Extending linear classification

- Linear classifiers can't model nonlinear class boundaries
- Simple trick:
  - Map attributes into new space consisting of combinations of attribute values
  - E.g.: all products of n factors that can be constructed from the attributes
- Example with two attributes and $n = 3$:

$$x = w_1 a_1^3 + w_2 a_1^2 a_2 + w_3 a_1 a_2^2 + w_4 a_2^3$$

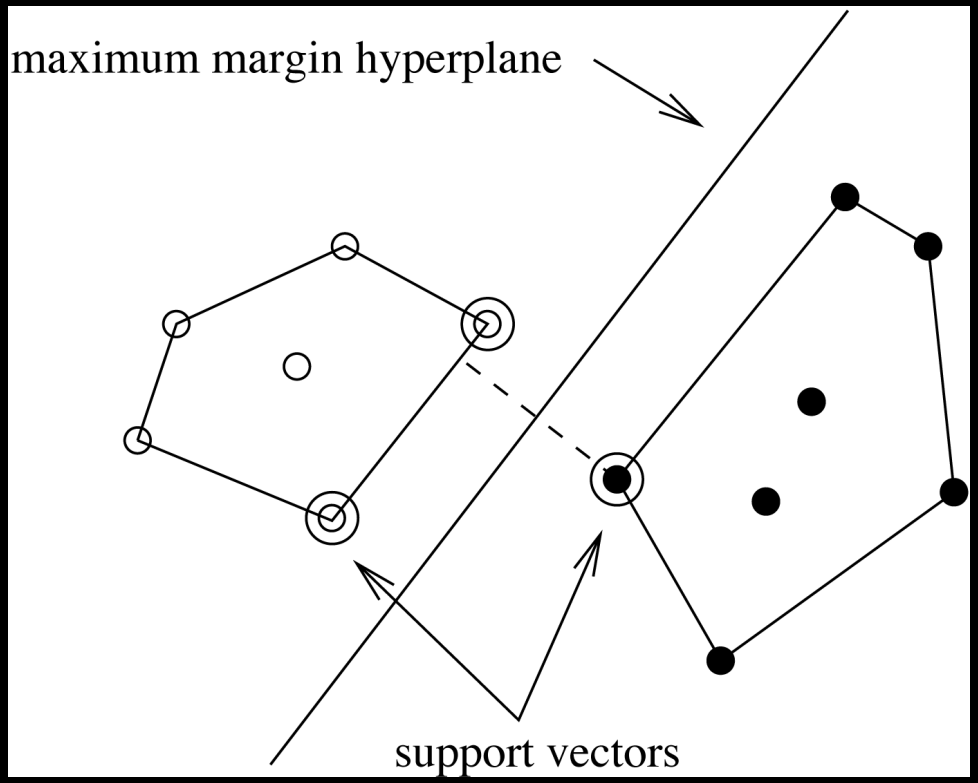
Problems with this approach

- 1st problem: speed
  - 10 attributes, and $n = 5 \Rightarrow$ 2000 coefficients
  - Use linear regression with attribute selection
  - Run time is cubic in number of attributes
- 2nd problem: overfitting
  - Number of coefficients is large relative to the number of training instances
  - *Curse of dimensionality* kicks in

Support vector machines

- *Support vector machines* are algorithms for learning linear classifiers
- Resilient to overfitting because they learn a particular linear decision boundary:
  - The *maximum margin hyperplane*
- Fast in the nonlinear case
  - Use a mathematical trick to avoid creating "pseudo-attributes"
  - The nonlinear space is created implicitly

