

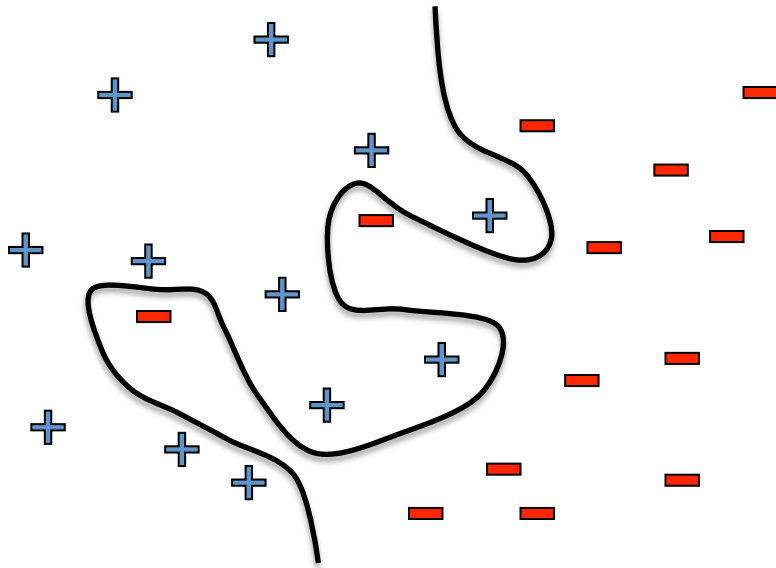
CSE446: Kernels and Kernelized Perceptron Winter 2015

Luke Zettlemoyer

Slides adapted from Carlos Guestrin

What if the data is not linearly separable?

**Use features of features
of features of features....**

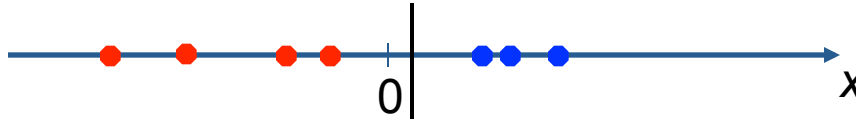


$$\phi(x) = \begin{pmatrix} x_1 \\ \dots \\ x_n \\ x_1 x_2 \\ x_1 x_3 \\ \dots \\ e_{x_1} \\ \dots \end{pmatrix}$$

Feature space can get really large really quickly!

Non-linear features: 1D input

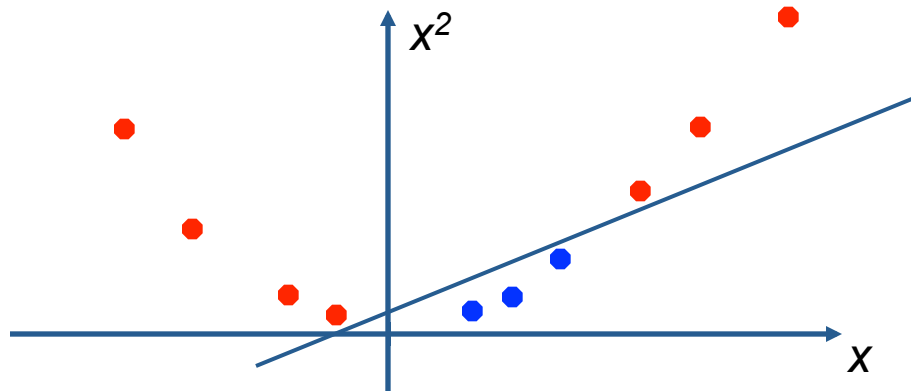
- Datasets that are linearly separable with some noise work out great:



- But what are we going to do if the dataset is just too hard?



