# Machine Learning Algorithms for Classification

Rob Schapire
Princeton University

www.cs.princeton.edu/∼schapire

Machine Learning

- studies how to <u>automatically learn</u> to make accurate <u>predictions</u> based on past observations

- <u>classification</u> problems:
  - classify examples into given set of categories

new
example

labeled
training
examples → machine learning
algorithm → classification
rule

predicted
classification
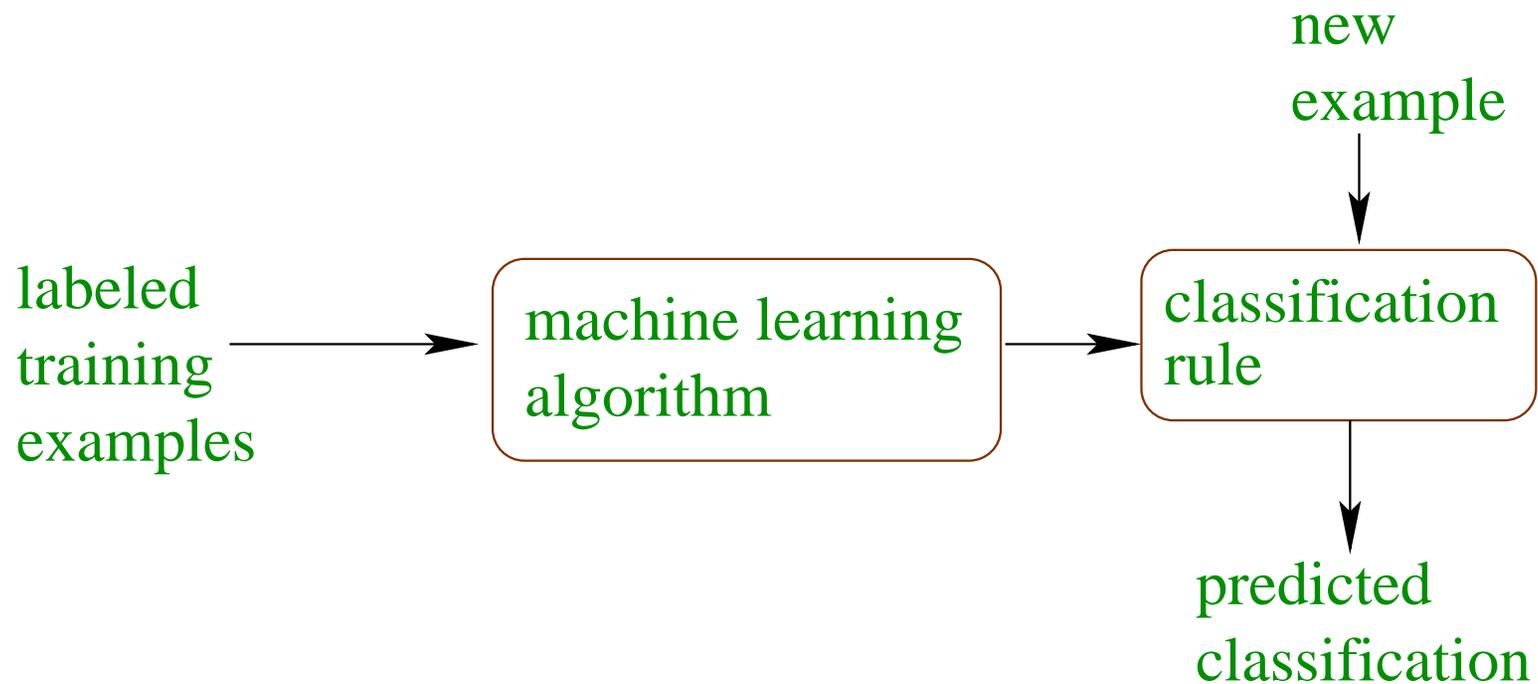
Examples of Classification Problems

- text categorization (e.g., spam filtering)

- fraud detection

- optical character recognition

- machine vision (e.g., face detection)

- natural-language processing (e.g., spoken language understanding)

- market segmentation (e.g.: predict if customer will respond to promotion)

- bioinformatics (e.g., classify proteins according to their function)
  ⋮

Why Use Machine Learning?

- advantages:

  - often much more accurate than human-crafted rules
    (since data driven)
  - humans often incapable of expressing what they know
    (e.g., rules of English, or how to recognize letters),
    but can easily classify examples
  - don't need a human expert or programmer
  - flexible — can apply to any learning task
  - cheap — can use in applications requiring many classifiers
    (e.g., one per customer, one per product, one per web page, ...)

- disadvantages

  - need a lot of labeled data
  - error prone — usually impossible to get perfect accuracy

Machine Learning Algorithms

- this talk:
  - decision trees
  - boosting
  - support-vector machines
  - neural networks

- others not covered:
  - nearest neighbor algorithms
 - Naive Bayes
  - bagging
  - ⋮

Decision Trees

Example: Good versus Evil

- problem: identify people as good or bad from their appearance

|	sex	mask	cape	tie	ears	smokes	class
			training data				
batman	male	yes	yes	no	yes	no	Good
robin	male	yes	yes	no	no	no	Good
alfred	male	no	no	yes	no	no	Good
penguin	male	no	no	yes	no	yes	Bad
catwoman	female	yes	no	no	yes	no	Bad
joker	male	no	no	no	no	no	Bad
			test data				
batgirl	female	yes	yes	no	yes	no	??
riddler	male	yes	no	no	no	no	??

Example (cont.)
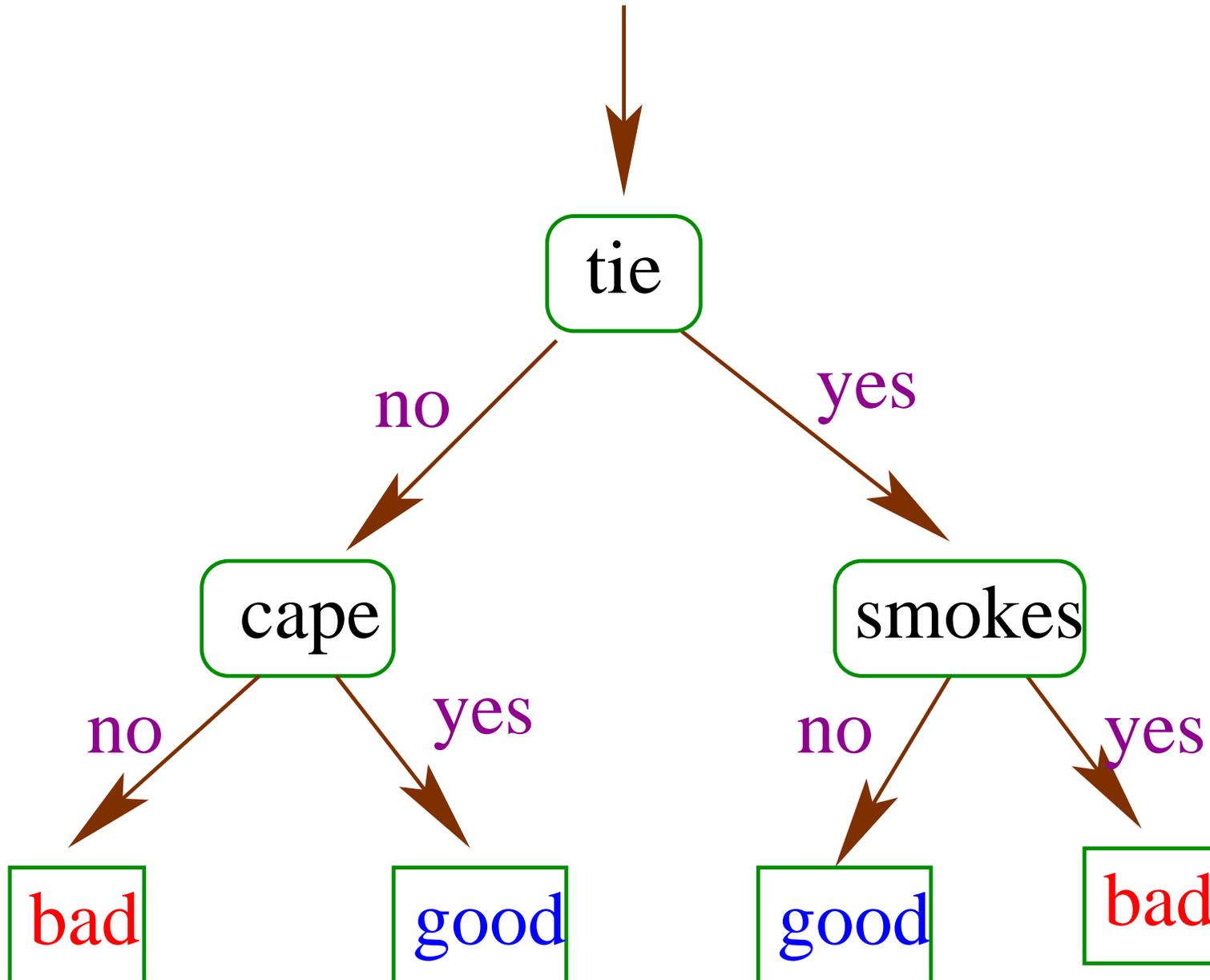
How to Build Decision Trees

