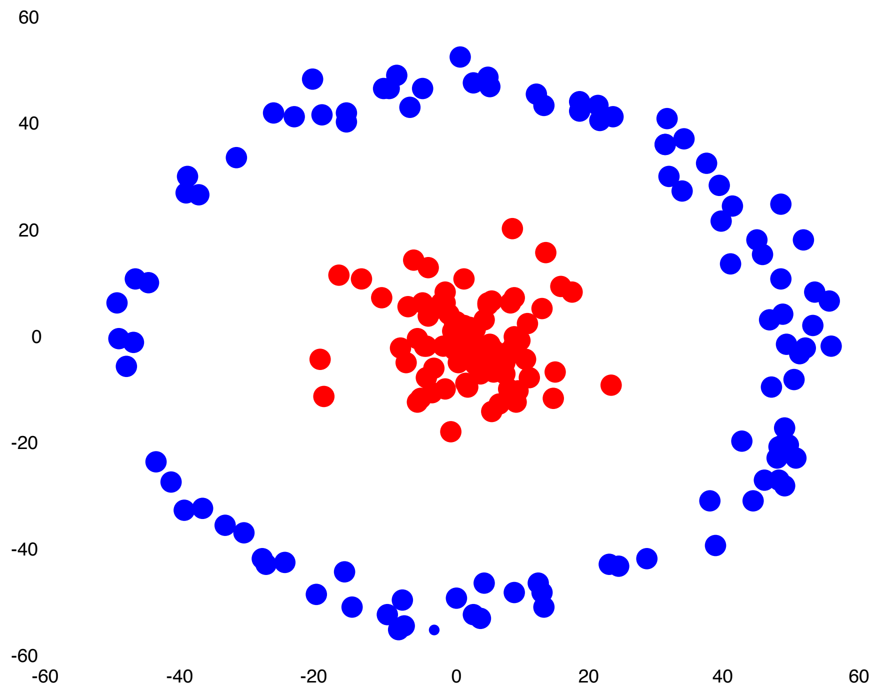
Gradient descent in α!

$$\alpha_{k,t+1} = \alpha_{k,t} - \eta \frac{\partial \varepsilon}{\partial \alpha_k}$$

