Principles of Machine Learning
Computational Learning Theory

Vasant Honavar
Artificial Intelligence Research Laboratory
College of Information Sciences and Technology
Bioinformatics and Genomics Graduate Program
The Huck Institutes of the Life Sciences
Pennsylvania State University

vhonavar@ist.psu.edu
http://vhonavar.ist.psu.edu
http://faculty.ist.psu.edu/vhonavar

Computational Learning Theory – What is it good for?

- To make explicit relevant aspects of the learner and the environment
- To identify easy and hard learning problems (and the precise conditions under which they are easy or hard)
- To guide the design of learning systems
- To shed light on natural learning systems
- To help analyze the performance of learning systems

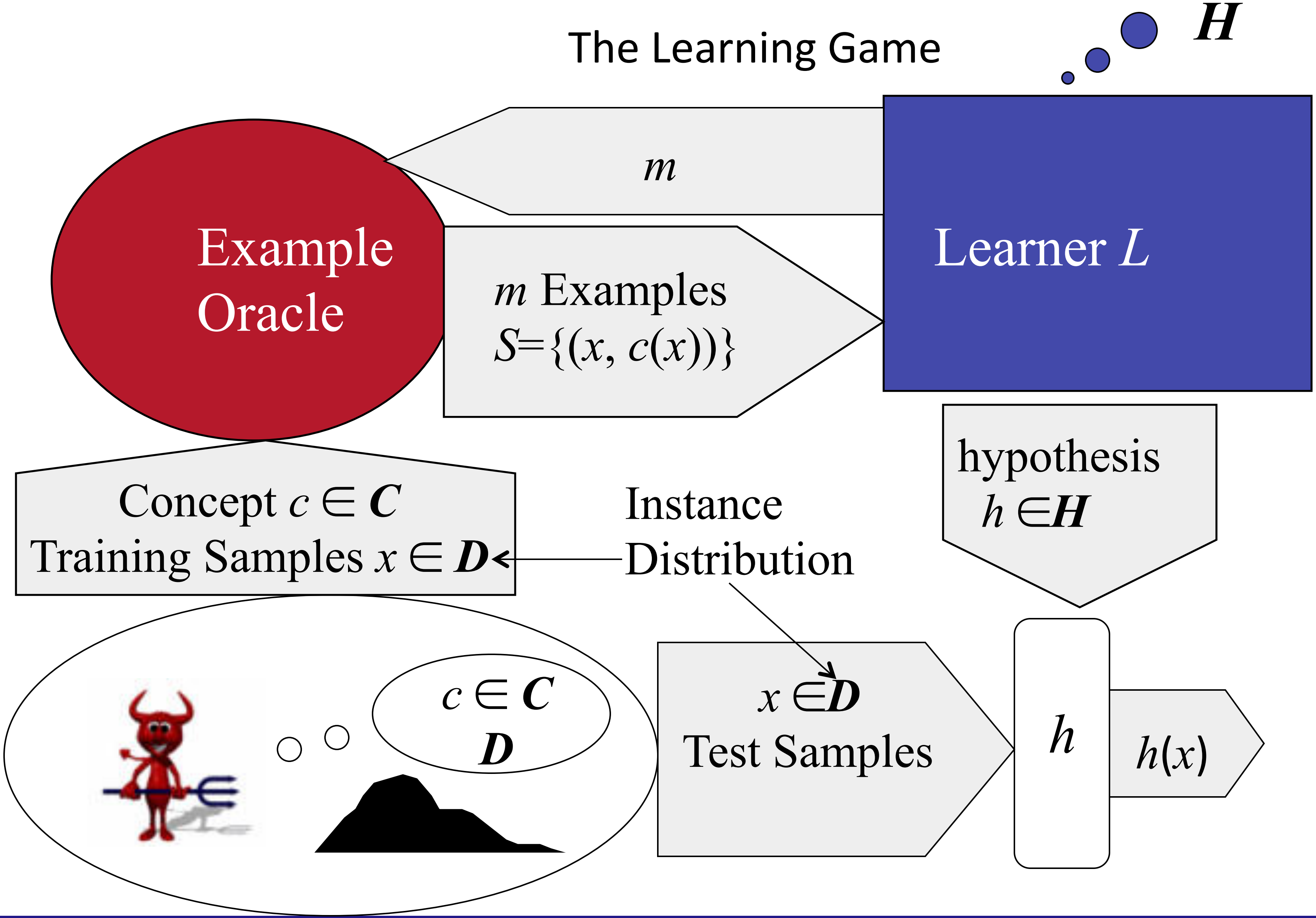
Computational Learning Theory

- Model of the Learner: Computational capabilities, sensors, effectors, knowledge representation, inference mechanisms, prior knowledge, etc.
- Model of the Environment: Tasks to be learned, information sources (teacher, queries, experiments), performance measures
- Key questions: Can a learner with a certain structure learn a specified task in a particular environment? Can the learner do so efficiently? If so, how? If not, why not?

Probably Approximately Correct (PAC) Learning

- Distribution-free models of learning
- Probably Approximately Correct (PAC) Learning
- Sample Complexity Analysis of Concept Classes
- Efficient PAC Learners – polynomial sample learning, polynomial time learning
- Vapnik-Chervonenkis (VC) dimension and Sample Complexity
- Occam's razor
- Learning under simple distributions
- Brief tour of other key results

The Learning Game



$\boldsymbol{H}$

m

Example Oracle

m Examples
$S=\{(x, c(x))\}$

Learner L

Concept $c \in \boldsymbol{C}$
Training Samples $x \in \boldsymbol{D}$

Instance Distribution

hypothesis
$h \in \boldsymbol{H}$

$c \in \boldsymbol{C}$
$\boldsymbol{D}$

$x \in \boldsymbol{D}$
Test Samples

h

$h(x)$

The Learning Game

- We assume
- An instance space $\boldsymbol{X}$
- A concept space $C = \{\, c : X \rightarrow \{0,1\} \,\}$
- A hypothesis space $H = \{\, h : X \rightarrow \{0,1\}\}$
- An unknown, arbitrary, not necessarily computable, stationary probability distribution $\boldsymbol{D}$ over the instance space $\boldsymbol{X}$