- choose rule to split on

- divide data using splitting rule into disjoint subsets

- repeat recursively for each subset

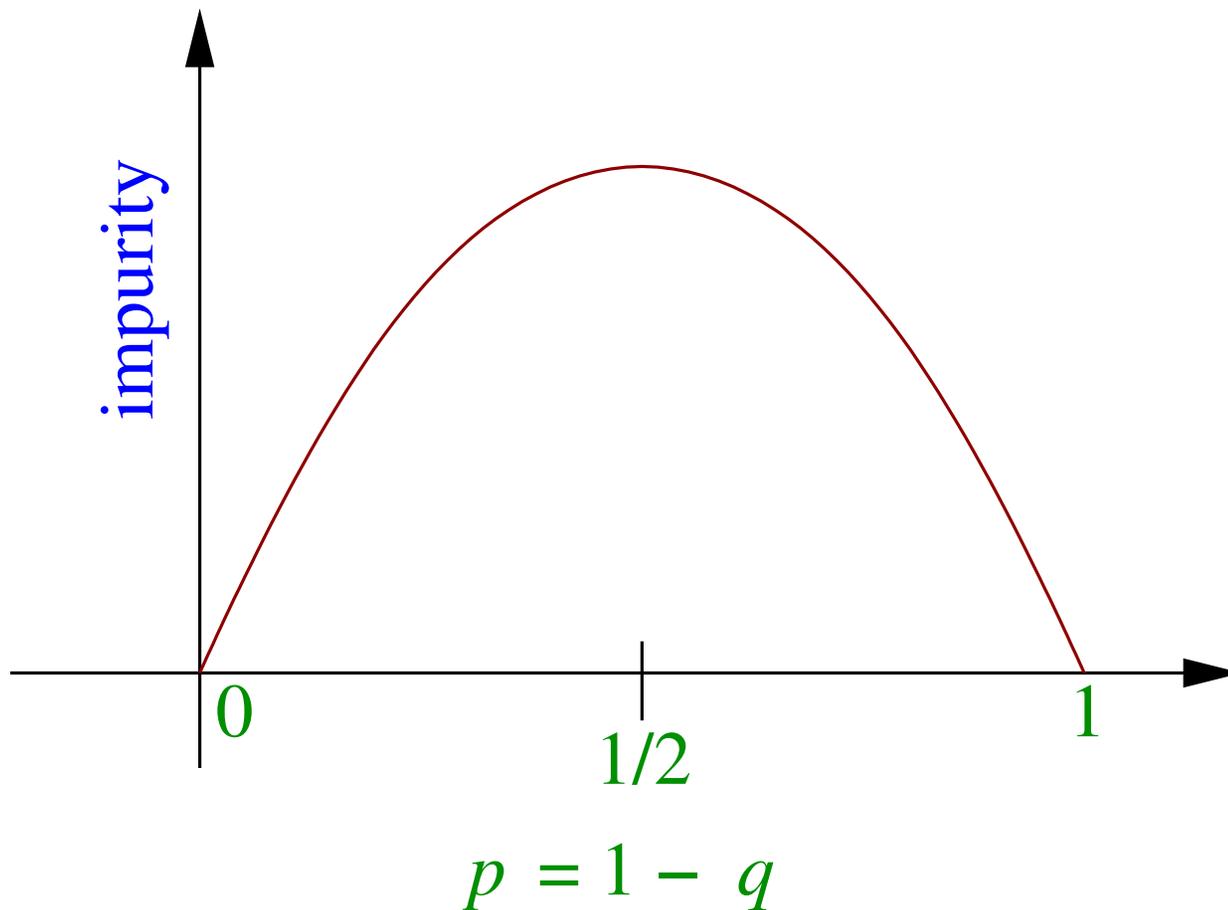- stop when leaves are (almost) "pure"

Choosing the Splitting Rule

- choose rule that leads to greatest increase in "purity":
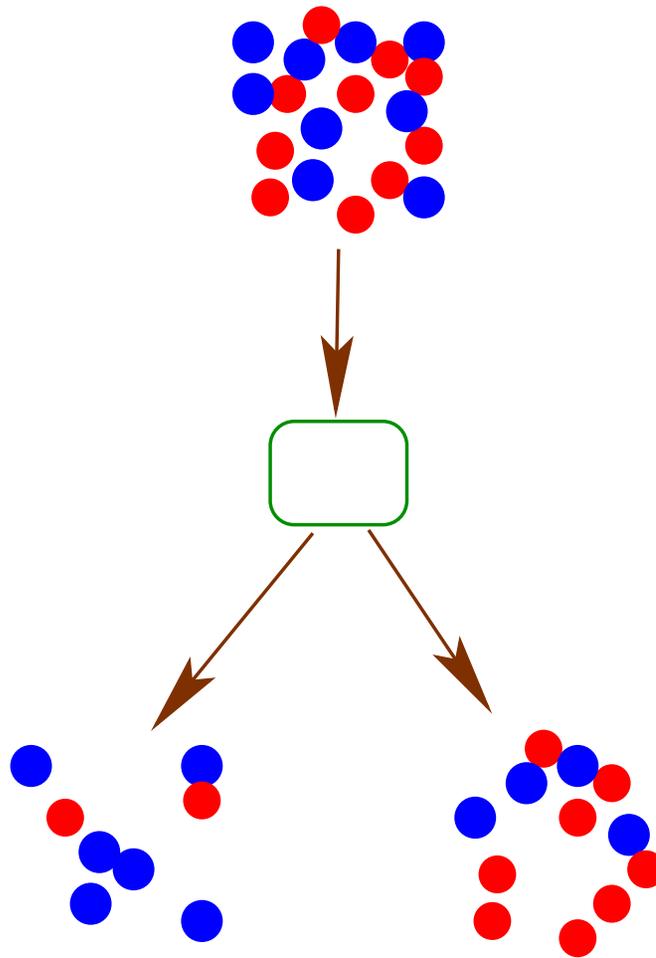
Choosing the Splitting Rule (cont.)

- (im)purity measures:
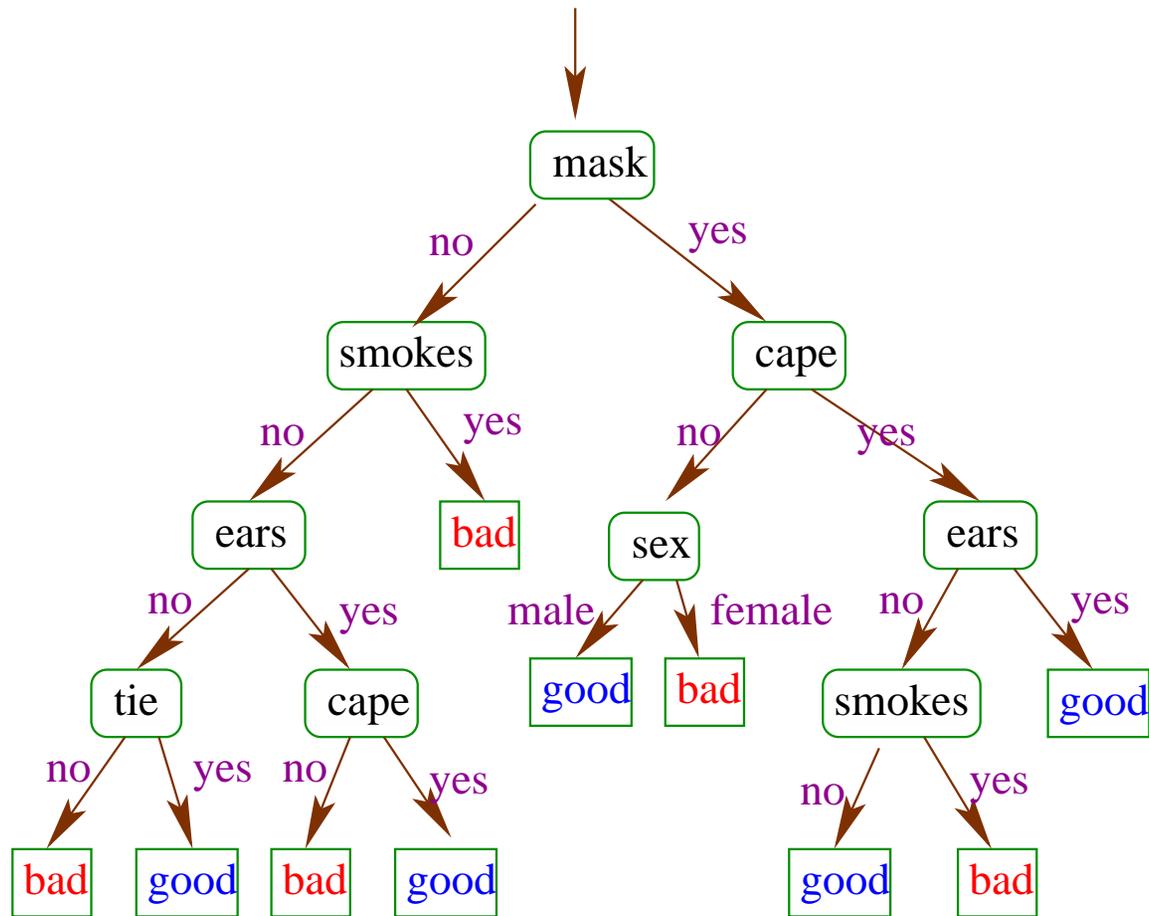 - entropy: $-p \ln p - q \ln q$
  - Gini index: pq