- How about... mapping data to a higher-dimensional space:



Feature spaces

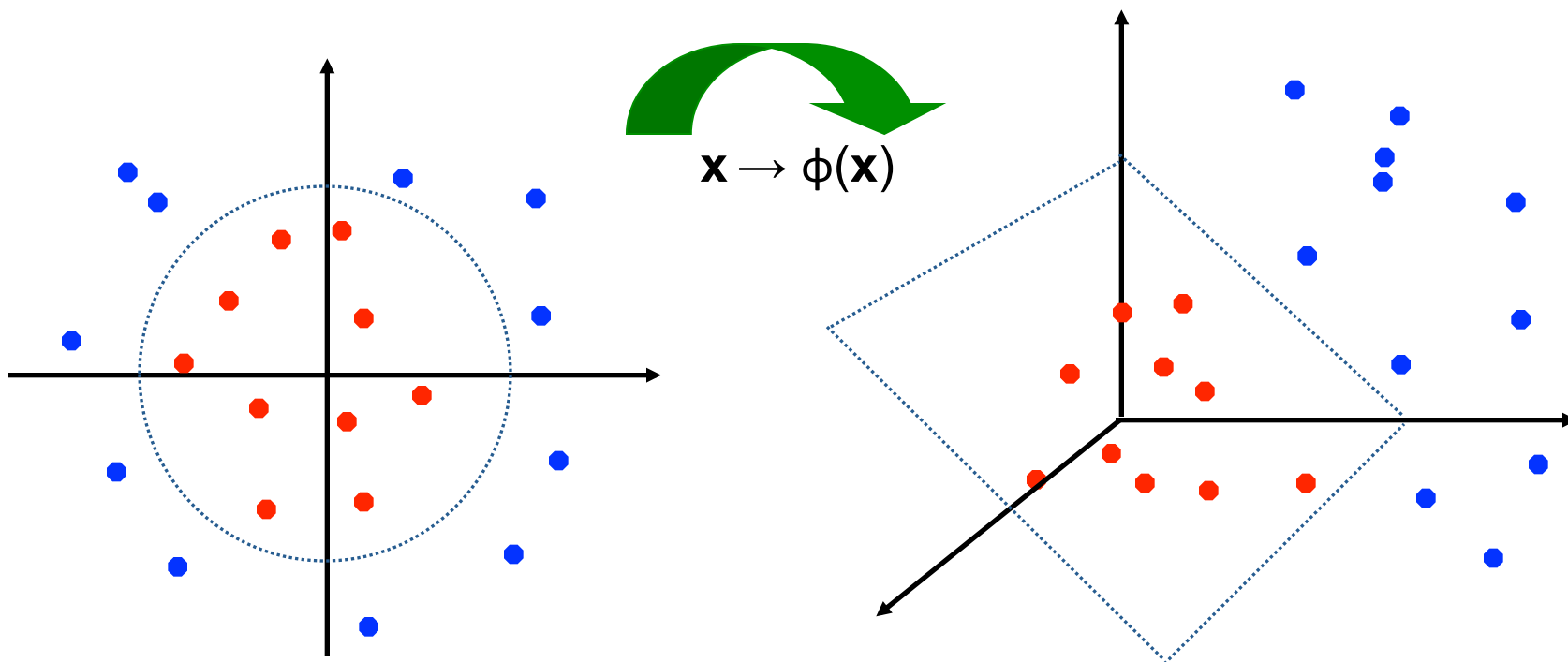
- **General idea:** map to higher dimensional space

- if \mathbf{x} is in \mathbb{R}^n , then $\phi(\mathbf{x})$ is in \mathbb{R}^m for $m > n$

- Can now learn feature weights \mathbf{w} in \mathbb{R}^m and predict:

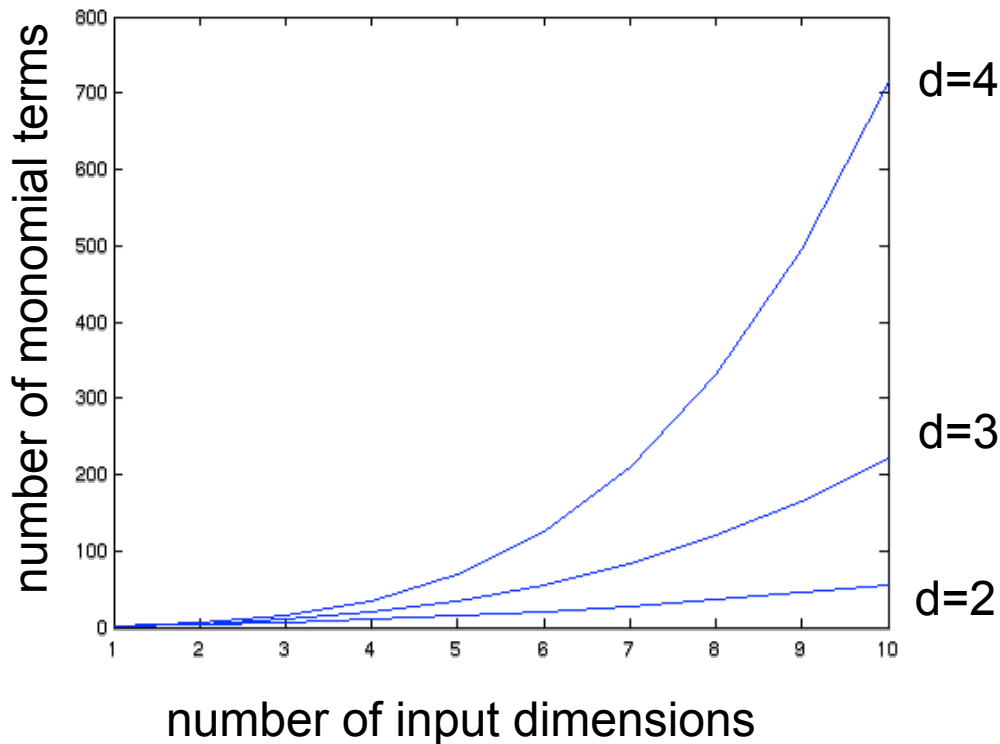
$$y = \text{sign}(\mathbf{w} \cdot \phi(\mathbf{x}))$$

- Linear function in the higher dimensional space will be non-linear in the original space



Higher order polynomials

$$\text{num. terms} = \binom{d + m - 1}{d} = \frac{(d + m - 1)!}{d!(m - 1)!}$$



m – input features
d – degree of polynomial

grows fast!
d = 6, m = 100
about 1.6 billion terms

Efficient dot-product of polynomials

Polynomials of degree exactly d

$d=1$

$$\phi(u) \cdot \phi(v) = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = u_1 v_1 + u_2 v_2 = u \cdot v$$

$d=2$

$$\begin{aligned} \phi(u) \cdot \phi(v) &= \begin{pmatrix} u_1^2 \\ u_1 u_2 \\ u_2 u_1 \\ u_2^2 \end{pmatrix} \cdot \begin{pmatrix} v_1^2 \\ v_1 v_2 \\ v_2 v_1 \\ v_2^2 \end{pmatrix} = u_1^2 v_1^2 + 2u_1 v_1 u_2 v_2 + u_2^2 v_2^2 \\ &= (u_1 v_1 + u_2 v_2)^2 \\ &= (u \cdot v)^2 \end{aligned}$$

For any d (we will skip proof):

$$K(u, v) = \phi(u) \cdot \phi(v) = (u \cdot v)^d$$

- **Cool!** Taking a dot product and an exponential gives same results as mapping into high dimensional space and then taking dot product

The “Kernel Trick”

- A *kernel function* defines a dot product in some feature space.

$$K(\mathbf{u}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{u}) \bullet \boldsymbol{\phi}(\mathbf{v})$$

- Example:

2-dimensional vectors $\mathbf{u} = [u_1 \ u_2]$ and $\mathbf{v} = [v_1 \ v_2]$; let $K(\mathbf{u}, \mathbf{v}) = (1 + \mathbf{u} \bullet \mathbf{v})^2$,