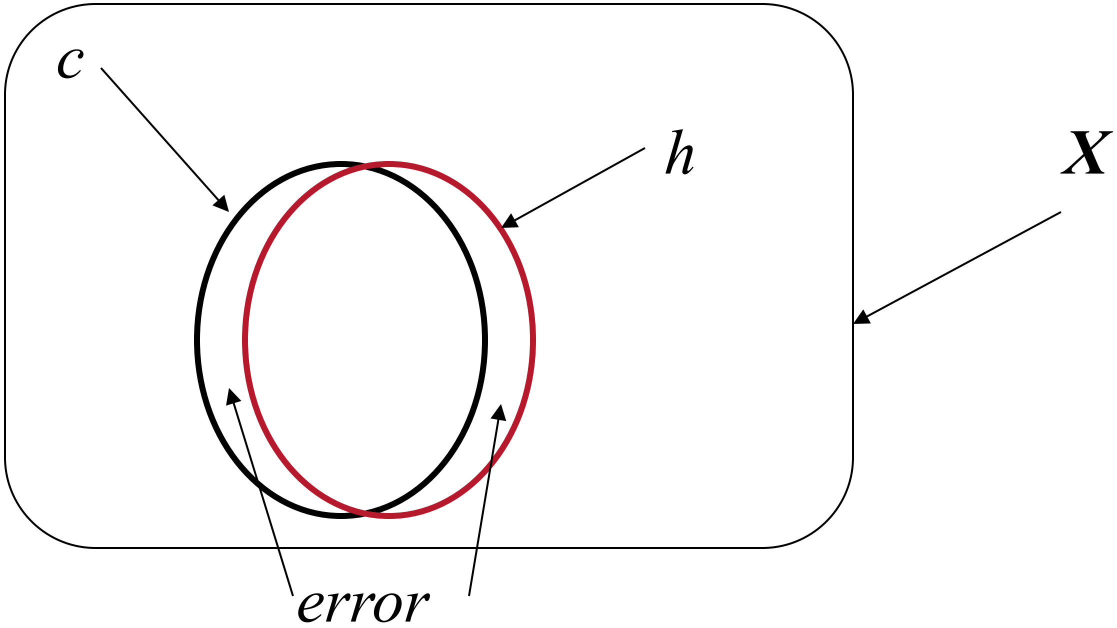
Rules of the Game

- An adversary selects a distribution $\boldsymbol{D}$ over a given instance space X and a target concept c from a given concept class $\boldsymbol{C}$
- An oracle samples the instance space according to $\boldsymbol{D}$ and provides a set $\boldsymbol{S}$ of labeled examples of an unknown concept c to the learner
- The learner's task is to output a hypothesis h from $\boldsymbol{H}$ that closely *approximates* the unknown concept c based on the examples it has encountered
- The learner is tested on samples drawn from the instance space according to the same probability distribution $\boldsymbol{D}$

Measuring the error of a hypothesis

- The error of a hypothesis *h* with respect to a concept *c* and distribution ***D***

$$error_{c,D}(h) = \Pr_{x \in D}\left(c(x) \neq h(x)\right)$$



Probably Approximately Correct Learning – Why?

Impossibility of learning with 0% error

- Because instances are sampled according to an unknown, arbitrary probability distribution ***D*** over the instance space, there is no way to be certain that the learner will see all the necessary examples to exactly learn an unknown concept – exact learning is impossible!

Impossibility of approximate learning with 100% confidence

- Approximate learning (with a specified error ε) cannot be guaranteed hundred percent of the time because of the vagaries of the sampling process

ε-approximation of a concept c

- We say that a hypothesis h is an ε-approximation of a concept c, with respect to an instance distribution $\boldsymbol{D}$ if and only if the probability that h and c disagree on an instance from the instance space drawn at random according to the distribution D is less than ε. That is,

$$error_{c,D}(h) < \varepsilon$$

PAC Learning – A preliminary definition

- A concept class $\boldsymbol{C}$ is said to be PAC-learnable using a hypothesis class $\boldsymbol{H}$ if there exists a learning algorithm L such that for all concepts $c \in \boldsymbol{C}$, for all distributions $\boldsymbol{D}$ on an instance space X, $\forall \varepsilon, \delta \ (0 < \varepsilon, \delta < 1)$, L, when allowed access to the *Example* oracle (that is, a finite set S of labeled examples of a target concept c), outputs with probability at least $(1-\delta)$, a hypothesis $h \in \boldsymbol{H}$ which is an ε-approximation of c. That is,

$$\forall D \text{ over } X, \forall c \in C, \forall \varepsilon, \delta : 0 < \varepsilon < 1, 0 < \delta < 1,$$
$$\Pr_{S \subset D}\left(error_{c,D}(h) < \in\right) \geq (1-\delta)$$

Such a learning algorithm L is called a PAC learning algorithm for the concept class C

Notes on the definition of PAC Learnability

- The definition of PAC learnability of a specified concept class *C* requires that there be a learning algorithm *L* that produces an ε-approximation of any concept in the concept class *C*, any instance distribution, and any choice of the error (ε) and confidence (δ) parameters.

- Specifying a learning algorithm requires the choice of an instance representation, the choice of a hypothesis (concept) representation, and an algorithm for determining the membership of an instance in a hypothesis (concept). More on this later.

How can we show that a concept class is PAC Learnable?

- In order to prove the PAC learnability of a concept class we have to demonstrate the existence of a learning algorithm which meets the necessary criteria specified in the definition of PAC learnability.
- It is even better if we can offer a constructive proof – that is, provide an algorithm that meets the PAC criteria.
- It turns out that we can often get away with using a rather dumb learning algorithm – one that simply outputs a hypothesis that is consistent with the training examples. (We assume that *H* is expressive enough to guarantee the existence of a consistent hypothesis).

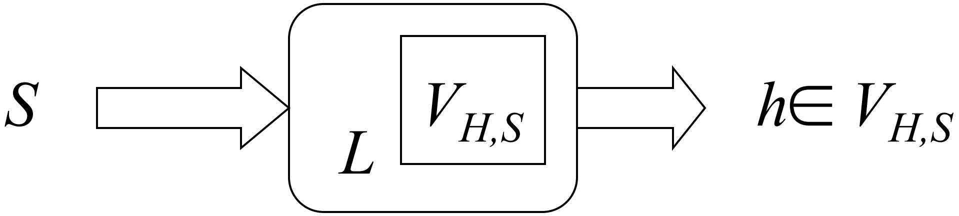
PAC Learnability of finite concept classes

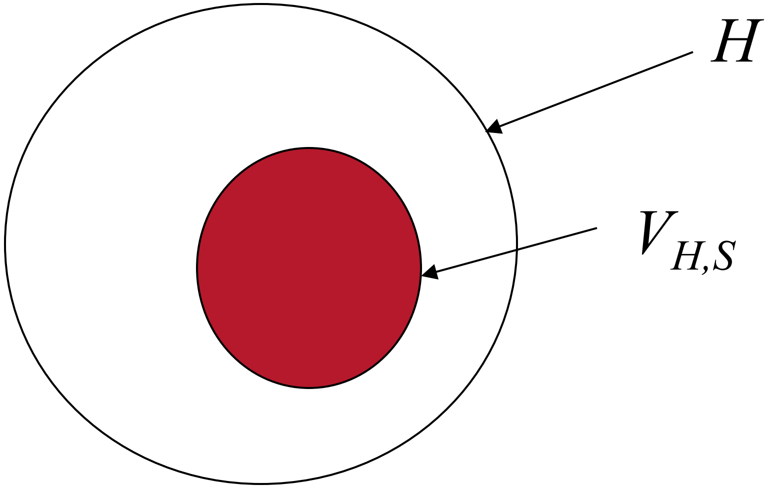
- Definition: A *consistent learner* is one that returns some hypothesis $h \in H$ that is consistent with a training set S of cardinality m.
- Theorem: A consistent learner L is a PAC learner. That is, given a sufficiently large number (m) of examples of c, the hypothesis produced by L is guaranteed, with probability at least 1-δ, to be an ε-approximation of c – for any choice of $c \in C$, any instance distribution D, and any choice of ε, δ such that 0< ε, δ < 1. Specifically, it suffices if

$$m > \frac{1}{\varepsilon} \ln \frac{|H|}{\delta}$$

A consistent learner

$$V_{H,S} = \left\{ h \in H \mid h \text{ is } consistent \text{ with examples in S} \right\}$$





Proof that a consistent learner is a PAC learner

Proof sketch

- There are two kinds of hypothesis in H, and hence in the version space $V_{H,S}$
  - good (ε-approximations of the target concept)
  - bad (not ε-approximations of the target concept).
- Given a sufficiently large number of examples of a target concept c, a sufficiently large fraction of the bad hypotheses get eliminated from the version space maintained by a consistent learner.
- Consequently, a randomly selected hypothesis from $V_{H,S}$ has a high probability (at least 1-δ) of being an ε-approximation of the target concept

A consistent learner is a PAC learner

Definition: A version space $V_{H,S}$ is said to be ε-exhausted with respect to an instance distribution *D* and a concept *c* if every hypothesis $h \in V_{H,S}$ is an ε-approximation of *c*. That is,

$$\forall h \in V_{H,S} \ error_{c,D}(h) < \in$$

Our goal is to make the training set *S* large enough to ensure that the probability that the version space is not ε-exhausted with respect to *c* and *D* is sufficiently small (less than δ) regardless of the choice of $c \in C$ and instance distribution *D* by an adversary.

