Introduction

- For non-linearly separable classes, performance of even the best linear classifier might not be good
- Thus we will remap feature vectors to new space where they are (almost) linearly separable

Outline:
- Multiple layers of neurons
  - Backpropagation
  - Sizing the network
- Polynomial remapping
- Gaussian remapping (radial basis functions)
- Efficiency issues (support vector machines)
- Other nonlinear classifiers (decision trees)

Getting Started: The XOR Problem

• Can’t represent with a single linear separator, but can with intersection of two:
  \[ g_1(x) = 1 \cdot x_1 + 1 \cdot x_2 - 1/2 \]
  \[ g_2(x) = 1 \cdot x_1 + 1 \cdot x_2 - 3/2 \]
  \[ \omega_1 = \{ x \in \mathbb{R}^2 : g_1(x) > 0 \text{ AND } g_2(x) < 0 \} \]
  \[ \omega_2 = \{ x \in \mathbb{R}^2 : g_1(x), g_2(x) < 0 \text{ OR } g_1(x), g_2(x) > 0 \} \]

Let \( y_i = \begin{cases} 0 & \text{if } g_i(x) < 0 \\ 1 & \text{otherwise} \end{cases} \)

<table>
<thead>
<tr>
<th>Class</th>
<th>( x_1, x_2 )</th>
<th>( g_1(x) )</th>
<th>( y_1 )</th>
<th>( g_2(x) )</th>
<th>( y_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \omega_1 )</td>
<td>B: (0,1)</td>
<td>1/2</td>
<td>1</td>
<td>-1/2</td>
<td>0</td>
</tr>
<tr>
<td>( \omega_1 )</td>
<td>C: (1,0)</td>
<td>1/2</td>
<td>1</td>
<td>-1/2</td>
<td>0</td>
</tr>
<tr>
<td>( \omega_2 )</td>
<td>A: (0,0)</td>
<td>-1/2</td>
<td>0</td>
<td>-3/2</td>
<td>0</td>
</tr>
<tr>
<td>( \omega_2 )</td>
<td>D: (1,1)</td>
<td>3/2</td>
<td>1</td>
<td>1/2</td>
<td>1</td>
</tr>
</tbody>
</table>

Now feed \( y_1, y_2 \) into:
\[ g(y) = 1 \cdot y_1 - 2 \cdot y_2 - 1/2 \]
Getting Started: The XOR Problem
(cont’d)

- In other words, we remapped all vectors \( x \) to \( y \) such that the classes are linearly separable in the new vector space

\[
\sum_{i} \omega_i x_i = 1
\]

\( \omega_1 = -1/2 \)

\( \omega_2 = -3/2 \)

\( \omega_3 = 1 \)

\( \omega_4 = -1 \)

Input Layer

Hidden Layer

Output Layer

- This is a two-layer perceptron or two-layer feedforward neural network

- Each neuron outputs 1 if its weighted sum exceeds its threshold, 0 otherwise

What Else Can We Do with Two Layers?
(cont’d)

- Define the \( p \)-dimensional unit hypercube as

\[
H_p = \{ [y_1, \ldots, y_p]^T \in \mathbb{R}^p, y_i \in [0, 1] \forall i \}
\]

- A hidden layer with \( p \) neurons maps an \( \ell \)-dim vector \( x \) to a \( p \)-dim vector \( y \) whose elements are corners of \( H_p \), i.e. \( y_i \in \{0, 1\} \forall i \)

- Each of the \( p \) neurons corresponds to an \( \ell \)-dim hyperplane

- The intersection* of the (pos. or neg.) half-spaces from these \( p \) hyperplanes maps to a vertex of \( H_p \)

- If the classes of \( H_p \)'s vertices are linearly separable, then a perfect two-layer network exists

- I.e. a 2-layer network can separate classes consisting of adjacent polyhedra

What Else Can We Do with Two Layers?
(cont’d)

- With two-layer networks, there exist unions of polyhedra not linearly separable on \( H_p \)

- I.e. there exist assignments of classes to points on \( H_p \) that are not linearly separable

- Solution: Add a second hidden layer of \( q \) neurons to partition \( H_p \) into regions based on class

- Output layer combines appropriate regions

- E.g. including 110 from Slide 6 in \( \omega_1 \) is possible using procedure similar to XOR solution

