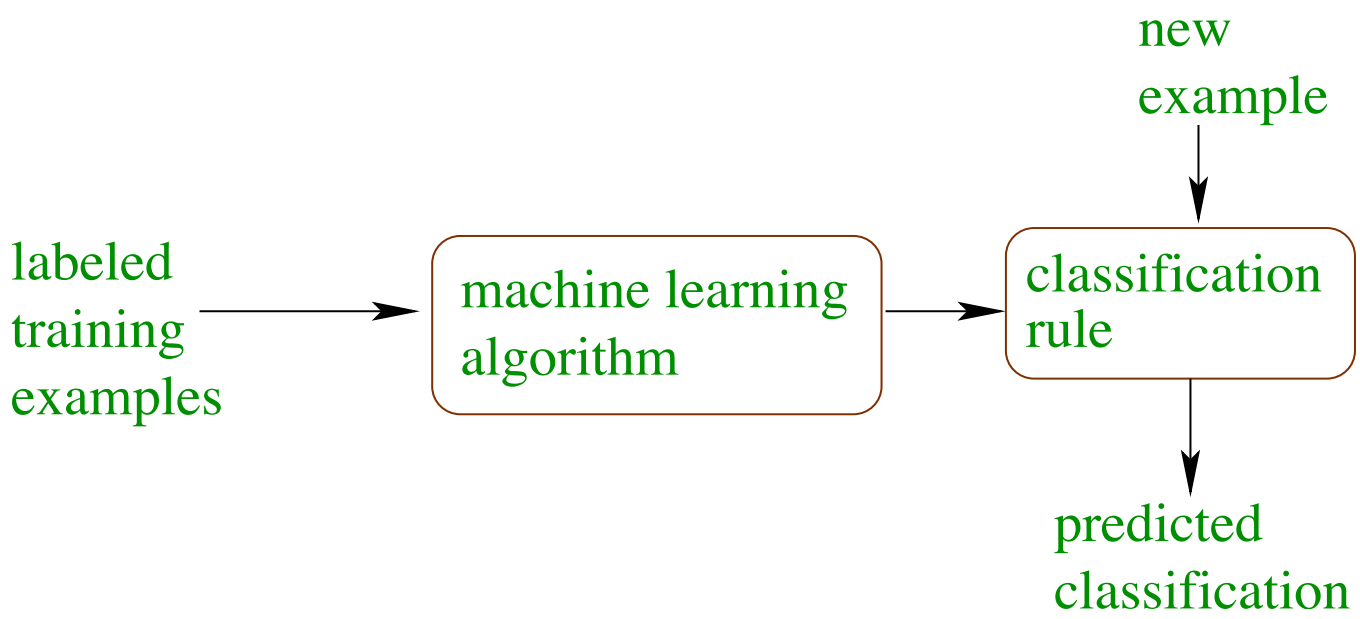
Machine Learning Algorithms for Classification

Rob Schapire
Princeton University

Machine Learning

- studies how to automatically learn to make accurate predictions based on past observations
- classification problems:
  - classify examples into given set of categories



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)
  ⋮

Characteristics of Modern Machine Learning

- primary goal: highly accurate predictions on test data
  - goal is not to uncover underlying "truth"
- methods should be general purpose, fully automatic and "off-the-shelf"
  - however, in practice, incorporation of prior, human knowledge is crucial
- rich interplay between theory and practice
- emphasis on methods that can handle large datasets

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
  - automatic method to search for hypotheses explaining data
  - cheap and flexible — can apply to any learning task
- disadvantages
  - need a lot of labeled data
  - error prone — usually impossible to get perfect accuracy

This Talk

- machine learning algorithms:
  - decision trees
    - conditions for successful learning
  - boosting
  - support-vector machines
- others not covered:
  - neural networks
  - nearest neighbor algorithms
  - Naive Bayes
  - bagging
  - random forests
  - ⋮
- practicalities of using machine learning algorithms

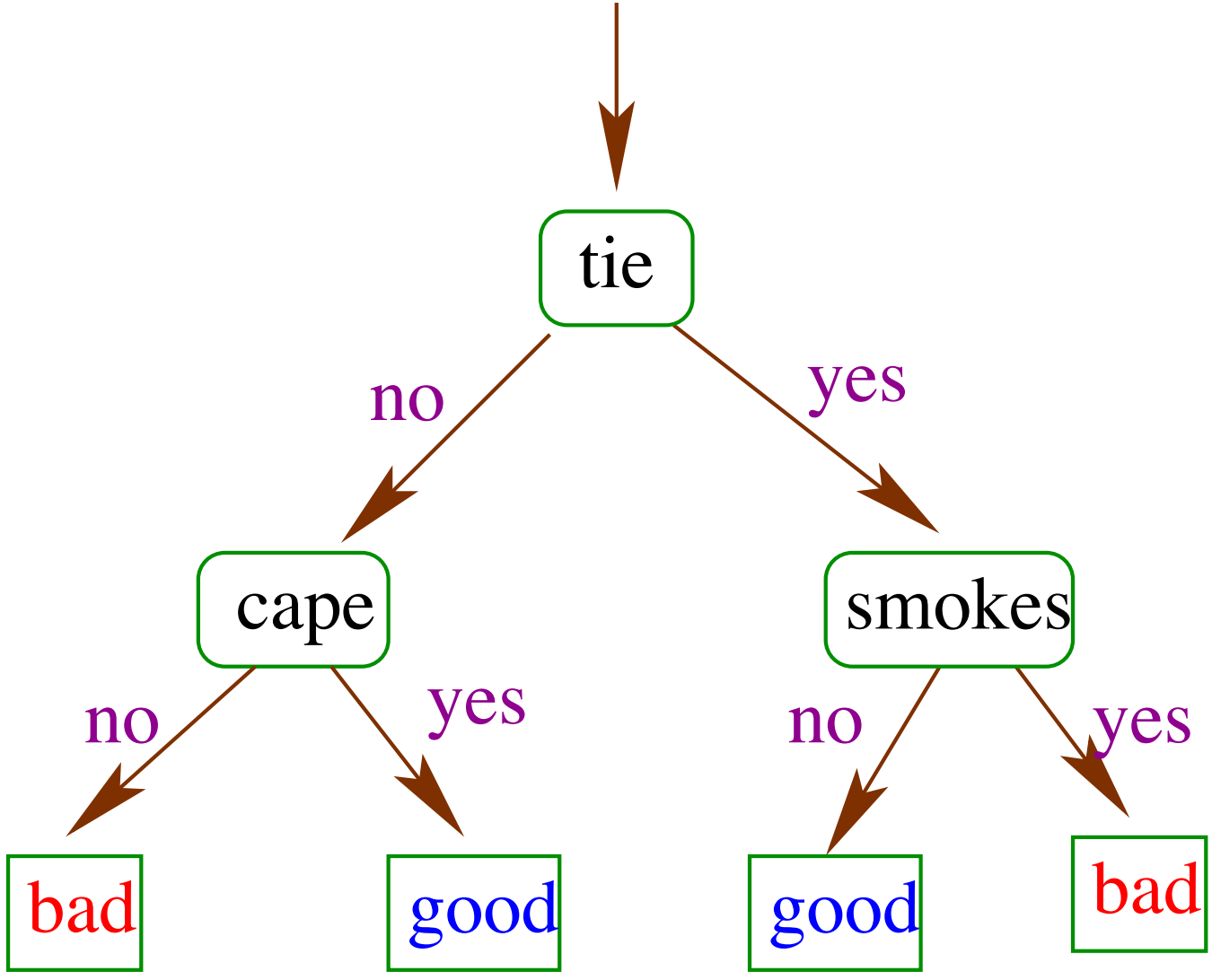
Decision Trees

Example: Good versus Evil

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

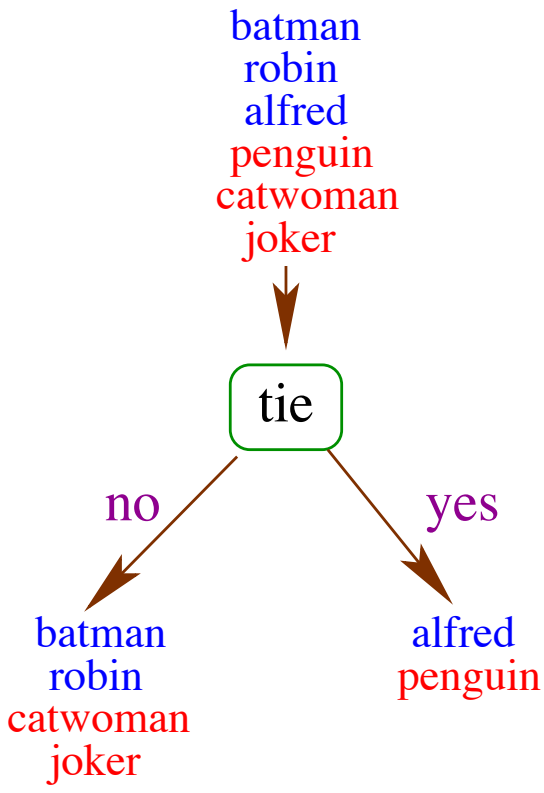
	sex	mask	cape	tie	ears	smokes	class
			training data				
batman	male	yes	yes	no	yes	no	Good
robin	male	yes	yes	no	no	no	Good
alfred	male	no	no	yes	no	no	Good
penguin	male	no	no	yes	no	yes	Bad
catwoman	female	yes	no	no	yes	no	Bad
joker	male	no	no	no	no	no	Bad
			test data				
batgirl	female	yes	yes	no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