A consistent learner is a PAC learner

Theorem: Suppose H is a finite hypothesis space, and S a set of m ($m \geq 1$) examples of some $c \in C$. Then for any ε (0< ε ≤ 1), the probability that the version space $V_{H,S}$ is not ε-exhausted with respect to an instance distribution D and a concept c is at most

$$|H|e^{-\varepsilon m}$$

Proof:

- Let H_{Bad} be the subset of hypothesis in $V_{H,S}$ that are not ε-approximations of c.

$$\forall h \in H_{Bad}, error_{c,D}(h) \geq \varepsilon$$

A consistent learner is a PAC learner

- The probability that a hypothesis $h \in H_{Bad}$ agrees with c on a random instance drawn according to D is at most $(1-\varepsilon)$
- The probability that a hypothesis $h \in H_{Bad}$ is consistent m independently drawn random examples is at most $(1-\varepsilon)^m$
- The probability that <u>some</u> hypothesis in $V_{H,S}$ survives m independently drawn random examples is at most

$$|H_{Bad}|(1-\varepsilon)^m \leq |H|(1-\varepsilon)^m \text{ since } H_{Bad} \subseteq H$$

- PAC learning requires that the probability of *L* returning a *bad* hypothesis is *small.* That is, $|H|(1-\varepsilon)^m < \delta$

A consistent learner is a PAC learner

PAC learning requires that the probability of *L* returning a *bad* hypothesis is small (at most δ). That is,

$$|H|(1-\varepsilon)^m \leq \delta$$

$$(0 < \varepsilon \leq 1) \Rightarrow \left\{ (1-\varepsilon) \leq e^{-\varepsilon} \right\}$$

$$\left\{ \begin{array}{l} |H| e^{-\varepsilon m} \leq \delta \\ \Rightarrow \left(\dfrac{|H|}{\delta} \right) \leq e^{\varepsilon m} \\ \Rightarrow e^{\varepsilon m} \geq \left(\dfrac{|H|}{\delta} \right) \\ \Rightarrow \varepsilon m \geq \ln\left(\dfrac{|H|}{\delta} \right) \\ \Rightarrow m \geq \left(\dfrac{1}{\varepsilon} \right)\left(\ln|H| + \ln\left(\dfrac{1}{\delta} \right) \right) \end{array} \right.$$

Hence, to ensure that a consistent learner is a PAC learner, it suffices that

Sample complexity of PAC Learning for finite hypothesis classes

The smallest integer m that satisfies the inequality

$$m > \frac{1}{\varepsilon} \ln \frac{|H|}{\delta}$$

is called the sample complexity of $\boldsymbol{H}$.

PAC- Easy and PAC-Hard Concept Classes for Consistent Learners

Conjunctive concepts are *easy* to learn: How?

Algorithm A.1

- Initialize $L=\{X_1, \sim X_1, ... X_N \sim X_N\}$
- Predict according to match between an instance and the conjunction of literals in L
- Whenever a mistake is made on a positive example, drop the offending literals from L

Example

N=4

Initialize $L = \{\sim X_1, X_1, \sim X_2, X_2, \sim X_3, X_3, \sim X_4, X_4\}$

(0111, 1) will result in $L = \{\sim X_1, X_2, X_3, X_4\}$

(1110, 1) will yield $L = \{X_2, X_3\}$

PAC- Easy and PAC-Hard Concept Classes for Consistent Learners

- Conjunctive concepts are *easy* to learn
- Total number of concepts considered $\left|H\right| = 3^N$
- Sample complexity $O\left(\frac{1}{\varepsilon}\left\{N \ln 3 + \ln \frac{1}{\delta}\right\}\right)$
- Time complexity is polynomial in the relevant parameters of interest
- The class of all Boolean concepts is hard to learn (Why?)
- Remark: Polynomial sample complexity is necessary but not sufficient for efficient (polynomial time) PAC learning – producing a consistent hypothesis may be NP-Hard

Representation

Distinction between a concept and its representation

- A concept is simply a set of instances – extensional definition
- A representation of a concept is a symbolic encoding of that set – intensional definition

Example

- A concept can be represented as a Boolean formula ϕ, or a Boolean formula φ that is logically equivalent to ϕ, or a truth table.

Representation

- Different representations of the same concept may differ radically in *size*
- Example
- Boolean parity function

$$f\left(x_1, x_2 \ldots x_n\right) = x_1 \oplus x_2 \oplus \ldots \oplus x_n$$

where ⊕ denotes the exclusive OR

- can be computed by a circuit of ∧, ∨, and ¬ gates whose size is bounded by a fixed polynomial in *n*
- but a DNF (disjunction of conjunctions) representation of the same function has size that is exponential in *n*.

Representation

- A given target concept has many representations
- The learner is oblivious to which, if any, representation is being used by the teacher or adversary to encode the target concept
- Yet it matters a great deal which of the many representations of hypotheses that the learner chooses – the size of the representation of a hypothesis *h* is a lower bound on the running time of an algorithm that outputs *h*

Representation

- A representation scheme for a concept class ***C*** is a function $R : \Sigma^* \rightarrow C$ where Σ is a finite alphabet of symbols.
- Any string $\sigma \in \Sigma^*$ such that $R(\sigma) = c$ is called a representation of c under *R*
- There may be many representations for a concept *c* under representation *R*
- When we need to use real numbers to represent concepts, we may allow $R : (\Sigma \cup \Re)^* \rightarrow C$

Representation size

$$R : \Sigma^* \rightarrow C$$

$$size : \Sigma^* \rightarrow \aleph$$

assigns a natural number *size*(σ) to each representation σ

The results obtained under a particular definition of *size* are meaningful only if the definition is *natural.*

Example

Σ={0,1}   *size*(σ) is the length of σ in bits

If real numbers are used to encode a concept, we may charge one unit of size to each real number – cannot translate this measure of size into size in bits unless the real numbers are finite precision

Size of a concept c under a representation R

$$size(c) = \min_{R(\sigma)=c} \{size(\sigma)\}$$

Size of a concept $c \in C$ under a representation scheme R for C is the size of the smallest representation of c under R

The larger the value of *size*(c), the more complex the concept c under the chosen representation

From now on, when we speak of learning a concept class C, we will mean learning C under a chosen representation R

Size of instances

- In a Boolean instance space $X_n = \{0,1\}^n$ the size of each instance is *n*
- In $X_n = \Re^n$ the size of each instance may be taken to be *n* (with the usual caveat).
- In $X_n = A^{\leq n}$ where A is a finite alphabet, the size of an instance is the length of the corresponding string (with maximum size being *n*)

Efficient (Polynomial Time) PAC Learning

- Definition: Let $\boldsymbol{C_n}$ be a concept class over X_n.
- Let $X = \cup_{n \geq 1} X_n$ and $C = \cup_{n \geq 1} C_n$
- C is said to be efficiently PAC-learnable if C is PAC-learnable using a learning algorithm L which runs in time that is polynomial in n (size of the instance representation), $size(c)$ (size of the representation of the target concept c), $\left(\frac{1}{\delta}\right)$ and $\left(\frac{1}{\varepsilon}\right)$
- We assume that the learner is given n and *size*(*c*) as input – however, these assumptions can be relaxed

Efficient (Polynomial Time) PAC Learning

- Necessary: Sample complexity must be polynomial in the relevant parameters
- Sufficient: Polynomial sample complexity and a polynomial time consistent learner
- More examples allowed to achieve lower error
- More examples allowed for achieving higher confidence
- More examples allowed for learning more complex concepts
- More examples allowed for learning from bigger instances

Conjunctive Concepts are Efficiently PAC Learnable

- Conjunctive concepts are efficiently PAC-learnable under a natural representation of conjunctions

- Sample complexity

$$O\left(\frac{1}{\varepsilon}\left\{N\ln 3+\ln\left(\frac{1}{\delta}\right)\right\}\right)$$

- Time complexity

$$O\left(\frac{1}{\varepsilon}\left\{N\ln 3+\ln\left(\frac{1}{\delta}\right)\right\}\right)$$

Quick review of computational complexity

P and NP, informally

- P and NP are two classes of problems that can be solved by computers.
- P problems can be solved *quickly.*
  - *Quickly* means seconds or minutes, maybe even hours.
- NP problems can be solved *slowly.*
  - *Slowly* can mean hundreds or thousands of years.