Example: Kernel perceptron summary

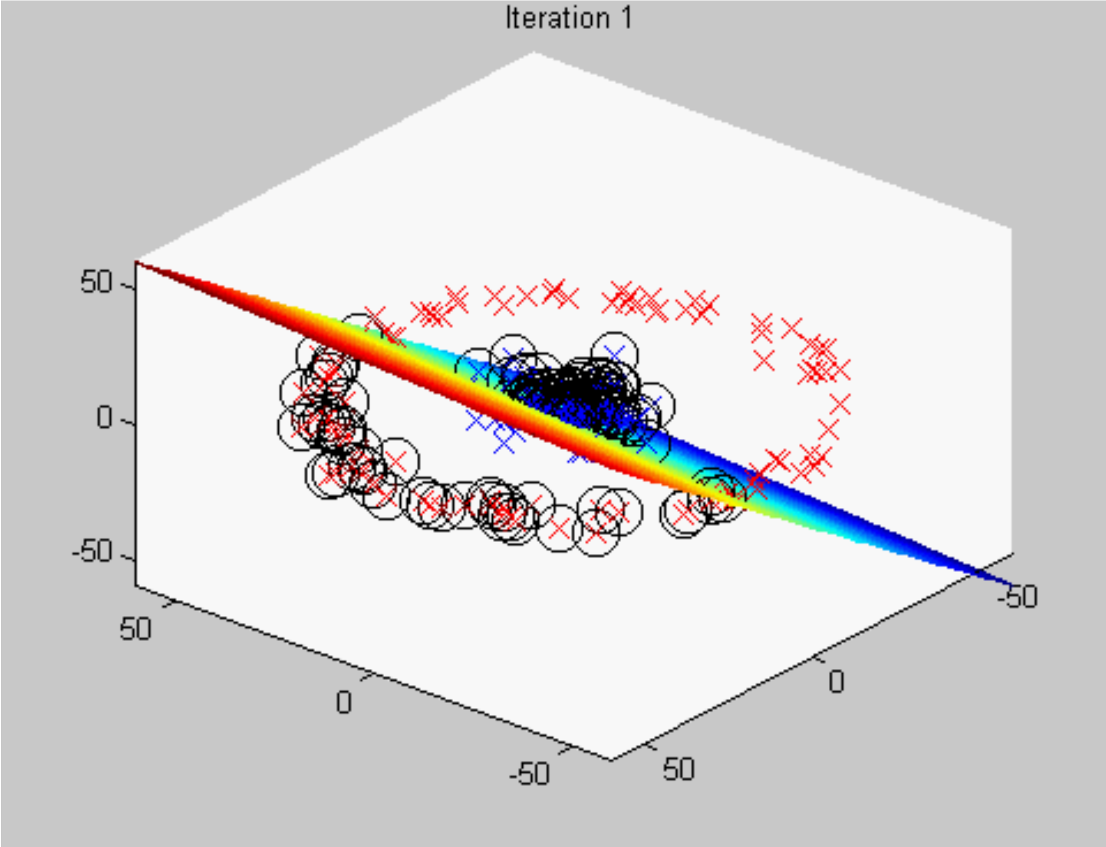
1. Showed that $\mathbf{w}^* = \sum_{i=1}^{N} \alpha_i \Phi(\mathbf{x}_i)$
2. Loss function in α: $\varepsilon(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \left(y_i - \sum_{j=1}^{N} \alpha_j \Phi(\mathbf{x}_j)^T \Phi(\mathbf{x}_i) \right)^2$
3. Choose kernel function: $\kappa(\mathbf{x}_j, \mathbf{x}_i) = \Phi(\mathbf{x}_j)^T \Phi(\mathbf{x}_i)$
4. Gradient descent in α: $\alpha_{k,t+1} = \alpha_{k,t} - \eta \frac{\partial \varepsilon}{\partial \alpha_k}$
5. Apply classifier: $f(\mathbf{x}; \boldsymbol{\alpha}) = \sum_{i=0}^{N} \alpha_i \kappa(\mathbf{x}_i, \mathbf{x})$

Kernel Perceptron example



Kernel Perceptron example, cont

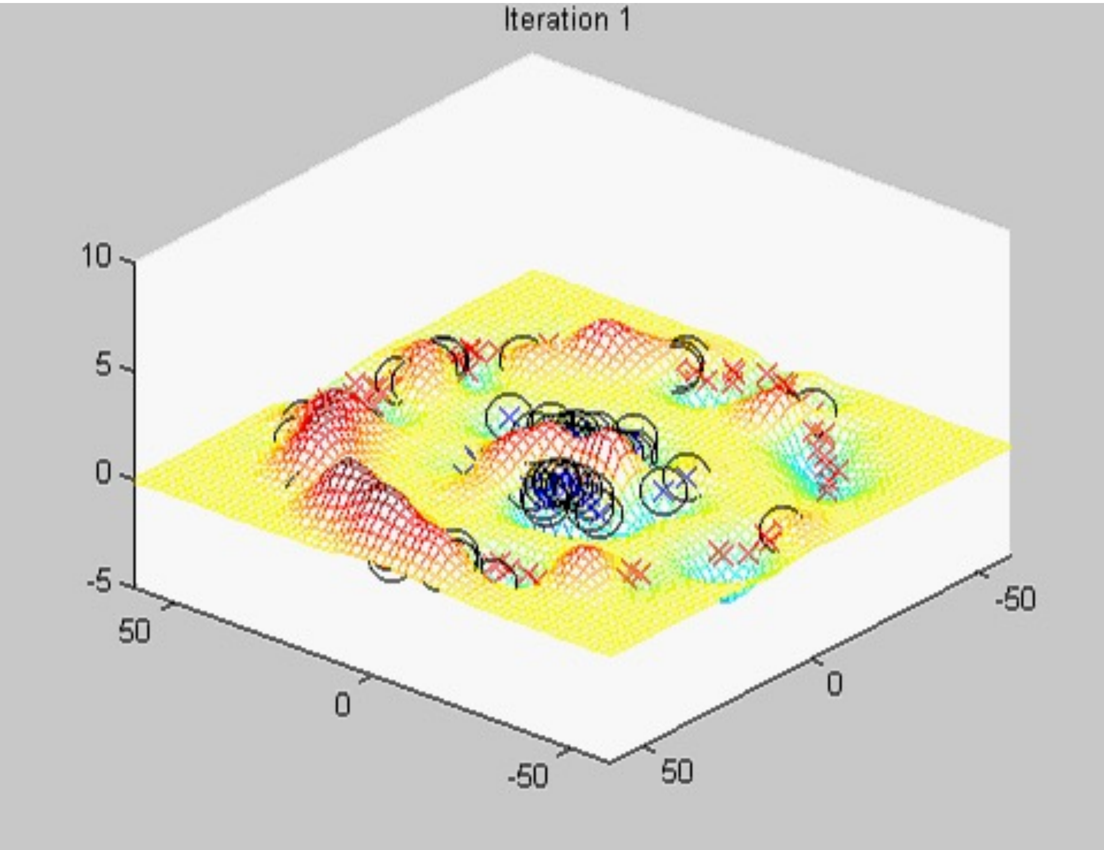
Movie!



