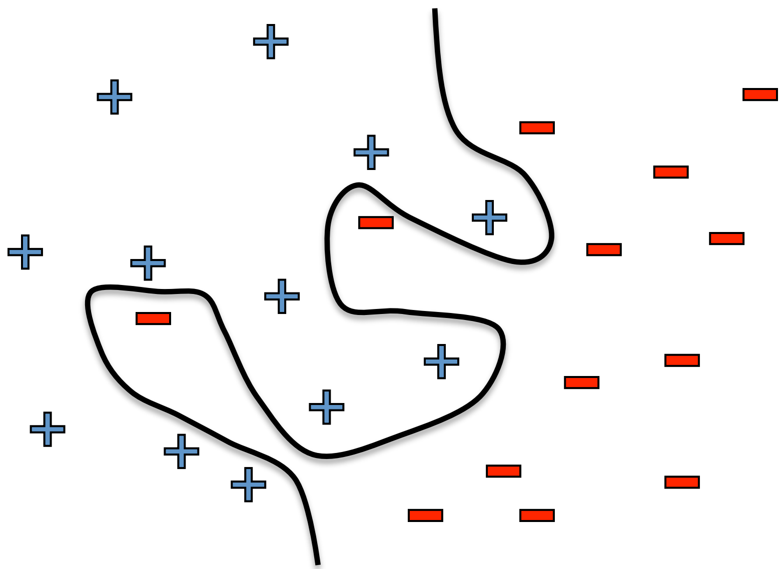
What if the data is not linearly separable?



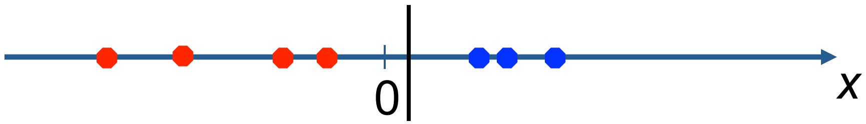
Use features of features of features of features….

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

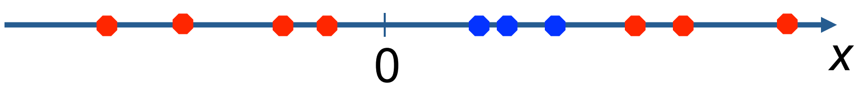
Feature space can get really large really quickly!

Non-linear features: 1D input

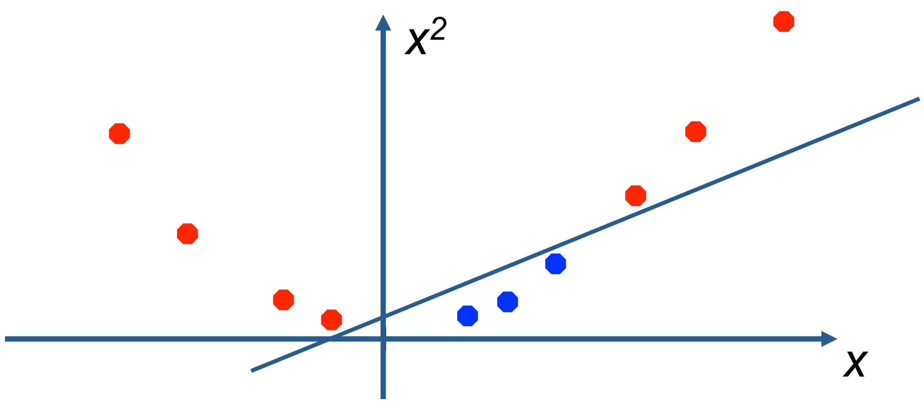
- Datasets that are linearly separable with some noise work out great:



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



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

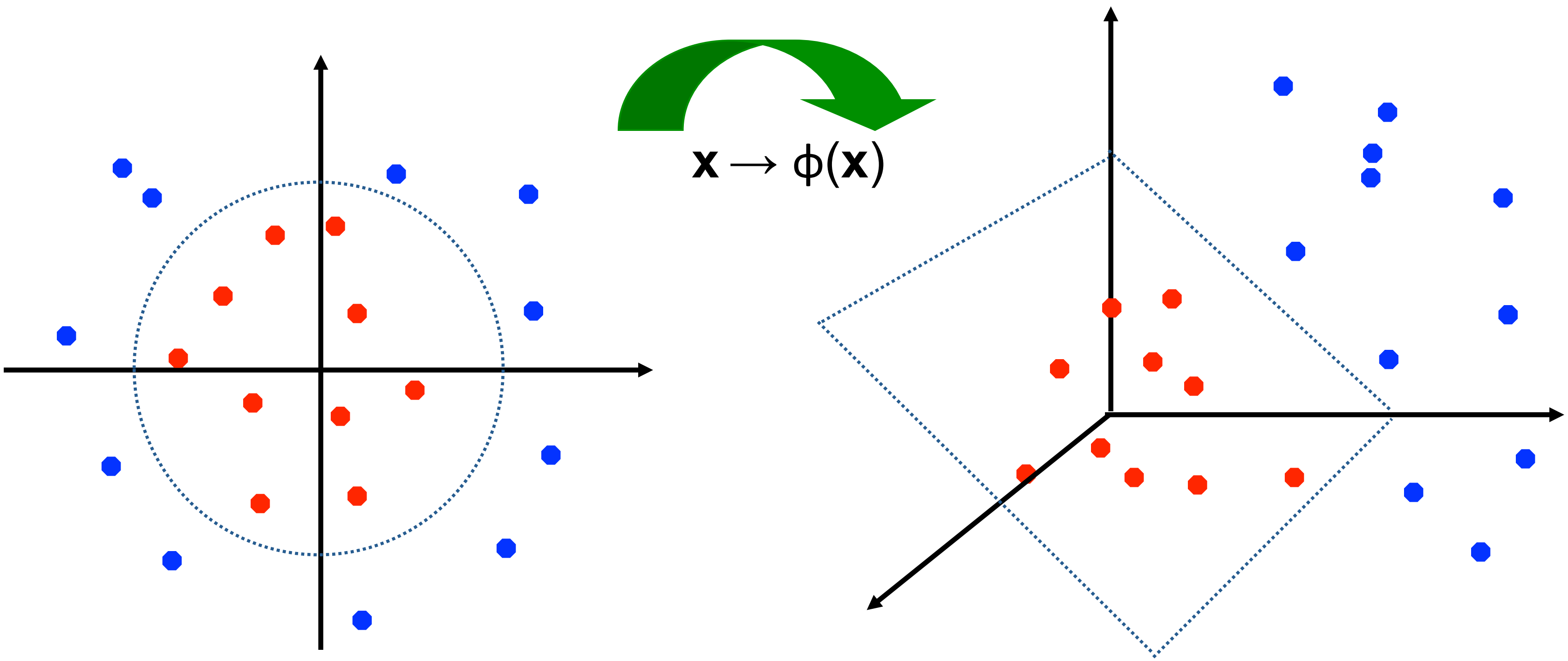


Feature spaces

- General idea: map to higher dimensional space
  - if **x** is in R^n, then $\phi(\mathbf{x})$ is in R^m for m>n
  - Can now learn feature weights **w** in R^m and predict:

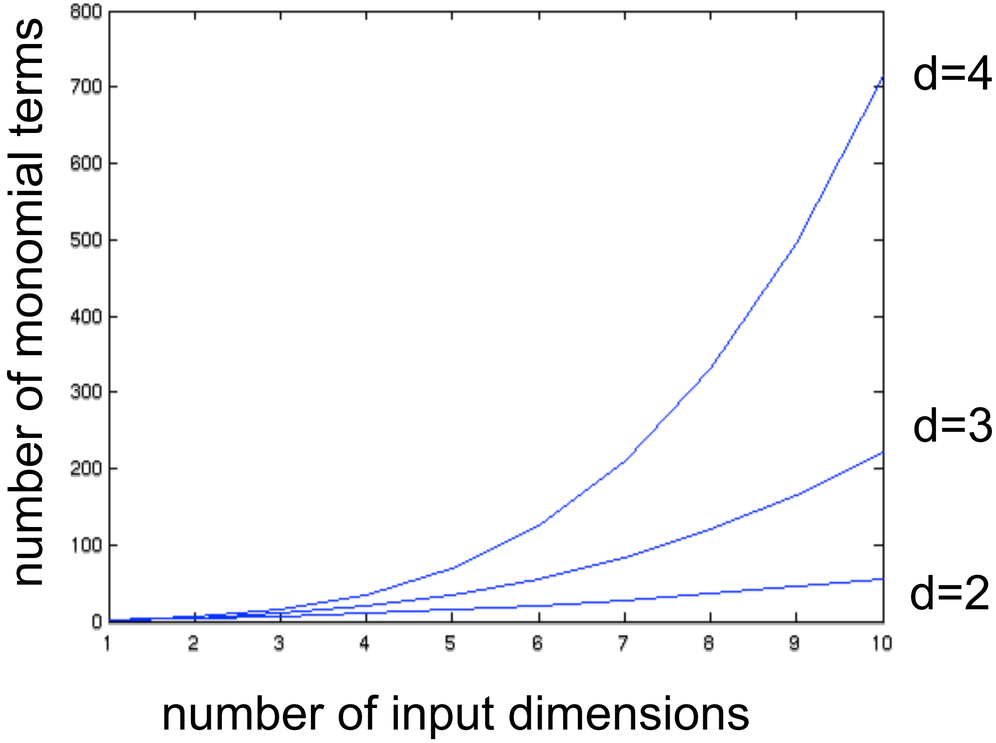
$$y = sign(\mathbf{w} \cdot \phi(\mathbf{x}))$$

  - 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

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

d=2

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

For any *d (we will skip proof):*

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

- 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 $K(\mathbf{x}_i,\mathbf{x}_j) = \boldsymbol{\phi}(\mathbf{x}_i) \bullet \boldsymbol{\phi}(\mathbf{x}_j)$:

  $K(\mathbf{u},\mathbf{v})=(1 + \mathbf{u}\bullet\mathbf{v})^2, = 1 + u_1^2 v_1^2 + 2\, u_1 v_1\, u_2 v_2 + u_2^2 v_2^2 + 2u_1 v_1 + 2u_2 v_2 =$

  $= [1,\, u_1^2,\, \sqrt{2}\, u_1 u_2,\;\; u_2^2,\, \sqrt{2}u_1,\, \sqrt{2}u_2] \bullet [1,\;\; v_1^2,\, \sqrt{2}v_1 v_2,\, v_2^2,\, \sqrt{2}v_1,\, \sqrt{2}v_2] =$

  $= \boldsymbol{\phi}(\mathbf{u}) \bullet \boldsymbol{\phi}(\mathbf{v})$, where $\boldsymbol{\phi}(\mathbf{x}) = [1,\;\; x_1^2,\, \sqrt{2}\, x_1 x_2,\;\; x_2^2,\;\; \sqrt{2}x_1,\, \sqrt{2}x_2]$

- 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) = Φ(u) • Φ(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:
    - $y = sign(w \cdot \phi(x^i))$
    - if $y \neq y^i$
      - $w = w + y^i \phi(x^i)$
      - a^i += y^i
  - At all times during learning:

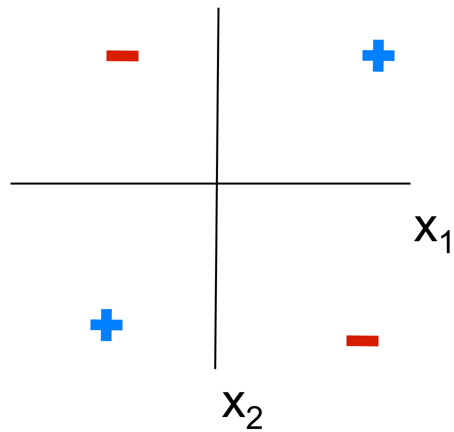
$$w = \sum_k a^k \phi(x^k)$$

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

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.,
$K(x^1,x^2)$
$= K([1,1],[-1,1])$
$= (1\text{x-}1+1\text{x}1)^2$
$= 0$

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

- $\Sigma_k a^k K(x^k,x^1)$ = 0x4+0x0+0x4+0x0 = 0, sign(0)=-1
- $a^1 \mathrel{+}= y^1 \rightarrow a^1 \mathrel{+}=1$, new a= [1,0,0,0]

t=1,i=2

- $\Sigma_k a^k K(x^k,x^2)$ = 1x0+0x4+0x0+0x4 = 0, sign(0)=-1

t=1,i=3

- $\Sigma_k a^k K(x^k,x^3)$ = 1x4+0x0+0x4+0x0 = 4, sign(4)=1

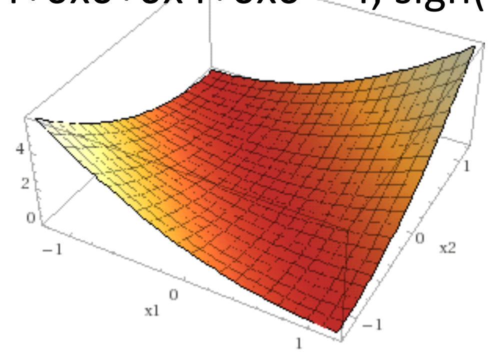
t=1,i=4

- $\Sigma_k a^k K(x^k,x^4)$ = 1x0+0x4+0x0+0x4 = 0, sign(0)=-1

t=2,i=1

- $\Sigma_k a^k K(x^k,x^1)$ = 1x4+0x0+0x4+0x0 = 4, sign(4)=1

...



Converged!!!

- $y=\Sigma_k a^k K(x^k,x)$
  $= 1\times K(x^1,x)+0\times K(x^2,x)+0\times K(x^3,x)+0\times K(x^4,x)$
  $= K(x^1,x)$
  $= K([1,1],x)$ (because $x^1=[1,1]$)
  $= (x_1+x_2)^2$ (because $K(u,v) = (u\bullet v)^2$)

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

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

- Sigmoid

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

- And many others: very active area of research!

Kernel of Infinite Dimensional Transform

infinite dimensional $\Phi(\mathbf{x})$? Yes, if $K(\mathbf{x}, \mathbf{x}')$ **efficiently computable**!

when $\mathbf{x} = (x)$, $K(x, x')$

$$\begin{aligned} K(x,x') &= \exp(-(x-x')^2) \\ &= \exp(-(x)^2)\exp(-(x')^2)\exp(2xx') \\ &\overset{\text{Taylor}}{=} \exp(-(x)^2)\exp(-(x')^2)\left(\sum_{i=0}^{\infty}\frac{(2xx')^i}{i!}\right) \\ &= \sum_{i=0}^{\infty}\left(\exp(-(x)^2)\exp(-(x')^2)\sqrt{\frac{2^i}{i!}}\sqrt{\frac{2^i}{i!}}(x)^i(x')^i\right) \\ &= \Phi(x)^T\Phi(x') \end{aligned}$$

with infinite dimensional $\Phi(x) = \exp(-x^2) \cdot \left(1, \sqrt{\frac{2}{1!}}x, \sqrt{\frac{2^2}{2!}}x^2, \ldots\right)$

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:

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

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

$$=\frac{1}{1+exp(w_0+\sum_j \alpha^j K(\mathbf{x}^j,\mathbf{x}))}$$

- 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