An equivalent question

- Is there a clever way to turn a *slow* algorithm into a *fast* one?
  - If P=NP, the answer is yes.
  - If P≠NP, the answer is no.

Why do we care?

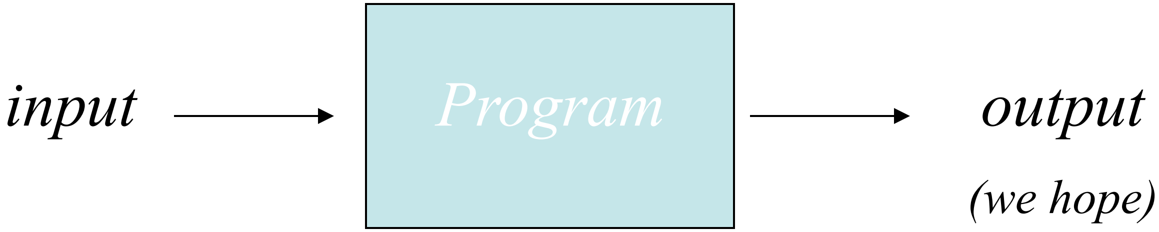
- People like things to work fast.
- Encrypting information
  - If there's an easy way to turn a slow algorithm into a fast one, there's an easy way to crack encrypted information.
  - This is bad for anyone in the business of protecting secrets and for people who like to buy things online.

Currently…

- Most people think P≠NP.

General computing

- First, consider computer programs and what they can do:



A couple of things to note

- There are *lots* of programs for any given problem.
- Some are faster than others.
  - We can always artificially slow them down.

Back to P and NP

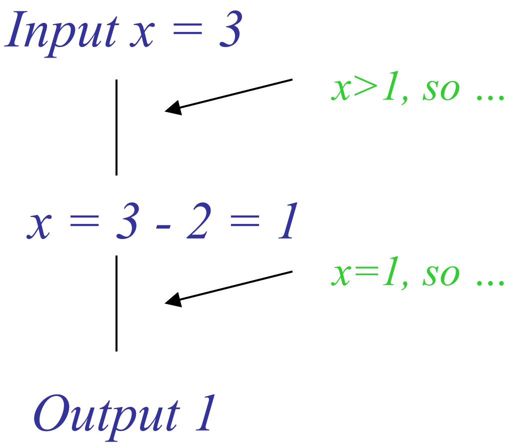
- P and NP are classes of *solvable* problems.
- *Solvable* means that there's a program that takes an input, runs for a while, but eventually stops and gives the answer.

Computation "trees" for solvable problems

Program:

```
     Input x
L1.  If x > 1,
       set x = x-2,
       and GoTo L1.
     If x = 0,
       output 0.
     If x = 1,
       output 1.
```

Example computation:



Input x = 3

x>1, so …

x = 3 - 2 = 1

x=1, so …

Output 1

More about the example…

- What does this program do?
  - Outputs 0 if input is even,
  - Outputs 1 if input is odd.
- Solves the problem "Is the input even or odd?"
- The length of the computation tree depends on the input.
  - *Time(3)= 2*
  - *Time(4)= 3*
  - *Time(x)≤ (x/2) + 1*

Solvability versus Tractability

- A problem is *solvable* if there is a program that always stops and gives the answer.
- The number of steps it takes depends on the input.
- A problem is *tractable* or *in the class P* if it is solvable and we can say *Time(x)≤(some polynomial(x)).*

P problems are “fast”

- These problems are comparatively fast.
- For example, consider a program that has $Time(x) \leq x^2$ compared to one with $Time(x) \leq 2^x$

.

  – On the input of 100, the computation times compare as follows

$$100^2 = 10{,}000 << 2^{100} = 1{,}267{,}650{,}600{,}228{,}229{,}401{,}496{,}703{,}205{,}376$$

Okay… so what about NP?

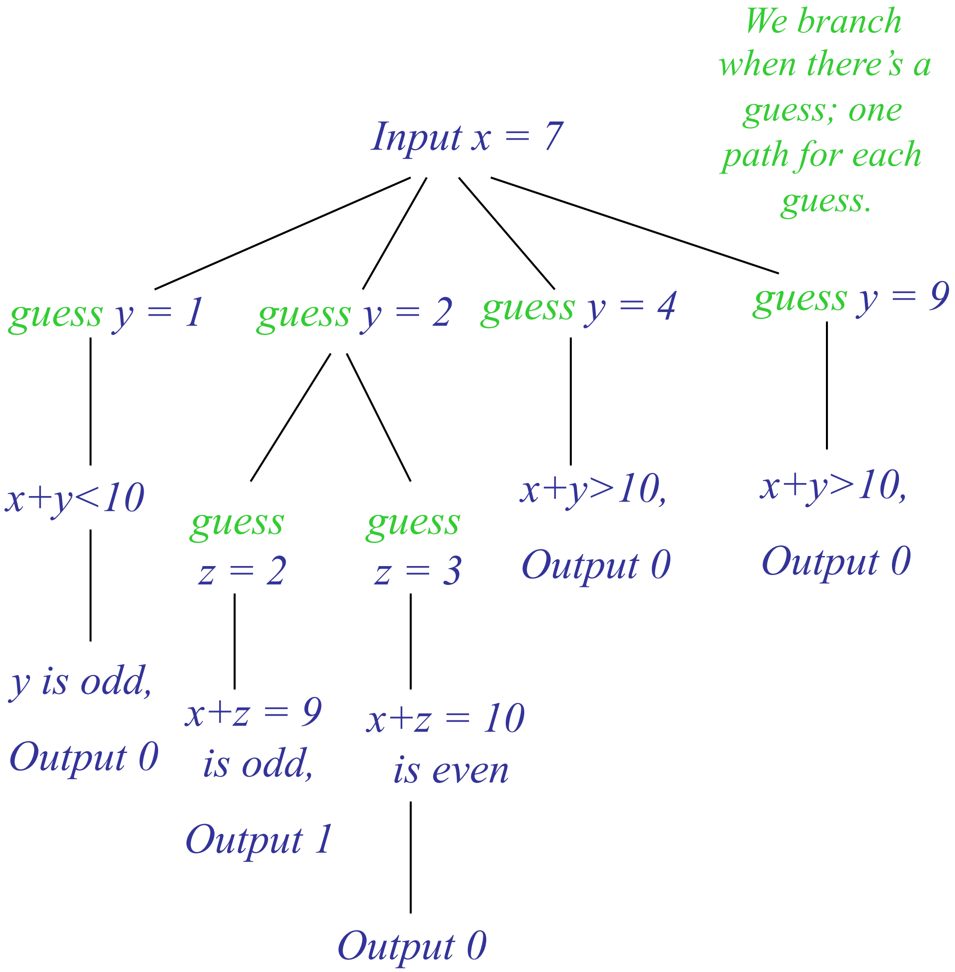
- The description involves *non-deterministic programming*.
  - NP stands for "non-deterministic, polynomial-time computable"
- The examples we' ve seen so far are examples of *deterministic programs.*
  - By the way, P stands for "polynomial-time computable"

An example of non-deterministic programming

- Non-deterministic programs use a new kind of command that normal programs can't really use.
- Basically, they can guess the answer and then check to see if the guess was right.
- And they can guess all possible answers simultaneously (as long as there are only finitely many of them).