The original linear perceptron algorithm will not work
because the classes are not linearly separable

Kernel Perceptron example, cont

Movie!



The surface shows the value of

$$\sum_{k=1}^{N} \alpha_k \kappa(\mathbf{x}_k, \mathbf{x})$$

for different **x**.

Gaussian kernel with σ=10

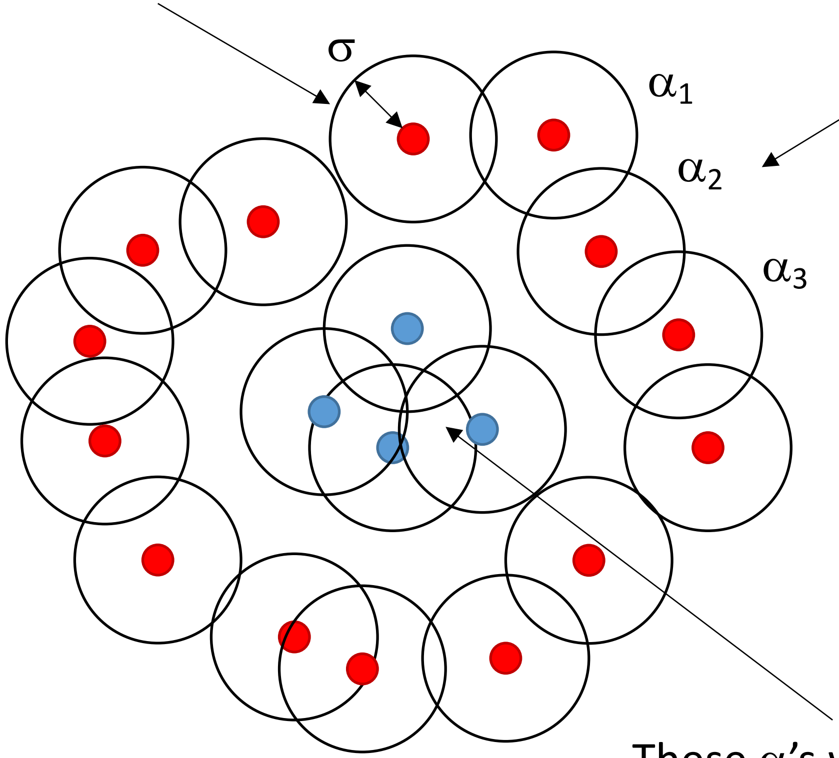
$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2} \right)$$