  where p (q) = fraction of positive (negative) examples
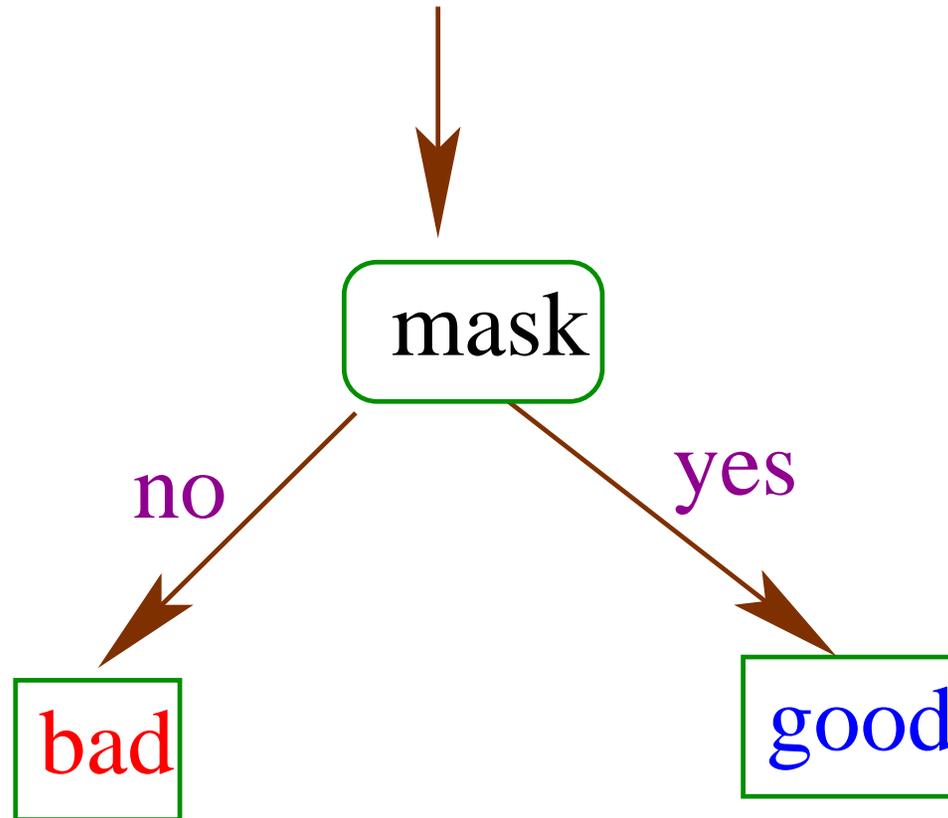

$$p = 1 - q$$

Kinds of Error Rates

- training error = fraction of training examples misclassified

- test error = fraction of test examples misclassified

- generalization error = probability of misclassifying new random example
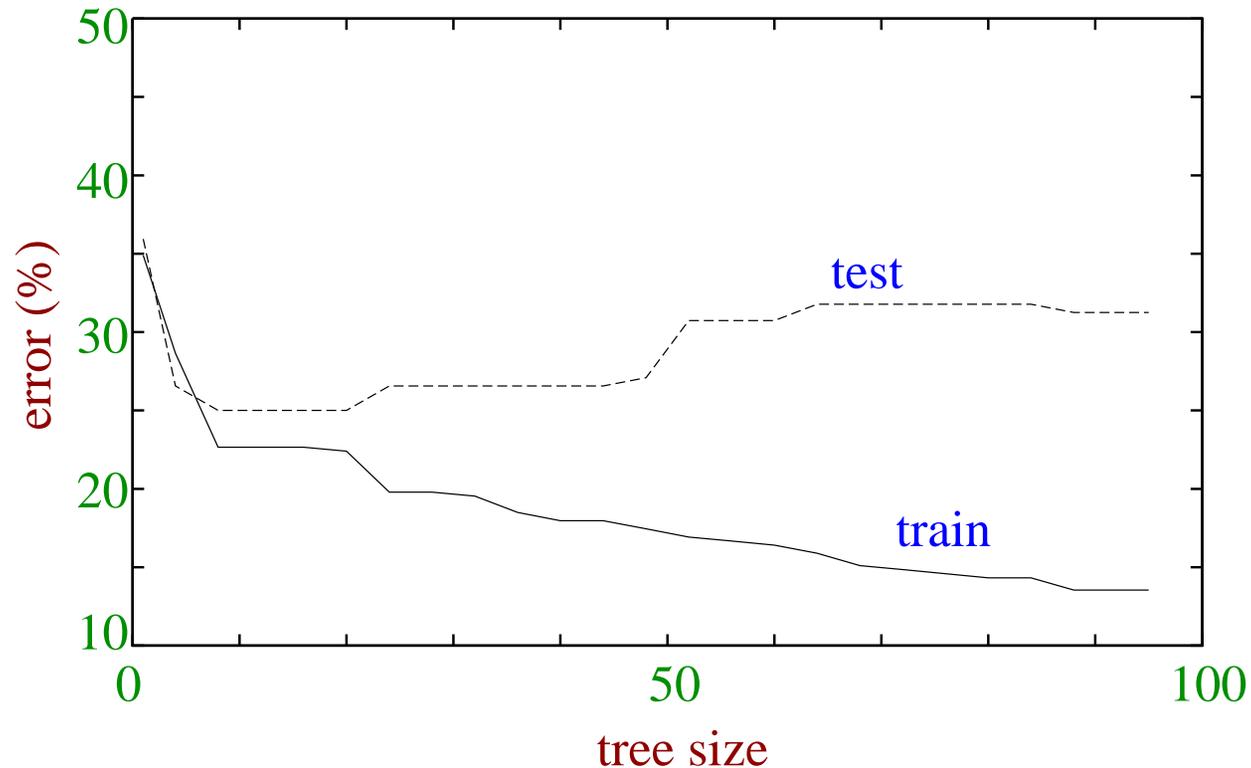
A Possible Classifier

- perfectly classifies training data
- BUT: intuitively, overly complex

Another Possible Classifier


- overly simple
- doesn't even fit available data

Tree Size versus Accuracy

- trees must be big enough to fit training data
  (so that "true" patterns are fully captured)

- BUT: trees that are too big may <u>overfit</u>
  (capture noise or spurious patterns in the data)

- <u>significant problem</u>: can't tell best tree size from training error
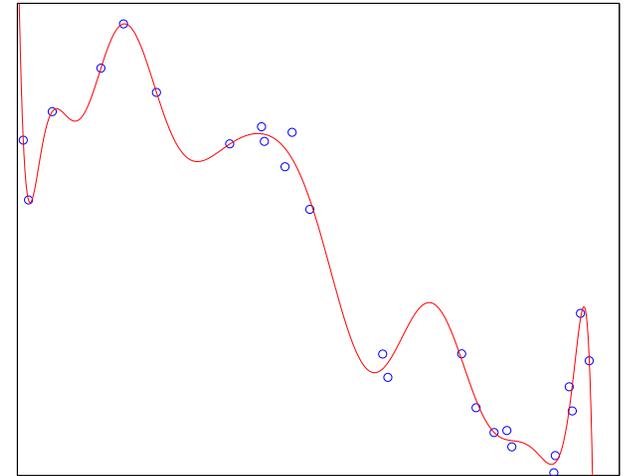
Overfitting Example

- fitting points with a polynomial
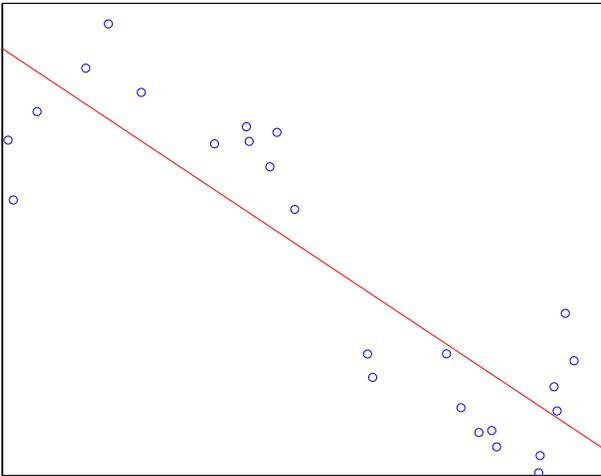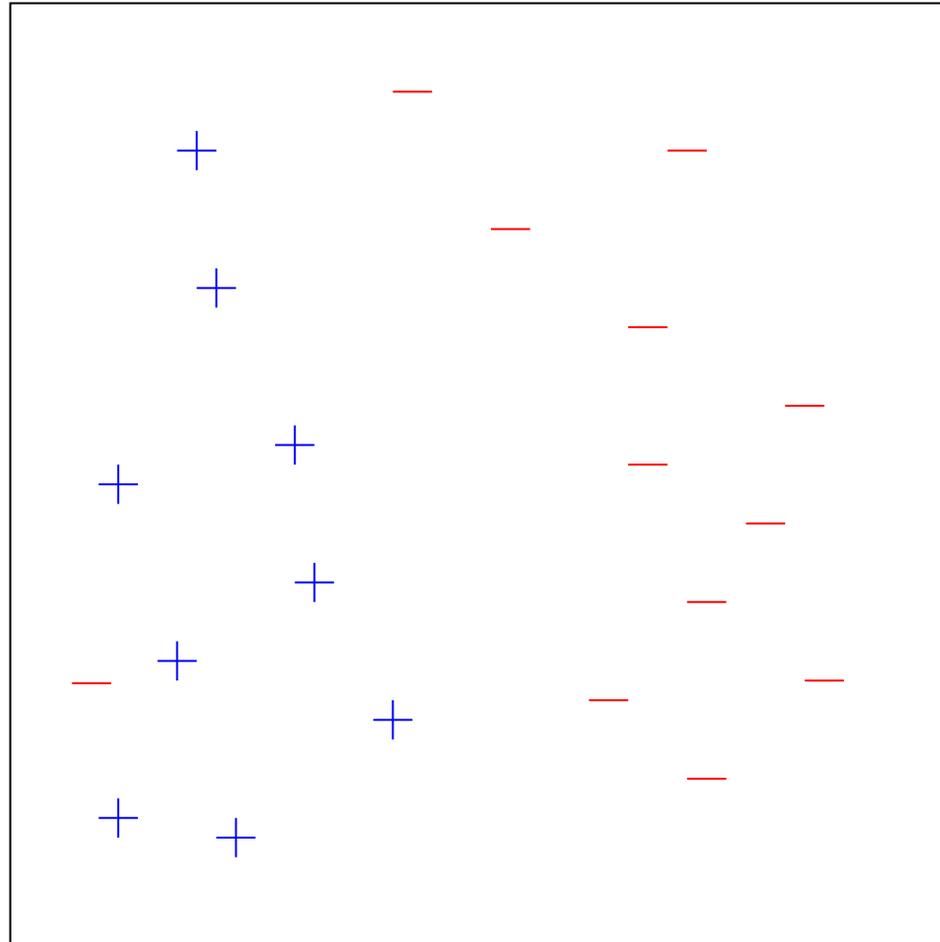

underfit
(degree = 1)
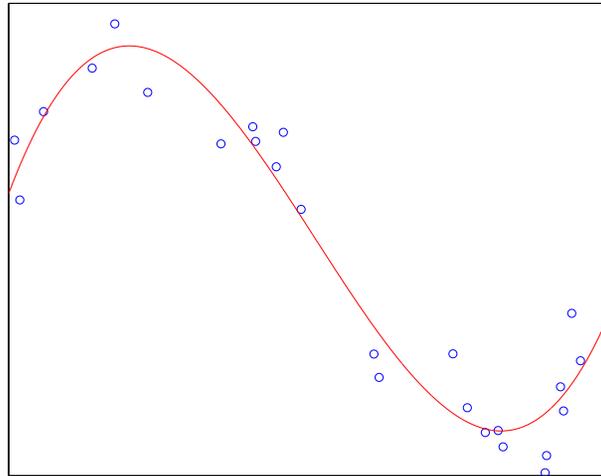
ideal fit
(degree = 3)

overfit
(degree = 20)