An example of non-deterministic programming

```
Program:
Input x.
Guess y in {1, 2, 4, 9}.
If x+y > 10,
     stop and output 0.
Otherwise,
If y is even,
     Guess z in {2, 3},
     If x+z is odd, stop,
     output 1.
Otherwise, output 0.
```



We branch when there's a guess; one path for each guess.

Input x = 7

guess y = 1 — *x+y<10* — *y is odd, Output 0*

guess y = 2 — *guess z = 2* — *x+z = 9 is odd, Output 1*

guess y = 2 — *guess z = 3* — *x+z = 10 is even* — *Output 0*

guess y = 4 — *x+y>10, Output 0*

guess y = 9 — *x+y>10, Output 0*

Non-deterministic programming

- Convention:
  - If <u>any</u> computation path ends with a 1, the answer to the problem is 1 (we count this as "yes").
  - If <u>all</u> computation paths end with a 0, the answer to the problem is 0 (we count this as "no").
  - Otherwise, we say the computation does not converge.
- Again, we're only interested in problems where this third case never happens – solvable problems.

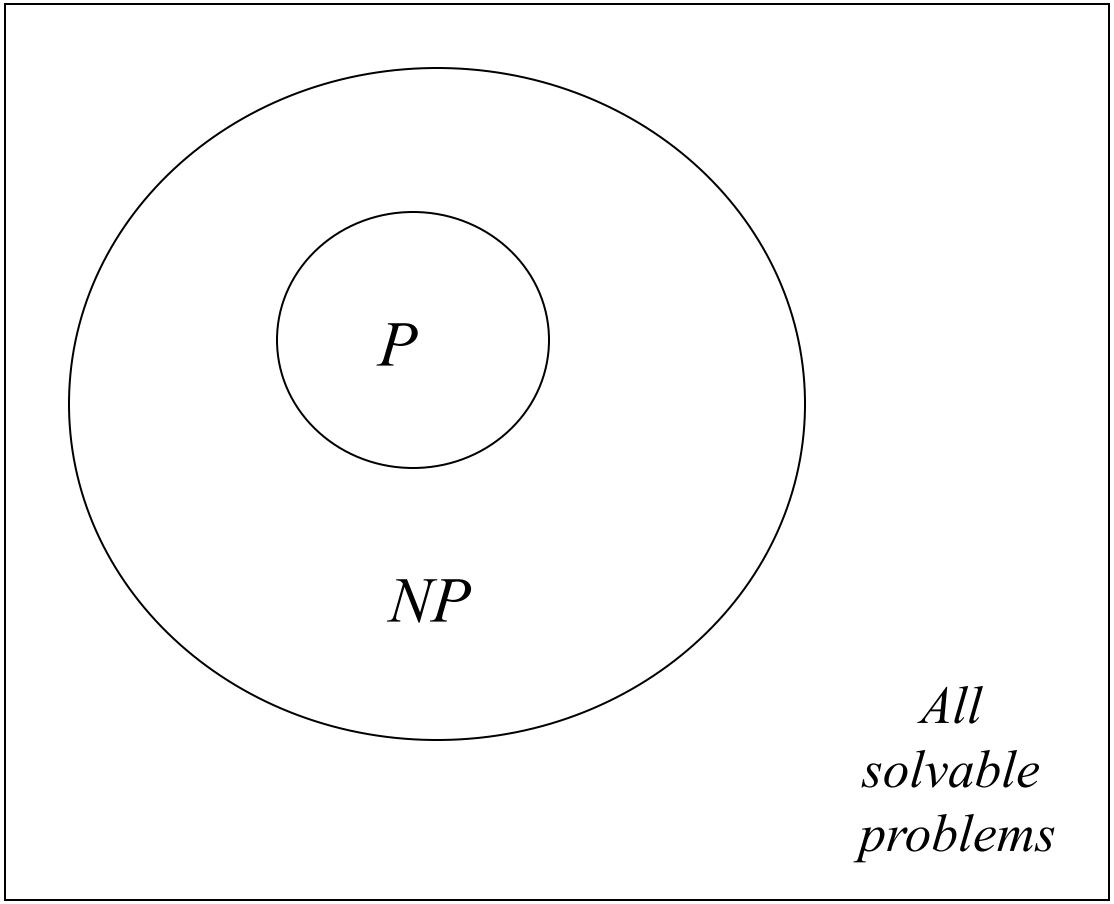
The class NP

- If the computation halts on input *x*, the length of the longest path is *NTime(x).*
- A problem is NP if it is solvable and there is a non-deterministic program that computes it so that *NTime(x)≤(Some polynomia(x)l).*

Non-deterministic → Deterministic

- A non-deterministic algorithm can be converted into a deterministic algorithm at the cost of *time.*
- Usually, the increase in computation time is exponential.
- This means, for normal computers, (deterministic ones), NP problems are *slow*.

The picture so far…



SAT (The problem of satisfiabity)

- Take a statement in propositional logic, (like $p \vee q \rightarrow p \wedge q$ for example).
- The problem is to determine if it is satisfiable. (In other words, is there a line in the truth table for this statement that has a "T" as its truth value.)
- This problem can be solved in polynomial time with a non-deterministic program.

SAT, cont.

- The length of each path in the computation tree is a polynomial function of the length of the input statement.
- SAT is an NP problem.

NP completeness

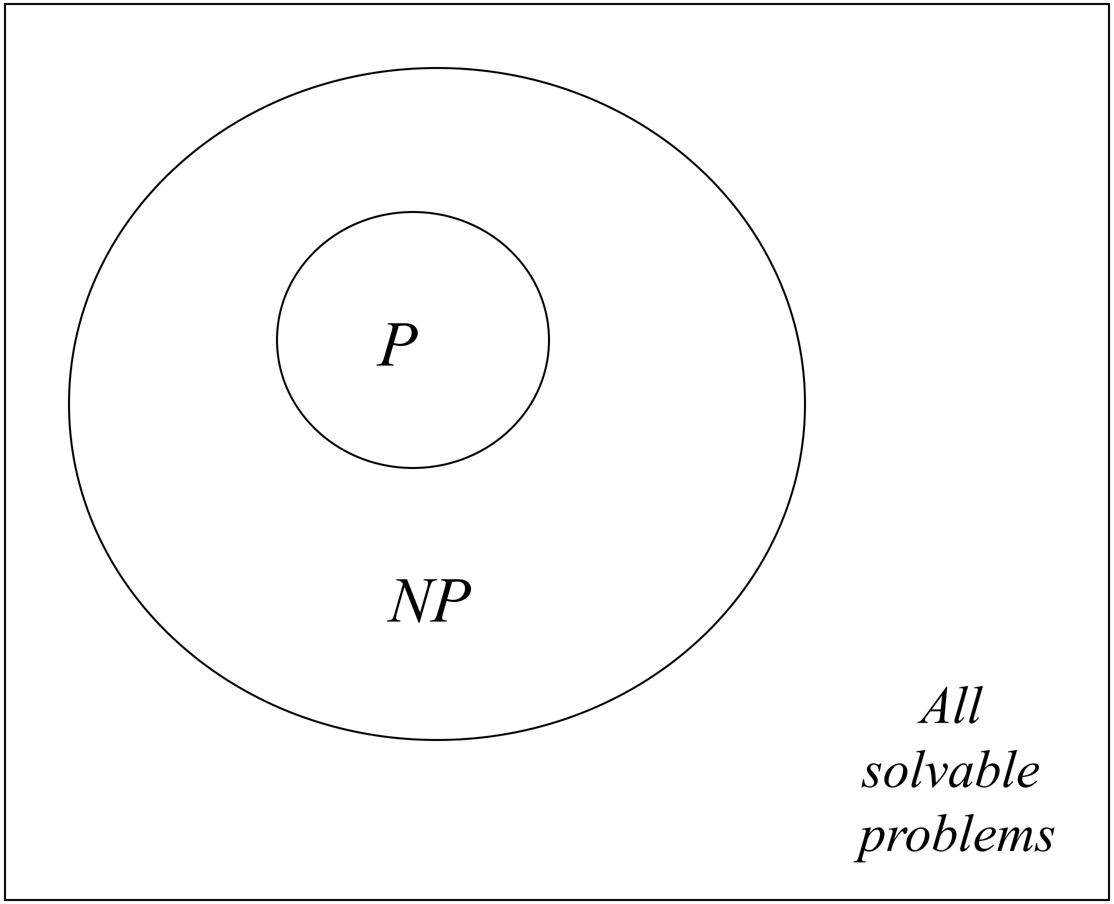
- If P≠NP, SAT is a witness of this fact, that is, SAT is NP but not P.
- It is among the "hardest" of the NP problems: any other NP problem can be coded into it in the following sense.