Structure of the classification function

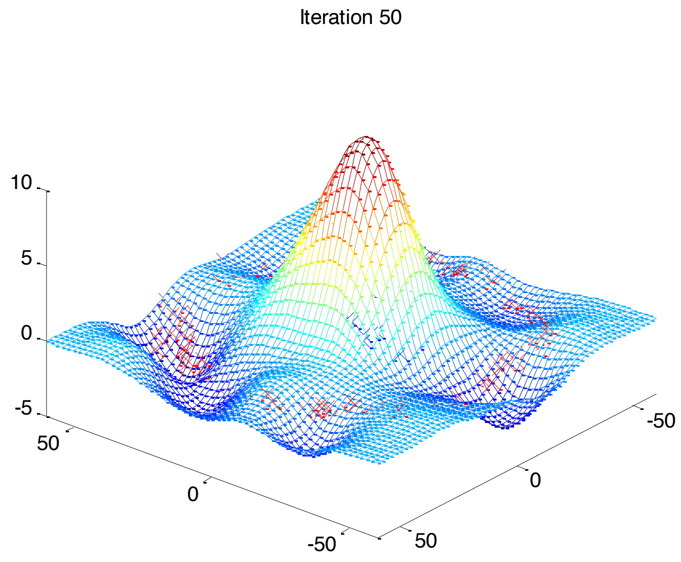
$$f(\mathbf{x}) = \sum_{k=1}^{N} \alpha_k \kappa(\mathbf{x}_k, \mathbf{x})$$