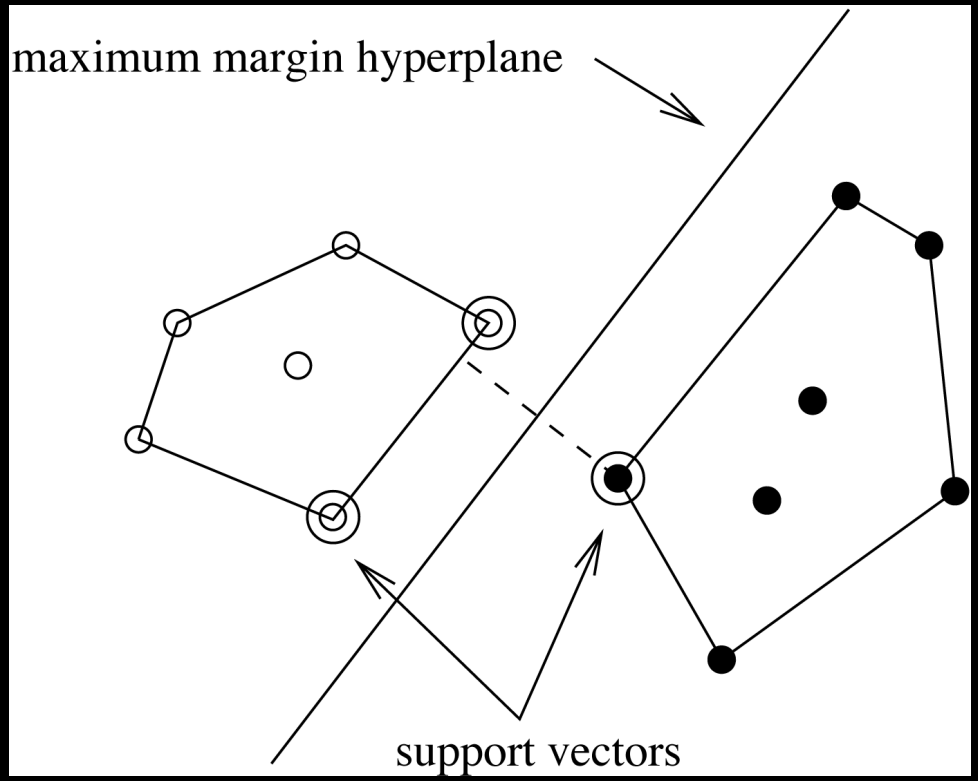
The maximum margin hyperplane



- The instances closest to the maximum margin hyperplane are called *support vectors*

Support vectors

- The support vectors define the maximum margin hyperplane
  - **All other instances can be deleted without changing its position and orientation**



- This means the hyperplane can be written as

$$x = w_0 + w_1 a_1 + w_2 a_2$$

$$x = b + \sum_{\text{i is supp. vector}} \alpha_i y_i \vec{a}(i) \cdot \vec{a}$$

Finding support vectors

$$x = b + \sum_{\text{i is supp. vector}} \alpha_i y_i \vec{a}(i) \cdot \vec{a}$$

- Support vector: training instance for which $\alpha_i > 0$
- Determine α_i and b?—
  A *constrained quadratic optimization* problem
  - Off-the-shelf tools for solving these problems
  - However, special-purpose algorithms are faster
  - Example: Platt's *sequential minimal optimization* algorithm (implemented in WEKA)
- Note: all this assumes separable data!

Nonlinear SVMs

- "Pseudo attributes" represent attribute combinations
- Overfitting not a problem because the maximum margin hyperplane is stable
  - There are usually few support vectors relative to the size of the training set
- Computation time still an issue
  - Each time the dot product is computed, all the "pseudo attributes" must be included

A mathematical trick

- Avoid computing the "pseudo attributes"
- Compute the dot product before doing the nonlinear mapping
- Example:

$$x = b + \sum_{\text{i is supp. vector}} \alpha_i y_i (\vec{a}(i) \cdot \vec{a})^n$$

- Corresponds to a map into the instance space spanned by all products of n attributes

Other kernel functions

- Mapping is called a "kernel function"
- Polynomial kernel

$$x = b + \sum_{\text{i is supp. vector}} \alpha_i y_i (\vec{a}(i) \cdot \vec{a})^n$$

- We can use others:

$$x = b + \sum_{\text{i is supp. vector}} \alpha_i y_i K(\vec{a}(i) \cdot \vec{a})$$

- Only requirement:

$$K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$$

- Examples:

$$K(\vec{x}_i, \vec{x}_j) = (\vec{x}_i \cdot \vec{x}_j + 1)^d$$

$$K(\vec{x}_i, \vec{x}_j) = \exp\left(\frac{-(\vec{x}_i - \vec{x}_j)^2}{2\sigma^2}\right)$$

