Kernel Methods

- Radial Basis Function (RBF) Networks
- Support Vector Machines (SVM)
- Kernel PCA
- Kernal k-Means

Revisiting Class Separability

A 2-class problem is linearly separable if a decision hyperplane (defined by $\boldsymbol{w}^*$) exists such that

$$(\boldsymbol{w}^*)^T \boldsymbol{x} > 0 \quad \forall \boldsymbol{x} \in \omega_1$$

$$(\boldsymbol{w}^*)^T \boldsymbol{x} < 0 \quad \forall \boldsymbol{x} \in \omega_2$$

Now, in a more general case, let us consider a mapping from the original input space to a space of a different dimension:

$$\varphi : \mathbb{R}^{m_0} \rightarrow \mathbb{R}^{m_1} \qquad \text{where} \qquad \varphi(\boldsymbol{x}) = \left[\varphi_1(\boldsymbol{x}) \;\; \varphi_2(\boldsymbol{x}) \ldots \varphi_{m_1}(\boldsymbol{x})\right]^T$$

Then we can define a φ-separability: A 2-class problem is φ-linearly separable if if there exists a $\boldsymbol{w}^*$ such that

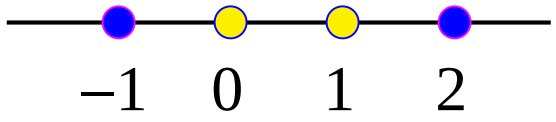
$$(\boldsymbol{w}^*)^T \varphi(\boldsymbol{x}) > 0 \quad \forall \boldsymbol{x} \in \omega_1$$

$$(\boldsymbol{w}^*)^T \varphi(\boldsymbol{x}) < 0 \quad \forall \boldsymbol{x} \in \omega_2$$

Cover's Theorem

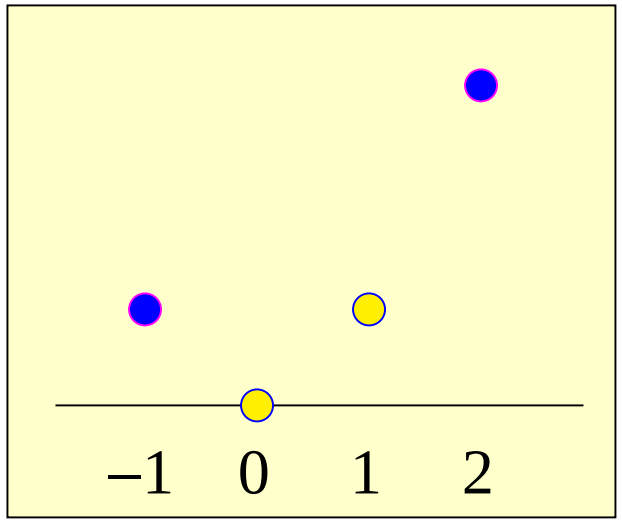
A complex pattern recognition problem cast in a high-dimensional space nonlinearly is more likely to be linearly separable than in a low-dimensional space. (1965)

Example: Non-separable 1-D data:



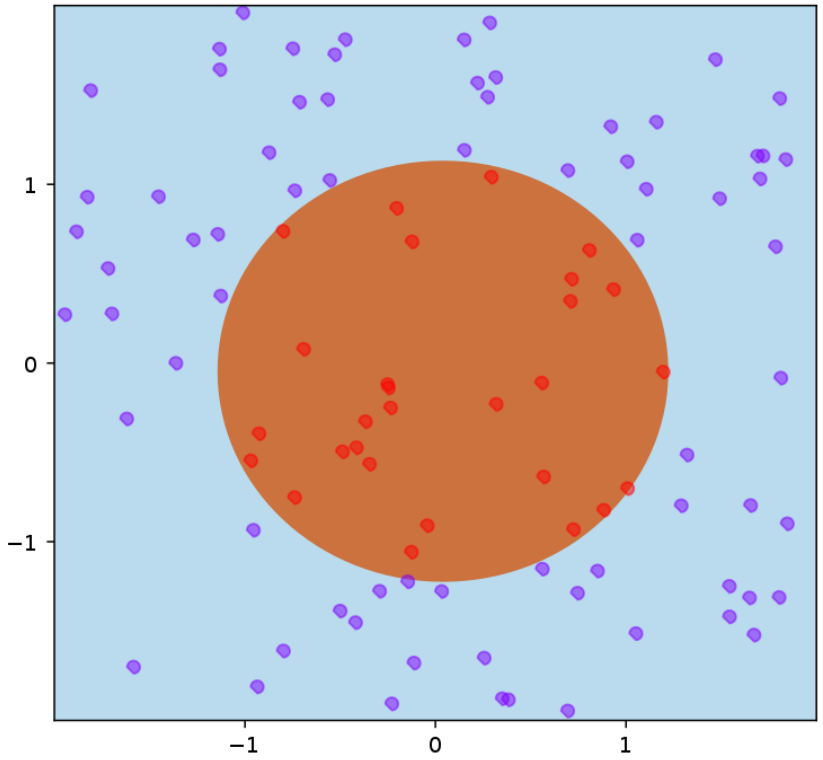
Mapped to 2-D by $\varphi(x) = [x \;\; x^2]^T$:

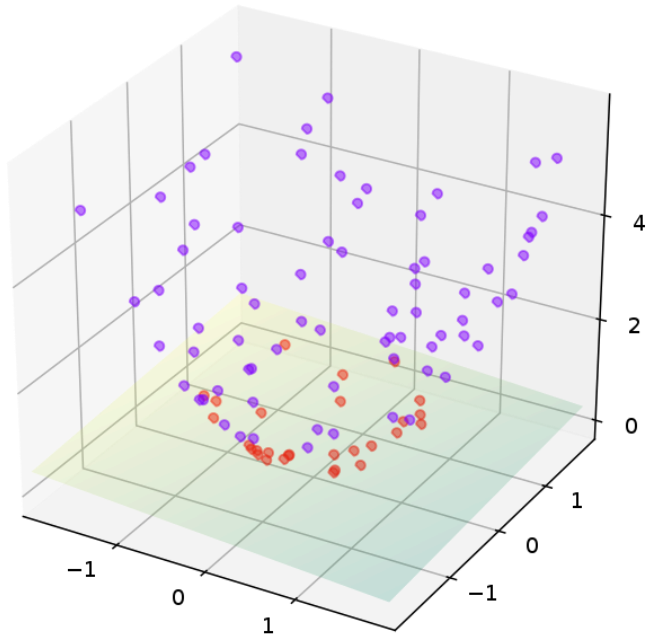
Now linearly separable



Cover's Theorem

An example 2-D to 3-D mapping to make the samples of two classes linearly separable:





https://en.wikipedia.org/wiki/Cover%27s_theorem

The Radial Basis Functions (RBF)

- Each radial basis function $\varphi(\boldsymbol{x})$ has a center $\boldsymbol{c}$.
- $\varphi(\boldsymbol{x})$ is determined by $\|\boldsymbol{x}-\boldsymbol{c}\|$, i.e., the distance between $\boldsymbol{x}$ and $\boldsymbol{c}$. (Euclidean distance is common.)
- A few forms of $\varphi(\boldsymbol{x})$:
  - Gaussian: $$\varphi(\boldsymbol{x}) = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{c}\|^2}{2\sigma^2}\right)$$
  - Multiquadrics: $$\varphi(\boldsymbol{x}) = \left(\|\boldsymbol{x}-\boldsymbol{c}\|^2 + a^2\right)^{1/2}$$
  - Inverse multiquadrics: $$\varphi(\boldsymbol{x}) = \left(\|\boldsymbol{x}-\boldsymbol{c}\|^2 + a^2\right)^{-1/2}$$

RBF Networks

Radial Basis Function (RBF) networks are neural networks where the activation functions of hidden neurons are radial basis functions.