- In general, can always use simple procedure of isolating each \( \omega_1 \) node in \( H_p \) with its own second-layer hyperplane and taking disjunction

- Thus, can use 3-layer network to perfectly classify any union of polyhedral regions

*Also known as polyhedra.
The Backpropagation Algorithm

- A popular way to train a neural network

- Assume the architecture is **fixed** and **complete**
  - $k_r = \text{number of nodes in layer } r$ (could have $k_L > 1$, $L = \text{number of layers}$)
  - $w_{ji}^r = \text{weight from neuron } i \text{ in layer } r-1 \text{ to neuron } j \text{ in layer } r$
  - $v_j^r = \sum_{k=1}^{k_r-1} w_{jk}^r y_k^{r-1} + w_{j0}^r$
  - $y_j^r = f(v_j^r) = \text{output of neuron } j \text{ in layer } r$

- During training we'll attempt to minimize a cost function, so use differentiable activation function, $f$, e.g.:
  - $f(v) = \frac{1}{1 + e^{-av}} \in [0, 1]$
  - OR
  - $f(v) = c \tanh (av) \in [-c, c]$

The Backpropagation Algorithm

Another Picture

The Backpropagation Algorithm

Intuition

- Recall derivation of Perceptron update rule:
  - Cost function:
    $$ U(w) = \sum_{i=1}^{\ell} (w_i(t+1) - w_i(t))^2 + \eta \left( y(t) - \sum_{i=1}^{\ell} w_i(t+1)x_i(t) \right)^2 $$
  - Take gradient w.r.t. $w(t+1)$, set to 0, approximate, and solve:
    $$ w_i(t+1) = w_i(t) + \eta \left( y(t) - \sum_{i=1}^{\ell} w_i(t)x_i(t) \right) x_i(t) $$
The Backpropagation Algorithm

Intuition: Output Layer

- Now use similar idea with jth node of output layer (layer $L$):
  
  - Cost function:
    
    $$ U \left( w^L_j \right) = \sum_{k=1}^{k-1} \left( w^L_{jk}(t+1) - w^L_{jk}(t) \right)^2 + $$
    
    $$ \eta \left[ \frac{\text{correct } y_j(t) - f \left( \sum_{k=1}^{k-1} w^L_{jk}(t+1)y^{L-1}_k(t) \right)}{y^L_k(t) \text{ with } w(t+1)} \right]^2 $$
    
    - Take gradient w.r.t. $w^L_j(t+1)$ and set to 0:
      
      $$ 0 = 2 \left( w^L_{jk}(t+1) - w^L_{jk}(t) \right) - 2\eta \left[ y_j(t) - f \left( \sum_{k=1}^{k-1} w^L_{jk}(t+1)y^{L-1}_k(t) \right) \right] $$
      
      $$ \cdot f' \left( \sum_{k=1}^{k-1} w^L_{jk}(t+1)y^{L-1}_k(t) \right) y^{L-1}_k(t) $$

The Backpropagation Algorithm

Intuition: The Other Layers

- How can we compute the “error term” for the hidden layers $r < L$ when there is no “target vector” $y$ for these layers?
  
  - Instead, propagate back error values from output layer toward input layers, scaling with the weights
  
  - Scaling with the weights characterizes how much of the error term each hidden unit is “responsible for”:
    
    $$ w^r_{jk}(t+1) = w^r_{jk}(t) + \eta y^{r-1}_k(t) \delta^r_j(t) $$
    
    where
    
    $$ \delta^r_j(t) = f' \left( v^r_j(t) \right) \sum_{k=1}^{h+1} \delta^r_k(t) w^r_{jk}(t+1) $$

  - Derivation comes from computing gradient of cost function w.r.t. $w^L_j(t+1)$ via chain rule

### Example

<table>
<thead>
<tr>
<th>target = y</th>
<th>trial 1: a = 1, b = 0, y = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(x) = 1/1 (1 + exp(-x))</td>
<td>trial 2: a = 0, b = 1, y = 0</td>
</tr>
</tbody>
</table>

13

14

15

16
The Backpropagation Algorithm

Issues

- When to stop iterating through training set?
  - When weights don’t change much
  - When value of cost function is small enough
  - Must also avoid overtraining