$$K(\vec{x}_i, \vec{x}_j) = \tanh(\beta \vec{x}_i \cdot \vec{x}_j + b) \quad *$$

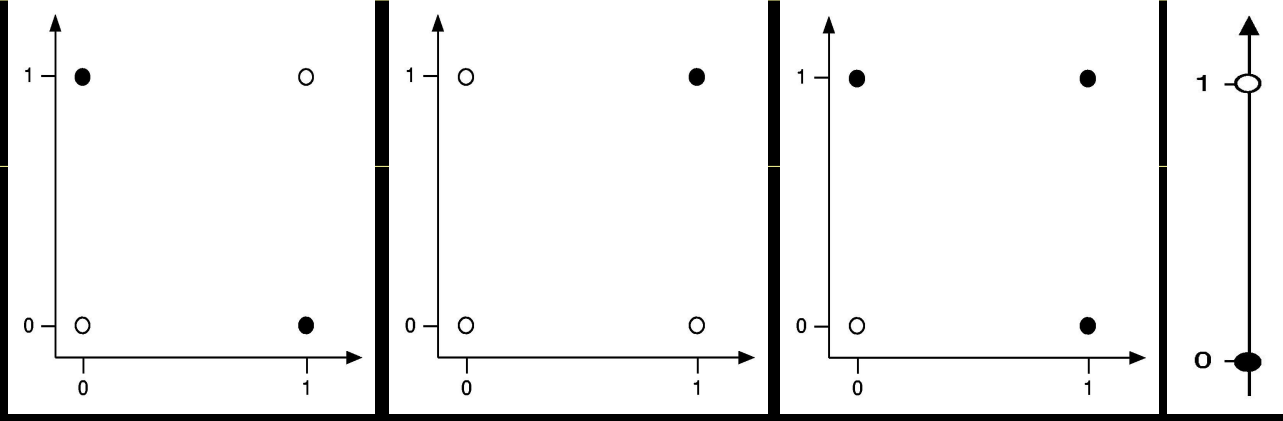
Applications

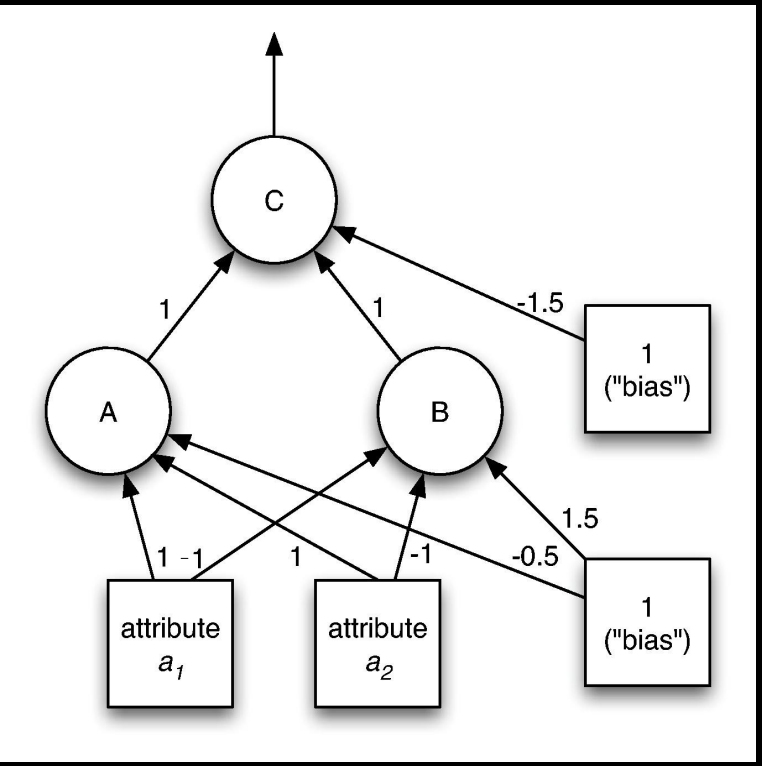
- Machine vision: e.g face identification
  - Outperforms alternative approaches (1.5% error)
- Handwritten digit recognition: USPS data
  - Comparable to best alternative (0.8% error)
- Bioinformatics: e.g. prediction of protein secondary structure
- Text classifiation
- Can modify SVM technique for numeric prediction problems

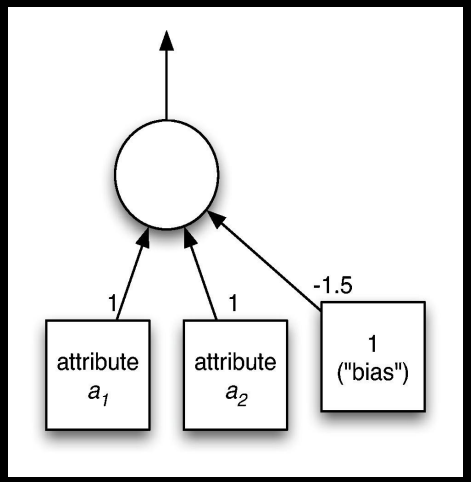
Multilayer perceptrons

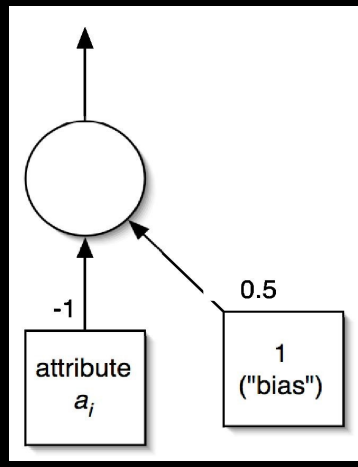
- Using kernels is only one way to build nonlinear classifier based on perceptrons
- Can create network of perceptrons to approximate arbitrary target concepts
- *Multilayer perceptron* is an example of an artificial neural network
  - Consists of: input layer, hidden layer(s), and output layer
- Structure of MLP is usually found by experimentation
- Parameters can be found using *backpropagation*

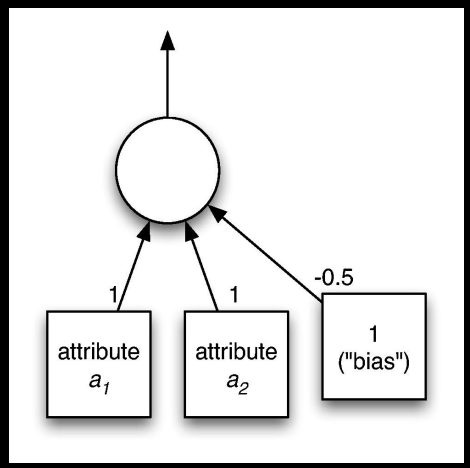
Examples


1
0
0
1
1
0
0
1
1
0
0
1
1
0


C
A
B
1
1
-1.5
1
("bias")
1
-1
1
-1
-0.5
1.5
attribute
a_1
attribute
a_2
1
("bias")