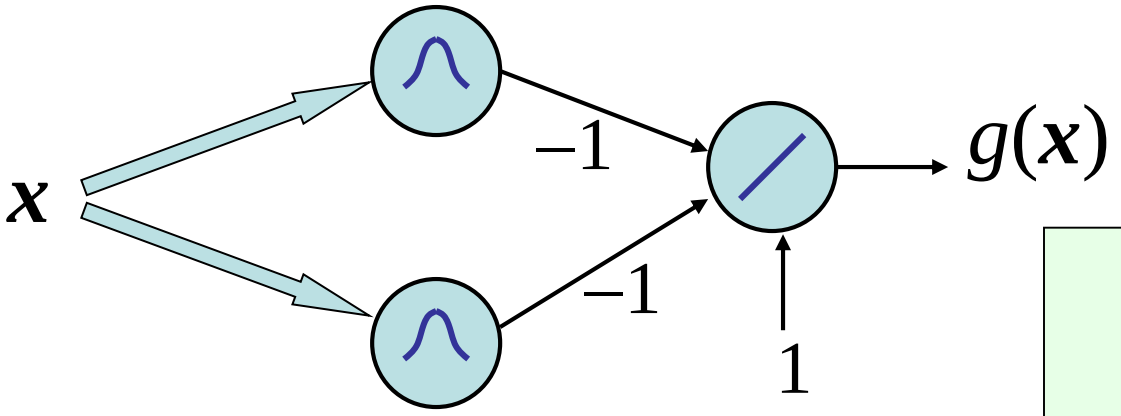
- A typical RBF network has one hidden layer.
- A typical RBF network uses linear activation for output neurons. Therefore, the output of a RBF network has the form of

$$F(\boldsymbol{x}) = \sum_{i=1}^{m_1} w_i \, \varphi(\| \boldsymbol{x} - \boldsymbol{c}_i \|)$$

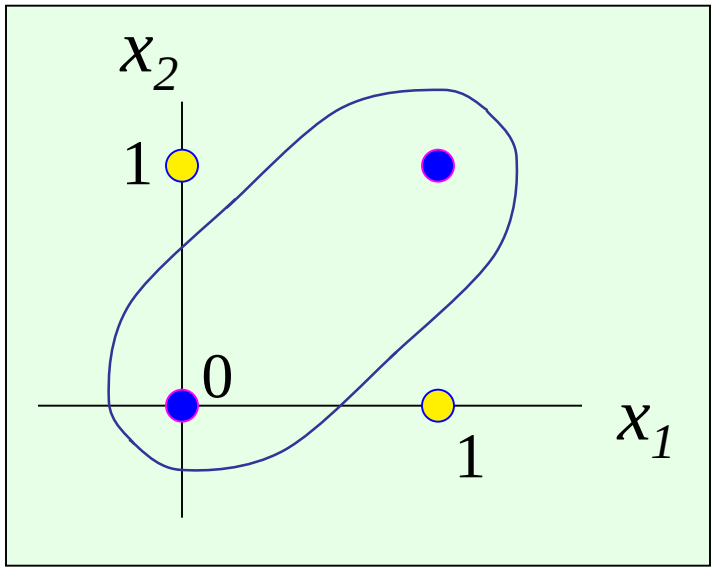
- RBF networks are universal approximators. (This can be easily understood by considering the case when we have one hidden neuron for each $\boldsymbol{x}_i$.
- The hidden neurons perform the mapping, as described for the Cover's theorem.

Revisiting the XOR Problem

Let us consider the following RBF network with two RBF units centered at $(0,0)$ and $(1,1)$ and $\sigma^2 = 0.5$.



The decision boundary ($g(\boldsymbol{x})=0$):



The final output is set to 1 for $g(\boldsymbol{x})>0$ and 0 otherwise, to be consistent with Boolean logic.

Training RBF Networks

Compared with MLPs, in addition to the weights, there are two more types of parameters that can be trained in RBF networks: the centers $\boldsymbol{c}_i$ and the standard deviations σ_i. This makes learning more complicated. The options are:

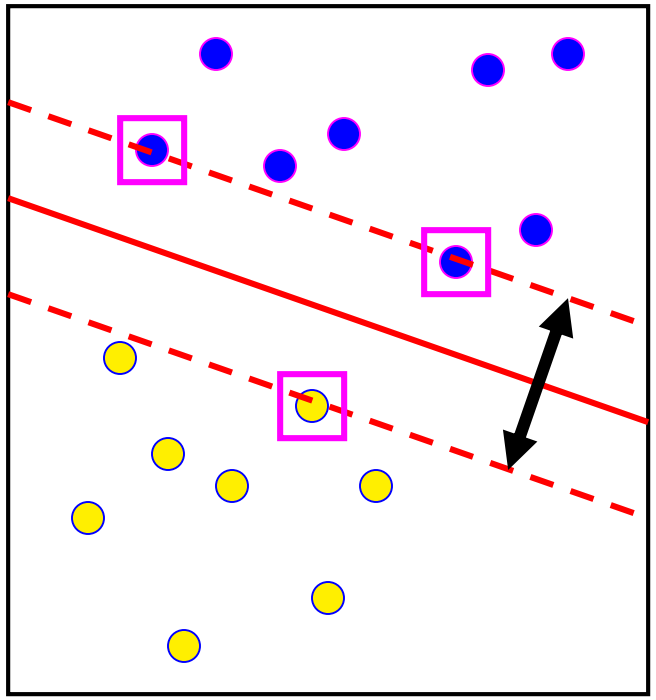
- Traing all $\boldsymbol{w}$, $\boldsymbol{c}_i$ and σ_i together using gradient descent. This is computationally more expensive.
- Two-stage training: Derive the RBF parameters ($\boldsymbol{c}_i$ and σ_i) first, and then the weights.

Support Vector Machines (SVM)

Idea (SVM as a linear classifier):
Choose the hyperplane as far as possible from any x

$\Rightarrow$ New samples near those x in the training set are more likely to be classified correctly.

$\Rightarrow$ Good generalization ability.



For linearly separable cases:

The ***margin*** is twice the distance from the hyperplane to the closest sample. The ***support vectors*** are the samples that are closest to the hyperplane.

Margin in SVM

The linear discriminant function is $g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0$

The distance between $\boldsymbol{x}$ and the decision boundary ($g(\boldsymbol{x})=0$) is given by $\dfrac{\left|\boldsymbol{w}^T \boldsymbol{x} + w_0\right|}{\|\boldsymbol{w}\|}$

To make the values of $\boldsymbol{w}$ and w_0 unique, we add the requirement: $\left|\boldsymbol{w}^T \boldsymbol{x} + w_0\right| = 1$ if $\boldsymbol{x}$ is a support vector.