Variations

- Can smooth oscillations of weight vector with momentum term $\alpha < 1$ that tends to keep it moving in the same direction as previous trials:
  \[
  \Delta w^r_j(t + 1) = \alpha \Delta w^r_j(t) + \eta y_{r-1}^r(t) \delta^r_j(t)
  \]
  \[
  w^r_j(t + 1) = w^r_j(t) + \Delta w^r_j(t + 1)
  \]

- Different training modes:
  - On-line (what we presented) has more randomness during training (might avoid local minima)
  - Batch mode (in text) averages gradients, giving better estimates and smoother convergence:
    * Before updating, first compute $\delta^r_j(t)$ for each vector $x_t$, $t = 1, \ldots, N$
    \[
    w^r_j(\text{new}) = w^r_j(\text{old}) + \eta \sum_{t=1}^{N} \delta^r_j(t) y^{r-1}(t)
    \]

The Backpropagation Algorithm

Issues (cont’d)

- How to set learning rate $\eta$ ($\mu$ in text)?
  - Small values slow convergence
  - Large values might overshoot minimum
  - Can adapt it over time, as with perceptron

- Might hit local minima that aren’t very good; try re-running with new random weights
  - Starting with weights near 0 $\Rightarrow$ output almost linear function of inputs $\Rightarrow$ error surface almost quadratic, reducing chances of bad local min

Variations (cont’d)

- A Recurrent network feeds output of e.g. layer $r$ to the input of some earlier layer $r' < r$
  - Allows predictions to be influenced by past predictions (for e.g. sequence data)

(a) Feedforward network

(b) Recurrent network
Variations
(cont’d)

• Can implement a “backprop” scheme with EG

• Other nonlinear optimization schemes:
  – Conjugate gradient
  – Newton’s method
  – Genetic algorithms
  – Simulated annealing

• Other cost functions, e.g. cross-entropy:
  \[ - \sum_{k=1}^{k_L} \left( \text{label} \left( y_k(t) \right) \ln \left( y_k^L(t) \right) + (1 - y_k(t)) \ln \left( 1 - y_k^L(t) \right) \right) \]

“blows up” if \( y_k(t) \approx 1 \) and \( y_k^L(t) \approx 0 \) or vice-versa (Section 4.8)

Sizing the Network

Before training, need to choose appropriate number of layers and size of each layer

  – Too small: Cannot learn what features make same classes similar and separate classes different
  – Too large: Adapts to details of the particular training set and cannot generalize well (called overfitting)
  – Also, increasing size increases complexity to train and use

• Approaches:
  – **Analytical methods**: Use knowledge of data to est. number of needed layers and neurons
  – **Pruning techniques**: Start with a large network and periodically remove weights and neurons that don’t affect output much
  – **Constructive techniques**: Start with small netw. and periodically add neurons and wts

Sizing the Network
Pruning Techniques [Also see Bishop, Sec. 9.5]

• **Approach 1**: Train with backprop, periodically computing effect of varying \( w_i \) on cost func:

  – From Taylor series expansion (p. 109),
  \[ \frac{\delta J}{\delta w_i} \approx \frac{1}{2} \sum_i h_{ii} \delta w_i^2 \quad \text{where} \quad h_{ii} = \frac{\partial^2 J}{\partial^2 w_i} \]

  – If \( h_{ii} w_i^2 / 2 \) (saliency factor) small, then \( w_i \) doesn’t have much impact and is removed
  – Now continue training with backprop

• Example (Sec 4.10): 480 wts pruned to 25

Sizing the Network
Pruning Techniques (cont’d)
[Also see Bishop, Sec. 9.5]

• **Approach 2**: Train with backprop, but add to the cost function \( J \) a term that penalizes large weights:

  \[ J' = J + \text{penalty} \]

  – If \( w_i \)’s contribution to network output is small, then its share of \( J \) is small
  – So penalty term dominates \( w_i \)’s share of \( J' \), driving it down
  – Periodically prune weights that get too low
Generalized Linear Classifiers
Section 4.12

• In XOR problem, used linear threshold funcs. in hidden layer to map non-lin. sep. classes to new space where they were lin. sep.
• Output layer gave sep. hyperplane in new space
• Replace hidden-layer lin. thresh. funcs. with family of nonlinear functions \(f_i: \mathbb{R}^\ell \to \mathbb{R}, \ i = 1, \ldots, k\)
• Hidden layer maps \(x \in \mathbb{R}^\ell\) to \(y = [f_1(x), \ldots, f_k(x)]^T\) and output layer finds separating hyperplane:

### Cover’s Theorem

(Justifies more features/higher dimensional space)

• For arbitrary set of \(N\) points, there are \(2^N\) ways to classify them into \(\omega_1\) and \(\omega_2\) (i.e. \(2^N\) dichotomies)
• If classification done by a single hyperplane, then the number of linear dichotomies is

\[
O(N, \ell) = 2 \sum_{i=0}^{\ell} \binom{N-1}{i}
\]

= \(2^N\) if \(N \leq \ell + 1\), else < \(2^N\)

14 linear dichotomies 8 linear dichotomies

• Thus if dimensionality \(\ell \geq N - 1\) then a perfect separating hyperplane is guaranteed to exist
• Otherwise \((N > \ell + 1)\) the fraction of dichotomies that are linear dichotomies is

\[
P = \frac{1}{2^N - 1} \sum_{i=0}^{\ell} \binom{N-1}{i}
\]

Let \(N = r(\ell + 1)\)

• Approximate \(g(x)\) by linear combination of up to order \(r\) polynomials over components of \(x\)
• E.g. for \(r = 2\)

\[
g(x) = w_0 + \sum_{i=1}^{\ell} w_i x_i + \sum_{i=1}^{\ell} \sum_{m=i+1}^{\ell} w_{im} x_i x_m + \sum_{i=1}^{\ell} \sum_{k=1}^{\ell} w_{ik} x_i^2, \quad k = \ell(\ell + 3)/2
\]

• For \(\ell = 2\), \(x = [x_1, x_2]^T\) and

\[
y = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]^T
\]

\[
g(x) = w^T y + w_0
\]

\(w^T = [w_1, w_2, w_{12}, w_{11}, w_{22}]\)
Generalized Linear Classifiers
Polynomial Classifiers
(cont’d)

• In general, will use all terms of form \(x_1^{p_1}x_2^{p_2} \cdots x_\ell^{p_\ell}\)
  for all \(p_1 + \cdots + p_\ell \leq r\)

• This gives size of \(y\) to be
  \[k = \frac{(\ell + r)!}{r!\ell!},\]

so time to classify and update exponential in \((\ell + r)\)

• Fortunately, EG’s loss bound logarithmic in \(k\),
  though run time still (in general) linear in \(k\)
  – Special cases can be made efficient with exact or approximate output computation

Example: XOR

• \(y = [x_1, x_2, x_1x_2]^T\)

  \[\begin{align*}
  \omega_1 & : [0, 1]^T, [0, 1, 0]^T \\
  \omega_1 & : [1, 0]^T, [1, 0, 0]^T \\
  \omega_2 & : [0, 0]^T, [0, 0, 0]^T \\
  \omega_2 & : [1, 1]^T, [1, 1, 1]^T
  \end{align*}\]

  \[
g(y) = y_1 + y_2 - 2y_3 - \frac{1}{4} \\
g(x) = -\frac{1}{4} + x_1 + x_2 - 2x_1x_2
\]

\[
\begin{array}{c}
> 0 \Rightarrow x \in \omega_1 \\
< 0 \Rightarrow x \in \omega_2
\end{array}
\]

Radial Basis Function Networks

• Argument of func. \(f_i\) is \(x\)’s Euclidian distance from designated center \(c_i\), e.g.
  \[f_i(x) = \exp\left(-\frac{\|x - c_i\|^2}{2\sigma_i^2}\right)\]

• So
  \[g(x) = w_0 + \sum_{i=1}^k w_i \exp\left(-\frac{(x - c_i)^T(x - c_i)}{2\sigma_i^2}\right)\]

• Exponential decrease in increased distance gives a very localized activation response

• Related to nearest neighbor approaches since only \(f_i\)’s with centers near \(x\) will have significant output