Need to show that $K(\mathbf{x}_i, \mathbf{x}_j) = \boldsymbol{\phi}(\mathbf{x}_i) \bullet \boldsymbol{\phi}(\mathbf{x}_j)$:

$$\begin{aligned} K(\mathbf{u}, \mathbf{v}) &= (1 + \mathbf{u} \bullet \mathbf{v})^2 = 1 + u_1^2 v_1^2 + 2 u_1 v_1 u_2 v_2 + u_2^2 v_2^2 + 2 u_1 v_1 + 2 u_2 v_2 = \\ &= [1, u_1^2, \sqrt{2} u_1 u_2, u_2^2, \sqrt{2} u_1, \sqrt{2} u_2] \bullet [1, v_1^2, \sqrt{2} v_1 v_2, v_2^2, \sqrt{2} v_1, \sqrt{2} v_2] = \\ &= \boldsymbol{\phi}(\mathbf{u}) \bullet \boldsymbol{\phi}(\mathbf{v}), \quad \text{where } \boldsymbol{\phi}(\mathbf{x}) = [1, x_1^2, \sqrt{2} x_1 x_2, x_2^2, \sqrt{2} x_1, \sqrt{2} x_2] \end{aligned}$$

- Thus, a kernel function *implicitly* maps data to a high-dimensional space (without the need to compute each $\boldsymbol{\phi}(\mathbf{x})$ explicitly).
- But, it isn't obvious yet how we will incorporate it into actual learning algorithms...

“Kernel trick” for The Perceptron!

- Never compute features explicitly!!!

- Compute dot products in closed form $K(u,v) = \Phi(u) \cdot \Phi(v)$

- Standard Perceptron:

- set $w_i=0$ for each feature i
- set $a^i=0$ for each example i
- For $t=1..T, i=1..n$:
 - $y = \text{sign}(w \cdot \phi(x^i))$
 - if $y \neq y^i$
 - $w = w + y^i \phi(x^i)$
 - $a^i += y^i$
- At all times during learning:

$$w = \sum_k a^k \phi(x^k)$$

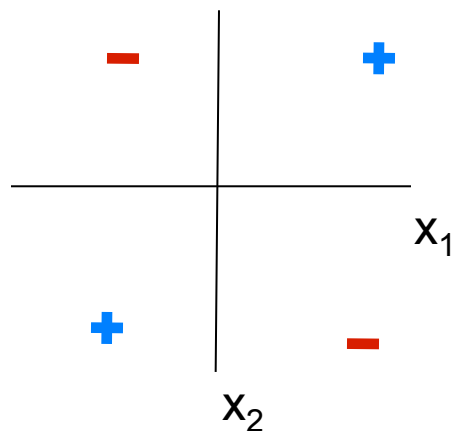
- Kernelized Perceptron:

- set $a^i=0$ for each example i
- For $t=1..T, i=1..n$:
 - $y = \text{sign}((\sum_k a^k \phi(x^k)) \cdot \phi(x^i))$
 $= \text{sign}(\sum_k a^k K(x^k, x^i))$
 - if $y \neq y^i$
 - $a^i += y^i$

Exactly the same
computations, but can use
 $K(u,v)$ to avoid enumerating
the features!!!

- set $a^i=0$ for each example i
- For $t=1..T$, $i=1..n$:
 - $y = \text{sign}(\sum_k a^k K(x^k, x^i))$
 - if $y \neq y^i$
 - $a^i += y^i$

x_1	x_2	y
1	1	1
-1	1	-1
-1	-1	1
1	-1	-1



$$K(u,v) = (u \bullet v)^2$$

e.g.,

$$\begin{aligned}
 K(x^1, x^2) &= K([1,1], [-1,1]) \\
 &= (1 \times -1 + 1 \times 1)^2 \\
 &= 0
 \end{aligned}$$

K	x^1	x^2	x^3	x^4
x^1	4	0	4	0
x^2	0	4	0	4
x^3	4	0	4	0
x^4	0	4	0	4

Initial:

- $a = [a^1, a^2, a^3, a^4] = [0,0,0,0]$

$t=1, i=1$

- $\sum_k a^k K(x^k, x^1) = 0 \times 4 + 0 \times 0 + 0 \times 4 + 0 \times 0 = 0$, $\text{sign}(0) = -1$