The margin is now given by $2/\|\boldsymbol{w}\|$. Maximizing the margin is the same as minimizing $\|\boldsymbol{w}\|$.

The Optimization Problem

We can phrase the task of finding the decision boundary in SVM as an optimization problem:

To minimize $J = \frac{1}{2} \| \boldsymbol{w} \|^2$

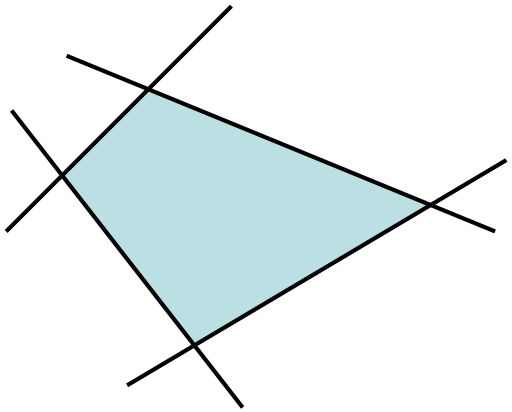
Subject to the constraints $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) \geq 1, \ \forall i$

Here $y_i = +1 \quad \forall \boldsymbol{x}_i \in \omega_1$

$y_i = -1 \quad \forall \boldsymbol{x}_i \in \omega_2$

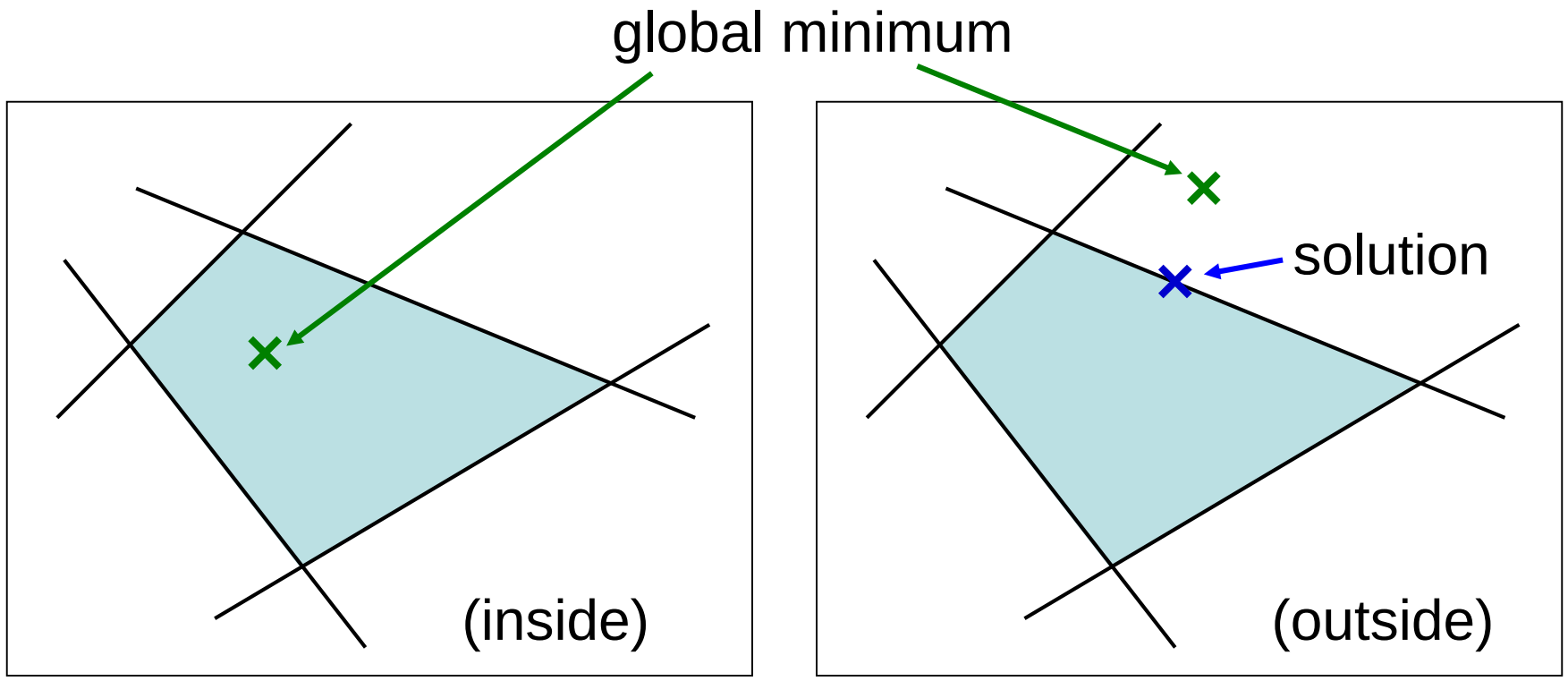
This is a primal problem, with a convex cost function (which has a single global minimum) and linear constraints. (Each training sample leads to a constraint.)

The (inequality) constraints define a *feasible region* of the solution (in the space of the weights/bias). Example:



The Optimization Problem

The global minimum may fall inside or outside of the feasible region. (The constrained minimum is unique.)



There are active and inactive constraints.

The Optimization Problem: Example

Consider the 1-D 2-class example:

x_1=1, x_2=2 in ω_1, and x_3= –1, x_4= –2 in ω_2.

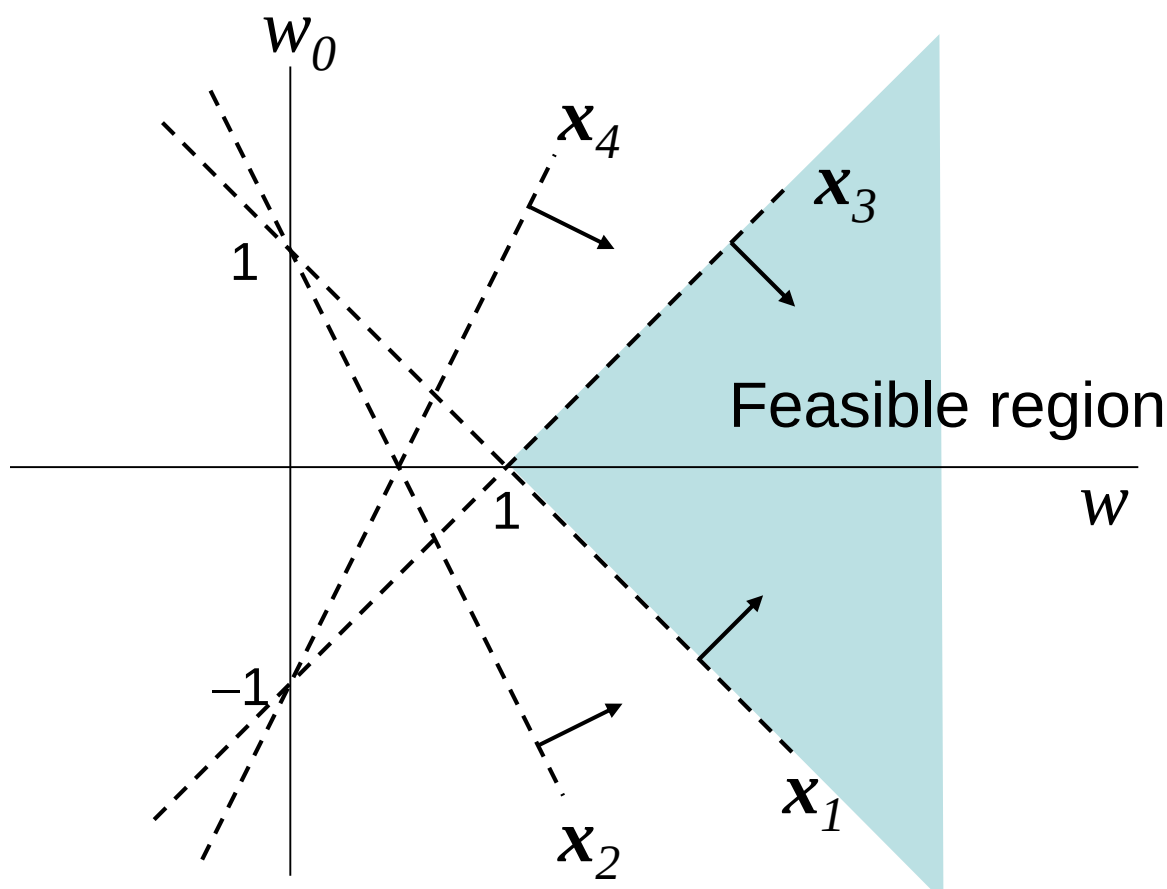
The constraints:

$\boldsymbol{x}_1$: $(w + w_0) \geq 1$

$\boldsymbol{x}_2$: $(2w + w_0) \geq 1$

$\boldsymbol{x}_3$: $-(-w + w_0) \geq 1$

$\boldsymbol{x}_4$: $-(-2w + w_0) \geq 1$



Finding the Decision Boundary

Using Lagrange multipliers to include the constraints:

$$L = \frac{1}{2} \| \boldsymbol{w} \|^2 - \sum_{i=1}^{N} \lambda_i \left[y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1 \right]$$

Minimizing L relative to $\boldsymbol{w}$ and w_0 by setting the partial derivatives to zero results in

$$\boldsymbol{w} = \sum_{i=1}^{N} \lambda_i y_i \boldsymbol{x}_i \quad \text{and} \quad \sum_{i=1}^{N} \lambda_i y_i = 0$$

These are the solutions for minimizing J with the constraints. The problem now is to determine the Lagrange multipliers.

Finding the Decision Boundary

We also get the following regarding the solution from the Karush-Kuhn-Tucker (KKT) conditions:

$$\lambda_i \left[y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1 \right] = 0, \ \forall i \quad \text{and} \quad \lambda_i \geq 0, \ \forall i$$

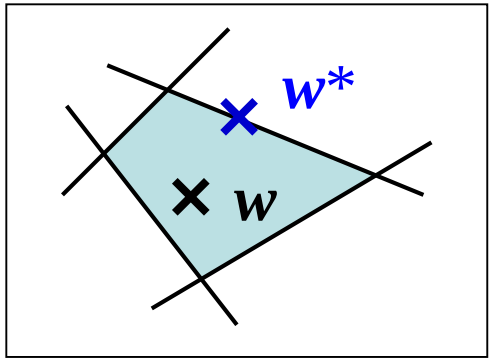
A Lagrange multiplier λ_i can be non-zero only if

$$y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) = 1$$

that is, when $\boldsymbol{x}_i$ is a support vector.

We can understand this using the concept of active and inactive constraints. The Lagrange multipliers are zero for the inactive constraints.

Finding the Decision Boundary



To understand the requirement that all $\lambda_i \geq 0$, consider the case when the solution (called $\boldsymbol{w}^*$) is on the boundary given by the i^{th} constraint, and $\boldsymbol{w}$ is a point inside the feasible region:

$$\left(\partial L/\partial \boldsymbol{w}\right)_{\boldsymbol{w}^*} = 0 = \left(\partial J/\partial \boldsymbol{w}\right)_{\boldsymbol{w}^*} - \lambda_i y_i \boldsymbol{x}_i$$

$(J(\boldsymbol{w}) > J(\boldsymbol{w}^*))$

$$(\boldsymbol{w}-\boldsymbol{w}^*)^T\left(\left(\partial J/\partial \boldsymbol{w}\right)_{\boldsymbol{w}^*}\right) = \lambda_i y_i (\boldsymbol{w}-\boldsymbol{w}^*)^T \boldsymbol{x}_i \geq 0$$

(KKT) assuming $\lambda_i \neq 0$

$$y_i(\boldsymbol{w}^{*T}\ \boldsymbol{x}_i + w_0) = 1$$

original constraint

$$\Rightarrow \lambda_i\left[y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1\right] \geq 0 \quad \Rightarrow \lambda_i \geq 0$$

This is still true when $\boldsymbol{w}^*$ falls on multiple constraint boundaries.

Finding the Decision Boundary

Now how do we obtain the Lagrange multipliers λ_i? After incorporating expressions for optimal $\boldsymbol{w}$, the problem now is to maximize

$$L = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$$

(Now L is a concave function with respect to $\boldsymbol{\lambda}$.)

Subject to the constraints $\sum_{i=1}^{N} \lambda_i y_i = 0$ and $\lambda_i \geq 0, \ \forall i$

This is easier to solve than the original optimization problem, as now we have equality instead of inequality constraints. Once the optimal Lagrange Multipliers are obtained through some optimization procedure, we can compute $\boldsymbol{w}$ using

$$\boldsymbol{w} = \sum_{i=1}^{N} \lambda_i y_i \boldsymbol{x}_i$$

Using any support vector $\boldsymbol{x}_i$, w_0 can be computed using $w_0 = y_i^{-1} - \boldsymbol{w}^T \boldsymbol{x}_i$

Example Continued

Continued example: $\boldsymbol{x}_1=1$, $\boldsymbol{x}_2=2$ in ω_1, and $\boldsymbol{x}_3=-1$, $\boldsymbol{x}_4=-2$ in ω_2.

The process of determining the Lagrange multipliers is not trivial in general. Based on what we know about the problem, we will set $\lambda_2=\lambda_4=0$ first. We then use

$$\frac{\partial L}{\partial \lambda_1} = \frac{\partial L}{\partial \lambda_3} = 0$$

This is combined with the constraint $\sum_{i=1}^{N} \lambda_i y_i = 0$

to yield the solution: $\lambda_1=\lambda_3=0.5$.

We can then determine $\boldsymbol{w}$ and w_0.

Nonseparable Cases

In most cases, the two classes are not linearly separable. This means that the previous constraints ($y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) \geq 1, \ \forall i$) are not valid.