1
1
-1.5
attribute
a_1
attribute
a_2
1
("bias")


-1
0.5
attribute
a_i
1
("bias")


1
1
-0.5
attribute
a_1
attribute
a_2
1
("bias")

Backpropagation

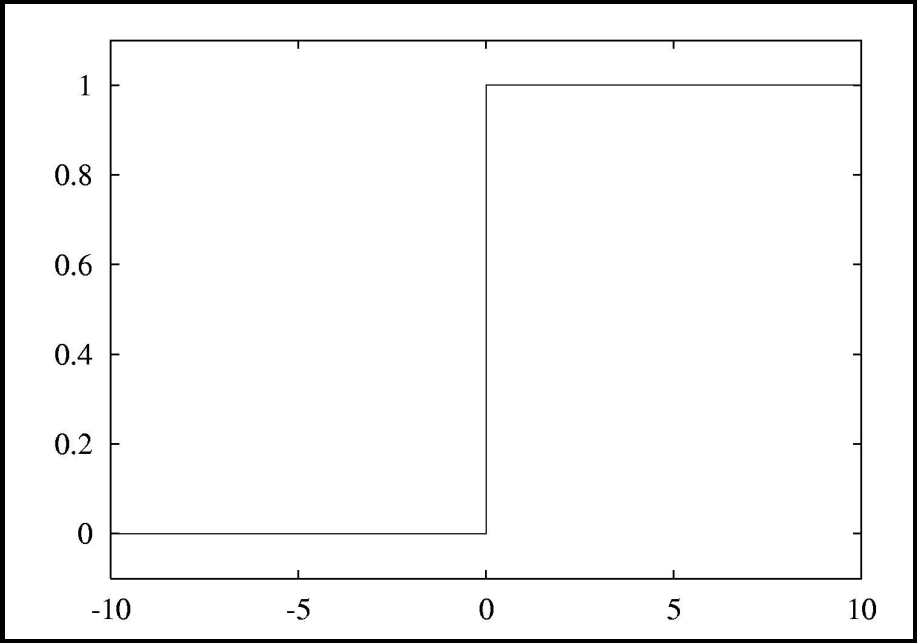
- How to learn weights given network structure?
  - Cannot simply use perceptron learning rule because we have hidden layer(s)
  - Function we are trying to minimize: error
  - Can use a general function minimization technique called *gradient descent*
    - Need differentiable *activation function*: use *sigmoid function* instead of threshold function

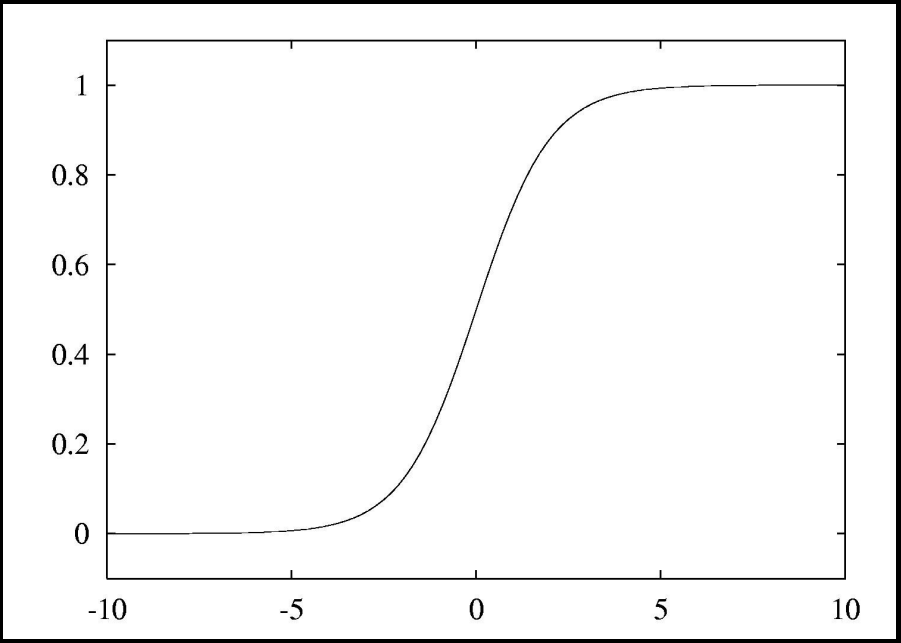
$$f(x)=\frac{1}{1+\exp(-x)}$$

    - Need differentiable error function: can't use zero-one loss, but can use squared error

