UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

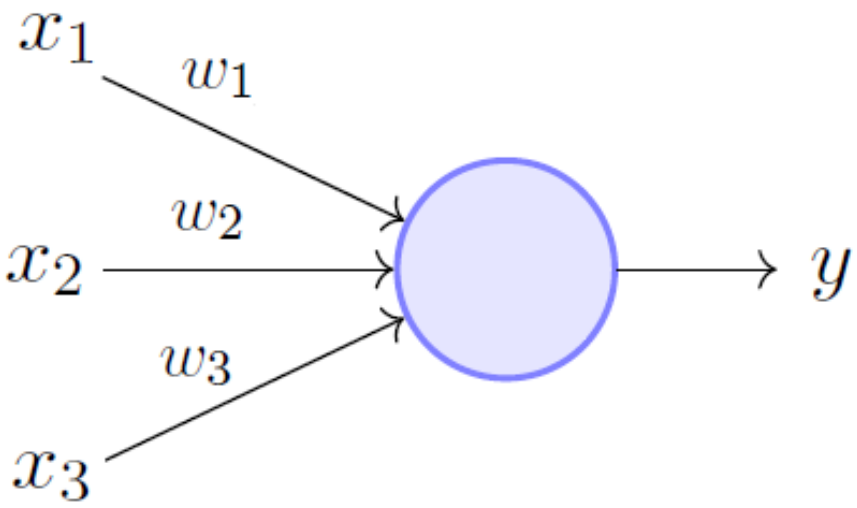
Introduction to ML
Lecture 12: Perceptron

Hamed Tabkhi
Department of Electrical and Computer Engineering,
University of North Carolina Charlotte (UNCC)
htabkhiv@uncc.edu

UNC CHARLOTTE

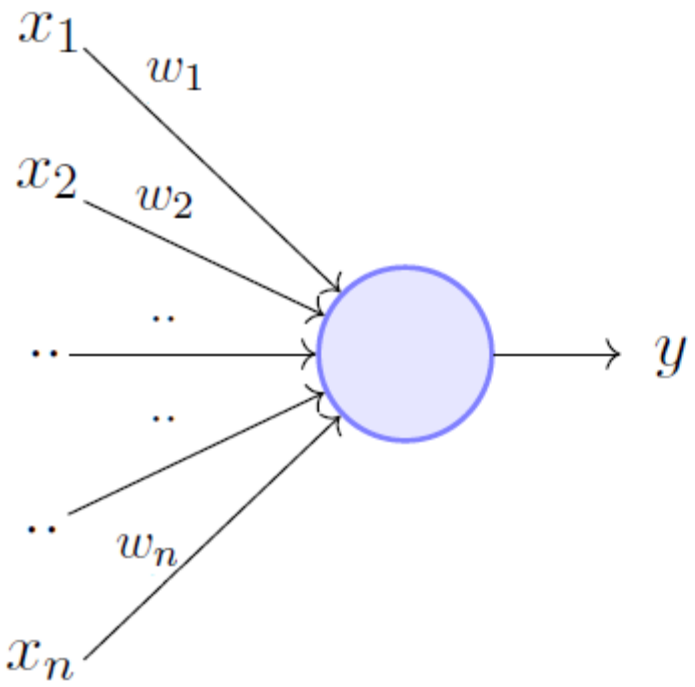
Perceptron

Perceptron



Perceptron Model (Minsky-Papert in 1969)

Perceptron



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

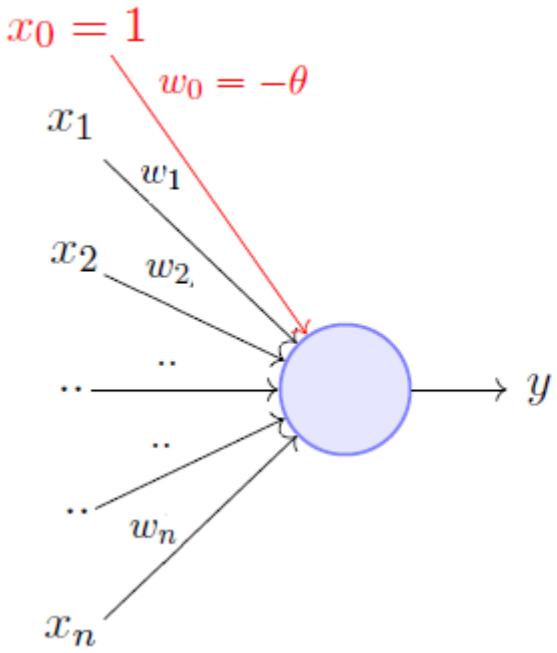
Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

Perceptron

However, according to the convention, instead of hand coding the thresholding parameter ***thetha***, we add it as one of the inputs, with the weight -***theta*** like shown below, which makes it learn-able (more on this in my next post — *Perceptron Learning Algorithm*).