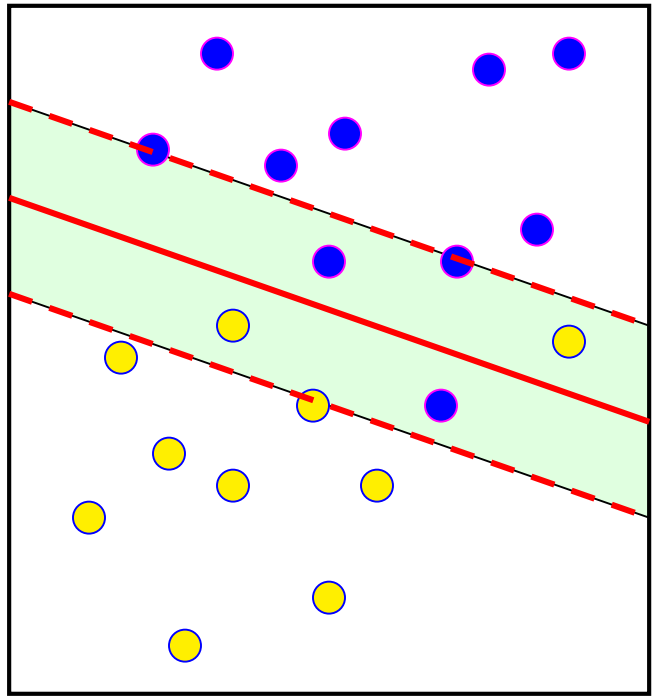
The constraints become

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) \geq 1 - \xi_i$$

where

$$\xi_i \geq 0$$

Cases with $\xi_i > 0$ correspond to those $\boldsymbol{x}_i$ that are misclassified or are within the "separation band".



Nonseparable Cases

Now we have two goals:

- To minimize the number of $\boldsymbol{x}$ that are either misclassified or fall in the separation band.
- To maximize the margin.

This leads to the cost function

$$J = \frac{1}{2}\| \boldsymbol{w} \|^2 + C\sum_{i=1}^{N} I(\xi_i) \quad \text{where} \quad I(\xi_i) = \begin{cases} 1 & \text{for } \xi_i > 0 \\ 0 & \text{for } \xi_i = 0 \end{cases}$$

This is discontinuous and too difficult to optimize. Instead, we use

$$J = \frac{1}{2}\| \boldsymbol{w} \|^2 + C\sum_{i=1}^{N} \xi_i \qquad (C > 0)$$

Nonseparable Cases

Using Lagrange multipliers to include the constraints:

$$L = \frac{1}{2}\| \boldsymbol{w} \|^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\lambda_i\left[y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1 + \xi_i\right] - \sum_{i=1}^{N}\alpha_i\xi_i$$

Minimizing L relative to $\boldsymbol{w}$, w_0, and ξ_i by setting the partial derivatives to zero results in two conditions as in linearly separable cases:

$$\boldsymbol{w} = \sum_{i=1}^{N}\lambda_i y_i \boldsymbol{x}_i \quad \text{and} \quad \sum_{i=1}^{N}\lambda_i y_i = 0$$

As well as

$$\lambda_i = C - \alpha_i$$

Nonseparable Cases

We also get the following from the KKT conditions:

$$\lambda_i \left[y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1 + \xi_i \right] = 0, \ \forall i$$

and $\alpha_i \xi_i = 0, \ \forall i$

A Lagrange multiplier λ_i can be non-zero only if

$$y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) = 1 - \xi_i$$

that is, when $\boldsymbol{x}_i$ is a support vector (inside or on the boundaries of the separation band, or misclassified).

In addition, each support vector $\boldsymbol{x}_i$ that is not on the boundaries of the separation band has $\lambda_i = C$.

Nonseparable Cases

Using the same techniques of the linearly separable cases, we end up with a very similar optimization problem: To maximize

$$L = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$$

Subject to the constraints

$$\sum_{i=1}^{N} \lambda_i y_i = 0 \quad \text{and} \quad 0 \le \lambda_i \le C, \ \forall i$$

Once we've found the optimal λ_i, we can compute $\boldsymbol{w}$ using

$$\boldsymbol{w} = \sum_{i=1}^{N} \lambda_i y_i \boldsymbol{x}_i$$

The value w_0 can be computed using $w_0 = y_i^{-1} - \boldsymbol{w}^T \boldsymbol{x}_i$

where $\boldsymbol{x}_i$ is a support vector on the boundary of the separation band, i.e., ξ_i=0.

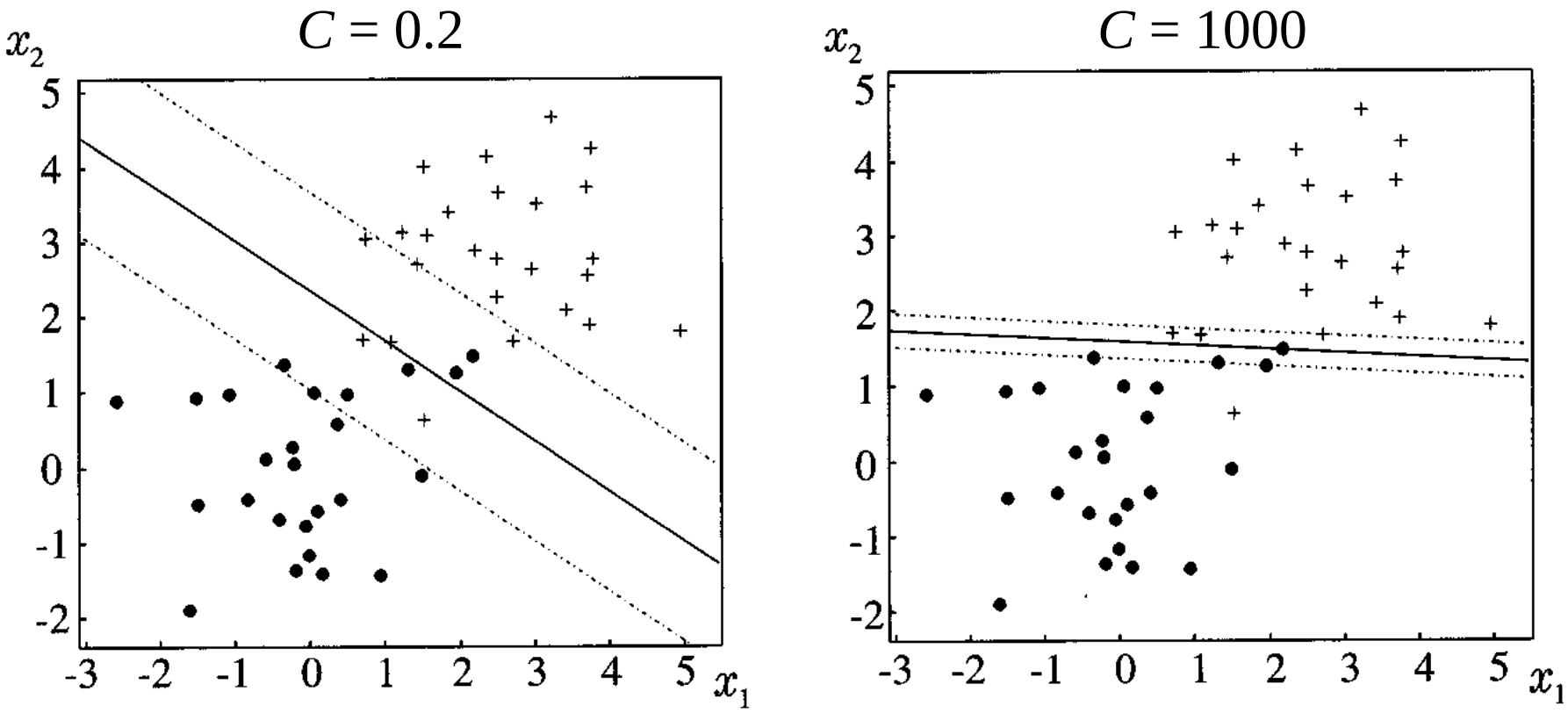
Nonseparable Cases

Some points about the parameter C:

- A trade-off between large margin and minimum classification error (in training data)
- Has to be selected separately (before SVM training). One common approach is to use cross-validation.
- For linearly separable cases, we can obtain the same results as obtained previously by setting $C \to \infty$.