Building an Accurate Classifier

- for good <u>test</u> peformance, need:
  - enough training examples
  - good performance on <u>training</u> set
  - classifier that is not too "complex" ("<u>Occam's razor</u>")
    - measure "complexity" by:
      - · number bits needed to write down
      - · number of parameters
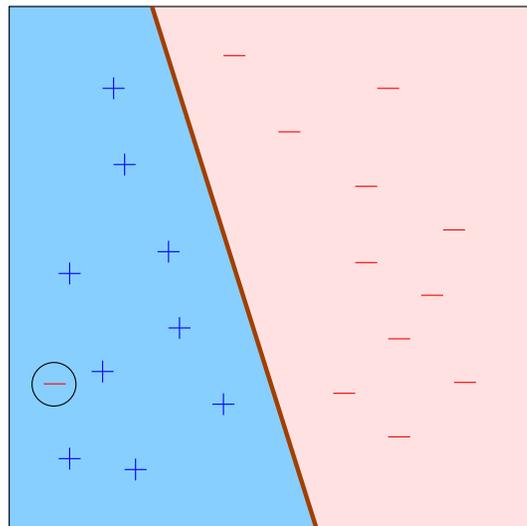   - · VC-dimension

Example
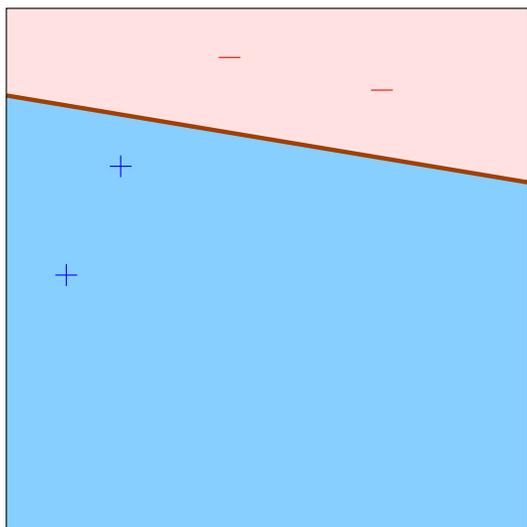
Training data:

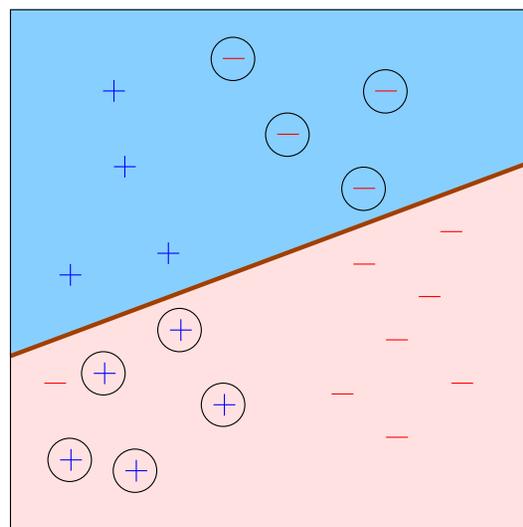# Good and Bad Classifiers

Good:

sufficient data
low training error
simple classifier

Bad:

insufficient data

training error
too high
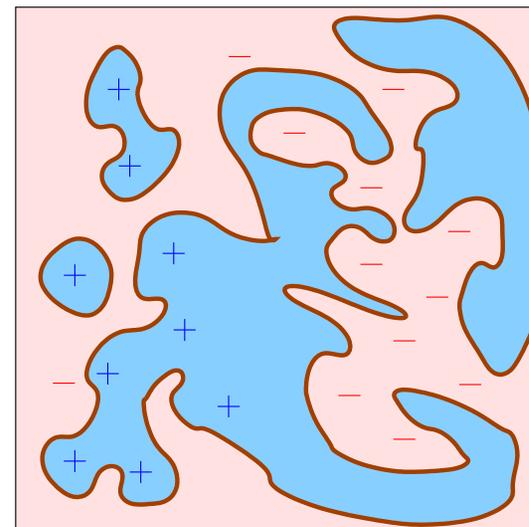
classifier
too complex

Theory

- can prove:

$$(\text{generalization error}) \leq (\text{training error}) + \tilde{O}\left(\sqrt{\frac{d}{m}}\right)$$

  with high probability
  - $d =$ VC-dimension
  - $m =$ number training examples

Controlling Tree Size

- typical approach: build very large tree that fully fits training data, then prune back
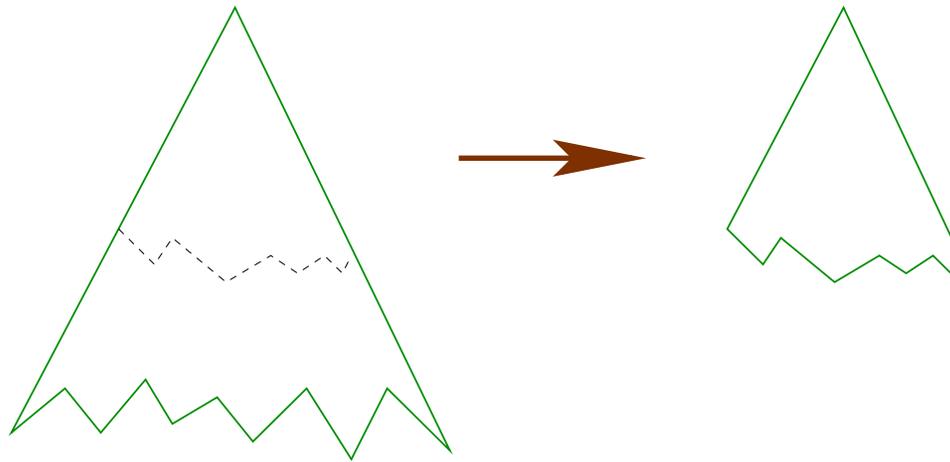
- pruning strategies:
  - grow on just part of training data, then find pruning with minimum error on held out part
  - find pruning that minimizes

$$(\text{training error}) + \text{constant} \cdot (\text{tree size})$$

Decision Trees

- best known:
    - C4.5 (Quinlan)
    - CART (Breiman, Friedman, Olshen & Stone)

- very fast to train and evaluate

- relatively easy to interpret

- but: accuracy often not state-of-the-art

Boosting

Example: Spam Filtering

- problem: filter out spam (junk email)

- gather large collection of examples of spam and non-spam:

  From: yoav@att.com   Rob, can you review a paper...   non-spam
  From: xa412@hotmail.com Earn money without working!!!! ... spam
  ⋮   ⋮   ⋮

- main observation:

  - easy to find "rules of thumb" that are "often" correct

    - *If* 'buy now' *occurs in message, then predict* 'spam'

  - hard to find single rule that is very highly accurate

The Boosting Approach

- devise computer program for deriving rough rules of thumb

- apply procedure to subset of emails

- obtain rule of thumb

- apply to 2nd subset of emails

- obtain 2nd rule of thumb

- repeat T times

Details

- how to <u>choose examples</u> on each round?
  - concentrate on "hardest" examples
    (those most often misclassified by previous rules of thumb)
- how to <u>combine</u> rules of thumb into single prediction rule?
 - take (weighted) majority vote of rules of thumb

Boosting

- boosting = general method of converting rough rules of thumb
  into highly accurate prediction rule
- technically:
  - assume given "weak" learning algorithm that can consistently find classifiers ("rules of thumb") at least slightly better than random, say, accuracy $\geq 55\%$
  - given sufficient data, a boosting algorithm can provably construct single classifier with very high accuracy, say, 99%

AdaBoost

- given training examples (x_i, y_i) where $y_i \in \{-1, +1\}$

- initialize $D_1 =$ uniform distribution on training examples

- for $t = 1, \ldots, T$:
  - train <u>weak classifier</u> ("rule of thumb") h_t on D_t
  - choose $\alpha_t > 0$
  - compute new distribution D_{t+1}:
    - for each example i:
      $$\text{multiply } D_t(x_i) \text{ by } \begin{cases} e^{-\alpha_t} \ (< 1) & \text{if } y_i = h_t(x_i) \\ e^{\alpha_t} \quad (> 1) & \text{if } y_i \neq h_t(x_i) \end{cases}$$
    - renormalize