Weighted distance to all training samples

Gaussian kernel



These α's will probably be negative



These α's will probably be positive

Kernelization of linear methods

- Perceptron
- LDA
- SVM
- PCA
- k-means

Linear classifiers

Linear structure discovery methods

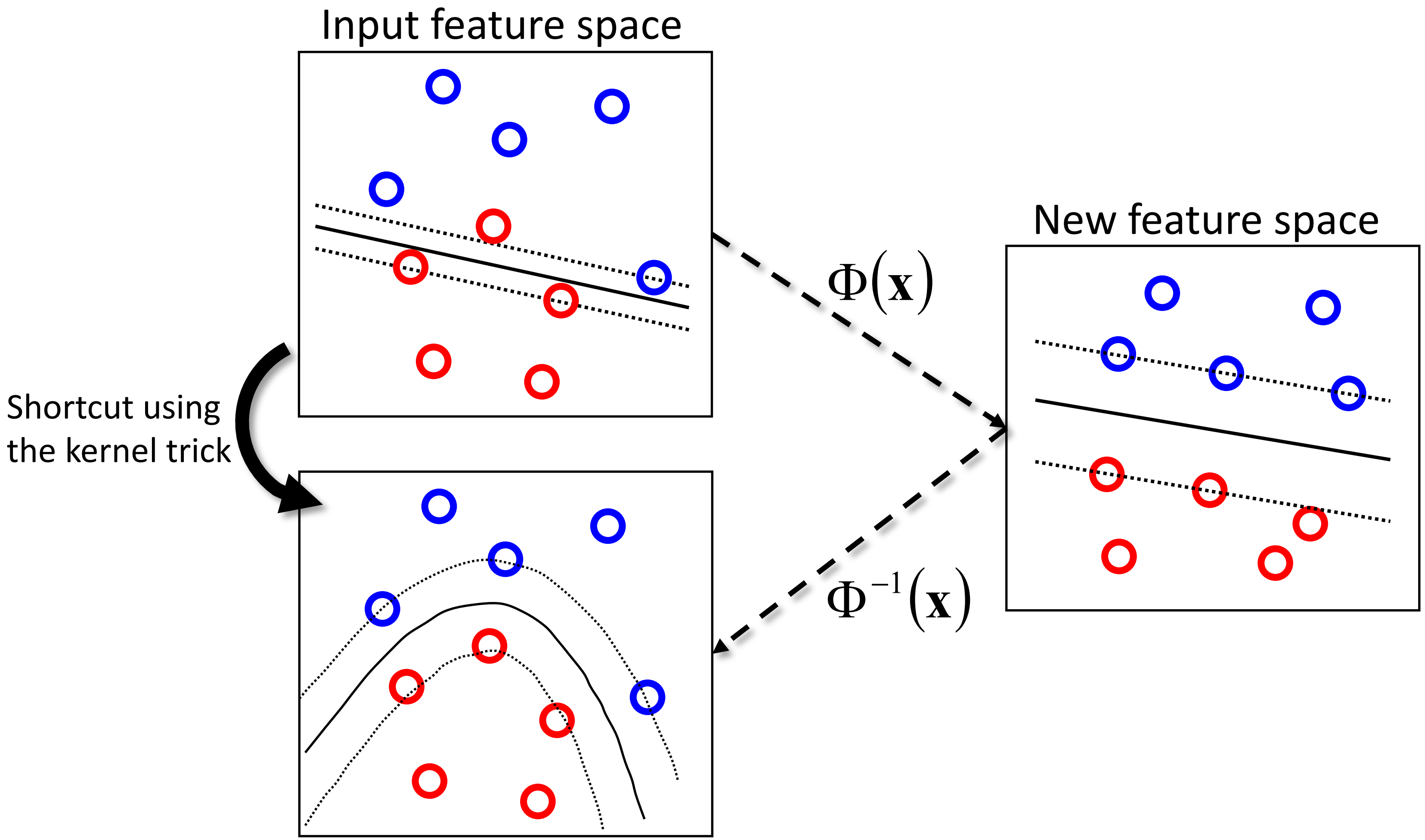
Clustering

Number of parameters equals the number of training samples

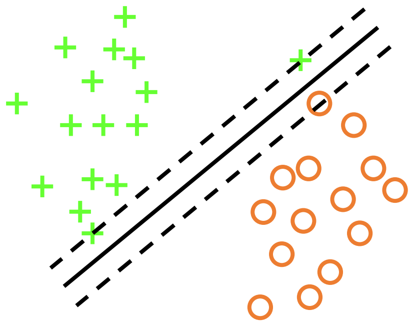
$$f(\mathbf{x}) = \sum_{k=1}^{N} \alpha_k \kappa(\mathbf{x}_k, \mathbf{x})$$

Have to store all training samples

Nonlinear SVM



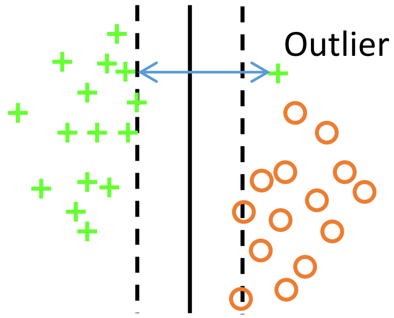
Kernelizing the linear SVM



$$\min_{\mathbf{w}} \mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{N}\xi_i$$

$$\text{subject to } y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1 - \xi_i$$

Assume again that $\mathbf{w} = \sum_{j=1}^{N}\alpha_j\mathbf{x}_j$



$$\min_{\boldsymbol{\alpha}} \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j\underline{\mathbf{x}_i^T\mathbf{x}_j} + C\xi_i$$

$$\text{subject to } y_i\left(\sum_{j=1}^{N}\alpha_j\underline{\mathbf{x}_i^T\mathbf{x}_j} + \alpha_0\right) \geq 1 - \xi_i$$

Nonlinear SVM

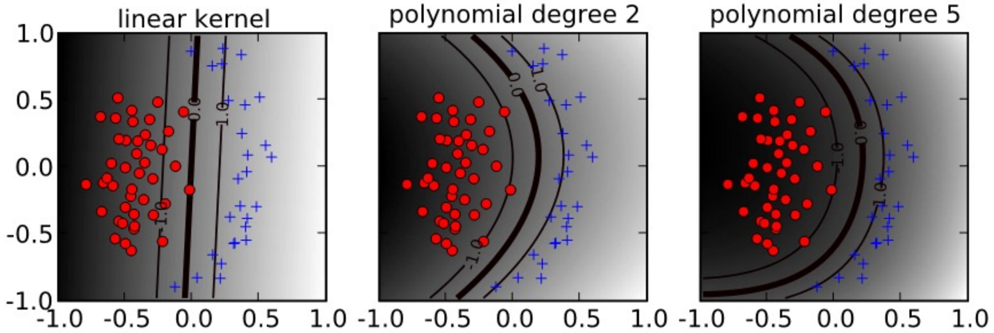
$$\min_{\boldsymbol{\alpha}} \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j \kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) + C\xi_i$$