A Decision Tree Classifier



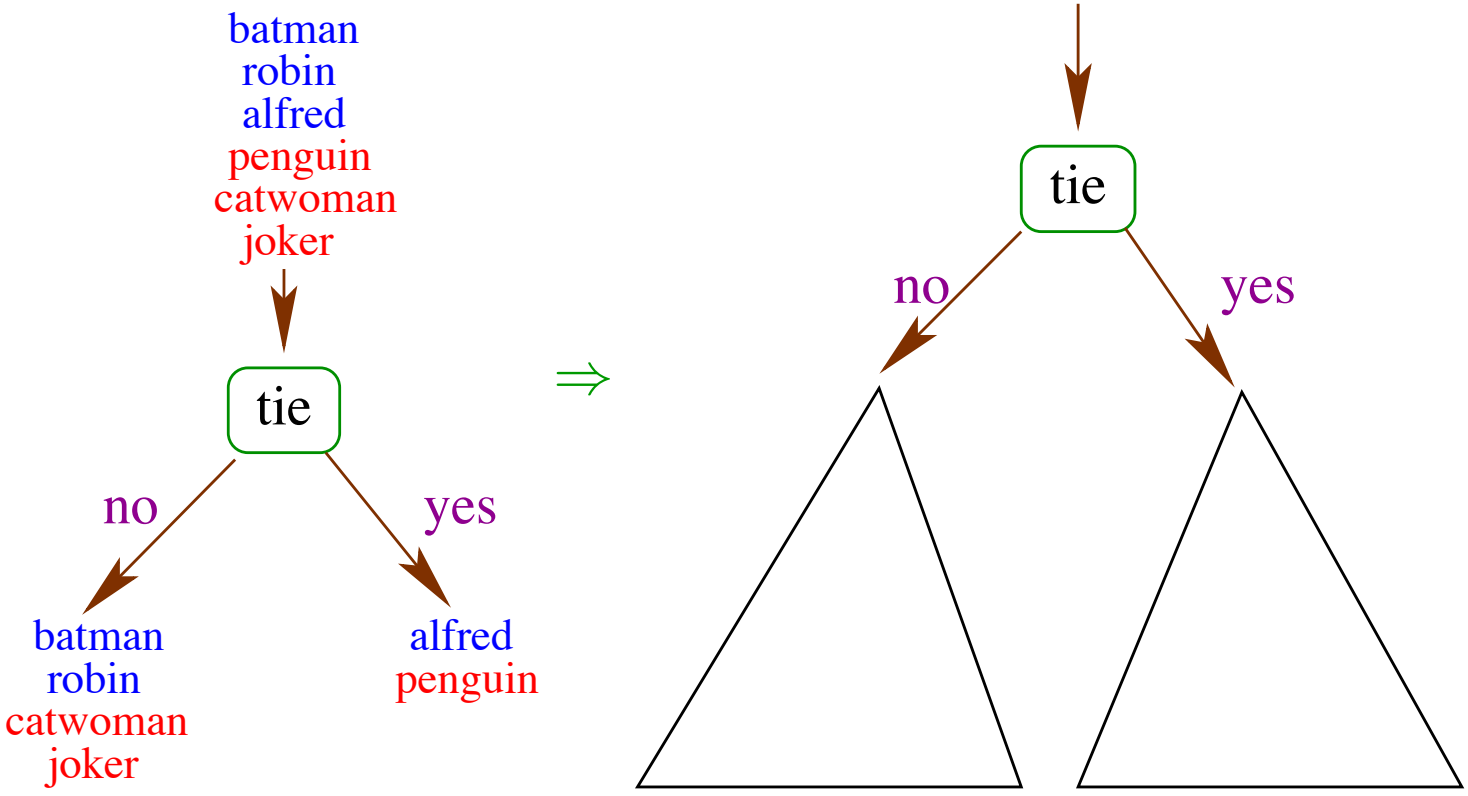
How to Build Decision Trees

- choose rule to split on
- divide data using splitting rule into disjoint subsets



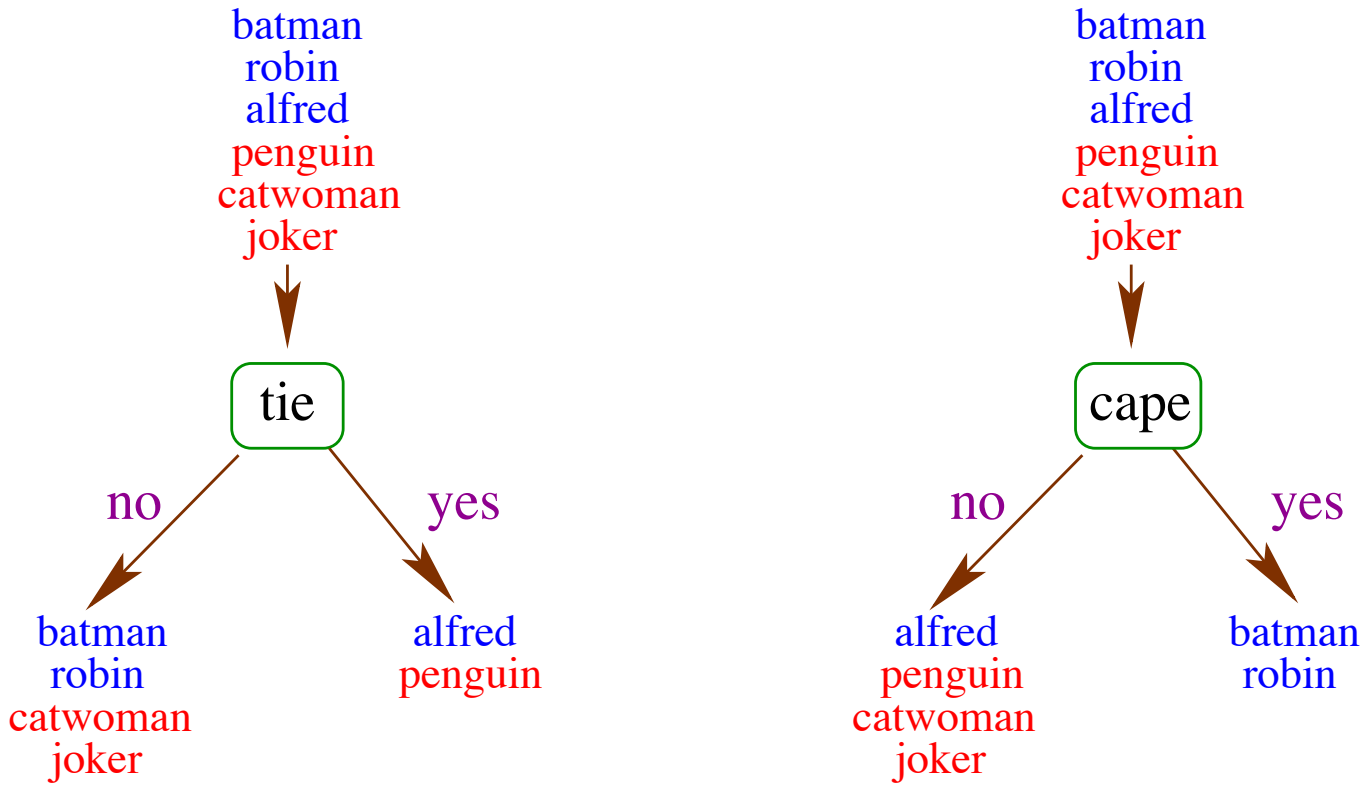
How to Build Decision Trees

- choose rule to split on
- divide data using splitting rule into disjoint subsets
- repeat recursively for each subset
- stop when leaves are (almost) "pure"



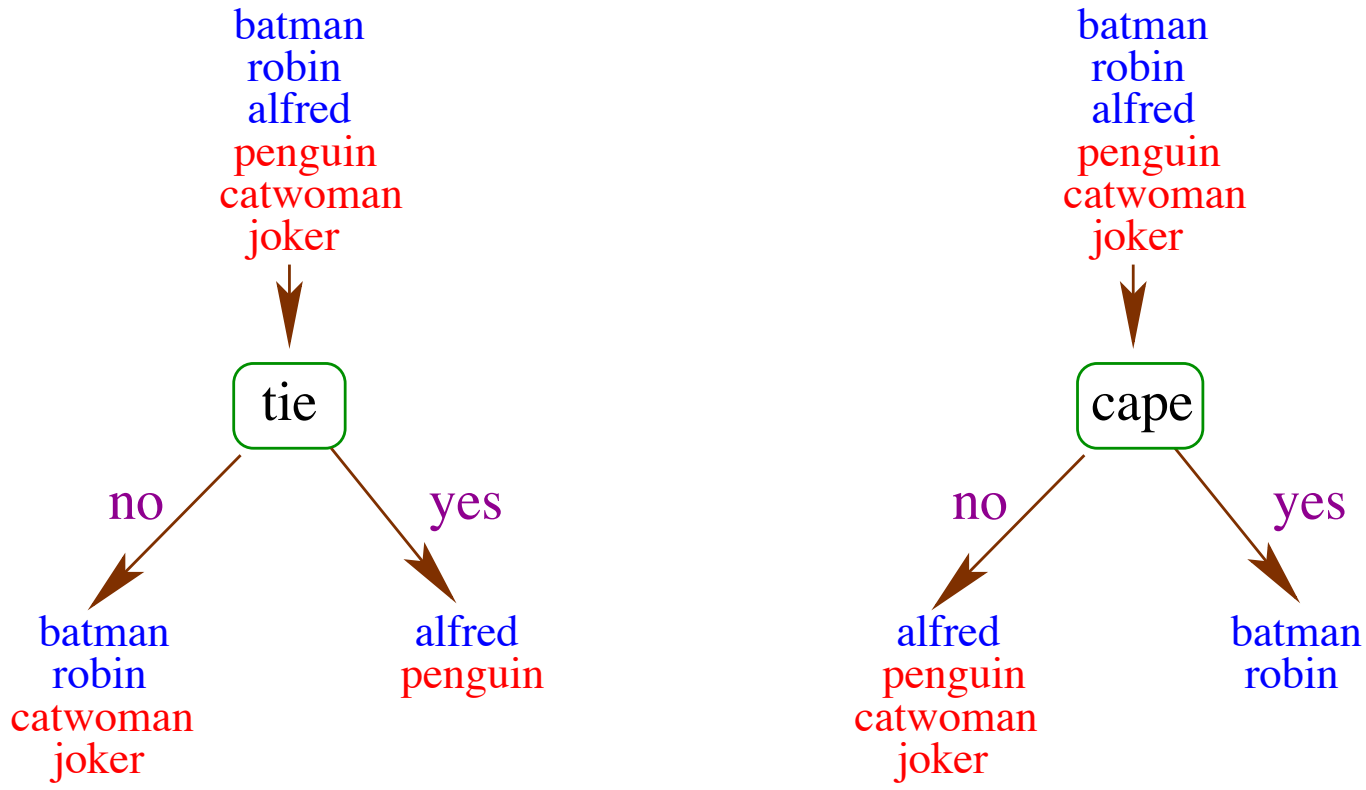
How to Choose the Splitting Rule

- key problem: choosing best rule to split on:



How to Choose the Splitting Rule

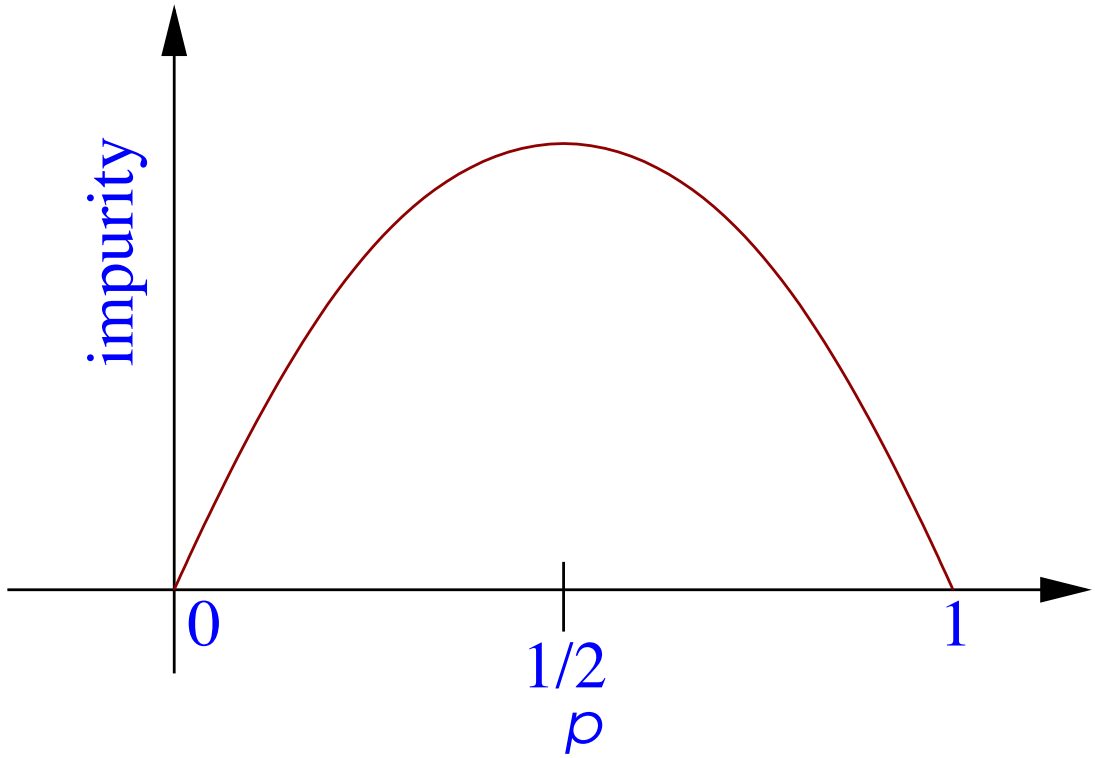
- key problem: choosing best rule to split on:



- idea: choose rule that leads to greatest increase in "purity"

How to Measure Purity

- want (im)purity function to look like this:
  (p = fraction of positive examples)

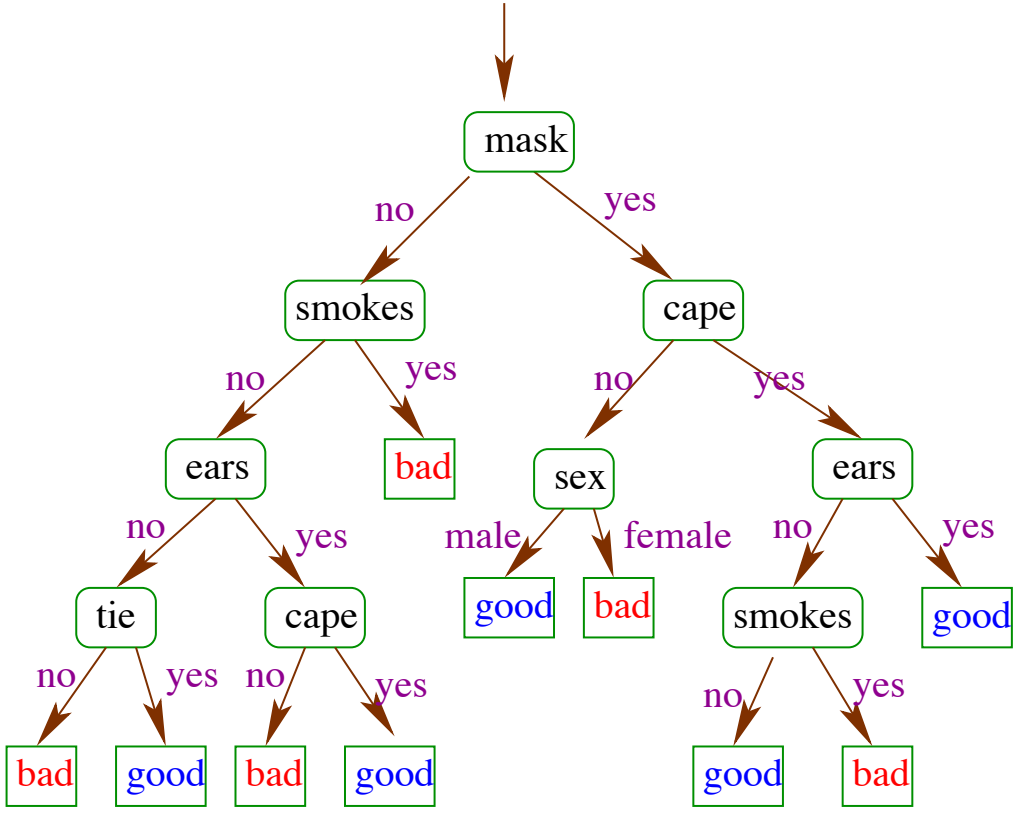


- commonly used impurity measures:
  - entropy: $-p \ln p - (1-p) \ln(1-p)$
  - Gini index: $p(1-p)$

Kinds of Error Rates

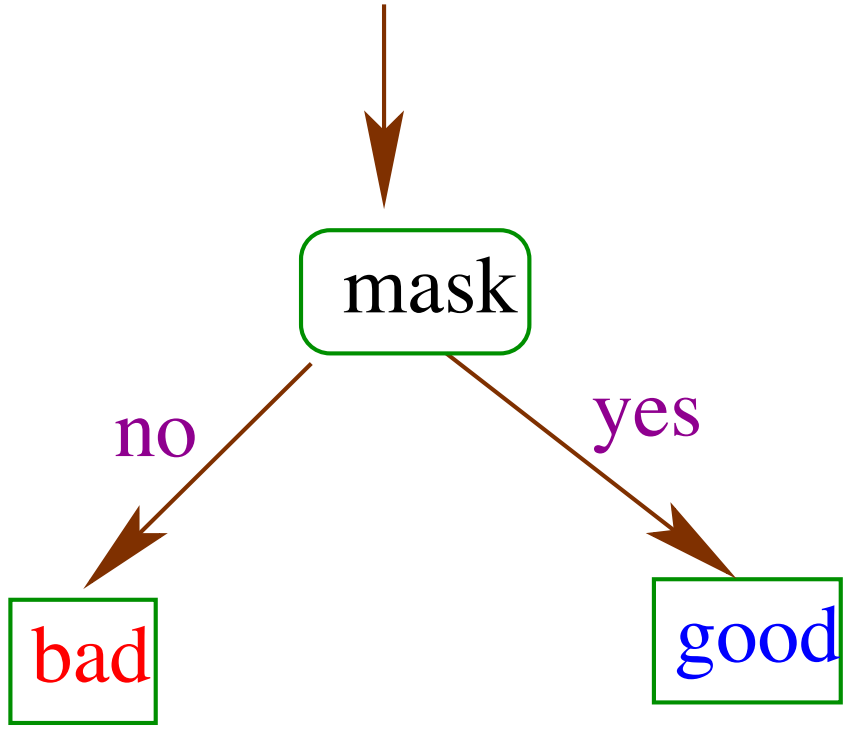
- training error = fraction of training examples misclassified
- test error = fraction of test examples misclassified
- generalization error = probability of misclassifying new random example

A Possible Classifier



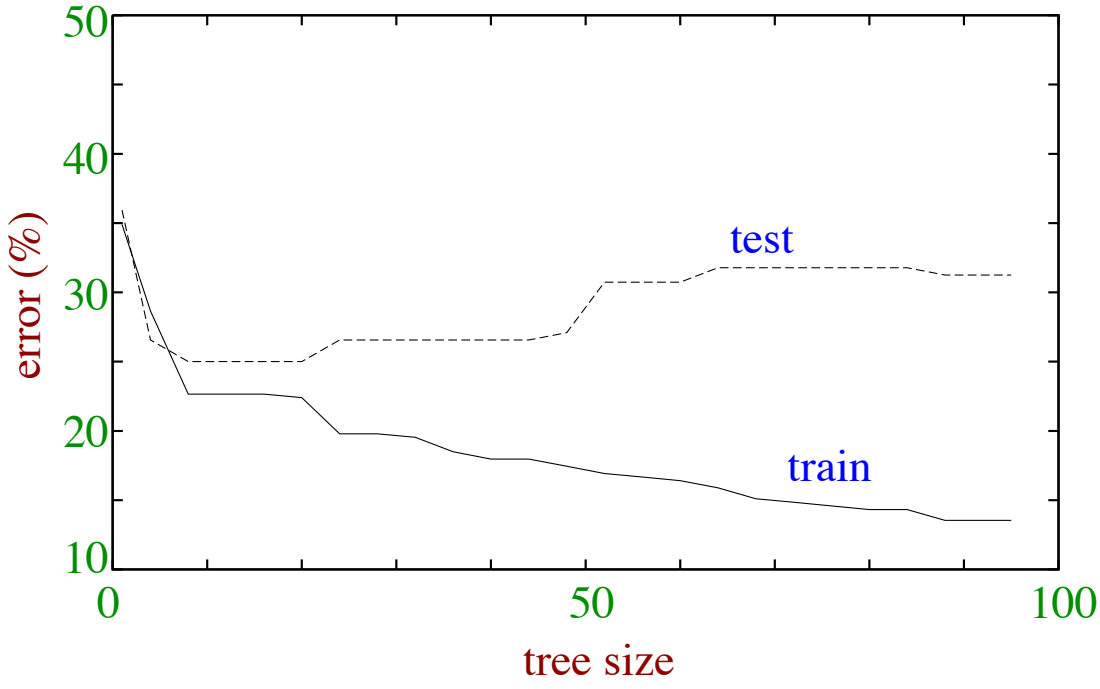
- perfectly classifies training data
- BUT: intuitively, overly complex