Nonseparable Cases

A computer experiment result for using different C values in a nonseparable case:



Nonlinear SVM

In SVM, the input vectors are used only in inner products. This includes $(\boldsymbol{x}_i^T \boldsymbol{x}_j)$ in L, and in the discriminant function:

$$L = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$$

$$g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0 = \sum_{i=1}^{N} \lambda_i y_i \boldsymbol{x}_i^T \boldsymbol{x} + w_0$$

This allows a "kernel trick": The inner-product of the high-dimensional mapping of two vectors can be expressed as a function of these two vectors.

Kernel Trick

- Some methods use explicit mappings of data into high-dimensional spaces. There are several challenges:
  - The need to choose the dimensionality and the mapping functions.
  - Increased computational cost.
- For many problems where the problem can be expressed using inner products of the data points (feature vectors), we do not need to do explicit mapping.
  - Use kernel functions in the original space to compute the inner products of high-dimensional vectors.
  - No need to select the dimensionality and compute the mapped high-dimensional vectors.
  - Support vector machines are the most well-known example.

Mercer's Theorem

Given any mapping $\boldsymbol{x} \to \boldsymbol{\varphi}(\boldsymbol{x}) \in H$,
there exists a symmetric $K(\boldsymbol{x},\boldsymbol{x}')$ that satisfies **(A)** and **(B)**

Given any symmetric $K(\boldsymbol{x},\boldsymbol{x}')$ that satisfies **(A)**,
there exists a mapping $\boldsymbol{x} \to \boldsymbol{\varphi}(\boldsymbol{x}) \in H$ that satisfies **(B)**

(A) $$\int\int K(\boldsymbol{x}, \boldsymbol{x}')\, f(\boldsymbol{x})\, f(\boldsymbol{x}')\, d\boldsymbol{x}\, d\boldsymbol{x}' \geq 0$$

for any $f(\boldsymbol{x})$ that satisfies $\int f(\boldsymbol{x})^2\, d\boldsymbol{x} < +\infty$

(B) $$K(\boldsymbol{x}, \boldsymbol{x}') = \sum_r \varphi_r(\boldsymbol{x})\, \varphi_r(\boldsymbol{x}')$$

meaning $K(\boldsymbol{x},\boldsymbol{x}')$ is the inner product of $\boldsymbol{\varphi}(\boldsymbol{x})$ and $\boldsymbol{\varphi}(\boldsymbol{x}')$.

SVM Tasks

The optimization task, with $\boldsymbol{x}$ now mapped to high-dimensional space, becomes the maximization of

$$L = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j \boldsymbol{\varphi}(\boldsymbol{x}_i)^T \boldsymbol{\varphi}(\boldsymbol{x}_j)$$

$$= \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

Subject to the constraints

$$\sum_{i=1}^{N} \lambda_i y_i = 0 \quad \text{and} \quad 0 \le \lambda_i \le C, \ \forall i$$

SVM Tasks

The linear discriminant function becomes

$$g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\varphi}(\boldsymbol{x}) + w_0 = \sum_{i=1}^{N} \lambda_i y_i \boldsymbol{\varphi}(\boldsymbol{x}_i)^T \boldsymbol{\varphi}(\boldsymbol{x}) + w_0$$

$$= \sum_{i=1}^{N} \lambda_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + w_0$$

As a result, we can select the function K (the inner-product kernel), and then do all the computation without calculating (or even knowing) $\boldsymbol{\varphi}(\boldsymbol{x})$!!

This "kernel trick" avoids the increased computational cost involved in working with high-dimensional data.

Types of Inner-Product Kernels

Different common types of $K(\boldsymbol{x},\boldsymbol{x}')$:

Polynomial

$$K(\boldsymbol{x},\boldsymbol{x}') = (\boldsymbol{x}^T\boldsymbol{x}'+1)^q \quad (q>0)$$

- q is specified by the user

RBF

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

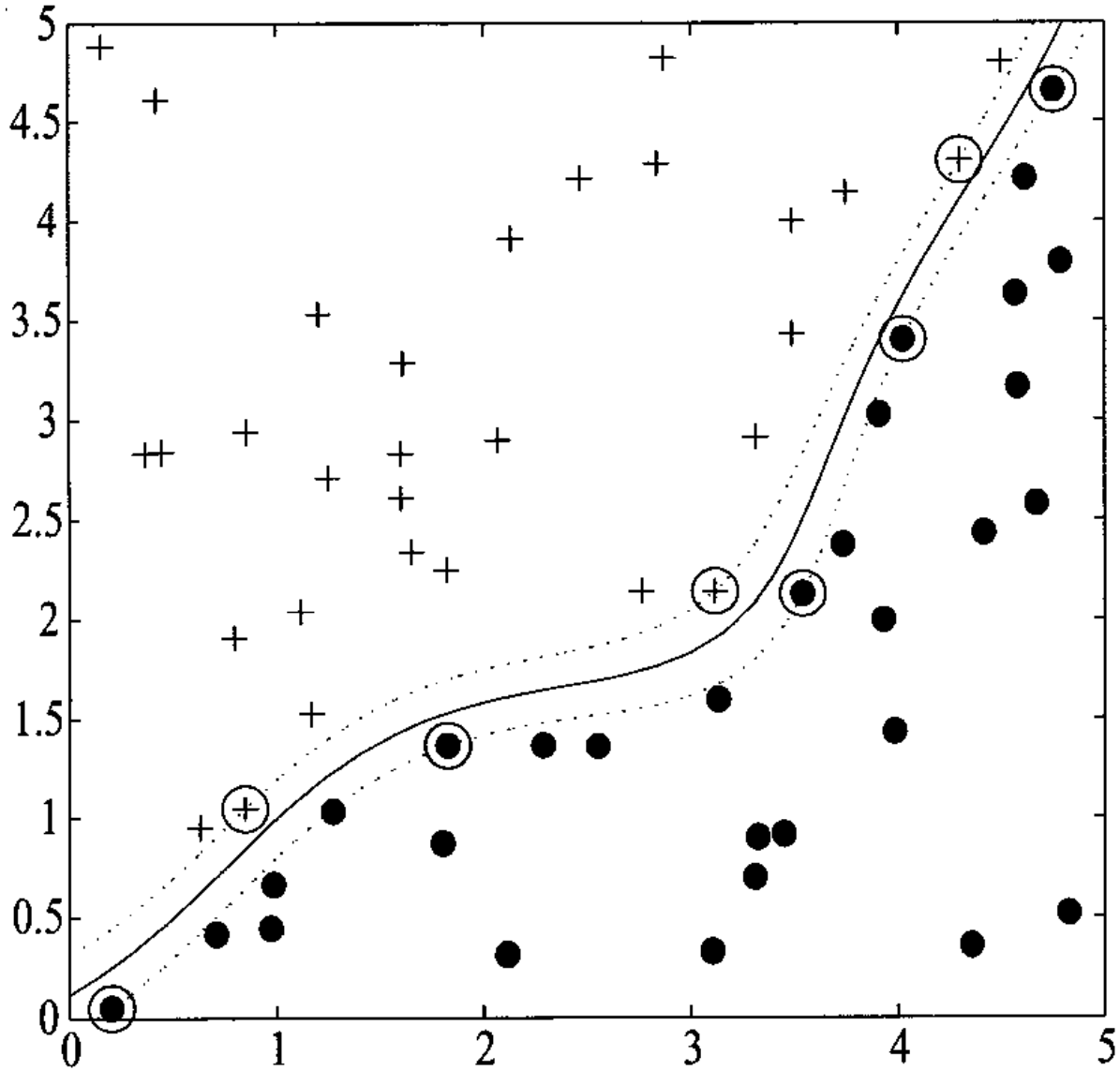
- σ is specified by the user and common to all kernels
- RBF network