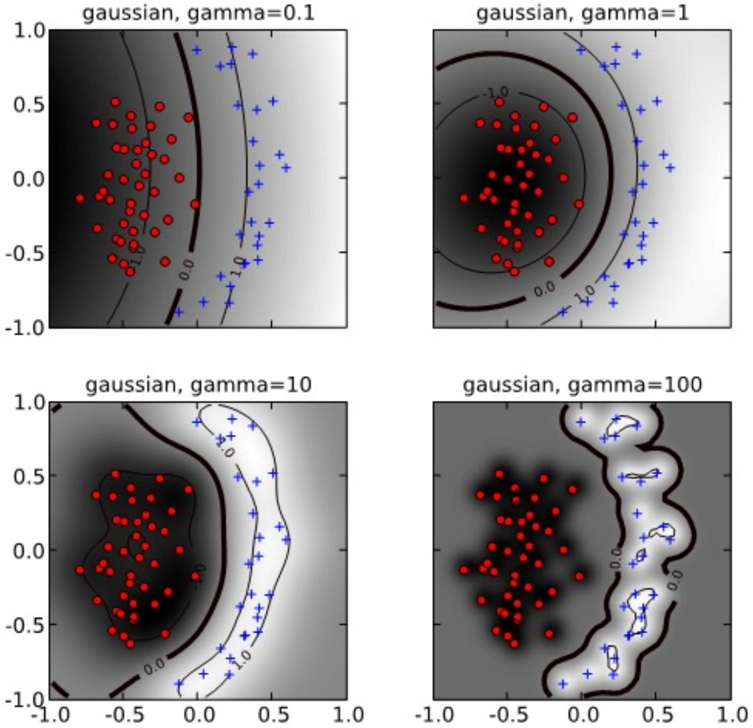
$$\text{subject to } y_i \left(\sum_{j=1}^{N} \alpha_n \kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) + \alpha_0\right) \geq 1 - \xi_i$$

C: Trade-off parameter between the importance of a low error on the training data vs. finding wide margins that may give better generalization on test data.

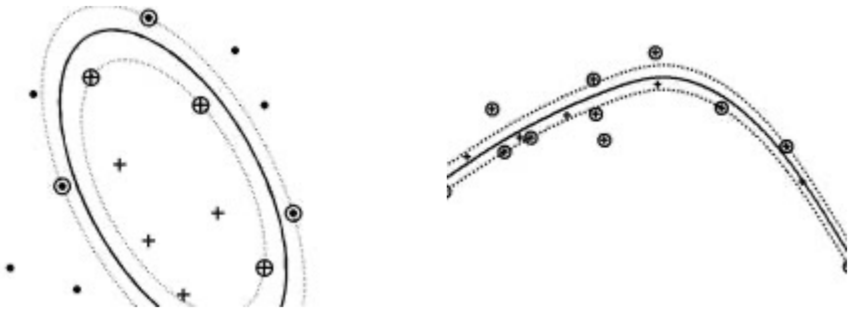
κ(.,.): Kernel function that determines the non-linear mapping. May contain additional parameters such as the width of a Gaussian kernel.

Nonlinear SVM - Examples





Source: A. Ben-Hur & J. Weston
A User's Guide to Support Vector Machines



Source: http://www.support-vector-machines.org/

Nonlinear SVM - Summary

- Brings two clever and independent concepts together:
  - Large margin principle for good generalization
  - Kernel trick for making linear methods nonlinear
- Loss function ”landscape” less complex than in, e.g., neural network training.
- Must store the support vectors, which can be many.
- Classification slower than, for example, boosting.

$$f(\mathbf{x}) = \sum_{k=1}^{N} \alpha_k \kappa(\mathbf{x}_k, \mathbf{x})$$

Kernel PCA

- Non-linear version of PCA.
- PCA can be written in terms of scalar products.
- Use the ”kernel trick”.