$$E=\frac{1}{2}(y-f(x))^2$$

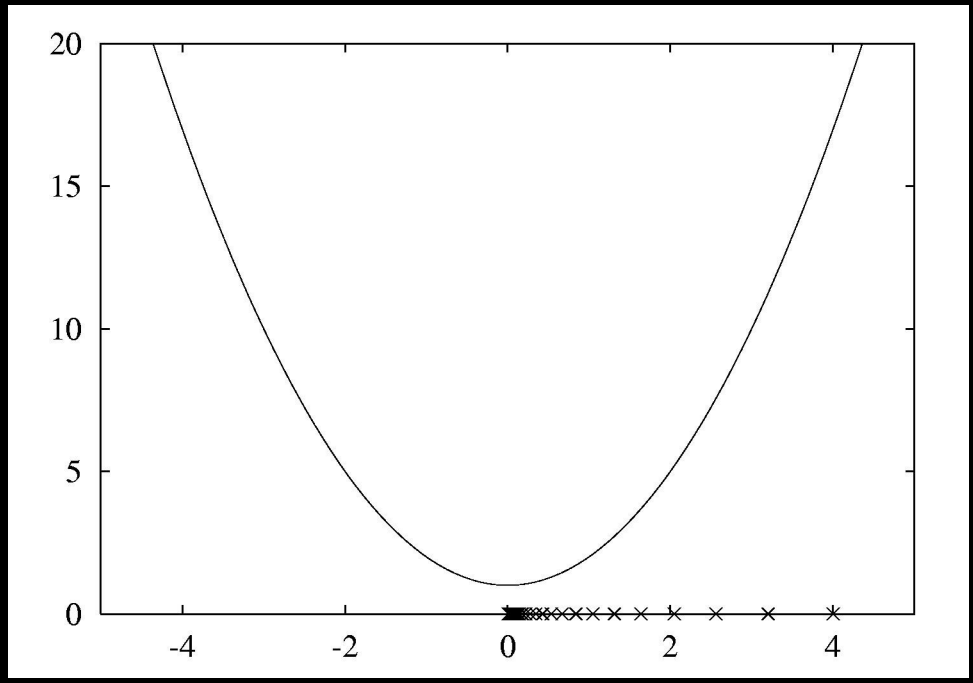
The two activation functions





Gradient descent example

- Function: x^2+1
- Derivative: $2x$
- Learning rate: 0.1
- Start value: 4



Can only find a local minimum!

Minimizing the error I

- Need to find partial derivative of error function for each parameter (i.e. weight)

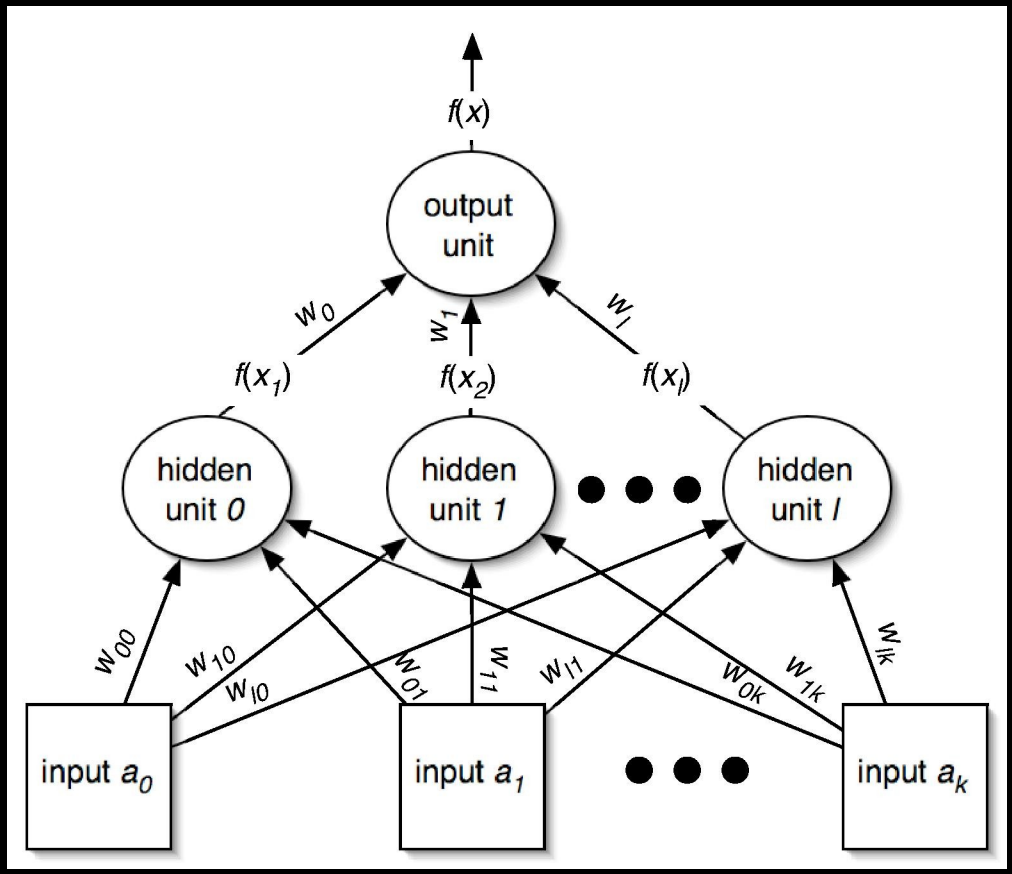
$$\frac{dE}{dw_i} = (y - f(x))\frac{df(x)}{dw_i}$$

$$\frac{df(x)}{dx} = f(x)(1 - f(x))$$

$$x = \sum_i w_i f(x_i)$$

$$\frac{df(x)}{dw_i} = f'(x) f(x_i)$$

$$\frac{dE}{dw_i} = (y - f(x)) f'(x) f(x_i)$$



Minimizing the error II

- What about the weights for the connections from the input to the hidden layer?