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

Dot Product

- Imagine you have two vectors oh size **n+1**, **w** and **x**, the dot product of these vectors (***w.x***) could be computed as follows:

$$\mathbf{w} = [w_0, w_1, w_2, ..., w_n]$$
$$\mathbf{x} = [1, x_1, x_2, ..., x_n]$$
$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w}^{\mathrm{T}}\mathbf{x} = \sum_{i=0}^{n} w_i * x_i$$

  - Here, **w** and **x** are just two lonely arrows in an **n+1 dimensional** space
  - Intuitively, their dot product quantifies how much one vector is going in the direction of the other
  - The decision boundary line which a perceptron gives out that separates positive examples from the negative ones is really just **w . x** = 0.

Perceptron Learning Algorithm

```
while !convergence do
    Pick random x ∈ P ∪ N ;
    if x ∈ P   and   w.x < 0 then
        w = w + x ;
    end
    if x ∈ N   and   w.x ≥ 0 then
        w = w − x ;
    end
end
```

- **Case 1:** When **x** belongs to *P* and its dot product **w.x** < 0
  Case 2: When **x** belongs to *N* and its dot product **w.x** ≥ 0

Example

Example: $(-1,2)$ − ✗

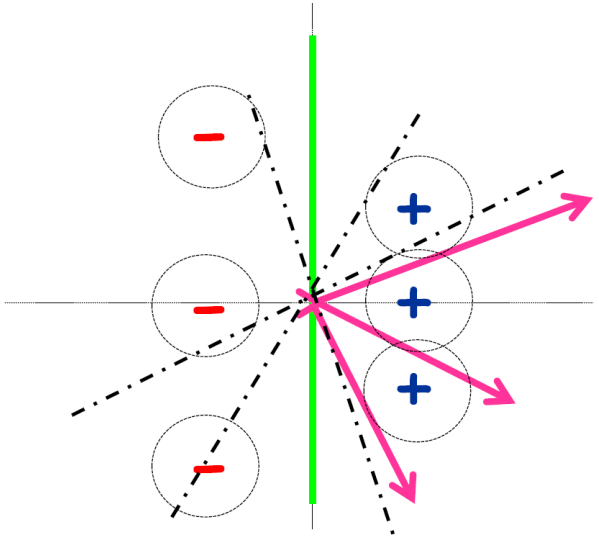
$(1,0)$ + ✓

$(1,1)$ + ✗

$(-1,0)$ − ✓

$(-1,-2)$ − ✗

$(1,-1)$ + ✓



$w_1 = (0,0)$

$w_2 = w_1 - (-1,2) = (1,-2)$

$w_3 = w_2 + (1,1) = (2,-1)$

$w_4 = w_3 - (-1,-2) = (3,1)$

- Are you wondering seemingly arbitrary operations of **x** and **w** would help you learn that perfect **w** that can perfectly classify *P* and *N*?
- We have already established that when **x** belongs to *P*, we want **w.x** > 0, basic perceptron rule.

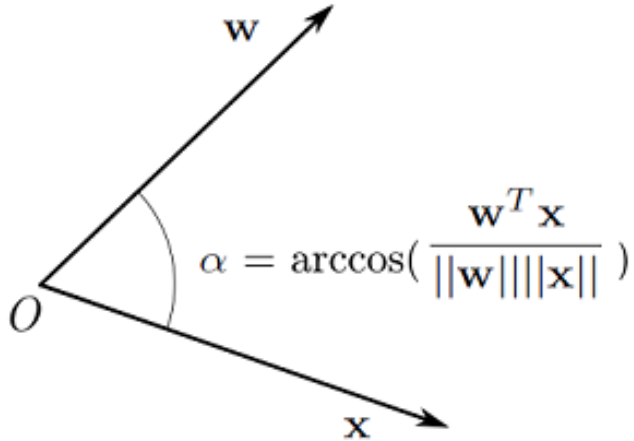
Geometrical Definition of Dot Product

- Intuitively, their dot product quantifies how much one vector is going in the direction of the other
- Now the same old dot product can be computed differently if only you knew the angle between the vectors and their individual magnitudes. Here's how:

$$\mathbf{w}^T\mathbf{x} = \|\mathbf{w}\|\|\mathbf{x}\| cos\alpha$$

- The other way around, you can get the angle between two vectors, if only you knew the vectors, given you know how to calculate vector magnitudes and their vanilla dot product.

$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||}$$



Geometrical Proof of Convergence

- So, when **x** belongs to *P*, the angle between **w** and **x** should be less than 90 because the cosine of the angle is proportional to the dot product.
- So, whatever the **w** vector may be, as long as it makes an angle less than 90 degrees with the positive example data vectors (**x** E *P*) and an angle more than 90 degrees with the negative example data vectors (**x** E *N*), we are cool.

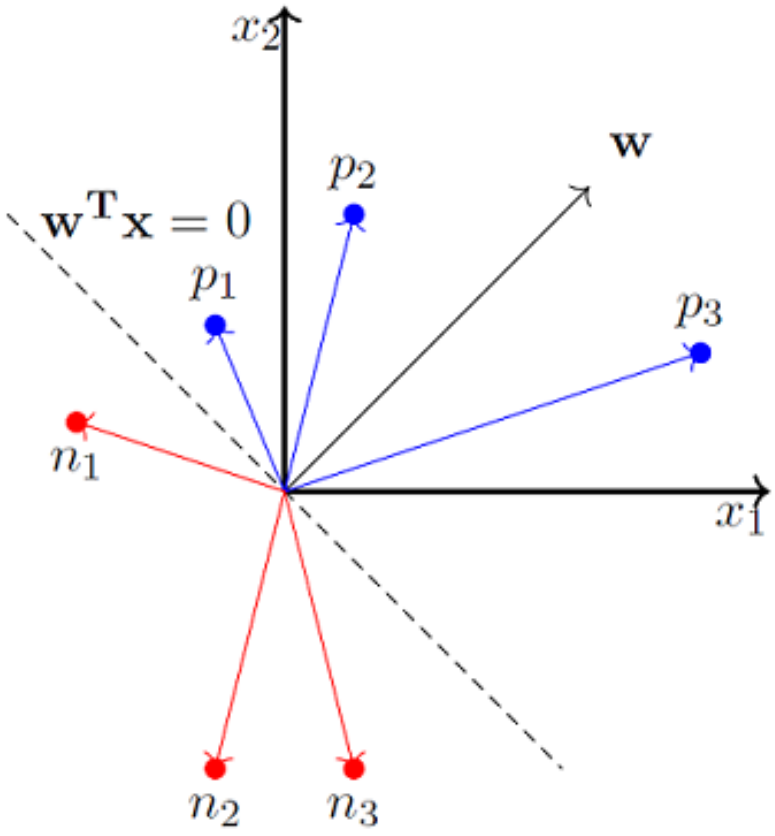
$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||} \quad \Big| \quad cos\alpha \propto \mathbf{w}^T\mathbf{x}$$

$$\text{So if } \mathbf{w}^T\mathbf{x} > 0 \Rightarrow cos\alpha > 0 \Rightarrow \alpha < 90$$

$$\text{Similarly, if } \mathbf{w}^T\mathbf{x} < 0 \Rightarrow cos\alpha < 0 \Rightarrow \alpha > 90$$

Geometrical Proof of Convergence

- So, when we are adding **x** to **w**, which we do when x belongs to P and **w.x** < 0 (Case 1), we are essentially **increasing the *cos(alpha)*** value.
- It means, we are **decreasing the *alpha* value**, the angle between **w** and **x**, **which is what we desire**.
- And the similar intuition works for the case when **x** belongs to *N* and **w.x** ≥ 0 (Case 2).