- output <u>final classifier</u> $H_{\text{final}}(x) = \text{sign}\left(\sum_t \alpha_t h_t(x) \right)$
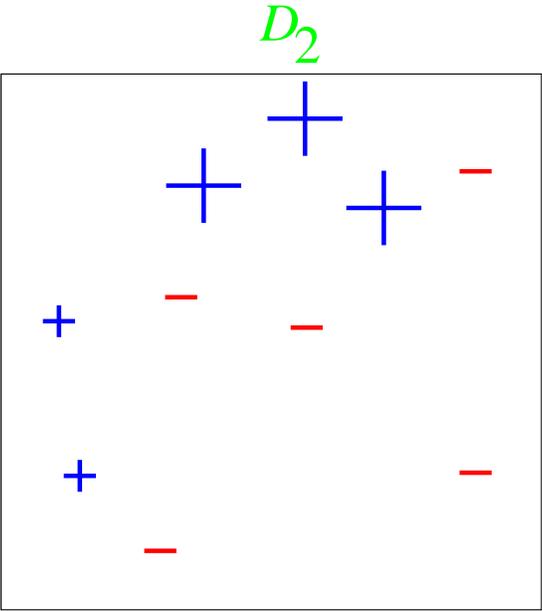
Toy Example

D_1
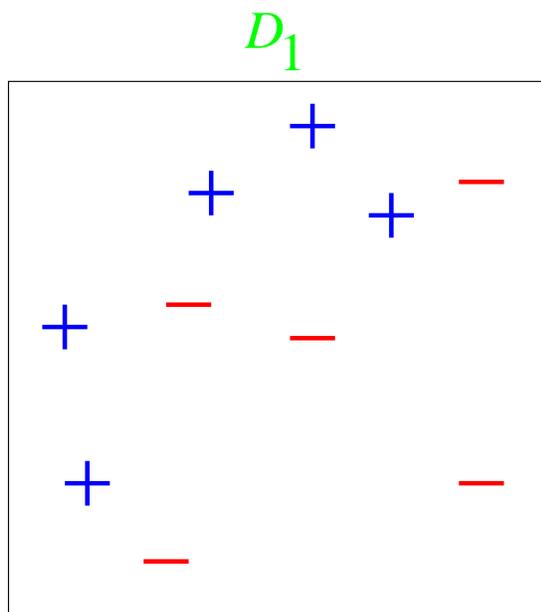
weak classifiers = vertical or horizontal half-planes

Round 1

h_1

D_2

$\varepsilon_1 = 0.30$

$\alpha_1 = 0.42$

Round 2

h_2

D_3

$\varepsilon_2 = 0.21$

$\alpha_2 = 0.65$
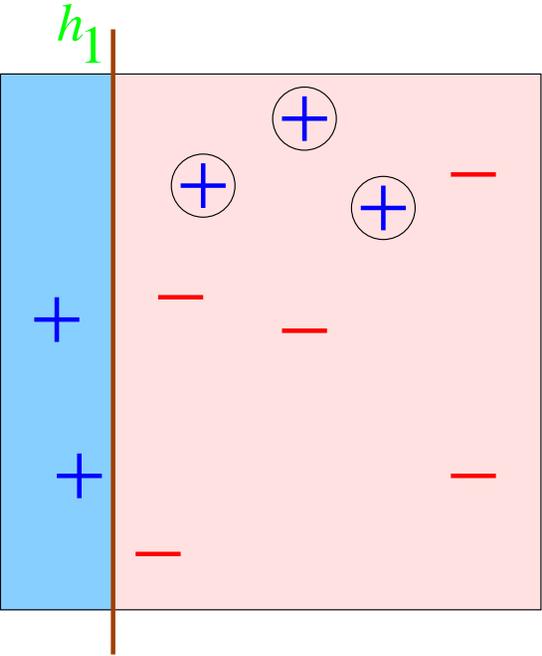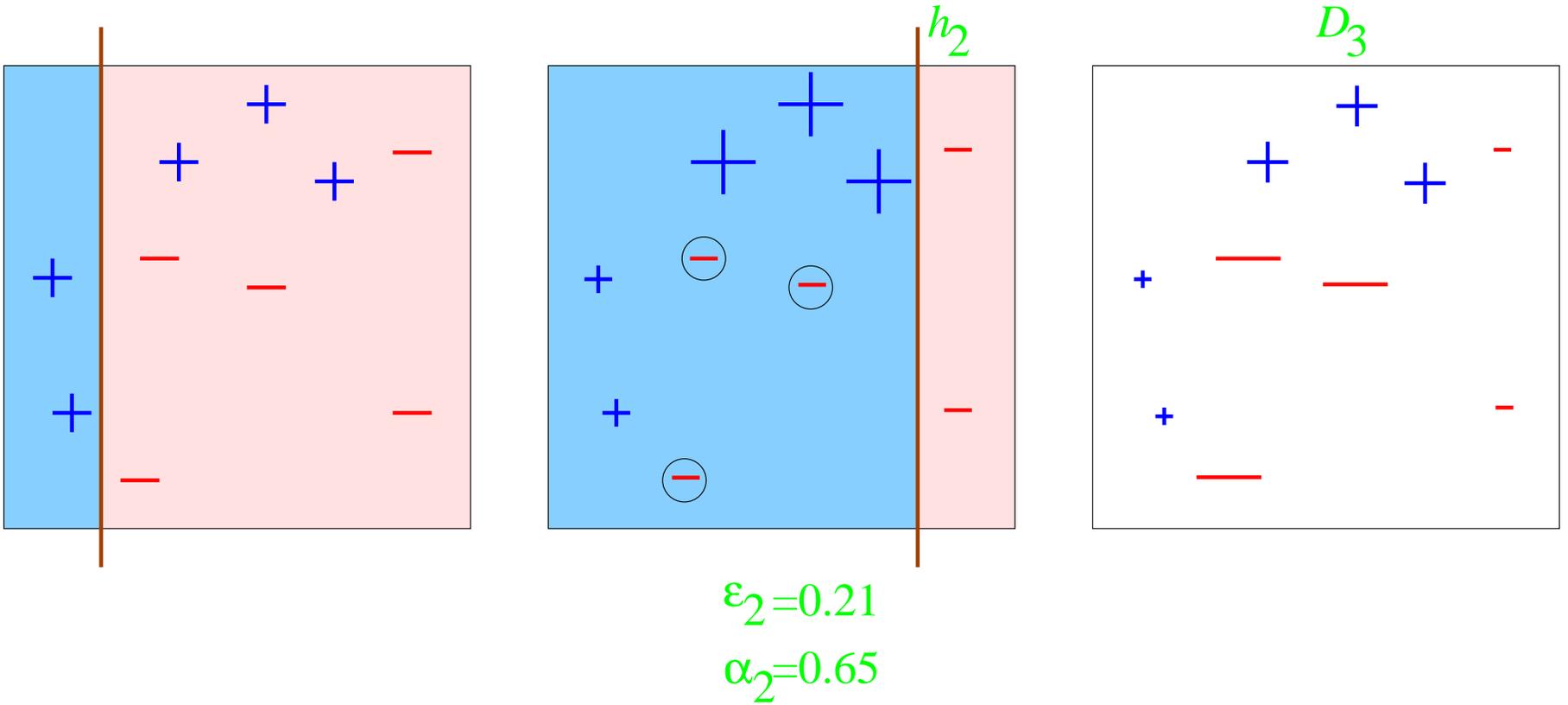
Round 3

h_3
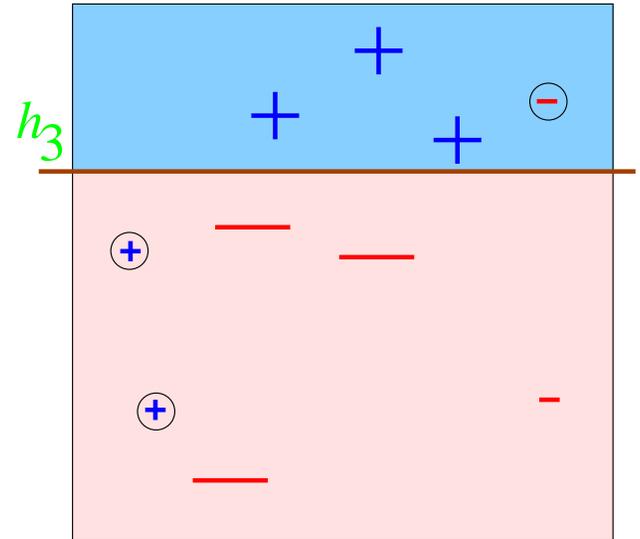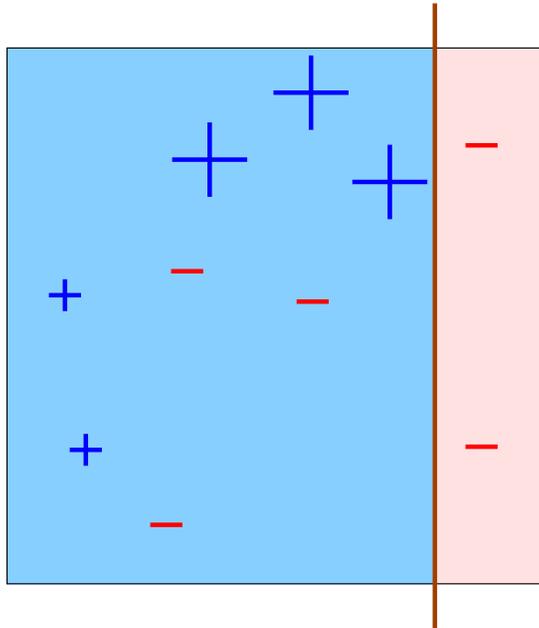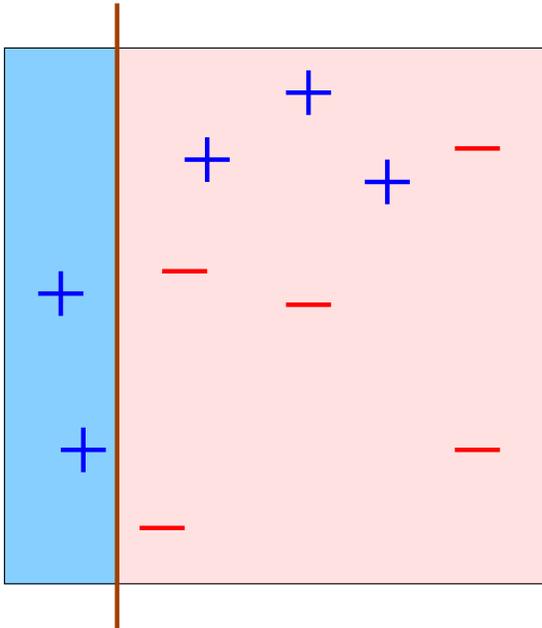
$\varepsilon_3 = 0.14$
$\alpha_3 = 0.92$