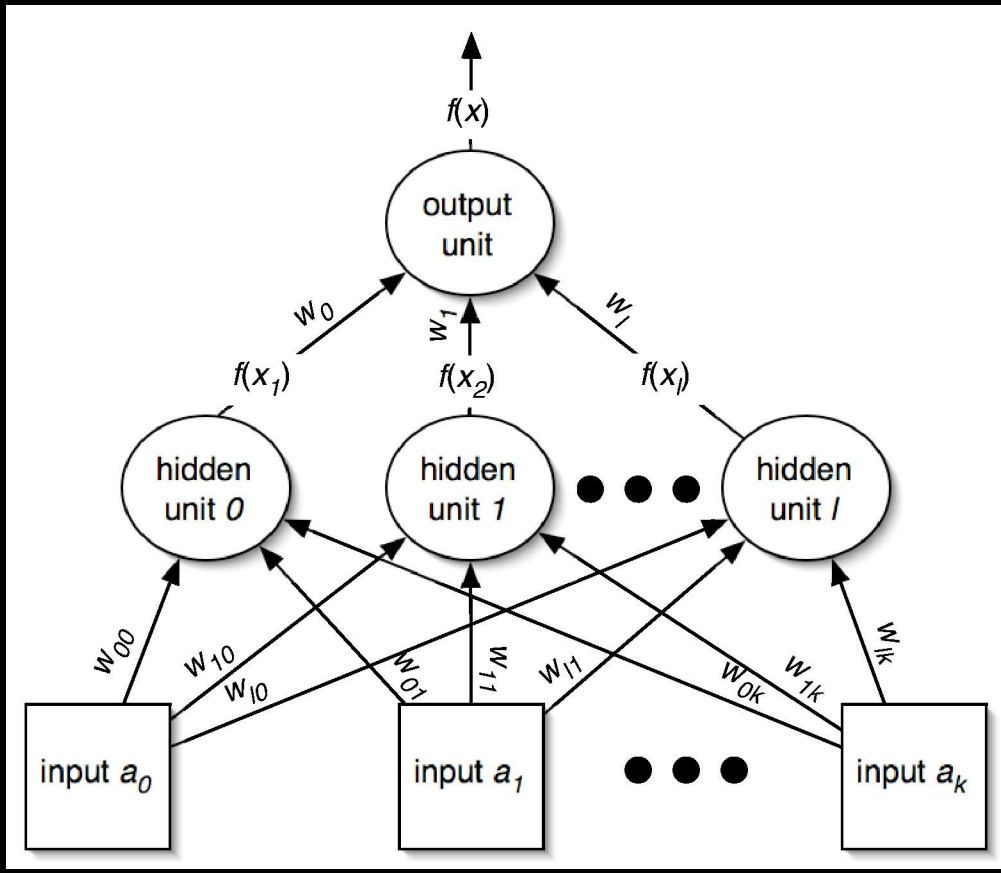
$$\frac{dE}{dw_{ij}} = \frac{dE}{dx}\frac{dx}{dw_{ij}} = (y - f(x))\, f'(x)\frac{dx}{dw_{ij}}$$

$$x = \sum_i w_i f(x_i)$$

$$\frac{dx}{dw_{ij}} = w_i \frac{df(x_i)}{dw_{ij}}$$

$$\frac{df(x_i)}{dw_{ij}} = f'(x_i)\frac{dx_i}{dw_{ij}} = f'(x_i)\, a_i$$

$$\frac{dE}{dw_{ij}} = (y - f(x))\, f'(x)\, w_i f'(x_i)\, a_i$$



Remarks

- Same process works for multiple hidden layers and multiple output units (eg. for multiple classes)
- Can update weights after all training instances have been processed or incrementally:
  - *batch learning* vs. *stochastic backpropagation*
  - Weights are initialized to small random values
- How to avoid overfitting?
  - *Early stopping*: use validation set to check when to stop
  - *Weight decay*: add penalty term to error function
- How to speed up learning?
  - *Momentum*: re-use proportion of old weight change
  - Use optimization method that employs 2nd derivative

Radial basis function networks

- Another type of *feedforward network* with two layers (plus the input layer)
- Hidden units represent points in instance space and activation depends on distance
  - To this end, distance is converted into similarity: Gaussian activation function
    - Width may be different for each hidden unit
  - Points of equal activation form hypersphere (or hyperellipsoid) as opposed to hyperplane
- Output layer same as in MLP