If R is a non-deterministic, polynomial-time algorithm that solves another NP problem, then for any input, *x*, we can quickly find a formula, *f*, so that *f* is satisfiable when R halts on *x* with output 1, and *f* is not satisfiable when R halts on *x* with output 0.

NP complete problems

- Problems with this property that all NP problems can be coded into them are called *NP-hard.*
- If they are also NP, they are called *NP-complete.*
- If P and NP are different, then the NP-complete problems are NP, but not P.

The picture so far…



Optimization & Decision Problems

- **Decision problems**
  - Given an input and a question regarding a problem, determine if the answer is yes or no
- **Optimization problems**
  - Find a solution with the "best" value
- Optimization problems can be cast as decision problems that are easier to study
  - *E.g.:* Shortest path: G = unweighted directed graph
    - Find a path between u and v that uses the fewest edges
    - *Does a path exist from u to v consisting of at most k edges?*

Algorithmic vs Problem Complexity

- The ***algorithmic complexity*** of a computation is some measure of how *difficult* is to perform the computation (i.e., specific to an algorithm)
- The **complexity of a computational *problem*** or *task* is the complexity of the algorithm with the **lowest** order of growth of complexity for solving that problem or performing that task.
  - *e.g.* the problem of searching an ordered list has *at most lgn* time complexity.
- **Computational Complexity**: deals with classifying problems by how hard they are.

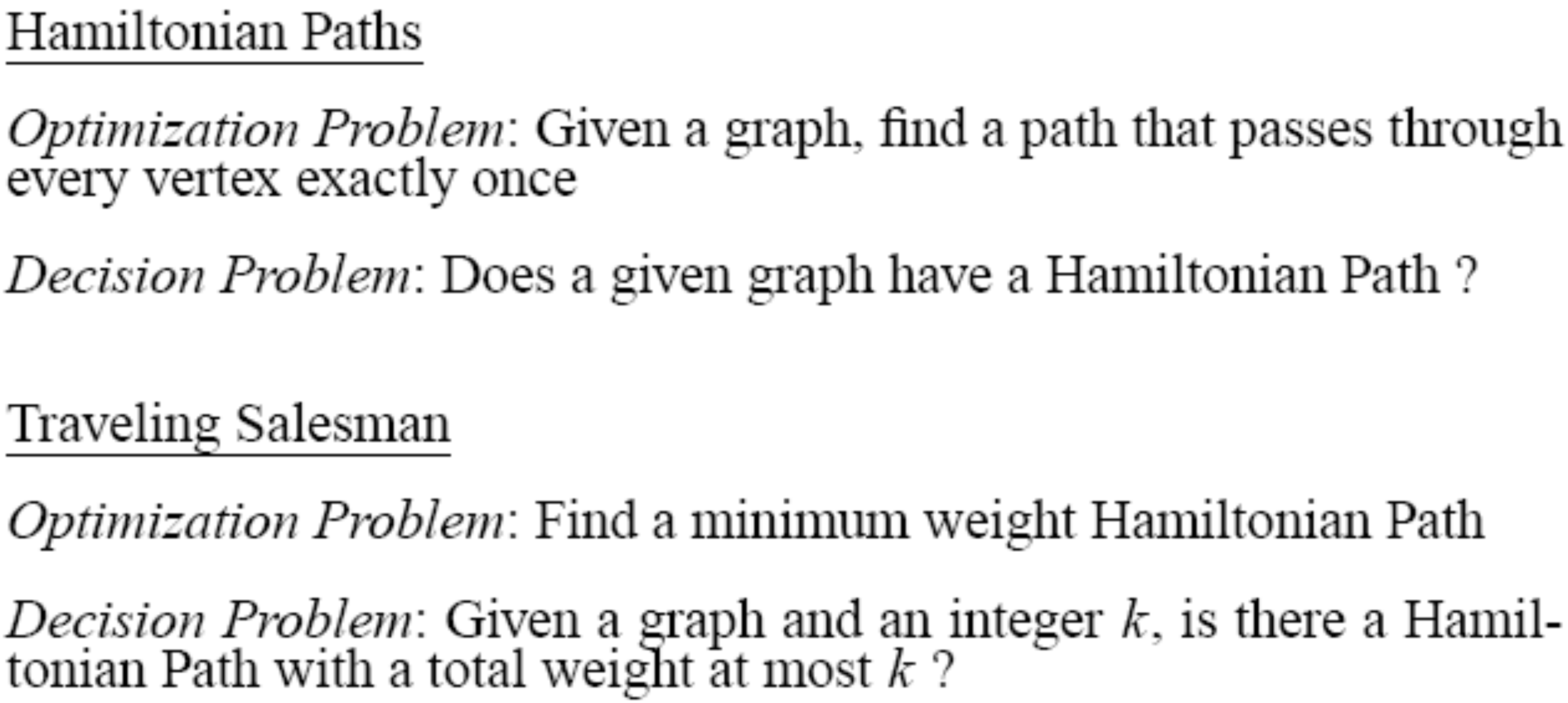
Class of “P” Problems

- **Class P** consists of (decision) problems that are solvable in polynomial time
- Polynomial-time algorithms
  - Worst-case running time is $O(n^k)$, for some constant k
- Examples of polynomial time:
  - $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- Examples of non-polynomial time:
  - $O(2^n)$, $O(n^n)$, $O(n!)$

Tractable/Intractable Problems

- Problems in P are also called **tractable**
- Problems **not** in P are **intractable**
  - Can be solved in reasonable time only for small inputs
  - Or, can not be solved at all

Examples of Intractable Problems



Hamiltonian Paths

Optimization Problem: Given a graph, find a path that passes through every vertex exactly once

Decision Problem: Does a given graph have a Hamiltonian Path ?

Traveling Salesman

Optimization Problem: Find a minimum weight Hamiltonian Path

Decision Problem: Given a graph and an integer k, is there a Hamiltonian Path with a total weight at most k ?