Final Classifier

$$H_{\text{final}} = \text{sign}\left(0.42 \quad \boxed{} \quad + 0.65 \quad \boxed{} \quad + 0.92 \quad \boxed{} \right)$$

=
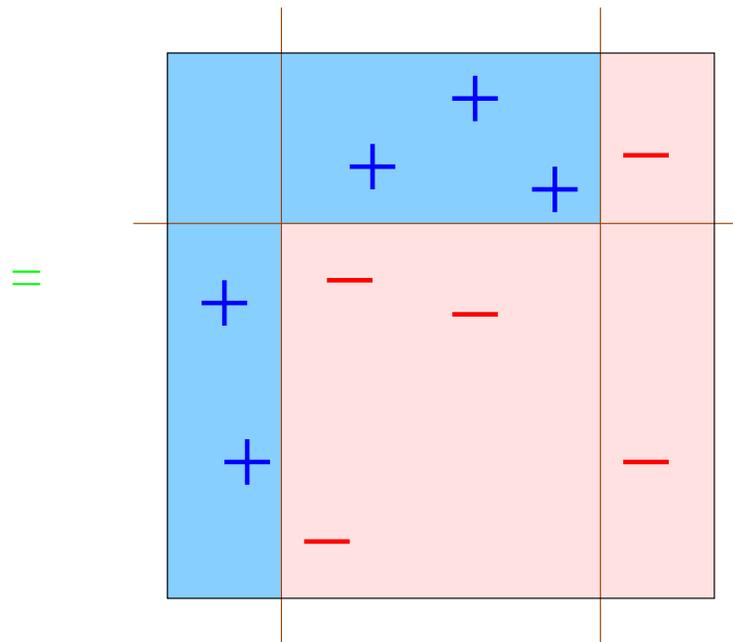
Theory: Training Error

- <u>weak learning assumption</u>: each weak classifier at least slightly better than random

    - i.e., (error of h_t on D_t) $\leq 1/2 - \gamma$ for some $\gamma > 0$

- given this assumption, can prove:

$$\text{training error}(H_{\text{final}}) \leq e^{-2\gamma^2 T}$$

How Will Test Error Behave? (A First Guess)
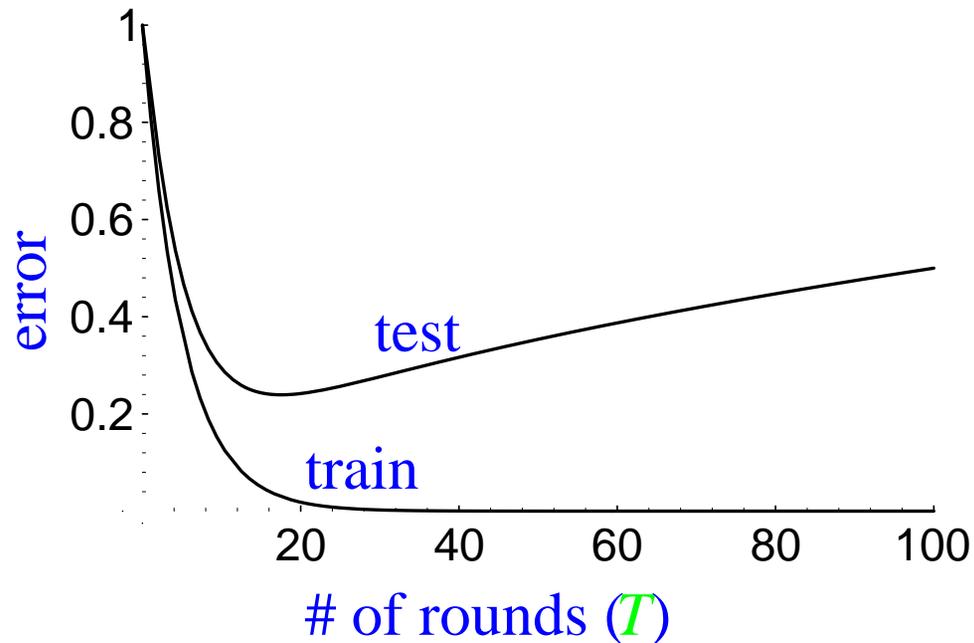
- expect:
  - training error to continue to drop (or reach zero)
  - test error to increase when H_{final} becomes "too complex" (overfitting)

Actual Typical Run
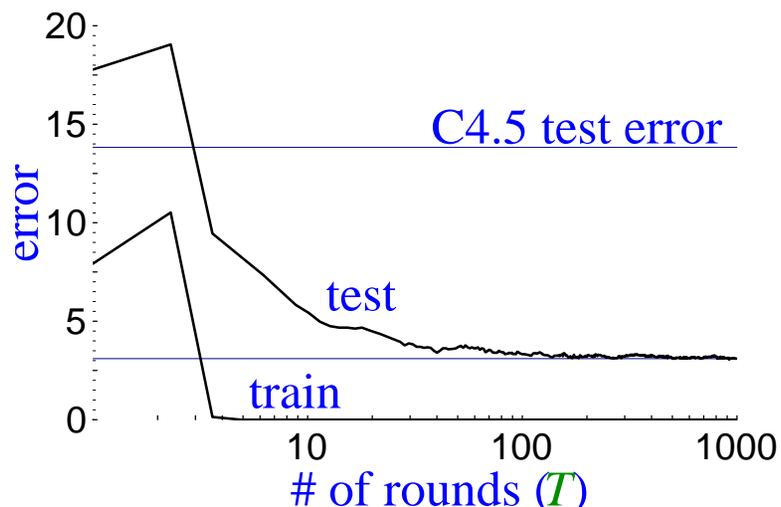
(boosting C4.5 on
"letter" dataset)

- test error does <u>not</u> increase, even after 1000 rounds
    - (total size $> 2,000,000$ nodes)

- test error continues to drop even after training error is zero!

	# rounds		
	5	100	1000
train error	0.0	0.0	0.0
test error	8.4	3.3	3.1

The Margins Explanation

- key idea:
  - training error only measures whether classifications are right or wrong
  - should also consider confidence of classifications

- recall: H_{final} is weighted majority vote of weak classifiers

- measure confidence by margin = strength of the vote

- empirical evidence and mathematical proof that:
  - large margins $\Rightarrow$ better generalization error (regardless of number of rounds)
  - boosting tends to increase margins of training examples (given weak learning assumption)

Application: Detecting Faces

[Viola & Jones]

- problem: find faces in photograph or movie

- weak classifiers: detect light/dark rectangles in image
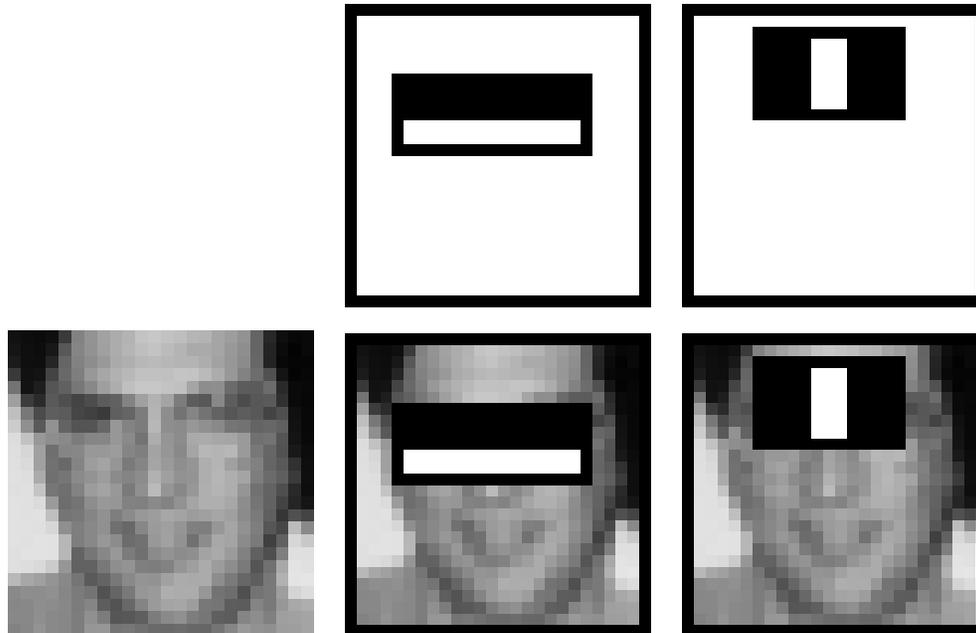
- many clever tricks to make extremely fast and accurate

Boosting

- fast (but not quite as fast as other methods)

- simple and easy to program

- flexible: can combine with <u>any</u> learning algorithm, e.g.
  - C4.5
  - very simple rules of thumb

- provable guarantees

- state-of-the-art accuracy

- tends not to overfit (but occasionally does)