Learning RBF networks

- Parameters: centers and widths of the RBFs + weights in output layer
- Can learn two sets of parameters independently and still get accurate models
  - Eg.: clusters from *k*-means can be used to form basis functions
  - Linear model can be used based on fixed RBFs
  - Makes learning RBFs very efficient
- Disadvantage: no built-in attribute weighting based on relevance
- RBF networks are related to RBF SVMs

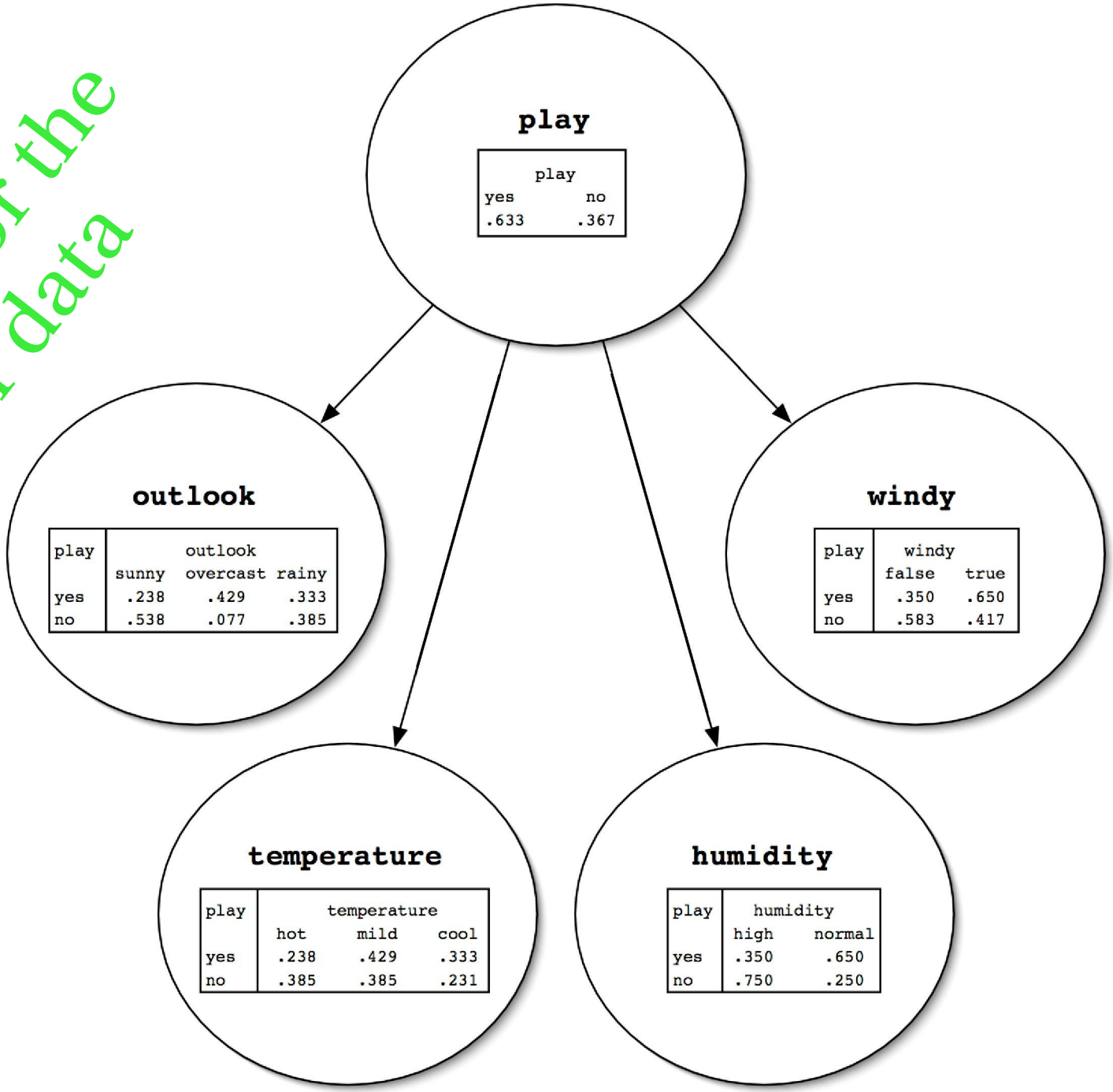
From naïve Bayes to Bayesian Networks

- Naïve Bayes assumes: attributes conditionally independent given the class
- Doesn't hold in practice but classification accuracy often high
- However: sometimes performance much worse than e.g. decision tree
- Can we eliminate the assumption?