Intractable Problems

- Can be classified in various categories based on their degree of difficulty, e.g.,
  - NP
  - NP-complete
  - NP-hard
- Let's define NP algorithms and NP problems …

Nondeterministic and NP Algorithms

Nondeterministic algorithm = two stage procedure:

1) Nondeterministic ("guessing") stage:

   generate randomly an arbitrary string that can be thought of as a candidate solution ("certificate")

2) Deterministic ("verification") stage:

   take the certificate and the instance to the problem and returns YES if the certificate represents a solution

NP algorithms (Nondeterministic polynomial)

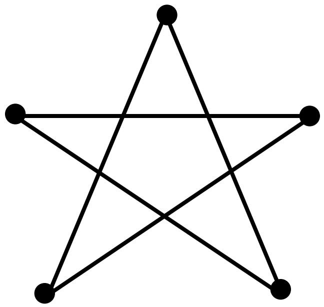
verification stage is polynomial

Class of "NP" Problems

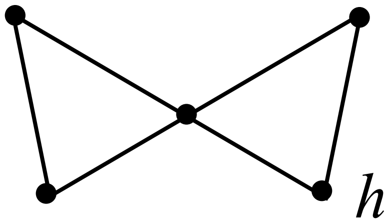
- **Class NP** consists of problems that could be solved by NP algorithms
  - i.e., verifiable in polynomial time
- If we were given a "certificate" of a solution, we could verify that the certificate is correct in time polynomial to the size of the input
- Warning: NP does **not** mean "non-polynomial"

E.g.: Hamiltonian Cycle

- **Given:** a directed graph G = (V, E), determine a simple cycle that contains each vertex in V
  - Each vertex can only be visited once
- **Certificate**:
  - Sequence: $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$

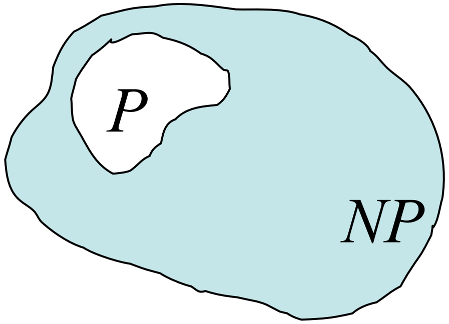


hamiltonian



not hamiltonian

Is P = NP?



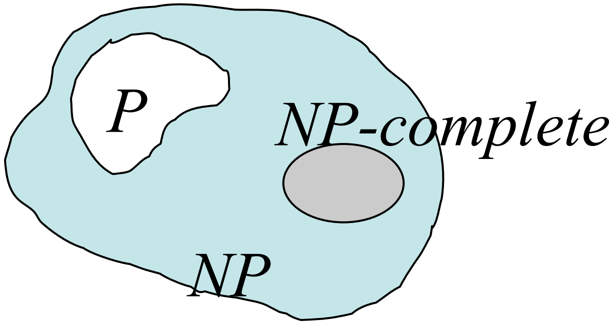
- Any problem in P is also in NP:

  $P \subseteq NP$

- The big (and **open question**) is whether $NP \subseteq P$ or $P = NP$
  - i.e., if it is always easy to check a solution, should it also be easy to find a solution?
- Most computer scientists believe that this is false but we do not have a proof …

NP-Completeness (informally)

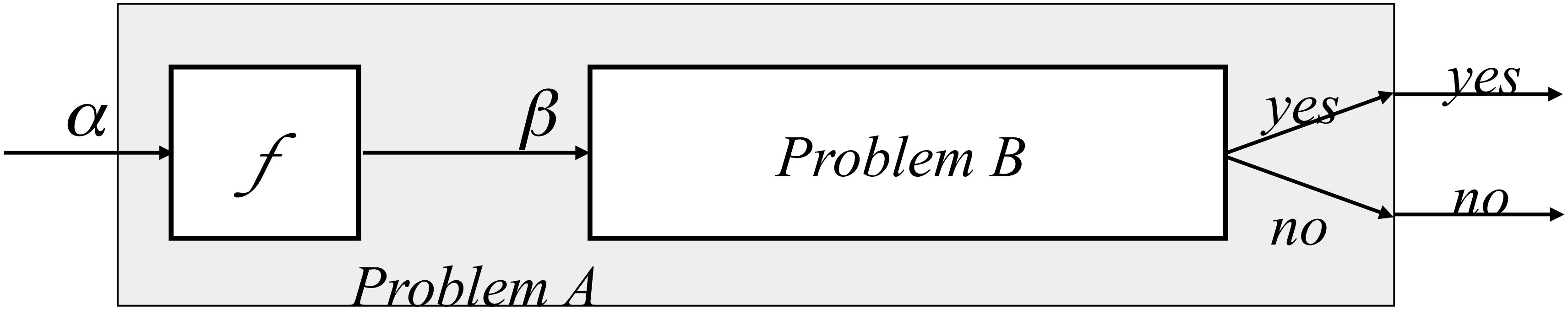
- **NP-complete** problems are defined as the hardest problems in NP



- Most practical problems turn out to be either P or NP-complete.
- Study NP-complete problems …

Reductions

- Reduction is a way of saying that one problem is **"easier"** than another.
- We say that problem A is easier than problem B, (i.e., we write **"A ≤ B"**) if we can solve A using the algorithm that solves B.
- **Idea:** transform the inputs of A to inputs of B

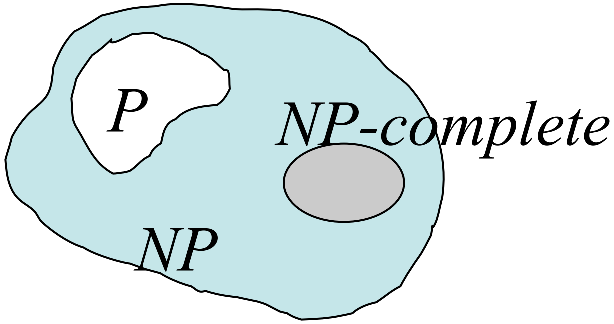


Polynomial Reductions

- Given two problems A, B, we say that A is polynomially **reducible** to B ($A \leq_p B$) if:

1. There exists a function f that converts the input of A to inputs of B in polynomial time
2. $A(i) = YES \Leftrightarrow B(f(i)) = YES$

NP-Completeness (formally)



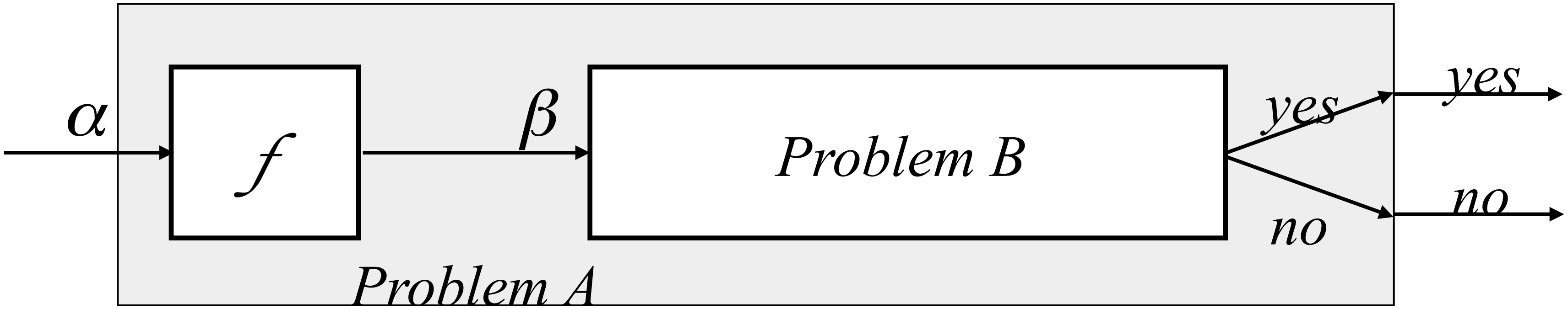
- A problem B is **NP-complete** if:

  (1) B ∈ **NP**

  (2) A $\leq_p$ B for all A ∈ **NP**

- If B satisfies only property (2) we say that B is **NP-hard**
- No polynomial time algorithm has been discovered for an **NP-Complete** problem
- No one has ever proven that no polynomial time algorithm can exist for any **NP-Complete** problem