- many applications

Support-Vector Machines

Geometry of SVM's
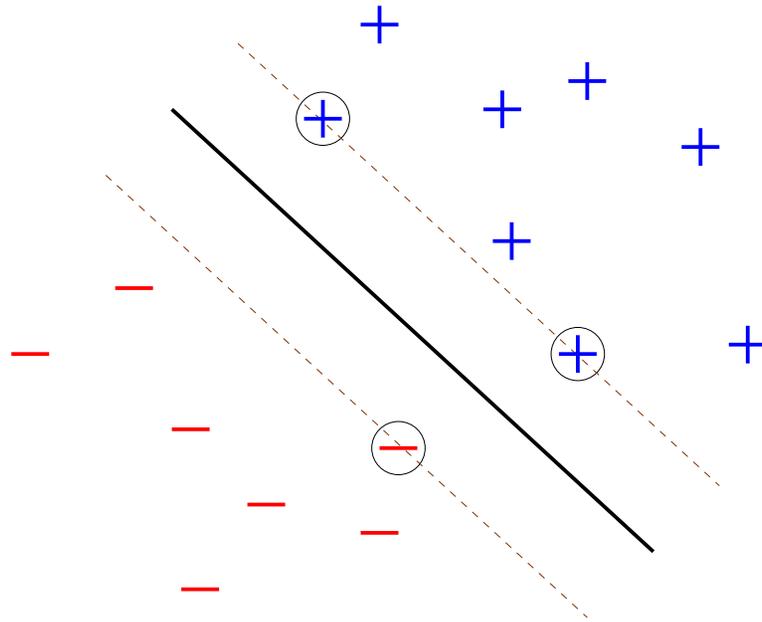
- given <u>linearly separable</u> data

- <u>margin</u> = distance to separating hyperplane

- choose hyperplane that maximizes minimum margin

- intuitively:

  - want to separate +'s from −'s as much as possible
  - margin = measure of confidence

Theoretical Justification

- let $\gamma =$ minimum margin
    $R =$ radius of enclosing sphere

- then

$$\text{VC-dim} \leq \left(\frac{R}{\gamma}\right)^2$$

  - so larger margins $\Rightarrow$ lower "complexity"
  - <u>independent</u> of number of dimensions

- in contrast, unconstrained hyperplanes in $\mathbb{R}^n$ have

$$\text{VC-dim} = (\# \text{ parameters}) = n + 1$$

Finding the Maximum Margin Hyperplane

- examples $\mathbf{x}_i, y_i$ where $y_i \in \{-1, +1\}$

- find hyperplane $\mathbf{v} \cdot \mathbf{x} = 0$ with $\| \mathbf{v} \| = 1$

- margin $= y(\mathbf{v} \cdot \mathbf{x})$

- maximize: γ
  subject to: $y_i(\mathbf{v} \cdot \mathbf{x}_i) \geq \gamma$ and $\| \mathbf{v} \| = 1$

- set $\mathbf{w} \leftarrow \mathbf{v}/\gamma \Rightarrow \gamma = 1/\| \mathbf{w} \|$

- minimize $\frac{1}{2} \| \mathbf{w} \|^2$
  subject to: $y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 1$

Convex Dual

- form Lagrangian, set $\partial/\partial\mathbf{w} = 0$

- get quadratic program:

- maximize $\sum\limits_i \alpha_i - \frac{1}{2} \sum\limits_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$

  subject to: $\alpha_i \geq 0$

- $\mathbf{w} = \sum\limits_i \alpha_i y_i \mathbf{x}_i$

- $\alpha_i =$ Lagrange multiplier
    - $> 0 \Rightarrow$ support vector

- key points:
    - optimal $\mathbf{w}$ is linear combination of support vectors
    - dependence on $\mathbf{x}_i$'s only through inner products
    - maximization problem is convex with no local maxima

What If Not Linearly Separable?

- answer #1: penalize each point by distance from margin 1, i.e., minimize:

$$\tfrac{1}{2} \|\mathbf{w}\|^2 + \text{constant} \cdot \sum_i \max\{0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i)\}$$

- answer #2: map into higher dimensional space in which data becomes linearly separable

Example
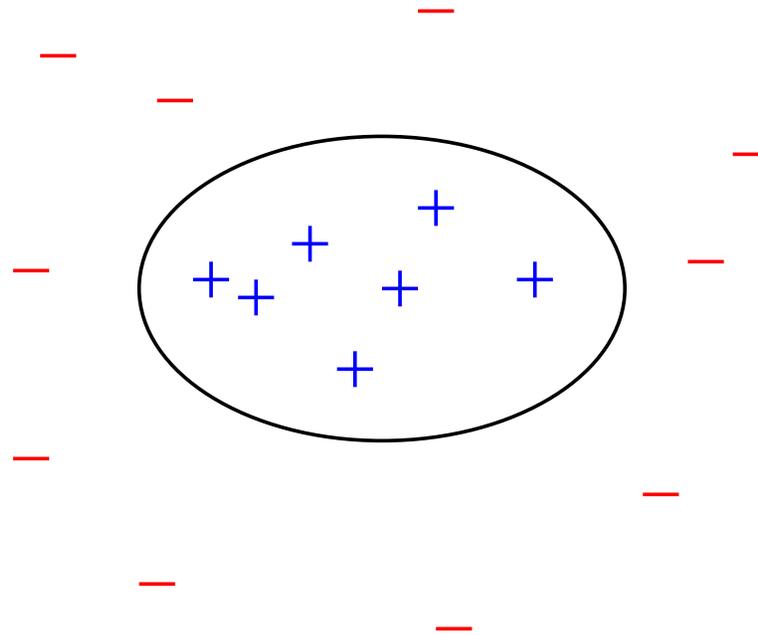
- <u>not</u> linearly separable
- map $\mathbf{x} = (x_1, x_2) \mapsto \Phi(\mathbf{x}) = (1, x_1, x_2, x_1 x_2, x_1^2, x_2^2)$
- hyperplane in mapped space has form
$$a + bx_1 + cx_2 + dx_1 x_2 + ex_1^2 + fx_2^2 = 0$$
  $=$ conic in original space
- linearly separable in mapped space

Higher Dimensions Don't (Necessarily) Hurt

- may project to very high dimensional space

- <u>statistically</u>, may not hurt since VC-dimension independent of number of dimensions ($(R/\gamma)^2$)

- <u>computationally</u>, only need to be able to compute inner products

$$\Phi(\mathbf{x}) \cdot \Phi(\mathbf{z})$$

- sometimes can do very efficiently using <u>kernels</u>

Example (cont.)

- modify Φ slightly:

$$\Phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2)$$

- then

$$\begin{aligned}
\Phi(\mathbf{x}) \cdot \Phi(\mathbf{z}) &= 1 + 2x_1z_1 + 2x_2z_2 + 2x_1x_2z_1z_2 + x_1^2z_1^2 + x_2^2 + z_2^2 \\
&= (1 + x_1z_1 + x_2z_2)^2 \\
&= (1 + \mathbf{x} \cdot \mathbf{z})^2
\end{aligned}$$

- in general, for polynomial of degree d, use $(1 + \mathbf{x} \cdot \mathbf{z})^d$

- very efficient, even though finding hyperplane in $O(n^d)$ dimensions

Kernels

- kernel = function K for computing
$$K(\mathbf{x}, \mathbf{z}) = \Phi(\mathbf{x}) \cdot \Phi(\mathbf{z})$$

- permits <u>efficient</u> computation of SVM's in very high dimensions

- K can be any symmetric, positive semi-definite function (Mercer's theorem)

- some kernels:
  - polynomials
  - Gaussian $\exp\left(-\|\mathbf{x} - \mathbf{z}\|^2 / 2\sigma\right)$
  - defined over structures (trees, strings, sequences, etc.)

- evaluation:
$$\mathbf{w} \cdot \Phi(\mathbf{x}) = \Sigma \, \alpha_i y_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}) = \Sigma \, \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})$$

  - time depends on # support vectors

SVM's versus Boosting

- both are large-margin classifiers
  (although with slightly different definitions of margin)

- both work in very high dimensional spaces
  (in boosting, dimensions correspond to weak classifiers)

- but different tricks are used:
  - SVM's use kernel trick
  - boosting relies on weak learner to select one dimension (i.e., weak classifier) to add to combined classifier

Application: Text Categorization

[Joachims]

- goal: classify text documents
  - e.g.: spam filtering
  - e.g.: categorize news articles by topic

- need to represent text documents as vectors in $\mathbb{R}^n$:
  - one dimension for each word in vocabulary
  - value = # times word occurred in particular document
  - (many variations)

- kernels don't help much

- performance state of the art

SVM's

- fast algorithms now available, but not so simple to program (but good packages available)