Kernel-PCA

$\mathbf{XX}^T\mathbf{e} = \lambda\mathbf{e}$ Ordinary PCA

Multiply from left with $\mathbf{X}^T$:

$$\mathbf{X}^T\mathbf{X}\underbrace{\mathbf{X}^T\mathbf{e}}_{\mathbf{f}} = \lambda\underbrace{\mathbf{X}^T\mathbf{e}}_{\mathbf{f}} \qquad \rightarrow \mathbf{X}^T\mathbf{X}\mathbf{f} = \lambda\mathbf{f}$$

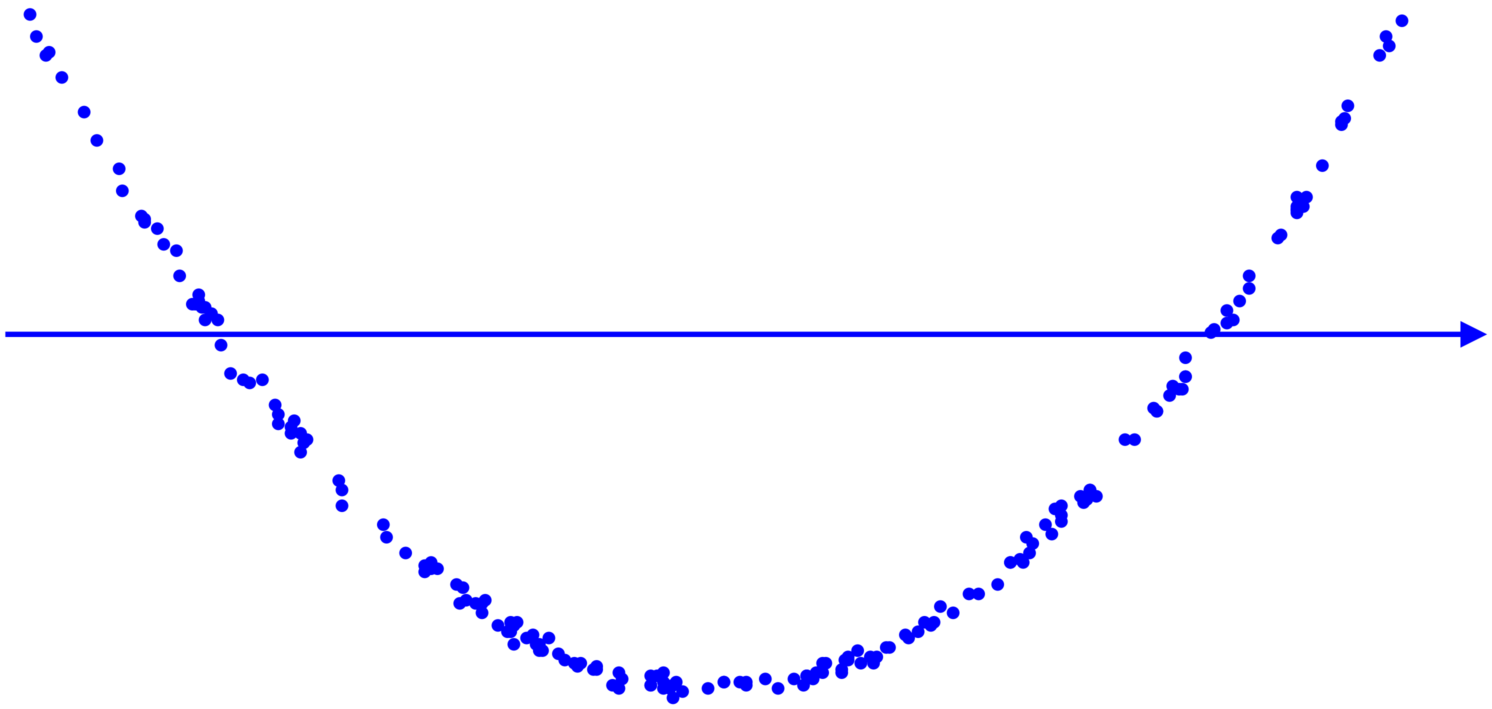
Eigen value problem on an inner product matrix
i.e. with coeficients defined by scalar products!

