What if the data is not linearly separable?

Use features of features of features of features...

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
Mapping to a higher dimensional space

Input feature space

Higher dimensional space

Polynomial of degree d

What can go wrong?
Higher order polynomials

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

- \( d \) – degree of polynomial
- \( m \) – input features

- \( d = 6, m = 100 \) about 1.6 billion terms
- Diagram shows the number of terms grows fast with increasing input dimensions and degree of polynomial.
Mapping to a higher dimensional space

Input feature space

Polynomial of degree d

Higher dimensional space

\[ \langle w \cdot \phi(x) \rangle \]
Efficient dot-product of polynomials

Polynomials of degree exactly $d$

$d=1$

$$\phi(u) \cdot \phi(v) = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = u_1v_1 + u_2v_2 = u \cdot v$$

$d=2$

$$\phi(u) \cdot \phi(v) = \begin{pmatrix} u_1^2 \\ u_1u_2 \\ u_2u_1 \\ u_2^2 \end{pmatrix} \cdot \begin{pmatrix} v_1^2 \\ v_1v_2 \\ v_2v_1 \\ v_2^2 \end{pmatrix} = u_1^2v_1^2 + 2u_1u_2v_1v_2 + u_2^2v_2^2$$

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.

\[ K(u,v) = \phi(u) \cdot \phi(v) \]

• Example:

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

Need to show that \( K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j) \):

\[ K(u,v) = (1 + u \cdot v)^2 = 1 + u_1^2v_1^2 + 2u_1v_1u_2v_2 + u_2^2v_2^2 + 2u_1v_1 + 2u_2v_2 = \]

\[ = [1, u_1^2, \sqrt{2}u_1u_2, u_2^2, \sqrt{2}u_1, \sqrt{2}u_2] \cdot [1, v_1^2, \sqrt{2}v_1v_2, v_2^2, \sqrt{2}v_1, \sqrt{2}v_2] = \]

\[ = \phi(u) \cdot \phi(v), \text{ where } \phi(x) = [1, x_1^2, \sqrt{2}x_1x_2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2] \]

• Thus, a kernel function implicitly maps data to a high-dimensional space (without the need to compute each \( \phi(x) \) 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 $t=1..T$, $i=1..n$:
  - $y = \text{sign} \left( \sum_k a^k K(x^k, x^i) \right)$
  - if $y \neq y^i$
    - $a^i += y^i$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

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

- $K(x^1, x^2) = K([1,1],[-1,1]) = (1x-1+1x1)^2 = 0$

<table>
<thead>
<tr>
<th>K</th>
<th>$x^1$</th>
<th>$x^2$</th>
<th>$x^3$</th>
<th>$x^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^1$</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>$x^2$</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>$x^3$</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>$x^4$</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Initial:
- $a = [a^1, a^2, a^3, a^4] = [0,0,0,0]$
- $t=1,i=1$
  - $\sum_k a^k K(x^k, x^1) = 0x4+0x0+0x4+0x0 = 0, \text{sign}(0)=-1$
  - $a^1 += y^1 \rightarrow a^1+=1$, new $a = [1,0,0,0]$
- $t=1,i=2$
  - $\sum_k a^k K(x^k, x^2) = 1x0+0x4+0x0+0x4 = 0, \text{sign}(0)=-1$
- $t=1,i=3$
  - $\sum_k a^k K(x^k, x^3) = 1x4+0x0+0x4+0x0 = 4, \text{sign}(4)=1$
- $t=1,i=4$
  - $\sum_k a^k K(x^k, x^4) = 1x0+0x4+0x0+0x4 = 0, \text{sign}(0)=1$
- $t=2,i=1$
  - $\sum_k a^k K(x^k, x^1) = 1x4+0x0+0x4+0x0 = 4, \text{sign}(4)=1$

... 

Converged!!!
Common kernels

- Polynomials of degree exactly $d$
  \[ K(u, v) = (u \cdot v)^d \]
- Polynomials of degree up to $d$
  \[ K(u, v) = (u \cdot v + 1)^d \]
- Gaussian kernels
  \[ K(u, v) = \exp \left( -\frac{||u - v||}{2\sigma^2} \right) \]
- Sigmoid
  \[ K(u, v) = \tanh(\eta u \cdot v + \nu) \]
- And many others: very active area of research!
Kernels in logistic regression

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

• Define weights in terms of data points:
  \[ w = \sum_j \alpha^j \phi(x^j) \]

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

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

• Derive gradient descent rule on \( \alpha^j, w_0 \)
• Similar tricks for all linear models: SVMs, etc
What you need to know

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