- state-of-the-art accuracy

- power and flexibility from kernels

- theoretical justification

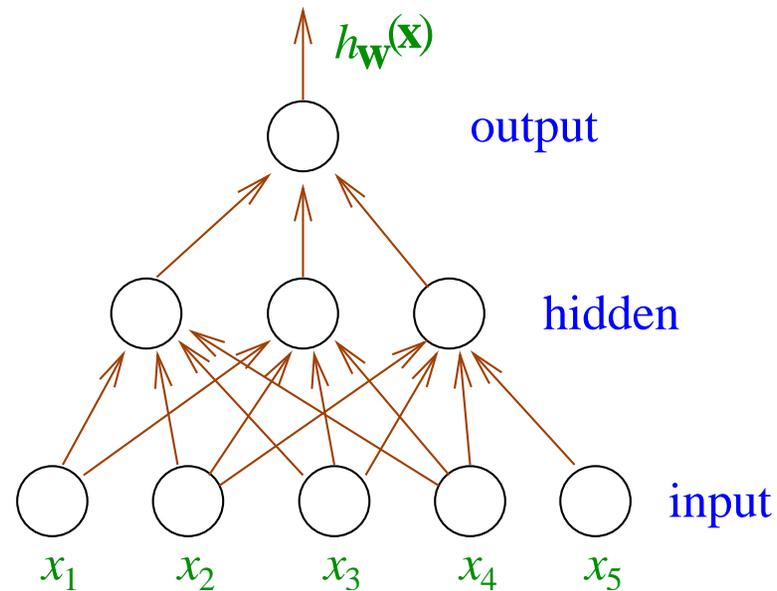- many applications

Neural Networks

The Neural Analogy

- perceptron (= linear threshold function) looks a lot like a neuron

Schematic of biological neuron.
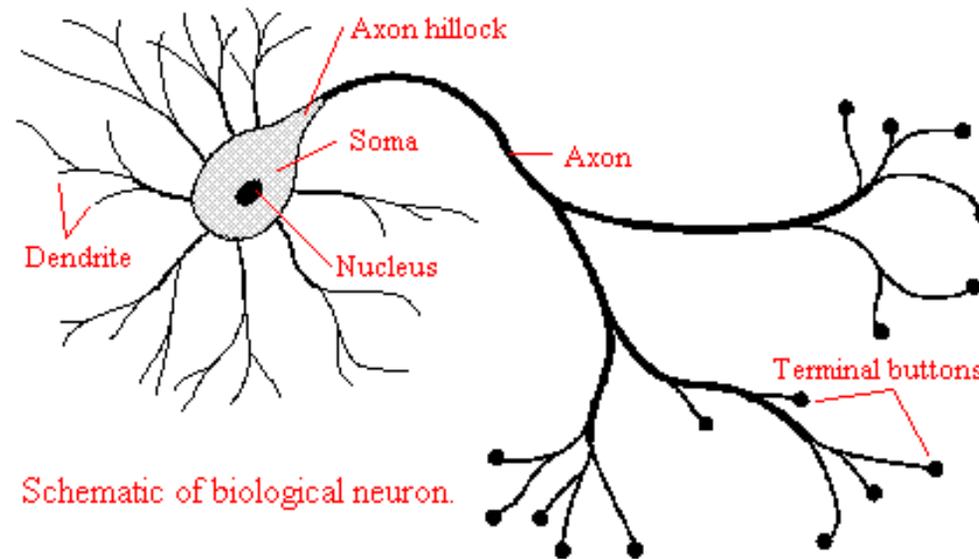
- other neurons fire (inputs)
- when electrical potential exceeds threshold, fires (output)
- inputs: $a_1, \ldots, a_n \in \{0, 1\}$
- weights: $w_1, \ldots, w_n \in \mathbb{R}$
- "activation" $= \begin{cases} 1 \text{ if } \sum w_i a_i > \theta \\ 0 \text{ else} \end{cases}$
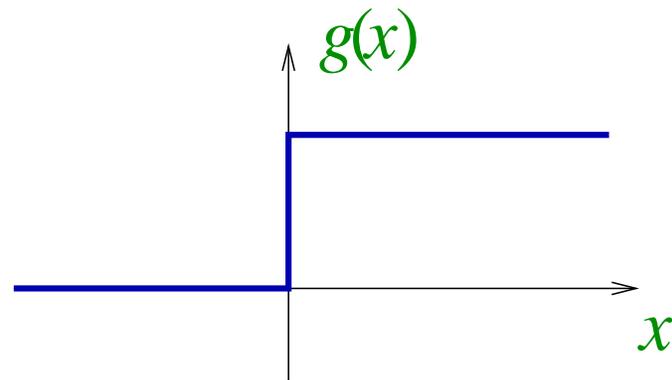
A Network of Neurons

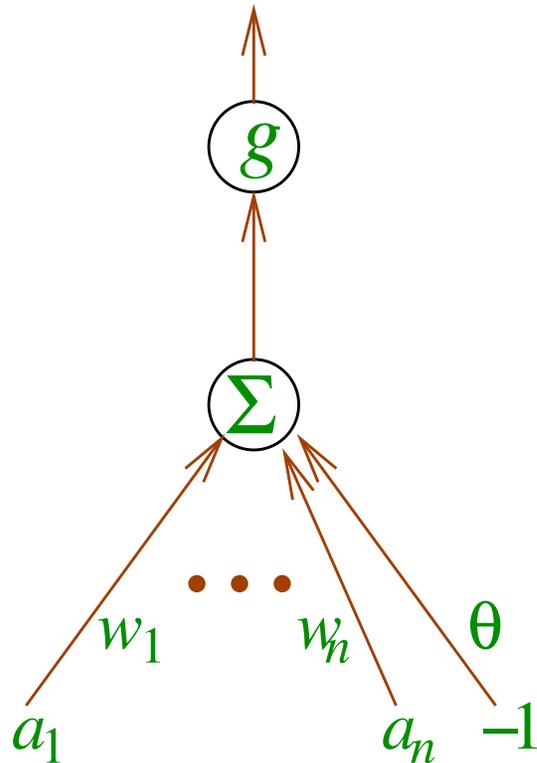- <u>idea</u>: put perceptrons in network

- weights on every edge

- each unit = perceptron

- dramatic increase in representation power
  (not necessarily a good thing for learning)

- great flexibility in choice of architecture

Perceptron Units

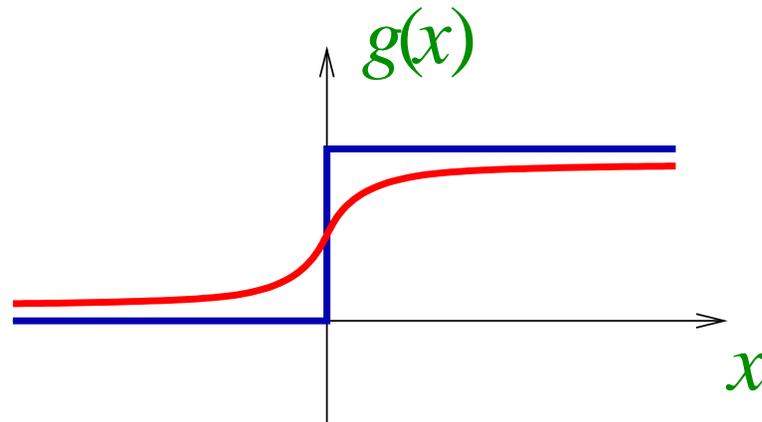- **problem**: overall network computation is horribly discontinuous because of g

  - optimizing network weights easier when everything continuous

Smoothed Threshold Functions

- idea: approximate g with smoothed threshold function

- e.g., use $g(x) = \dfrac{1}{1 + e^{-x}}$

- now $h_{\mathbf{w}}(\mathbf{x})$ is continuous and differentiable in both inputs $\mathbf{x}$ and weights $\mathbf{w}$

Finding Weights

- given $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ where $y_i \in \{0, 1\}$

- how to find weights $\mathbf{w}$?

- want network output $h_{\mathbf{w}}(\mathbf{x}_i)$ "close" to y_i

- typical measure of closeness:

$$\text{"energy"} \quad E(\mathbf{w}) = \sum_i \left(h_{\mathbf{w}}(\mathbf{x}_i) - y_i \right)^2$$

Minimizing Energy

- E is a continuous and differentiable function of **w**

- minimize using <u>gradient descent</u>:
  - start with any **w**
  - repeatedly adjust **w** by taking tiny steps in direction of steepest descent

- easy to compute gradients
  - turns out to have simple recursive form in which error signal is <u>backpropagated</u> from output to inputs

Implementation Details

- often do gradient descent step based just on single example (and repeat for all examples in training set)

- often slow to converge
  - speed up using techniques like conjugate gradient descent

- can get stuck in local minima or large flat regions

- can overfit
  - use <u>regularization</u> to keep weights from getting too large

$$E(\mathbf{w}) = \sum_i \left(h_{\mathbf{w}}(\mathbf{x}_i) - y_i \right)^2 + \beta \|\mathbf{w}\|^2$$

Application: Optical Character Recognition

[LeCun, Bottou, Bengio & Haffner]

- problem: recognize handwritten characters

- LeNet-5:
  - 7 layers (plus inputs) specially designed for OCR
  - extended for segmentation
  - very high accuracy

Neural Nets

- can be slow to converge

- can be difficult to get right architecture, and difficult to tune parameters

- not state-of-the-art as a general method

- with proper care, can do very well on particular problems, often with specialized architecture

Further reading on machine learning in general:

Ethem Alpaydin. *Introduction to machine learning*. MIT Press, 2004.

Luc Devroye, Lázló Györfi and Gábor Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer, 1996.

Richard O. Duda, Peter E. Hart and David G. Stork. *Pattern Classification (2nd ed.)*. Wiley, 2000.

Trevor Hastie, Robert Tibshirani and Jerome Friedman. *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*. Springer, 2001.

Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

Vladimir N. Vapnik. *Statistical Learning Theory*. Wiley, 1998.

Decision trees:

Leo Breiman, Jerome H. Friedman, Richard A. Olshen and Charles J. Stone. *Classification and Regression Trees*. Wadsworth & Brooks, 1984.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

Boosting:

Ron Meir and Gunnar Rätsch. An Introduction to Boosting and Leveraging. In *Advanced Lectures on Machine Learning (LNAI2600)*, 2003. http://www.boosting.org/papers/MeiRae03.pdf

Robert E. Schapire. The boosting approach to machine learning: An overview. In *Nonlinear Estimation and Classification*, Springer, 2003. Available from: www.cs.princeton.edu/∼schapire/boost.html.

Many more papers, tutorials, etc. available at www.boosting.org.

Support-vector machines:

Nello Cristianni and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000. See www.support-vector.net.

Many more papers, tutorials, etc. available at www.kernel-machines.org.

Neural nets:

Christopher M. Bishop. *Neural networks for Pattern Recognition*. Oxford University Press, 1995.