Hyperbolic

$$K(\boldsymbol{x},\boldsymbol{x}') = \tanh(\beta\boldsymbol{x}^T\boldsymbol{x}'+\gamma)$$

- Only some values of β and γ can be used, e.g., $\beta=2$ and $\gamma=1$.
- Two-layer perceptron with $\tanh$ activation functions.

SVM Experiments



Kernel PCA

- When PCA is not effective for the original data distribution, mapping to a high-dimensional space might lead to more useful PCA results to subsequent tasks.
- Mapping: $\boldsymbol{x} \leftarrow \varphi(\boldsymbol{x})$
- Assuming that the projection of data points to the high-dimensional space is centered (having zero mean), the covariance matrix becomes:

$$\Sigma = \frac{1}{N}\sum_{i=1}^{N} \varphi(\boldsymbol{x}_i)\varphi(\boldsymbol{x}_i)^T$$

- However, we never actually compute the high-dimensional vectors, so this covariance matrix cannot be used directly.

Derivation of Kernel PCA

- As PCA involves the eigen-decomposition of the covariance matrix, let as consider this form ($\boldsymbol{u}$ is an eigenvector here):

$$\Sigma \boldsymbol{u} = \lambda \boldsymbol{u}$$

- We can express $\boldsymbol{u}$ as a linear combination of the projected high-dimensional data points, as the subspace spanned by the eigenvectors with nonzero eigenvalues is the same as the subspace spanned by the data points:

$$\boldsymbol{u} = \sum_{i=1}^{N} a_i \varphi(\boldsymbol{x}_i)$$

Derivation of Kernel PCA

- Substituting $\boldsymbol{u}$ back to the eigen-decomposition of Σ:

$$\Rightarrow \quad \left[\frac{1}{N}\sum_{i=1}^{N}\varphi(\boldsymbol{x}_i)\varphi(\boldsymbol{x}_i)^T\right]\left[\sum_{j=1}^{N}a_j\varphi(\boldsymbol{x}_j)\right]=\lambda\left[\sum_{i=1}^{N}a_i\varphi(\boldsymbol{x}_i)\right]$$

- Exchange of terms:

$$\Rightarrow \quad \frac{1}{N}\sum_{i=1}^{N}\varphi(\boldsymbol{x}_i)\left[\sum_{j=1}^{N}a_j\varphi(\boldsymbol{x}_i)^T\varphi(\boldsymbol{x}_j)\right]=\lambda\sum_{i=1}^{N}a_i\varphi(\boldsymbol{x}_i)$$

- To convert everything involving the high-dimensional vectors to inner products:

$$\Rightarrow \quad \frac{1}{N}\sum_{i=1}^{N}\varphi(\boldsymbol{x}_k)^T\varphi(\boldsymbol{x}_i)\left[\sum_{j=1}^{N}a_j\varphi(\boldsymbol{x}_i)^T\varphi(\boldsymbol{x}_j)\right]=\lambda\sum_{i=1}^{N}a_i\varphi(\boldsymbol{x}_k)^T\varphi(\boldsymbol{x}_i)$$

Derivation of Kernel PCA

- Now, express this with the inner-product kernels:

$$\Rightarrow \quad \frac{1}{N}\sum_{i=1}^{N} K(\boldsymbol{x}_k, \boldsymbol{x}_i)\left[\sum_{j=1}^{N} a_j K(\boldsymbol{x}_j, \boldsymbol{x}_i)\right] = \lambda \sum_{i=1}^{N} a_i K(\boldsymbol{x}_k, \boldsymbol{x}_i)$$

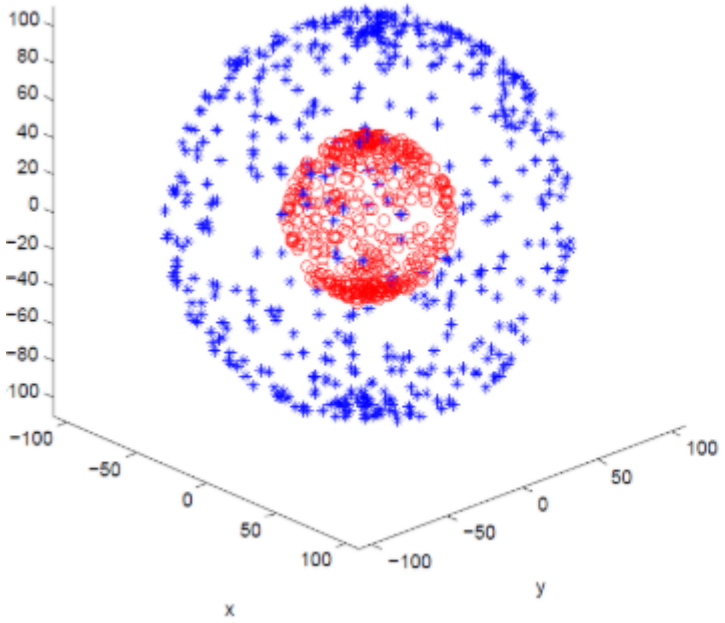
- Express this in matrix form: $\boldsymbol{K}(\boldsymbol{K}\boldsymbol{\alpha}) = N\lambda \boldsymbol{K}\boldsymbol{\alpha}$
- Here $\boldsymbol{K}$ is the kernel matrix, and $\boldsymbol{\alpha}$ is the vector of the coefficients. We can remove one $\boldsymbol{K}$ from both sides because we are not concerned with eigenvectors of zero eigenvalues:

$$\boldsymbol{K}\boldsymbol{\alpha} = N\lambda\boldsymbol{\alpha}$$

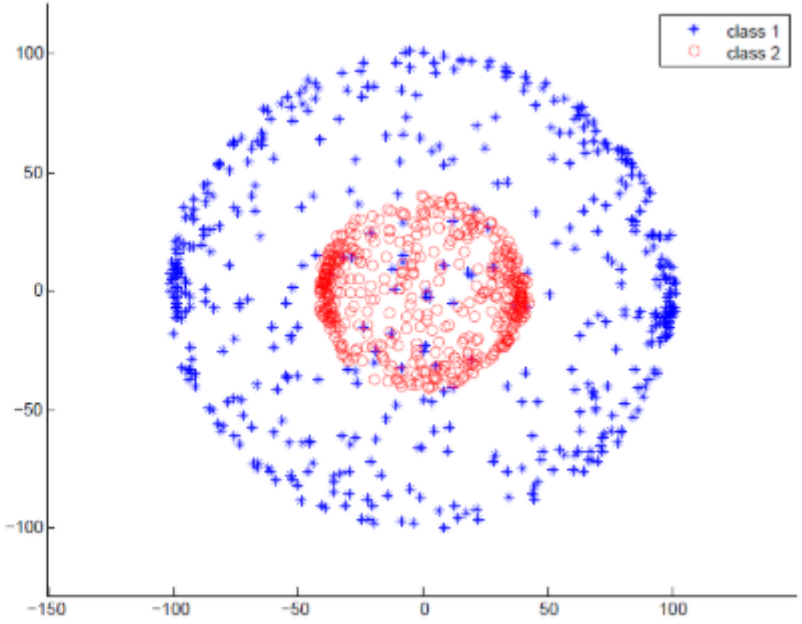
- Now we again have an eigen-decomposition problem. We can then project the data points to a low-dimension subspace spanned by a subset of the eigenvectors (with the largest eigenvalues) of $\boldsymbol{K}$.