UNC CHARLOTTE

Geometrical Proof of Convergence

- So we now strongly believe that the angle between **w** and **x** should be less than 90 when **x** belongs to *P* class and the angle between them should be more than 90 when **x** belongs to *N* class.
- Here's why the update works:

(α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$
\begin{aligned}
cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T\mathbf{x} \\
&\propto (\mathbf{w}+\mathbf{x})^T\mathbf{x} \\
&\propto \mathbf{w}^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \\
&\propto cos\alpha + \mathbf{x}^T\mathbf{x} \\
cos(\alpha_{new}) &> cos\alpha
\end{aligned}
$$

(α_{new}) when $\mathbf{w_{new}} = \mathbf{w} - \mathbf{x}$

$$
\begin{aligned}
cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T\mathbf{x} \\
&\propto (\mathbf{w}-\mathbf{x})^T\mathbf{x} \\
&\propto \mathbf{w}^T\mathbf{x} - \mathbf{x}^T\mathbf{x} \\
&\propto cos\alpha - \mathbf{x}^T\mathbf{x} \\
cos(\alpha_{new}) &< cos\alpha
\end{aligned}
$$

Kernelization

Perceptron Learning Algorithm

- Our goal is to find the **w** vector that can perfectly classify positive inputs and negative inputs in our data.

- Set t=1, start with the all zero vector w_1.
- Given example x, predict positive iff $w_t \cdot x \geq 0$
- On a mistake, update as follows:
  - Mistake on positive, then update $w_{t+1} \leftarrow w_t + x$
  - Mistake on negative, then update $w_{t+1} \leftarrow w_t - x$

Perceptron Learning Algorithm

- Set t=1, start with the all zero vector w_1.
- Given example x, predict positive iff $w_t \cdot x \geq 0$
- On a mistake, update as follows:
  - Mistake on positive, then update $w_{t+1} \leftarrow w_t + x$
  - Mistake on negative, then update $w_{t+1} \leftarrow w_t - x$

Note: w_t is weighted sum of incorrectly classified examples

$$w_t = a_{i_1} x_{i_1} + \cdots + a_{i_k} x_{i_k}$$

$$w_t \cdot x = a_{i_1} x_{i_1} \cdot x + \cdots + a_{i_k} x_{i_k} \cdot x$$

Feature Mapping

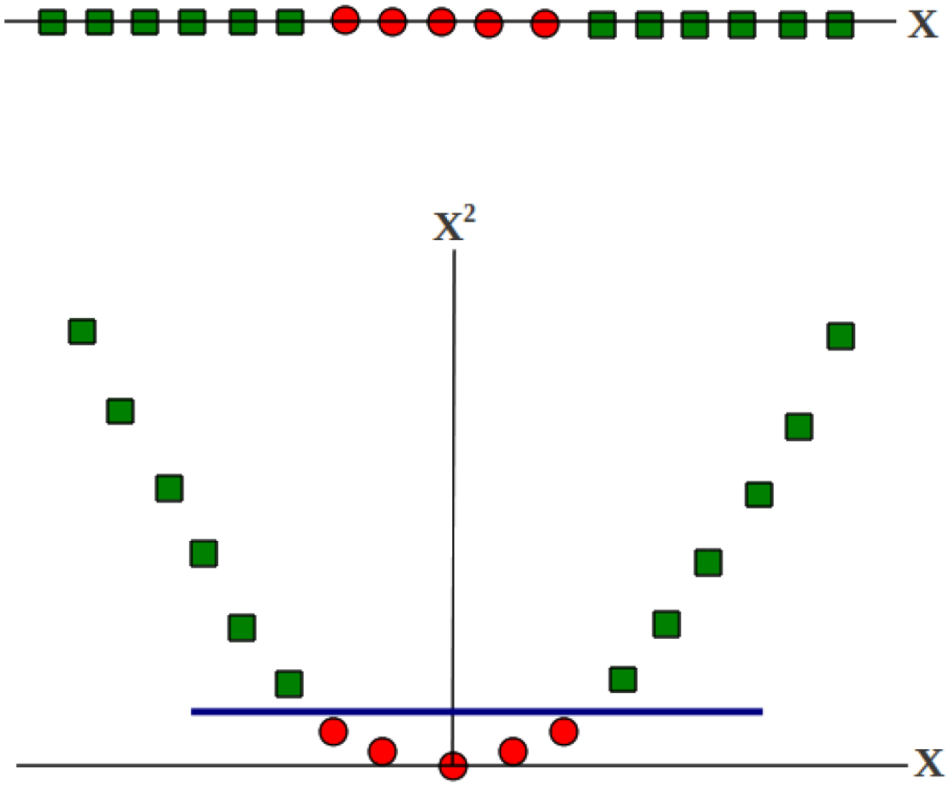
Problem: data not linearly separable in the most natural feature representation.