- $a^1 += y^1 \rightarrow a^1 += 1$, new $a = [1,0,0,0]$

$t=1, i=2$

- $\sum_k a^k K(x^k, x^2) = 1 \times 0 + 0 \times 4 + 0 \times 0 + 0 \times 4 = 0$, $\text{sign}(0) = -1$

$t=1, i=3$

- $\sum_k a^k K(x^k, x^3) = 1 \times 4 + 0 \times 0 + 0 \times 4 + 0 \times 0 = 4$, $\text{sign}(4) = 1$

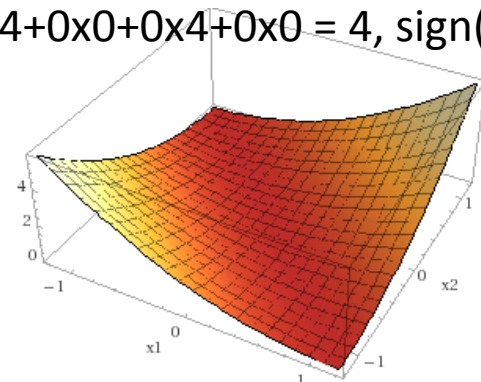
$t=1, i=4$

- $\sum_k a^k K(x^k, x^4) = 1 \times 0 + 0 \times 4 + 0 \times 0 + 0 \times 4 = 0$, $\text{sign}(0) = -1$

$t=2, i=1$

- $\sum_k a^k K(x^k, x^1) = 1 \times 4 + 0 \times 0 + 0 \times 4 + 0 \times 0 = 4$, $\text{sign}(4) = 1$

...



Converged!!!

- $y = \sum_k a^k K(x^k, x)$

$$\begin{aligned}
 &= 1 \times K(x^1, x) + 0 \times K(x^2, x) + 0 \times K(x^3, x) + 0 \times K(x^4, x) \\
 &= K(x^1, x) \\
 &= K([1,1], x) \quad (\text{because } x^1 = [1,1]) \\
 &= (x_1 + x_2)^2 \quad (\text{because } K(u,v) = (u \bullet v)^2)
 \end{aligned}$$

Common kernels

- Polynomials of degree exactly d

$$K(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v})^d$$

- Polynomials of degree up to d

$$K(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v} + 1)^d$$

- Gaussian kernels

$$K(\mathbf{u}, \mathbf{v}) = \exp\left(-\frac{\|\mathbf{u} - \mathbf{v}\|^2}{2\sigma^2}\right)$$

- Sigmoid

$$K(\mathbf{u}, \mathbf{v}) = \tanh(\eta \mathbf{u} \cdot \mathbf{v} + \nu)$$

- And many others: very active area of research!

Overfitting?

- Huge feature space with kernels, what about overfitting???
- Often robust to overfitting, e.g. if you don't make too many Perceptron updates
- SVMs (which we will see next) will have a clearer story for avoiding overfitting
- But everything overfits sometimes!!!
 - Can control by:
 - Choosing a better Kernel
 - Varying parameters of the Kernel (width of Gaussian, etc.)

Kernels in logistic regression

$$P(Y = 0 | \mathbf{X} = \mathbf{x}, \mathbf{w}, w_0) = \frac{1}{1 + \exp(w_0 + \mathbf{w} \cdot \mathbf{x})}$$

- Define weights in terms of data points:

$$\mathbf{w} = \sum_j \alpha^j \phi(\mathbf{x}^j)$$

$$\begin{aligned} P(Y = 0 | \mathbf{X} = \mathbf{x}, \mathbf{w}, w_0) &= \frac{1}{1 + \exp(w_0 + \sum_j \alpha^j \phi(\mathbf{x}^j) \cdot \phi(\mathbf{x}))} \\ &= \frac{1}{1 + \exp(w_0 + \sum_j \alpha^j K(\mathbf{x}^j, \mathbf{x}))} \end{aligned}$$

- Derive gradient descent rule on α^j, w_0
- Similar tricks for all linear models: SVMs, etc

What you need to know

- The kernel trick
- Derive polynomial kernel
- Common kernels
- Kernelized perceptron