Another Possible Classifier



- overly simple
- doesn't even fit available data

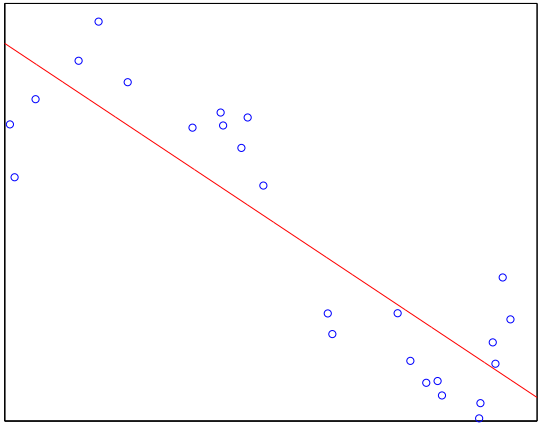
Tree Size versus Accuracy



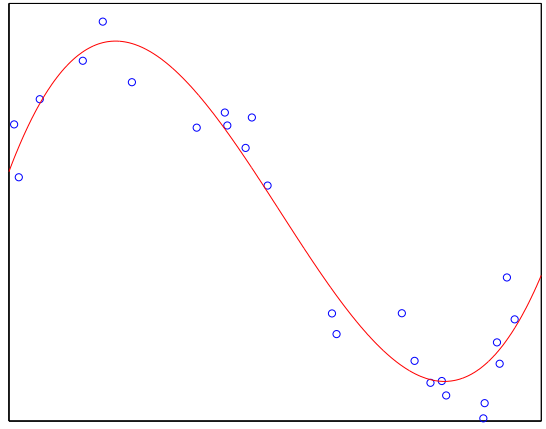
- trees must be big enough to fit training data
  (so that "true" patterns are fully captured)
- BUT: trees that are too big may overfit
  (capture noise or spurious patterns in the data)
- significant problem: can't tell best tree size from training error

Overfitting Example

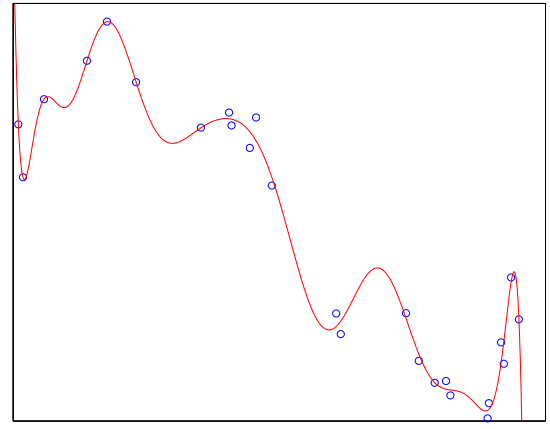
- fitting points with a polynomial



underfit
(degree = 1)



ideal fit
(degree = 3)



overfit
(degree = 20)

Building an Accurate Classifier

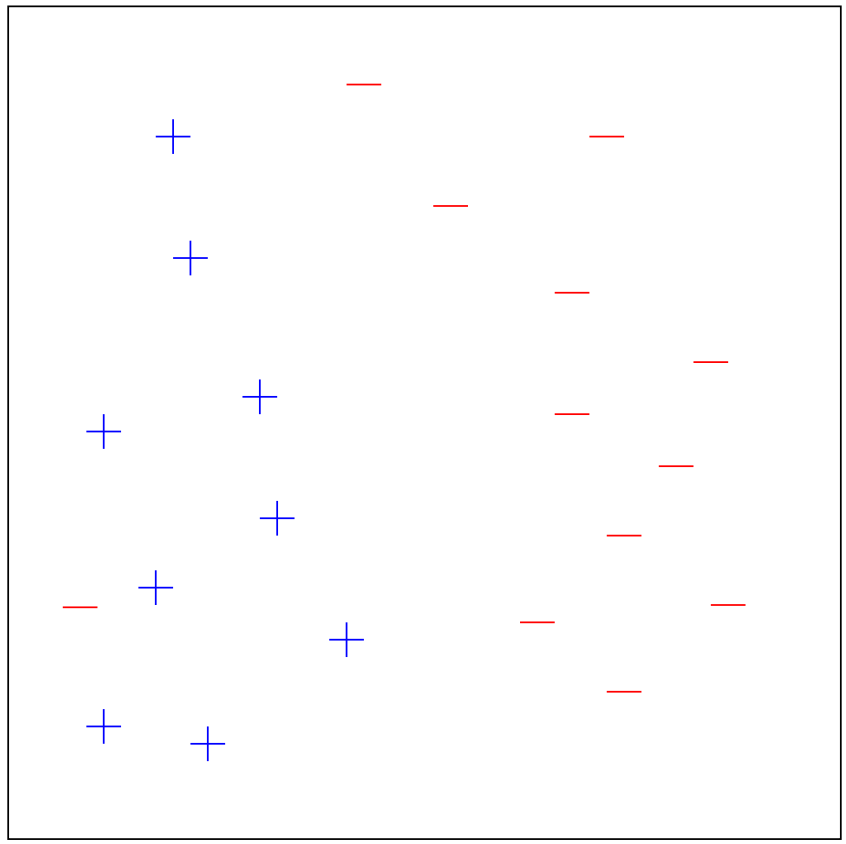
- for good test peformance, need:
  - enough training examples
  - good performance on training set
  - classifier that is not too "complex" ("Occam's razor")
- classifiers should be "as simple as possible, but no simpler"
- "simplicity" closely related to prior expectations

Building an Accurate Classifier

- for good test peformance, need:
  - enough training examples
  - good performance on training set
  - classifier that is not too "complex" ("Occam's razor")
- classifiers should be "as simple as possible, but no simpler"
- "simplicity" closely related to prior expectations
- measure "complexity" by:
  - number bits needed to write down
  - number of parameters
  - VC-dimension

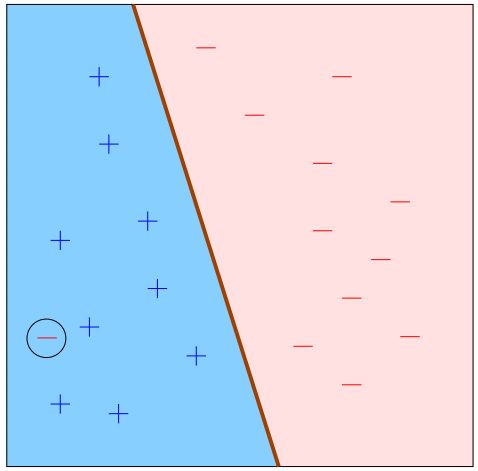
Example

Training data:



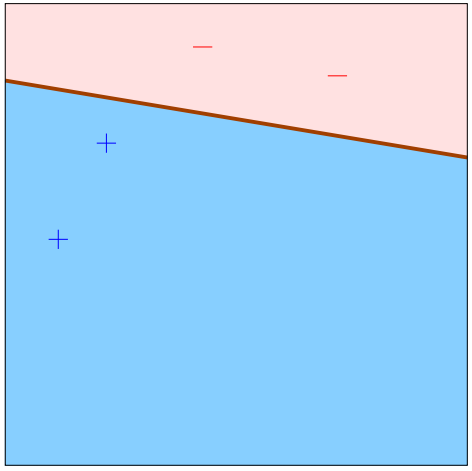
Good and Bad Classifiers

Good:

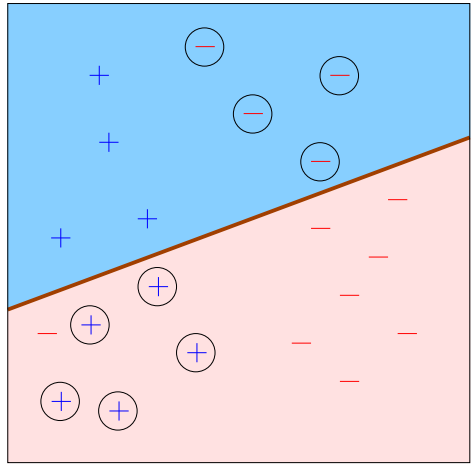


sufficient data
low training error
simple classifier

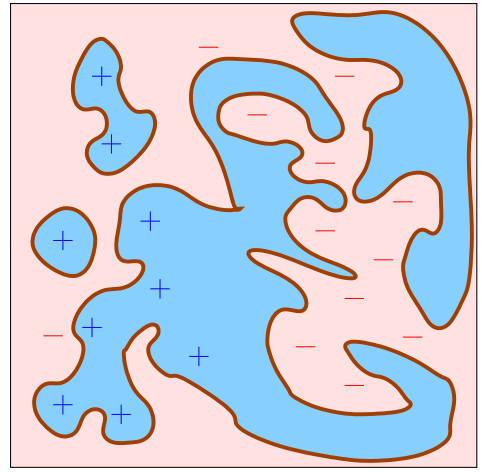
Bad:



insufficient data



training error
too high



classifier
too complex

Theory

- can prove:

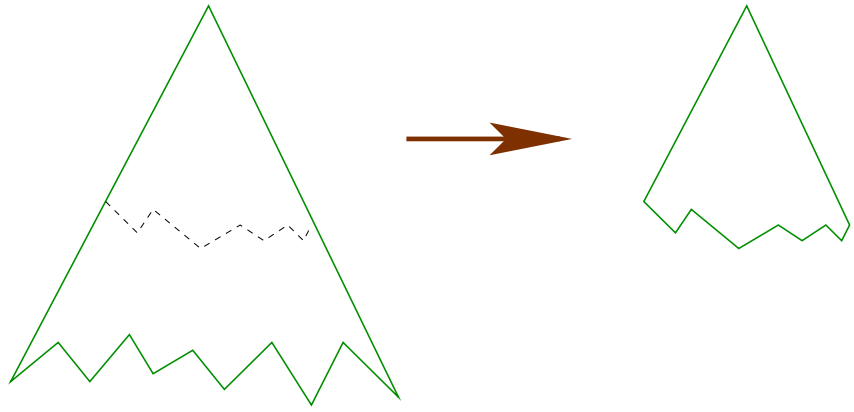
$$\text{(generalization error)} \leq \text{(training error)} + \tilde{O}\left(\sqrt{\frac{d}{m}}\right)$$

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
⋮	⋮	⋮

- goal: have computer learn from examples to distinguish spam from non-spam

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
⋮	⋮	⋮

- goal: have computer learn from examples to distinguish spam from non-spam
- main observation:
  - easy to find "rules of thumb" that are "often" correct
    - *If* 'v1agr@' *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 choose examples on each round?
  - concentrate on "hardest" examples (those most often misclassified by previous rules of thumb)
- how to combine 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 $y_i \in \{-1, +1\}$

AdaBoost

- given training examples (x_i, y_i) where $y_i \in \{-1, +1\}$
- for $t = 1, \ldots, T$:
  - train weak classifier ("rule of thumb") h_t on D_t

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 weak classifier ("rule of thumb") h_t on D_t

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 weak classifier ("rule of thumb") h_t on D_t
  - choose $\alpha_t > 0$
  - compute new distribution D_{t+1}:
    - for each example i:

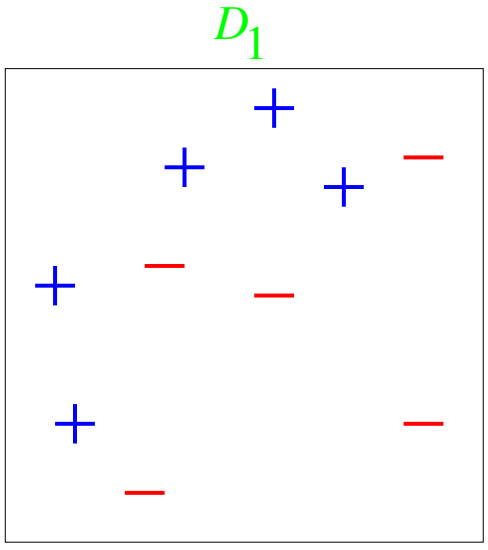
      multiply $D_t(x_i)$ by $\begin{cases} e^{-\alpha_t} & (< 1) \quad \text{if } y_i = h_t(x_i) \\ e^{\alpha_t} & (> 1) \quad \text{if } y_i \neq h_t(x_i) \end{cases}$
    - renormalize

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 weak classifier ("rule of thumb") h_t on D_t
  - choose $\alpha_t > 0$
  - compute new distribution D_{t+1}:
    - for each example i:

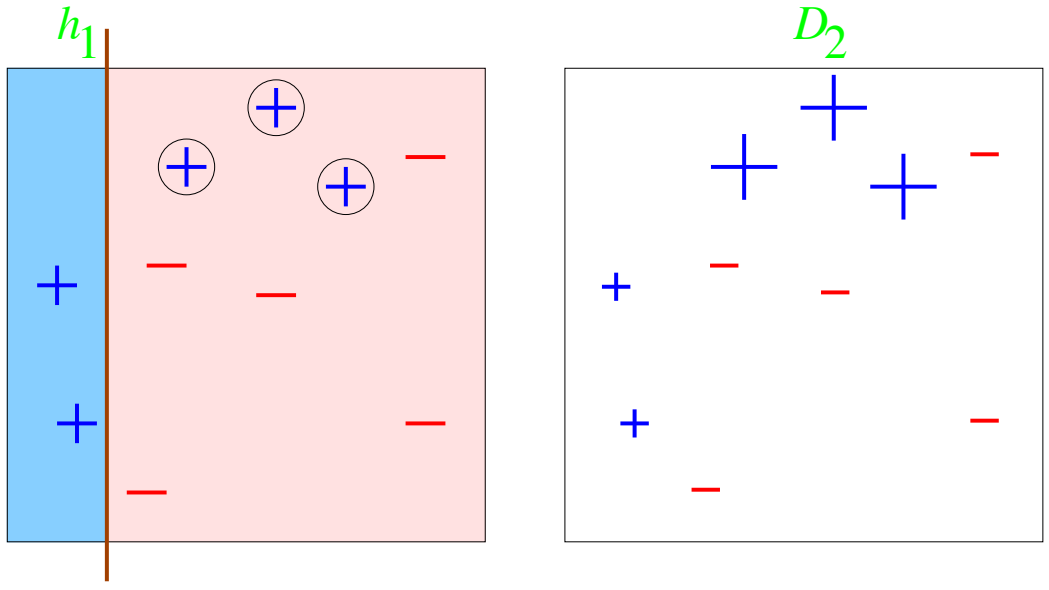
      multiply $D_t(x_i)$ by $\begin{cases} e^{-\alpha_t} & (< 1) \quad \text{if } y_i = h_t(x_i) \\ e^{\alpha_t} & (> 1) \quad \text{if } y_i \neq h_t(x_i) \end{cases}$
    - renormalize
- output final classifier $H_{\text{final}}(x) = \text{sign}\left(\sum_t \alpha_t h_t(x)\right)$