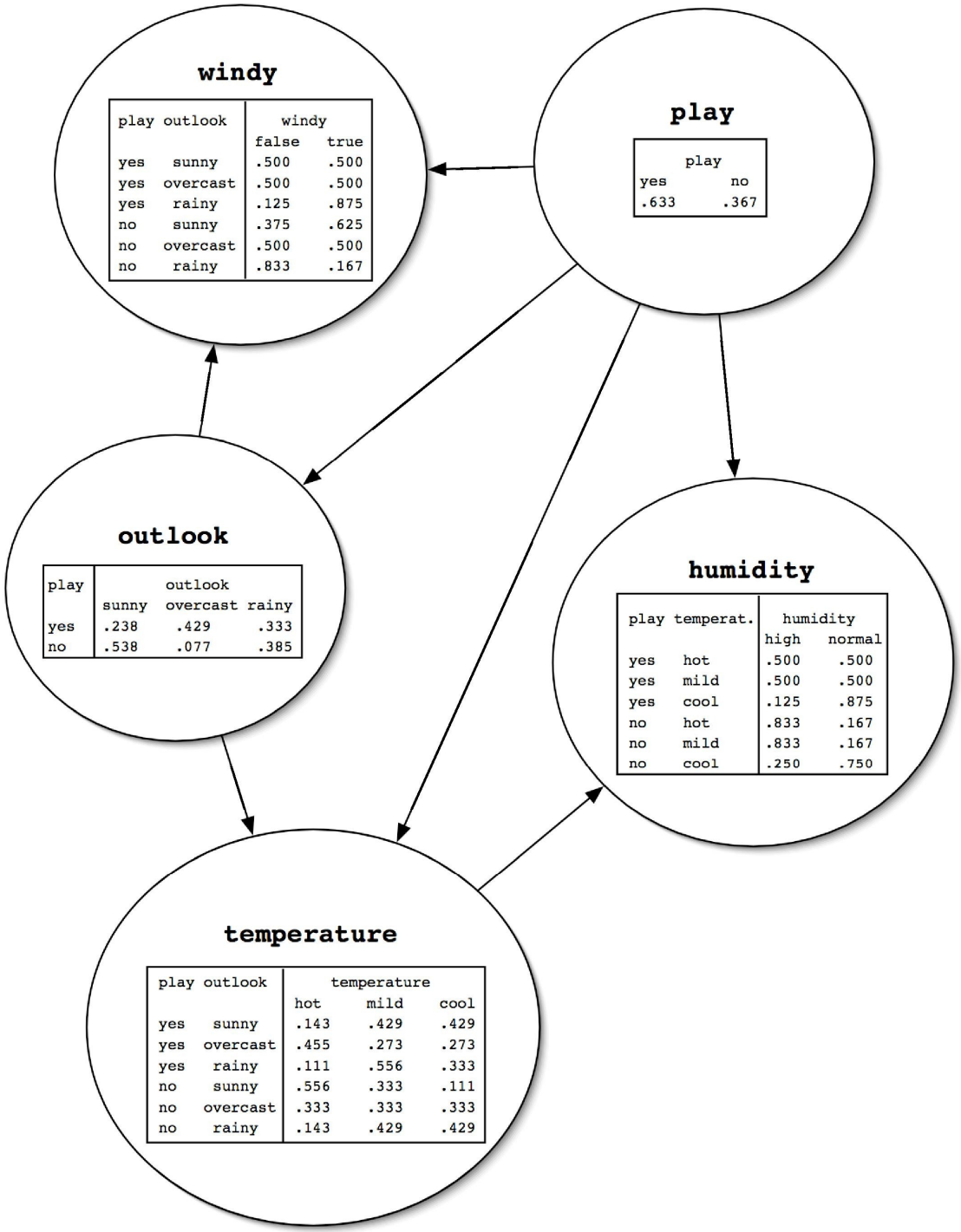
Enter Bayesian networks

- Graphical models that can represent any probability distribution
- Graphical representation: directed acyclic graph, one node for each attribute
- Overall probability distribution factorized into component distributions
- Graph's nodes hold component distributions (conditional distributions)

Network for the weather data



Network for the weather data



Computing the class probabilities

- Two steps: computing a product of probabilities for each class and normalization
  - For each class value
    - Take all attribute values and class value
    - Look up corresponding entries in conditional probability distribution tables
    - Take the product of all probabilities
  - Divide the product for each class by the sum of the products (normalization)

Why can we do this? (Part I)

- Single assumption: values of a node's parents completely determine probability distribution for current node

$$Pr[\text{node}|\text{ancestors}] = Pr[\text{node}|\text{parents}]$$

Means that node/attribute is conditionally independent of other ancestors given parents

Why can we do this? (Part II)

- Chain rule from probability theory:

$$Pr[a_1, a_2, \ldots, a_n] = \prod_{i=1}^{n} Pr[a_i | a_{i-1}, \ldots, a_1]$$

Because of our assumption from the previous slide:

$$Pr[a_1, a_2, \ldots, a_n] = \prod_{i=1}^{n} Pr[a_i | a_{i-1}, \ldots, a_1] = \prod_{i=1}^{n} Pr[a_i | a_i\text{'s parents}]$$

Learning Bayes nets

- Basic components of algorithms for learning Bayes nets:
  - Method for evaluating the goodness of a given network
    - Measure based on probability of training data given the network (or the logarithm thereof)
  - Method for searching through space of possible networks
    - Amounts to searching through sets of edges because nodes are fixed