Example: XOR

• \(c_1 = [1, 1]^T, c_2 = [0, 0]^T, f_i(x) = \exp\left(-\|x - c_i\|^2\right)\)

  \[
  \begin{align*}
  \omega_1 (A) & : [0, 1]^T, [0.368, 0.368]^T \\
  \omega_1 (A) & : [1, 0]^T, [0.368, 0.368]^T \\
  \omega_2 (B) & : [0, 0]^T, [0.135, 1]^T \\
  \omega_2 (B) & : [1, 1]^T, [1, 0.135]^T
  \end{align*}\]

  \[
g(y) = y_1 + y_2 - 1 \\
g(x) = -1 + e^{-\|x - c_1\|^2} + e^{-\|x - c_2\|^2}
\]

\[
\begin{array}{c}
< 0 \Rightarrow x \in \omega_1 \\
> 0 \Rightarrow x \in \omega_2
\end{array}
\]
Generalized Linear Classifiers
Radial Basis Function Networks
Choosing the Centers

• Randomly select from the training set
  − Might work well if training set representative of probability distribution over data

• Learn the $c_i$'s and $\sigma_i^2$'s via gradient descent
  − Frequently computationally complex

• First cluster the data (Chapters 11–16) and use results to find centers

• Use methods similar to constructive and pruning techniques when sizing neural network
  − Add new center when perceived as needed, delete unnecessary centers
  − E.g. if new input vector $x$ far from all current centers and error high, then new center necessary, so add $x$ as new center

Support Vector Machines
[See refs. on slides page]

• Introduced in 1992

• State-of-the-art technique for classification and regression

• Techniques can also be applied to e.g. clustering and principal components analysis

• Similar to polynomial classifiers and RBF networks in that it remaps inputs and then finds a hyperplane
  − Main difference is how it works

• Features of SVMs:
  − Maximization of margin
  − Duality
  − Use of kernels
  − Use of problem convexity to find classifier (often without local minima)

Support Vector Machines
Margins

• A hyperplane's margin $\gamma$ is the shortest distance from it to any training vector

• Intuition: larger margin $\Rightarrow$ higher confidence in classifier's ability to generalize
  − Guaranteed generalization error bound in terms of $1/\gamma^2$

• Definition assumes linear separability (more general definitions exist that do not)

Support Vector Machines
Large Margin Perceptron Algorithm

• $w(0) \leftarrow 0, b(0) \leftarrow 0, k \leftarrow 0, R \leftarrow \max_{1 \leq i \leq N} ||x_i||_2$
  ($R =$ radius of ball centered at origin containing training vectors), $y_i \in \{-1, +1\} \forall i$

• Update slope same as before, update offset differently

• While mistakes are made on training set
  − For $i = 1$ to $N$ (=$\#$ training vectors)
    * If $y_i (w_k \cdot x_i + b_k) \leq 0$
      $w_{k+1} \leftarrow w_k + \eta y_i x_i$
      $b_{k+1} \leftarrow b_k + \eta y_i R^2$
      $k \leftarrow k + 1$

• Final predictor: $h(x) = \text{sgn}(w_k \cdot x + b_k)$
Support Vector Machines

Duality

- Another way of representing predictor:
  \[ h(x) = \text{sgn} \left( w \cdot x + b \right) = \text{sgn} \left( \sum_{i=1}^{N} (\alpha_i y_i x_i) \cdot x + b \right) \]
  \( = \text{sgn} \left( \sum_{i=1}^{N} \alpha_i y_i (x_i \cdot x) + b \right) \)

\( (\alpha_i = \# \text{ mistakes on } x_i, \eta > 0 \text{ ignored}) \)

- So perceptron alg has equivalent dual form:
  \( \alpha \leftarrow 0, b \leftarrow 0, R \leftarrow \max_{1 \leq i \leq N} \| x_i \|_2 \)

- While mistakes are made in For loop
  - For \( i = 1 \) to \( N \) (= # training vectors)
    - If \( y_i \left( \sum_{j=1}^{N} \alpha_j y_j (x_j \cdot x_i) + b \right) \leq 0 \)
      - \( \alpha_i \leftarrow \alpha_i + 1 \)
      - \( b \leftarrow b + y_i R^2 \)
  - Now data only in dot products

Kernels

- Duality lets us remap to many more features!
- Let \( \phi : \mathbb{R}^d \rightarrow F \) be nonlinear map of f.v.s, so
  \[ h(x) = \text{sgn} \left( \sum_{i=1}^{N} \alpha_i y_i (\phi(x_i) \cdot \phi(x)) + b \right) \]