Toy Example



weak classifiers = vertical or horizontal half-planes

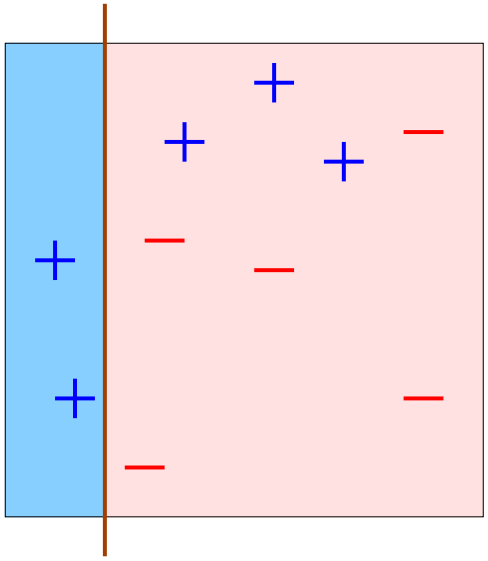
Round 1

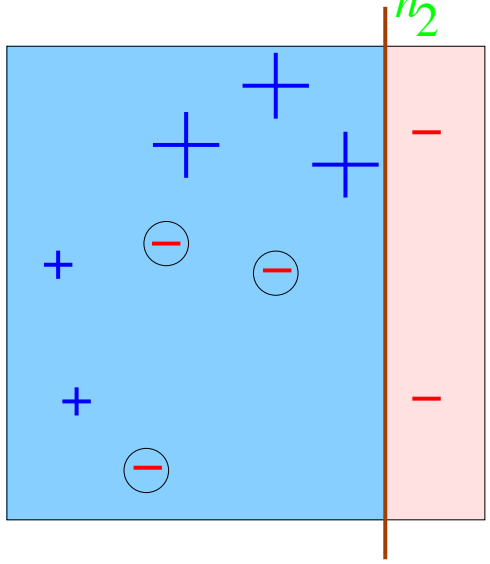


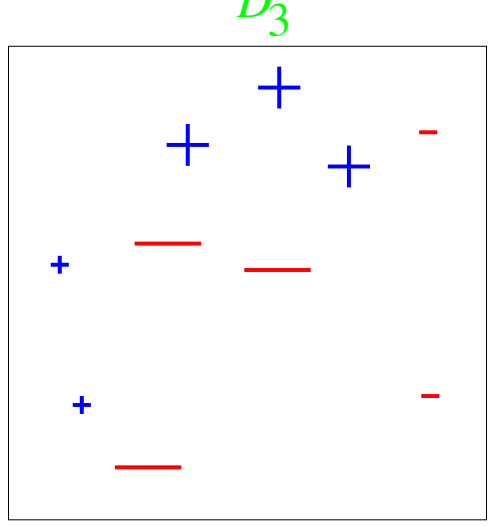
$\varepsilon_1 = 0.30$

$\alpha_1 = 0.42$

Round 2



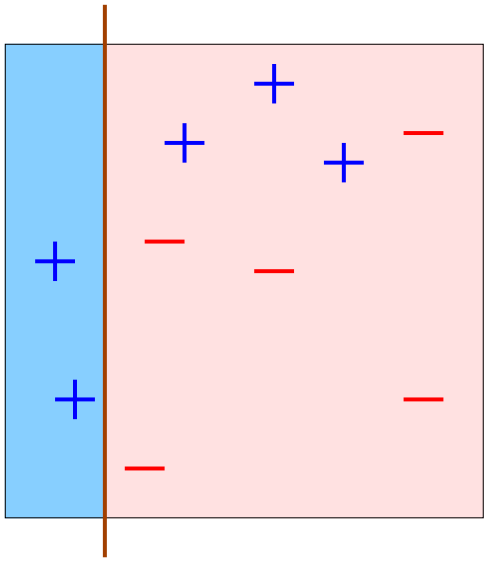


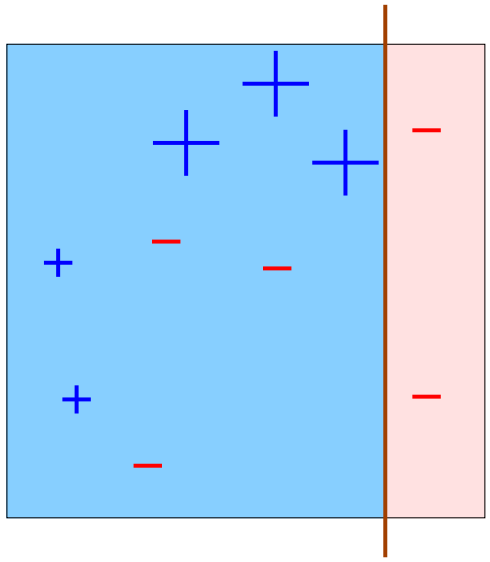


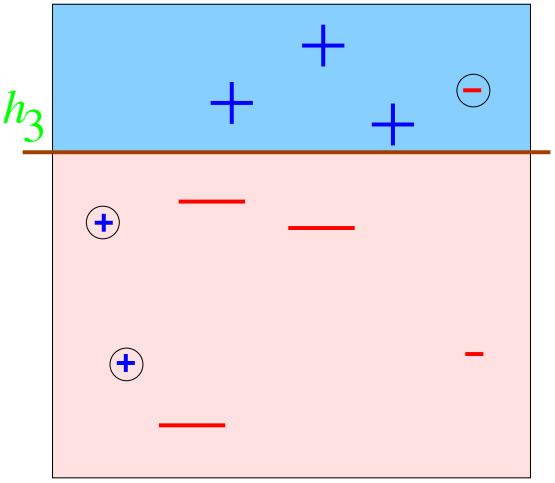
$\varepsilon_2 = 0.21$

$\alpha_2 = 0.65$

Round 3



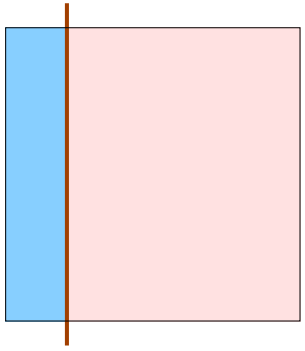
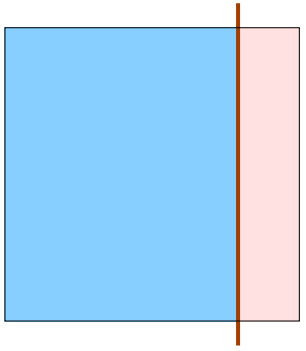
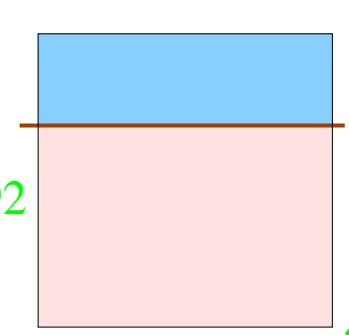


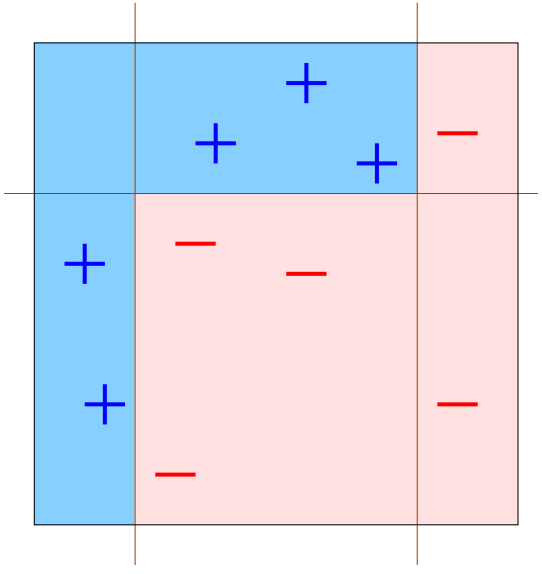


$\varepsilon_3 = 0.14$

$\alpha_3 = 0.92$

Final Classifier

$H_{\text{final}} = \text{sign}\Big($ 0.42  $+\,0.65$  $+\,0.92$  $\Big)$

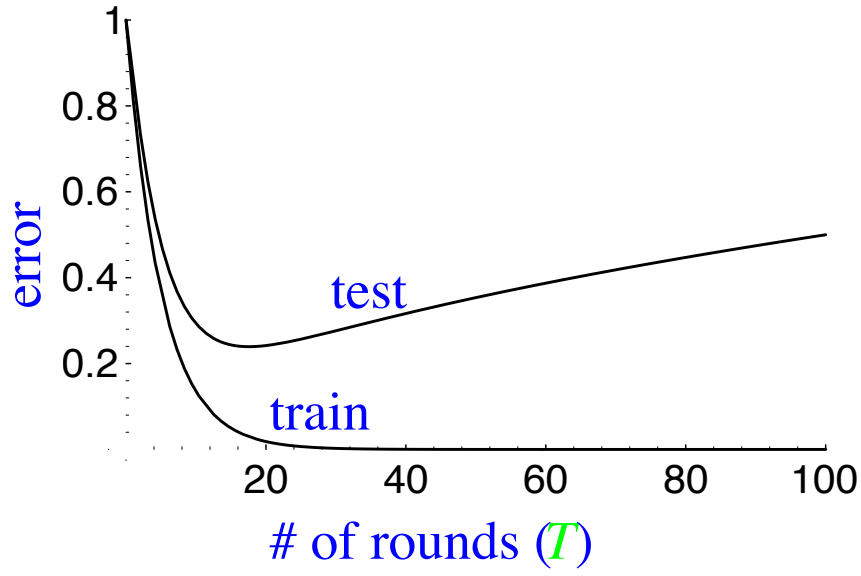
$=$ 

Theory: Training Error

- weak learning assumption: 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:

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

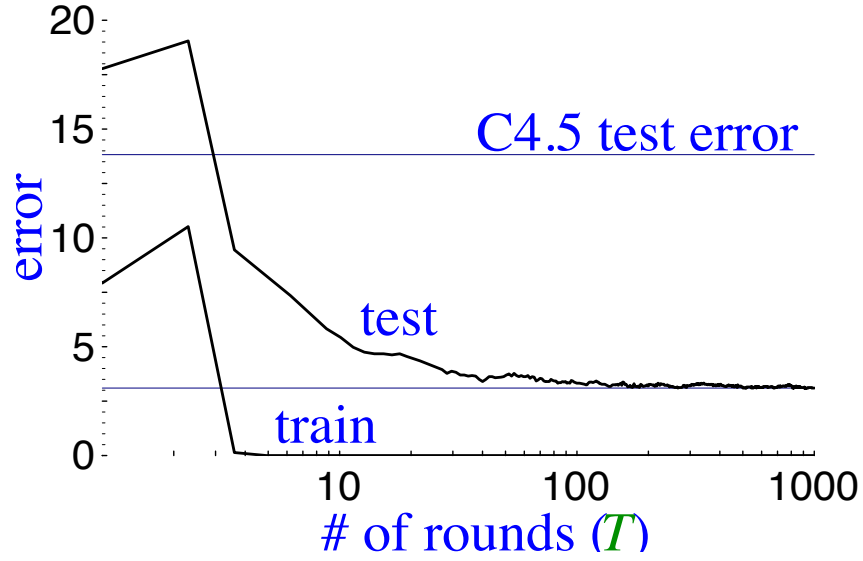
How Will Test Error Behave? (A First Guess)



expect:

- training error to continue to drop (or reach zero)
- test error to increase when H_{final} becomes "too complex"
  - "Occam's razor"
  - overfitting
    - hard to know when to stop training

Actual Typical Run



(boosting C4.5 on "letter" dataset)

- test error does not 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

- Occam's razor wrongly predicts "simpler" rule is better

The Margins Explanation

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

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

The Margins Explanation

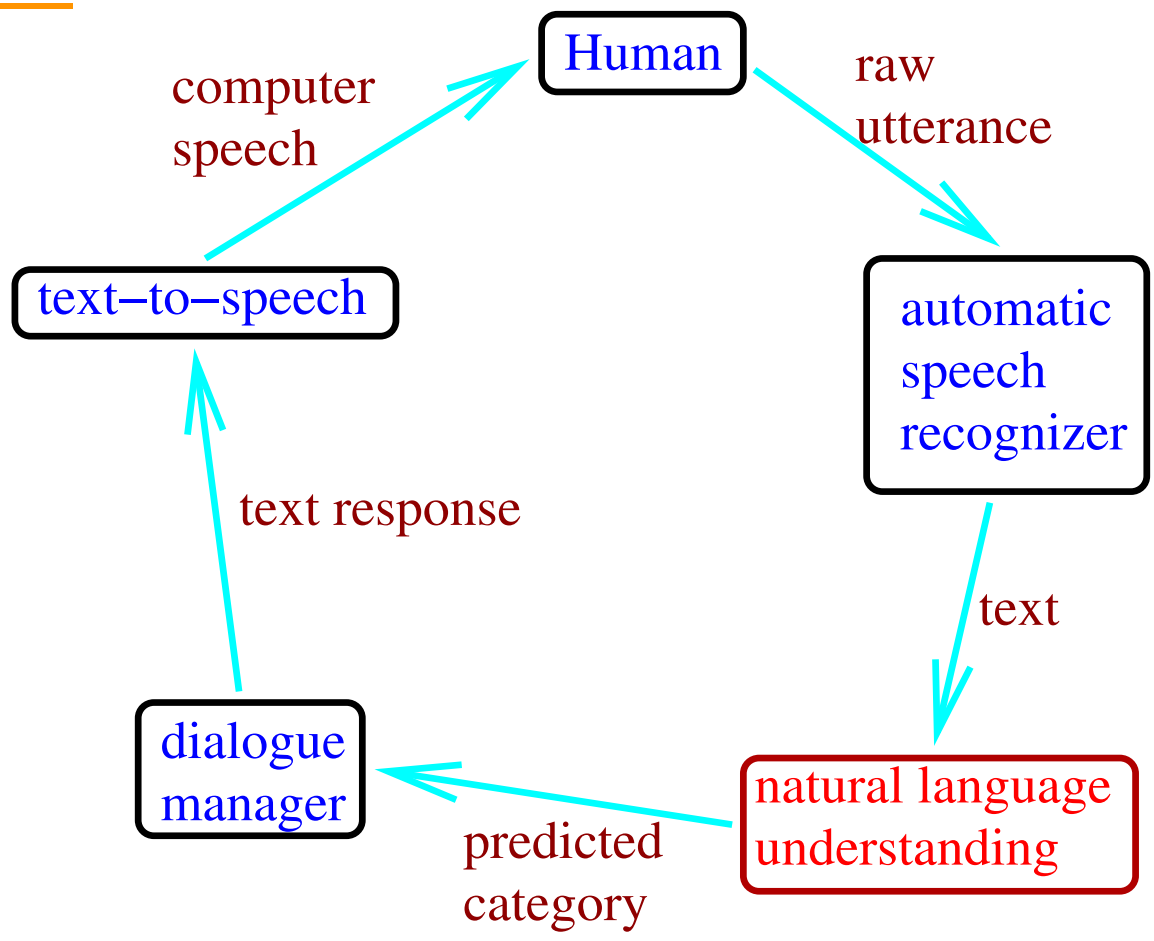
- key idea:
  - training error only measures whether classifications are right or wrong
  - should also consider confidence of classifications
- recall: H_{final} is weighted majority vote of weak classifiers
- measure confidence by margin = strength of the vote
- empirical evidence and mathematical 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: Human-computer Spoken Dialogue