Solution: Feature Mapping and Kernelization

- Map an original feature vector $x = \langle x_1, x_2, x_3, \ldots, x_D \rangle$ to an expanded version $\phi(x)$

- The aim is to make nonlinear classification problem to a linearly separatable problem at higher dimensions.
- If data is linearly separable by large margin in the Φ-space, then don't overfit (i.e., good sample complexity).

Feature Mapping Example

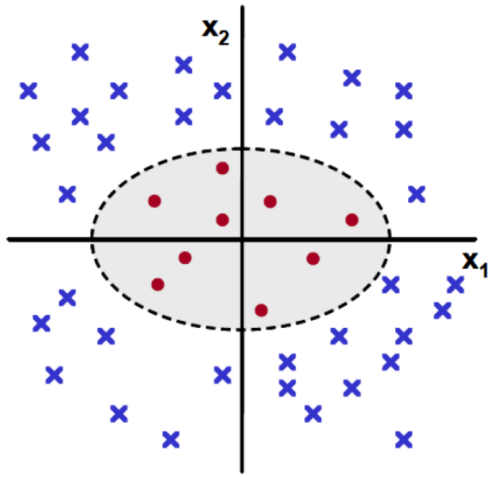


Non-linearly separable data in 1D

Becomes linearly separable in new 2D space
defined by the following mapping:

$$x \rightarrow \{x, x^2\}$$

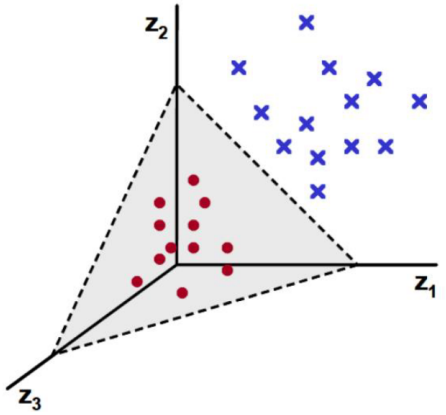
Feature Mapping Example



Non-linearly
separable data in 2D

Becomes linearly separable in the 3D space
defined by the following transformation:

$$\mathbf{x} = \{x_1, x_2\} \rightarrow \mathbf{z} = \{x_1^2, \sqrt{2}x_1x_2, x_2^2\}$$



UNC CHARLOTTE

Kernelization

Definition

$K(\cdot,\cdot)$ is a kernel if it can be viewed as a legal definition of inner product:

- $\exists\ \phi: X \to R^N$ s.t. $K(x,z) = \phi(x) \cdot \phi(z)$
  - Range of ϕ is called the Φ-space.
  - N can be very large.

- But think of ϕ as implicit, not explicit!!!!

- A kernel K is a replacement over baseline legal dot-product that creates an implicit mapping to higher dimension Φ.
- So, if replace x·z with K(x,z) they act implicitly as if data was in the higher-dimensional Φ-space.

$$\phi: R^2 \to R^3,\ (x_1, x_2) \to \Phi(x) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

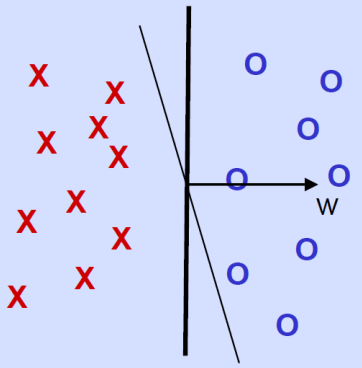
$$\phi(x) \cdot \phi(z) = \left(x_1^2, x_2^2, \sqrt{2}x_1x_2\right) \cdot (z_1^2, z_2^2, \sqrt{2}z_1z_2)$$

$$= (x_1z_1 + x_2z_2)^2 = (x \cdot z)^2 = K(x,z)$$

UNC CHARLOTTE

Perceptron with Kernelization Trick

- Set t=1, start with the all zero vector w_1.
- Given example x, predict + iff $w_t \cdot x \geq 0$
- On a mistake, update as follows:
  - Mistake on positive, $w_{t+1} \leftarrow w_t + x$
  - Mistake on negative, $w_{t+1} \leftarrow w_t - x$