Kernel-PCA

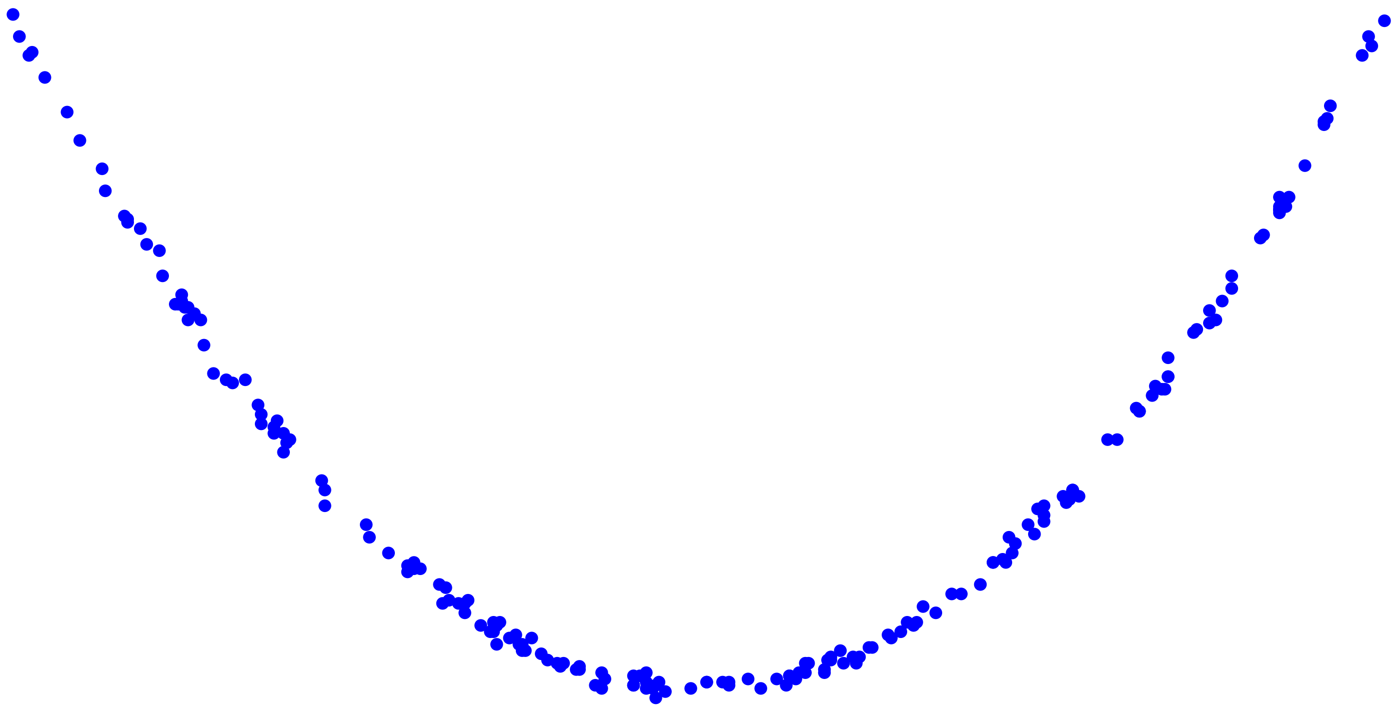
- Similarly, PCA can be performed on any kernel matrix $\mathbf{K}$ whose components k_{ij} are defined by a kernel function $k_{ij} = \boldsymbol{\varphi}(\mathbf{x}_i)^T \boldsymbol{\varphi}(\mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j)$

- The principal components are linear in the feature space but non-linear in the input space.

Linear PCA



KPCA with quadratic kernel



Kernels – Pros and cons

- Well understood linear methods carried out in a high-dimensional space where linear separability is more likely.
- Can achieve good performance

- How to choose the kernel and the kernel parameters?
- Have to store the training data.
- Need all combinations of training samples:
  (# samples)^2
- Training and classification can be computationally intensive