[with Rahim, Di Fabbrizio, Dutton, Gupta, Hollister & Riccardi]

- application: automatic "store front" or "help desk" for AT&T Labs' Natural Voices business
- caller can request demo, pricing information, technical support, sales agent, etc.
- interactive dialogue

How It Works

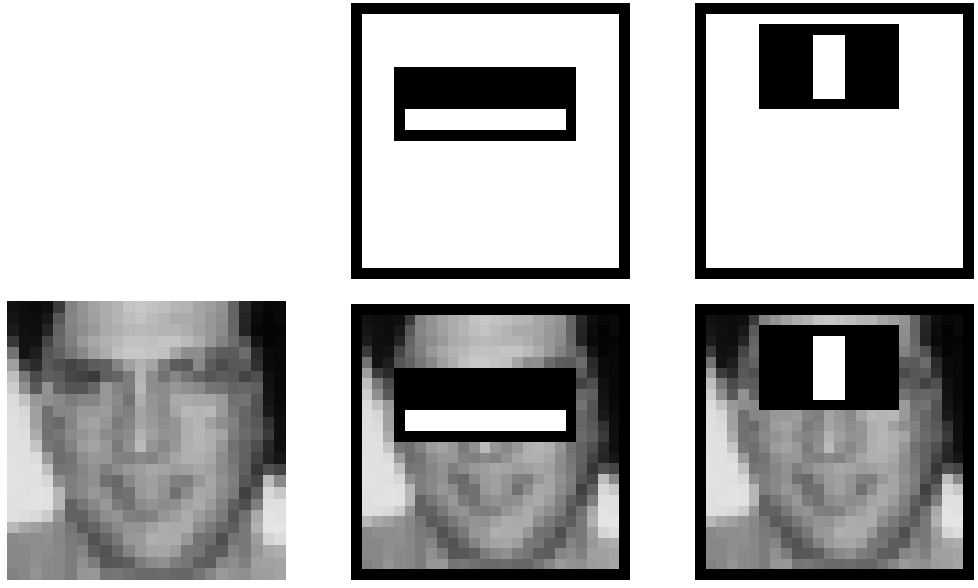


- NLU's job: classify caller utterances into 24 categories (demo, sales rep, pricing info, yes, no, etc.)
- weak classifiers: test for presence of word or phrase

Application: Detecting Faces

[Viola & Jones]

- problem: find faces in photograph or movie
- weak classifiers: detect light/dark rectangles in image



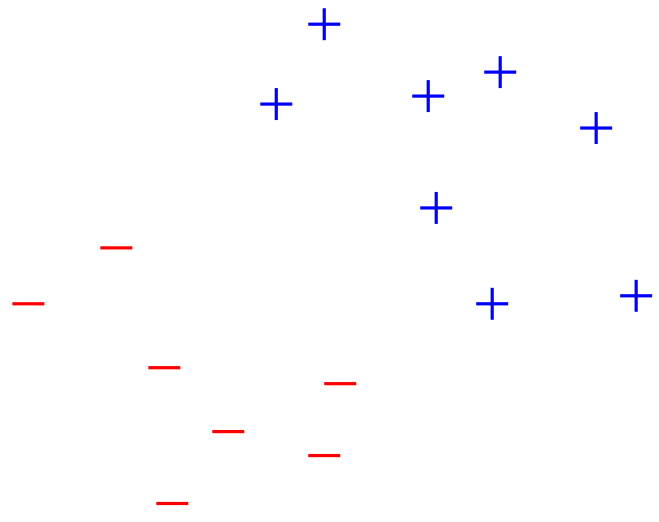
- many clever tricks to make extremely fast and accurate

Boosting

- fast (but not quite as fast as other methods)
- simple and easy to program
- flexible: can combine with any 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

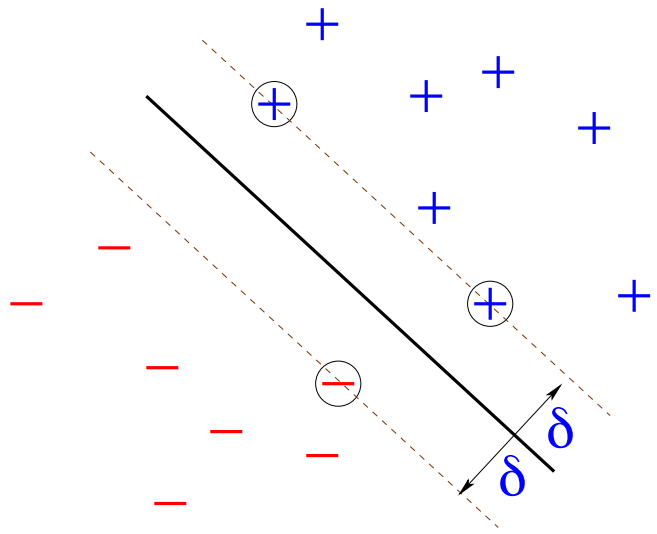
Support-Vector Machines

Geometry of SVM's



- given linearly separable data

Geometry of SVM's



- given linearly separable data
- margin = 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 $\delta =$ minimum margin
  $R =$ radius of enclosing sphere
- then

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

  - so larger margins $\Rightarrow$ lower "complexity"
  - independent 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 $\mathbf{x}_i \in \mathbb{R}^n$, $y_i \in \{-1, +1\}$
- find hyperplane $\mathbf{v} \cdot \mathbf{x} = 0$ with $\|\mathbf{v}\| = 1$

Finding the Maximum Margin Hyperplane

- examples $\mathbf{x}_i, y_i$ where $\mathbf{x}_i \in \mathbb{R}^n$, $y_i \in \{-1, +1\}$
- find hyperplane $\mathbf{v} \cdot \mathbf{x} = 0$ with $\| \mathbf{v} \| = 1$
- margin $= y(\mathbf{v} \cdot \mathbf{x})$
- maximize: δ
  subject to: $y_i(\mathbf{v} \cdot \mathbf{x}_i) \geq \delta$ and $\| \mathbf{v} \| = 1$

Finding the Maximum Margin Hyperplane

- examples $\mathbf{x}_i, y_i$ where $\mathbf{x}_i \in \mathbb{R}^n$, $y_i \in \{-1, +1\}$
- find hyperplane $\mathbf{v} \cdot \mathbf{x} = 0$ with $\| \mathbf{v} \| = 1$
- margin $= y(\mathbf{v} \cdot \mathbf{x})$
- maximize: δ
  subject to: $y_i(\mathbf{v} \cdot \mathbf{x}_i) \geq \delta$ and $\| \mathbf{v} \| = 1$
- set $\mathbf{w} = \mathbf{v}/\delta \quad \Rightarrow \quad \| \mathbf{w} \| = 1/\delta$

Finding the Maximum Margin Hyperplane

- examples $\mathbf{x}_i, y_i$ where $\mathbf{x}_i \in \mathbb{R}^n$, $y_i \in \{-1, +1\}$
- find hyperplane $\mathbf{v} \cdot \mathbf{x} = 0$ with $\| \mathbf{v} \| = 1$
- margin $= y(\mathbf{v} \cdot \mathbf{x})$
- maximize: δ
  subject to: $y_i(\mathbf{v} \cdot \mathbf{x}_i) \geq \delta$ and $\| \mathbf{v} \| = 1$
- set $\mathbf{w} = \mathbf{v}/\delta \quad \Rightarrow \quad \| \mathbf{w} \| = 1/\delta$
- minimize: $\frac{1}{2} \| \mathbf{w} \|^2$
  subject to: $y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 1$

Convex Dual

- form Lagrangian, set $\partial/\partial \mathbf{w} = 0$
- $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$
- get quadratic program:
- maximize $\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$

  subject to: $\alpha_i \geq 0$
- $\alpha_i =$ 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

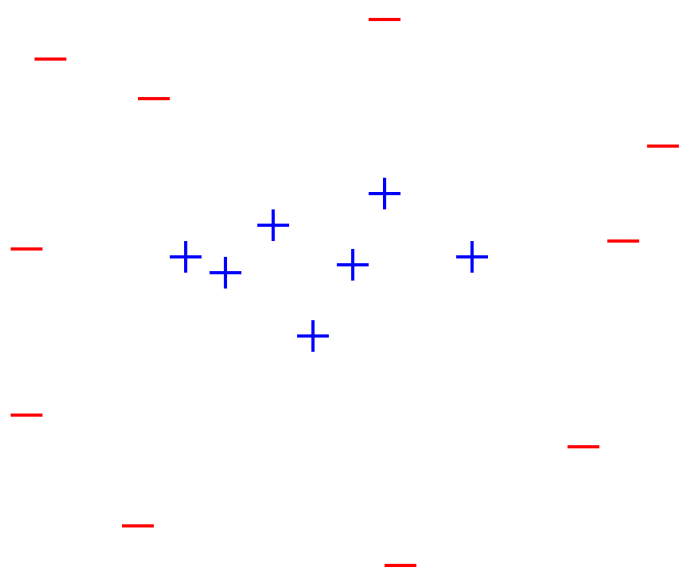
What If Not Linearly Separable?