Easy to kernelize since w_t is weighted sum of incorrectly classified examples $w_t = a_{i_1} x_{i_1} + \cdots + a_{i_k} x_{i_k}$

Replace $w_t \cdot x = a_{i_1} x_{i_1} \cdot x + \cdots + a_{i_k} x_{i_k} \cdot x$ with

$a_{i_1} K(x_{i_1}, x) + \cdots + a_{i_k} K(x_{i_k}, x)$

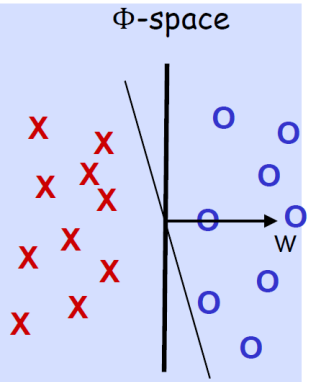
UNC CHARLOTTE

Perceptron with Kernelization Trick

- Given x, predict + iff $\phi(x_{i_{t-1}}) \cdot \phi(x)$

$$a_{i_1} K(x_{i_1}, x) + \cdots + a_{i_{t-1}} K(x_{i_{t-1}}, x) \geq 0$$

- On the t th mistake, update as follows:
  - Mistake on positive, set $a_{i_t} \leftarrow 1$; store x_{i_t}
  - Mistake on negative, $a_{i_t} \leftarrow -1$; store x_{i_t}



Perceptron $w_t = a_{i_1} x_{i_1} + \cdots + a_{i_k} x_{i_k}$

$$w_t \cdot x = a_{i_1} x_{i_1} \cdot x + \cdots + a_{i_k} x_{i_k} \cdot x \quad \rightarrow \quad a_{i_1} K(x_{i_1}, x) + \cdots + a_{i_k} K(x_{i_k}, x)$$

Exact same behavior/prediction rule as if mapped data in the ϕ-space and ran Perceptron there!

Do this implicitly, so computational savings!!!!!

UNC CHARLOTTE

Common Kernel Functions

Linear: $\mathrm{K}(\mathrm{x}, \mathrm{z}) = \mathrm{x} \cdot z$

Polynomial: $\mathrm{K}(\mathrm{x}, z) = (\mathrm{x} \cdot z)^{\mathrm{d}}$ or $\mathrm{K}(\mathrm{x}, z) = (1 + \mathrm{x} \cdot z)^{\mathrm{d}}$

Gaussian: $\mathrm{K}(\mathrm{x}, z) = \exp\left[-\frac{||x-z||^2}{2\,\sigma^2}\right]$

Laplace Kernel: $\mathrm{K}(\mathrm{x}, z) = \exp\left[-\frac{||x-z||}{2\,\sigma^2}\right]$

Common Kernel Functions

Name	Kernel Function (implicit dot product)	Feature Space (explicit dot product)
Linear	$K(\mathbf{x}, \mathbf{z}) = \mathbf{x}^T\mathbf{z}$	Same as original input space
Polynomial (v1)	$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T\mathbf{z})^d$	All polynomials **of** degree d
Polynomial (v2)	$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T\mathbf{z} + 1)^d$	All polynomials **up to** degree d
Gaussian	$K(\mathbf{x}, \mathbf{z}) = \exp(-\frac{\|\|\mathbf{x} - \mathbf{z}\|\|_2^2}{2\sigma^2})$	Infinite dimensional space
Hyperbolic Tangent (Sigmoid) Kernel	$K(\mathbf{x}, \mathbf{z}) = \tanh(\alpha\mathbf{x}^T\mathbf{z} + c)$	(With SVM, this is equivalent to a 2-layer neural network)

Kernelization Summary

Pros:
can help turn non-linear classification problem into linear problem

Cons:
"feature explosion" creates issues when training linear classifier in new feature space
–More computationally expensive to train
–More training examples needed to avoid overfitting