- Can we compute \( \phi(x_i) \cdot \phi(x) \) without evaluating \( \phi(x_i) \) and \( \phi(x) \)? **YES!**
- \( x = [x_1, x_2], z = [z_1, z_2] \):
  \[ (x \cdot z)^2 = (x_1 z_1 + x_2 z_2)^2 \]
  \[ = x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2 \]
  \[ = \left[ x_1^2, x_2^2, \sqrt{2} x_1, x_2 \right] \cdot [z_1^2, z_2^2, \sqrt{2} z_1 z_2] \]
  \( \phi(x) \)

- LHS requires 2 mults + 1 squaring to compute, RHS takes 3 mults
- In general, \( (x \cdot z)^d \) takes \( \ell \) mults + 1 expon., vs. \( \left( \frac{\ell + d - 1}{d} \right) \geq \left( \frac{4 + d - 1}{d} \right) \) mults if compute \( \phi \) first

Support Vector Machines

Finding a Hyperplane

- Can show [Cristianini & Shawe-Taylor] that if data linearly separable in remapped space, then get maximum margin classifier by minimizing \( w \cdot w \) subject to \( y_i (w \cdot x_i + b) \geq 1 \)
- Can reformulate this into a convex quadratic program, which can be solved optimally, i.e. won’t encounter local optimas
- Can always find a kernel that will make training set linearly separable, but beware of choosing a kernel that is too powerful (overfitting)
- If kernel doesn’t separate, can optimize subject to \( y_i (w \cdot x_i + b) \geq 1 - \xi_i \), where \( \xi_i \) are slack variables that soften the margin (can still solve optimally)
- If number of training vectors is very large, may opt to approximately solve these problems to save time and space
- Use e.g. gradient ascent and sequential minimal optimization (SMO) [Cristianini & Shawe-Taylor]
- When done, can throw out non-SVs

Kernels (cont’d)

- In general, a kernel is a function \( K \) such that \( \forall x, z, K(x, z) = \phi(x) \cdot \phi(z) \)
- Typically start with kernel and take the feature mapping that it yields
- E.g. Let \( \ell = 1, x = x, z = z, K(x, z) = \sin(x - z) \)
- By Fourier expansion,
  \[ \sin(x - z) = a_0 + \sum_{n=1}^{\infty} a_n \sin(nx) \sin(nz) \]
  \[ + \sum_{n=1}^{\infty} a_n \cos(nx) \cos(nz) \]
  for Fourier coefficients \( a_0, a_1, \ldots \)
- This is the dot product of two infinite sequences of nonlinear functions:
  \[ \{ \phi_i(x) \}_{i=0}^{\infty} = [1, \sin(x), \cos(x), \sin(2x), \cos(2x), \ldots] \]
- I.e. there are an infinite number of features in this remapped space!
**Decision Trees** [Also Mitchell, ch. 3]

- Start at root and work down tree until leaf reached; output that classification
- E.g. $x = [1/2, 1/4]^T$ classified as $\omega_3$

---

**Learning Good Trees** [Also Mitchell, ch. 3]

- Feature at root is one that yields highest information gain, equivalent to max. reduction of entropy (class impurity) in training data:
  
  $S = \text{set of } N \text{ feature vectors} \quad N_i = \text{number in } \omega_i$

  
  $p_i = N_i/N \quad \text{Ent}(S) = \sum_{i=1}^{M} -p_i \log_2(p_i)$

- First partition along dimensions into set $A$ of features and places where classes change, e.g.
  
  $A = \{(x_1, 0), (x_1, 1/4), (x_1, 1/2), (x_1, 3/4), (x_2, 0), (x_2, 1/2), (x_2, 3/4)\}$

- For $a = (x_i, b) \in A$, define
  
  $S_a = \{x \in S : x_i > b\} \quad S'_a = \{x \in S : x_i \leq b\}$

  $\text{Gain}(S, a) = \text{Ent}(S) - \left( \frac{|S_a|}{|S|} \text{Ent}(S_a) + \frac{|S'_a|}{|S|} \text{Ent}(S'_a) \right)$

  $= 0 \text{ for } (x_1, 1/4)$

- Choose $a$ from $A$ that maximizes $\text{Gain}$, place it at root, then recursively call on $S_a$ and $S'_a$

- Forms basis of algorithms ID3 and C4.5

- Can avoid overfitting by pruning

---

**Topic summary due in 1 week!**