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

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

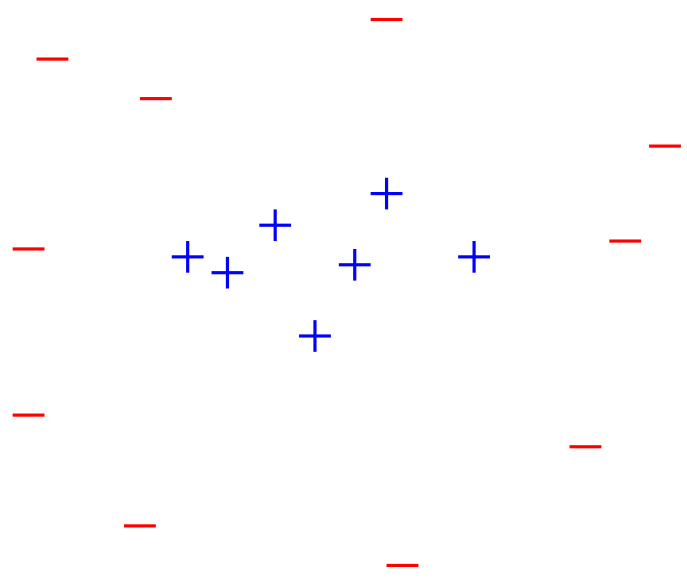
- answer #2: map into higher dimensional space in which data becomes linearly separable

Example



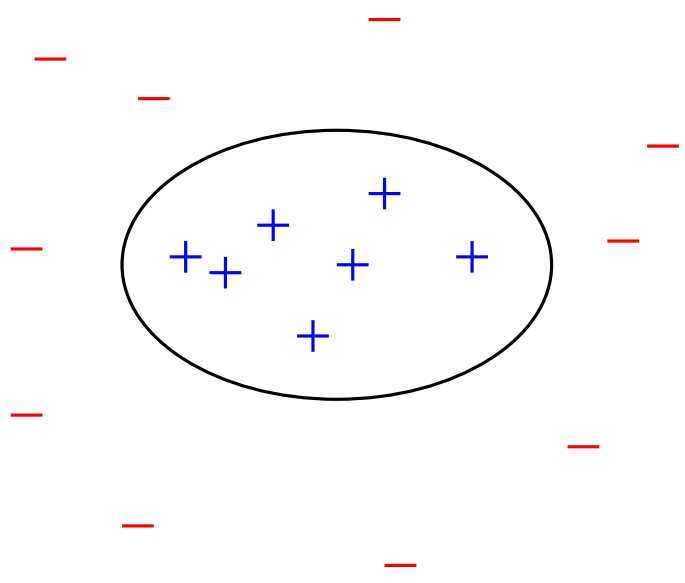
- not linearly separable

Example



- not linearly separable
- map $\mathbf{x} = (x_1, x_2) \mapsto \Phi(\mathbf{x}) = (1, x_1, x_2, x_1 x_2, x_1^2, x_2^2)$

Example



- not 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

Why Mapping to High Dimensions Is Dumb

- can carry idea further
  - e.g., add all terms up to degree d
  - then n dimensions mapped to $O(n^d)$ dimensions
  - huge blow-up in dimensionality

Why Mapping to High Dimensions Is Dumb

- can carry idea further
  - e.g., add all terms up to degree d
  - then n dimensions mapped to $O(n^d)$ dimensions
  - huge blow-up in dimensionality
- statistical problem: amount of data needed often proportional to number of dimensions ("curse of dimensionality")
- computational problem: very expensive in time and memory to work in high dimensions

How SVM's Avoid Both Problems

- statistically, may not hurt since VC-dimension independent of number of dimensions ($(R/\delta)^2$)
- computationally, only need to be able to compute inner products

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

  - sometimes can do very efficiently using kernels

Example (continued)

- modify Φ slightly:

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

Example (continued)

- modify Φ slightly:

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

Example (continued)

- modify Φ slightly:

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

- then

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

- simply use in place of usual inner product

Example (continued)

- modify Φ slightly:

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

- then

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

- simply use in place of usual inner product
- 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 efficient 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:

$$\mathbf{w} \cdot \Phi(\mathbf{x}) = \sum \alpha_i y_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}) = \sum \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})$$

  - 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

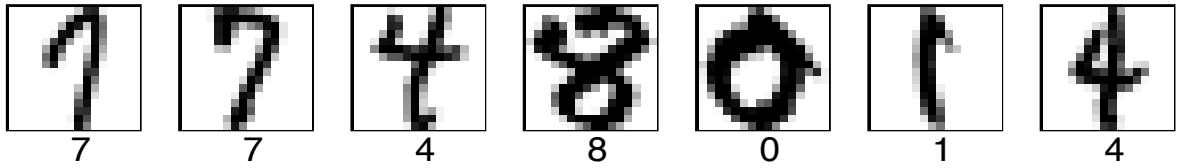
[Joachims]

- 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

Application: Recognizing Handwritten Characters

[Cortes & Vapnik]

- examples are 16×16 pixel images, viewed as vectors in $\mathbb{R}^{256}$



- kernels help:

degree	error	dimensions
1	12.0	256
2	4.7	≈ 33000
3	4.4	$\approx 10^6$
4	4.3	$\approx 10^9$
5	4.3	$\approx 10^{12}$
6	4.2	$\approx 10^{14}$
7	4.3	$\approx 10^{16}$
human	2.5	

- to choose best degree:
  - train SVM for each degree
  - choose one with minimum VC-dimension $\approx (R/\delta)^2$

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

Other Machine Learning Problem Areas

- supervised learning
  - classification
  - regression – predict real-valued labels
  - rare class / cost-sensitive learning
- unsupervised – no labels
  - clustering
  - density estimation
- semi-supervised
  - in practice, unlabeled examples much cheaper than labeled examples
  - how to take advantage of both labeled and unlabeled examples
  - active learning – how to carefully select which unlabeled examples to have labeled
- on-line learning – getting one example at a time

Practicalities

Getting Data

- more is more
- want training data to be like test data
- use your knowledge of problem to know where to get training data, and what to expect test data to be like

Choosing Features

- use your knowledge to know what features would be helpful for learning
- redundancy in features is okay, and often helpful
  - most modern algorithms do not require independent features
- too many features?
  - could use feature selection methods
  - usually preferable to use algorithm designed to handle large feature sets

Choosing an Algorithm

- first step: identify appropriate learning paradigm
  - classification? regression?
  - labeled, unlabeled or a mix?
  - class proportions heavily skewed?
  - goal to predict probabilities? rank instances?
  - is interpretability of the results important? (keep in mind, no guarantees)

Choosing an Algorithm

- first step: identify appropriate learning paradigm
  - classification? regression?
  - labeled, unlabeled or a mix?
  - class proportions heavily skewed?
  - goal to predict probabilities? rank instances?
  - is interpretability of the results important? (keep in mind, no guarantees)
- in general, no learning algorithm dominates all others on all problems
  - SVM's and boosting decision trees (as well as other tree ensemble methods) seem to be best off-the-shelf algorithms
  - even so, for some problems, difference in performance among these can be large, and sometimes, much simpler methods do better

Choosing an Algorithm (cont.)

- sometimes, one particular algorithm seems to naturally fit problem, but often, best approach is to try many algorithms
  - use knowledge of problem and algorithms to guide decisions
    - e.g., in choice of weak learner, kernel, etc.
  - usually, don't know what will work until you try
  - be sure to try simple stuff!
  - some packages (e.g. weka) make easy to try many algorithms, though implementations are not always optimal

Testing Performance

- does it work? which algorithm is best?
- train on part of available data, and test on rest
  - if dataset large (say, in 1000's), can simply set aside $\approx$ 1000 random examples as test
  - otherwise, use 10-fold cross validation
    - break dataset randomly into 10 parts
    - in turn, use each block as a test set, training on other 9 blocks

Testing Performance

- does it work? which algorithm is best?
- train on part of available data, and test on rest
  - if dataset large (say, in 1000's), can simply set aside $\approx$ 1000 random examples as test
  - otherwise, use 10-fold cross validation
    - break dataset randomly into 10 parts
    - in turn, use each block as a test set, training on other 9 blocks
- repeat many times
- use same train/test splits for each algorithm
- might be natural split (e.g., train on data from 2004-06, test on data from 2007)
  - however, can confound results — bad performance because of algorithm, or change of distribution?

Selecting Parameters

- sometimes, theory can guide setting of parameters, possibly based on statistics measurable on training set
- other times, need to use trial and test, as before
- danger: trying too many combinations can lead to overfitting test set
  - break data into train, validation and test sets
    - set parameters using validation set
    - measure performance on test set for selected parameter settings
  - or do cross-validation within cross-validation
- trying many parameter settings is also very computationally expensive

Running Experiments

- automate everything!
  - write one script that does everything at the push of a single button
  - fewer errors
  - easy to re-run (for instance, if computer crashes in middle of experiment)
  - have explicit, scientific record in script of exact experiments that were executed

Running Experiments

- automate everything!
  - write one script that does everything at the push of a single button
  - fewer errors
  - easy to re-run (for instance, if computer crashes in middle of experiment)
  - have explicit, scientific record in script of exact experiments that were executed
- if running many experiments:
  - put result of each experiment in a separate file
  - use script to scan for next experiment to run based on which files have or have not already been created
  - makes very easy to re-start if computer crashes
  - easy to run many experiments in parallel if have multiple processors/computers
  - also need script to automatically gather and compile results

If Writing Your Own Code

- R and matlab are great for easy coding, but for speed, may need C or java
- debugging machine learning algorithms is very tricky!
  - hard to tell if working, since don't know what to expect
  - run on small cases where can figure out answer by hand
  - test each module/subroutine separately
  - compare to other implementations
    (written by others, or written in different language)
  - compare to theory or published results

Summary

- central issues in machine learning:
  - avoidance of overfitting
  - balance between simplicity and fit to data
- machine learning algorithms:
  - decision trees
  - boosting
  - SVM's
  - many not covered
- looked at practicalities of using machine learning methods (will see more in lab)

Further reading on machine learning in general:

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

Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

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.

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.
www-ee.technion.ac.il/∼rmeir/Publications/MeiRae03.pdf

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

Support-vector machines:

Christopher J. C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.
research.microsoft.com/∼cburges/papers/SVMTutorial.pdf

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