Kernel PCA Example

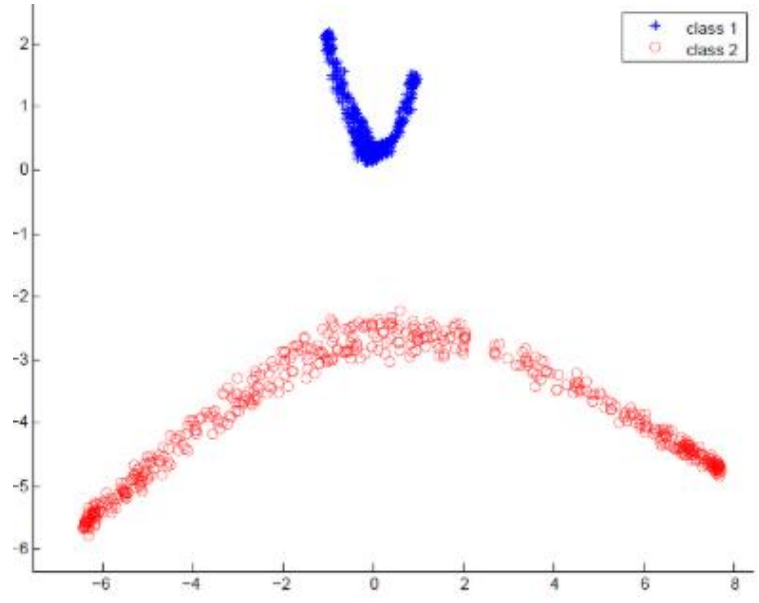
Original Data (3D)



Linear PCA (2D)



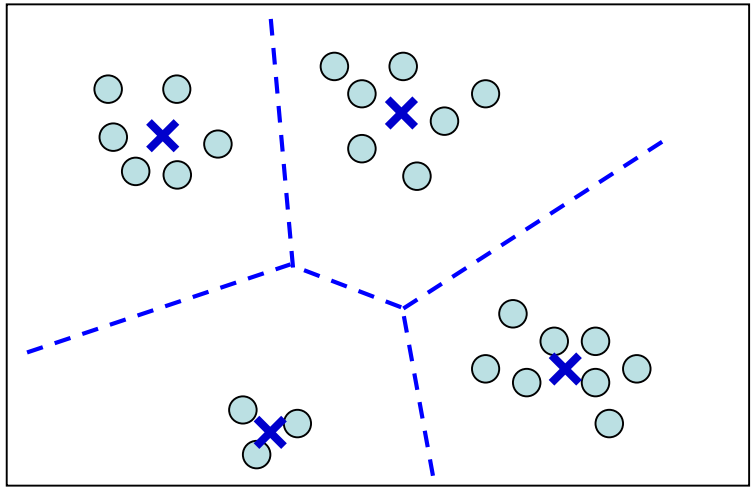
Kernel PCA (2D)
with Gaussian Kernels



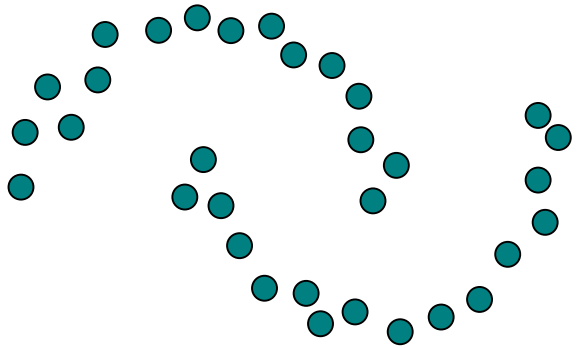
Wang (2012) Kernel Principal Component Analysis and its Applications https://arxiv.org/abs/1207.3538

Kernel k-Means: Motivation

- The “kernel” version of k-means aims to solve another limitation of k-means: Linear cluster boundaries.
- In standard k-means, the prototypes define “cluster boundaries” that are hyperplanes in the feature space:



- Q: Can we separate the "clusters" in this example?



Kernel Trick in Kernel k-Means

- Let us rewrite the cost function of k-means:

$$J = \sum_{j=1}^{k} \sum_{\boldsymbol{x} \in C_j} \left\| \boldsymbol{x} - \boldsymbol{m}_j \right\|^2$$

  - C_j : The set of samples in cluster #j.
  - $\boldsymbol{m}_j$: Centroid (prototype) of cluster #j.
- Now after the mapping to a high-dimensional space:

$$\boldsymbol{x} \leftarrow \varphi(\boldsymbol{x})$$

$$\boldsymbol{m}_j \leftarrow \frac{1}{\left| C_j \right|} \sum_{\boldsymbol{x} \in C_j} \varphi(\boldsymbol{x})$$

Kernel Trick in Kernel k-Means

The squared distance in the cost function can be rewritten as

$$\varphi(\boldsymbol{x})^{\mathrm{T}}\varphi(\boldsymbol{x}) - \frac{2}{|C|}\sum_{\boldsymbol{x}_j \in C}\varphi(\boldsymbol{x})^{\mathrm{T}}\varphi(\boldsymbol{x}_j) + \frac{1}{|C|^2}\sum_{\boldsymbol{x}_j, \boldsymbol{x}_k \in C}\varphi(\boldsymbol{x}_j)^{\mathrm{T}}\varphi(\boldsymbol{x}_k)$$

assuming that $\boldsymbol{x}$ belongs to cluster C.

We can see that everything is expressed in inner products of mapped vectors. These inner products can be replaced with the corresponding kernel function:

$$K(\boldsymbol{x}, \boldsymbol{x}) - \frac{2}{|C|}\sum_{\boldsymbol{x}_j \in C}K(\boldsymbol{x}, \boldsymbol{x}_j) + \frac{1}{|C|^2}\sum_{\boldsymbol{x}_j, \boldsymbol{x}_k \in C}K(\boldsymbol{x}_j, \boldsymbol{x}_k)$$

Kernel k-Means

- Iterative update of cluster assignments only: In each iteration, each sample is assigned to the cluster with the smallest sample-cluster distance.
- No explicit computation of the prototypes. (They are in the high dimensional space.) ➔ Not suitable if the actual prototypes are needed.
- When N is very large, processing of the NxN kernel matrix can still be computationally expensive.