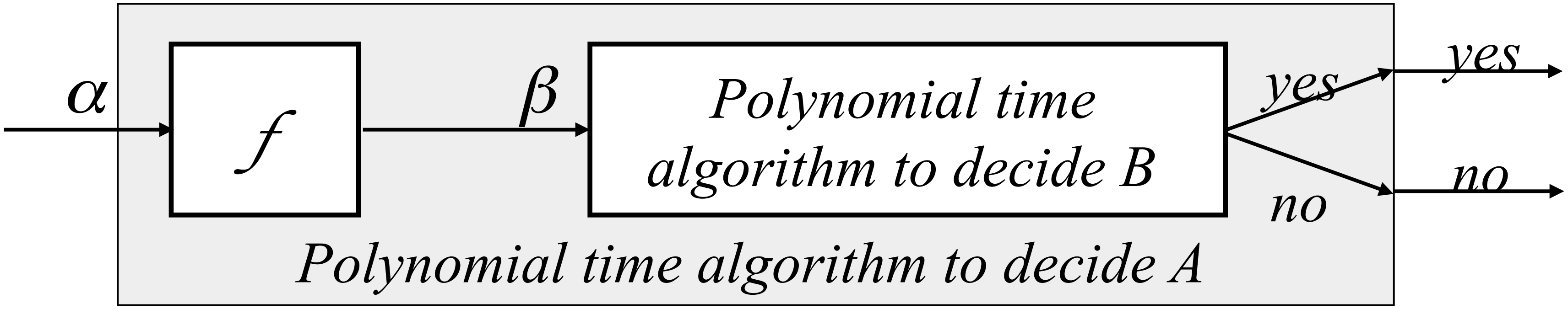
Implications of Reduction



- If A $\leq_p$ B and B $\in$ P, then A $\in$ P

- if A $\leq_p$ B and A $\notin$ P, then B $\notin$ P

Proving Polynomial Time

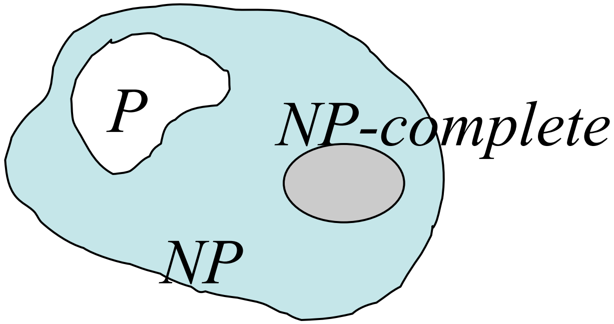

α
f
β
Polynomial time algorithm to decide B
yes
no
yes
no
Polynomial time algorithm to decide A

1. Use a **polynomial time** reduction algorithm to transform A into B
2. Run a known **polynomial time** algorithm for B
3. Use the answer for B as the answer for A

Proving NP-Completeness In Practice

- Prove that the problem B is in NP
  - A randomly generated string can be checked in polynomial time to determine if it represents a solution
- Show that **one known** NP-Complete problem can be transformed to B in polynomial time
  - No need to check that **all** NP-Complete problems are reducible to B

Revisit "Is P = NP?"



Theorem: If any NP-Complete problem can be solved in polynomial time ⇒ then P = NP.

P & NP-Complete Problems

- **Euler tour**
  - G = (V, E) a connected, directed graph find a cycle that traverses each edge of G exactly once (may visit a vertex multiple times)
  - Polynomial solution O(E)
- **Hamiltonian cycle**
  - G = (V, E) a connected, directed graph find a cycle that visits each vertex of G exactly once
  - NP-complete

Satisfiability Problem (SAT)

- **Satisfiability problem:** given a logical expression Φ, find an assignment of values (F, T) to variables x_i that causes Φ to evaluate to T

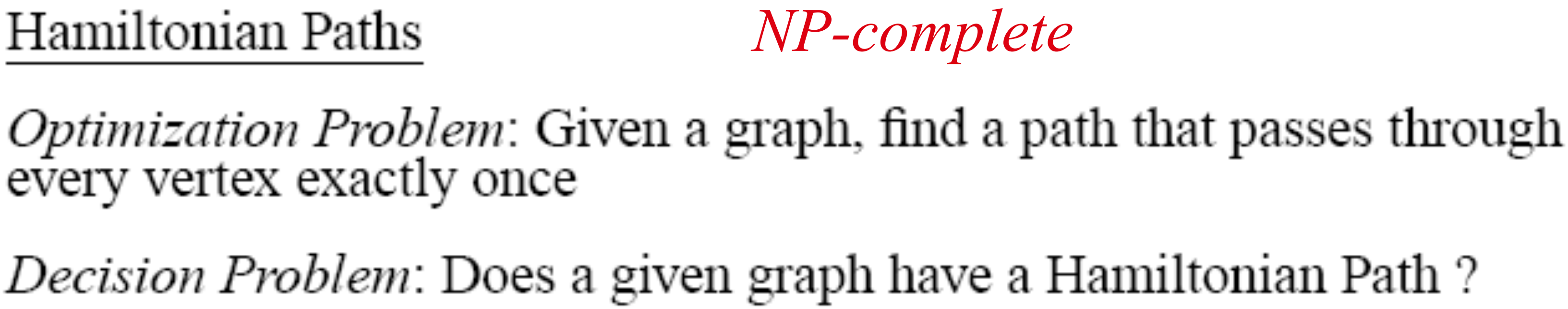
$$\Phi = x_1 \vee \neg x_2 \wedge x_3 \vee \neg x_4$$

- SAT was the first problem shown to be NP-complete!

NP-naming convention

- **NP-complete -** means problems that are 'complete' in NP, i.e. the most difficult to solve in NP
- **NP-hard** - stands for 'at least' as hard as NP (but not necessarily **in** NP);
- **NP-easy** - stands for 'at most' as hard as NP (but not necessarily **in** NP);
- **NP-equivalent** - means equally difficult as NP, (but not necessarily **in** NP);

Examples NP-complete and NP-hard problems

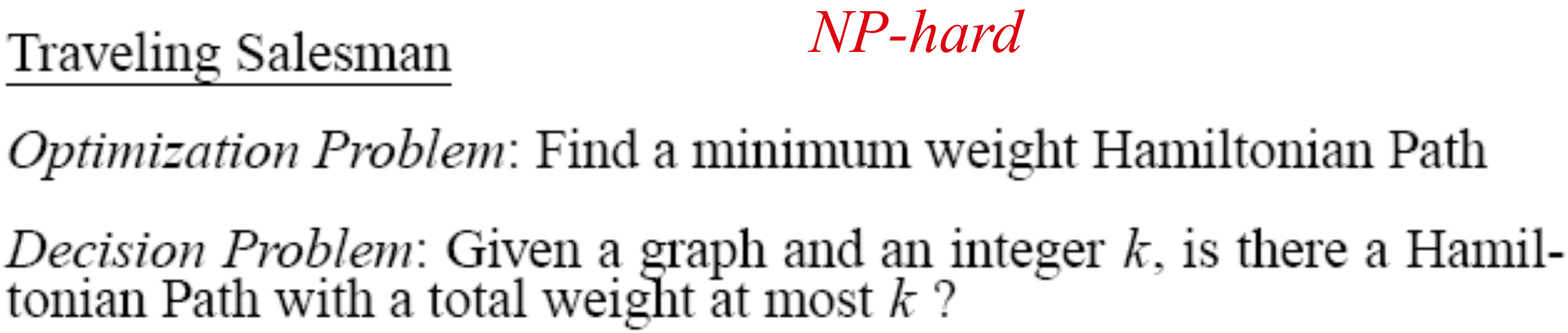


Hamiltonian Paths

NP-complete

Optimization Problem: Given a graph, find a path that passes through every vertex exactly once

Decision Problem: Does a given graph have a Hamiltonian Path ?



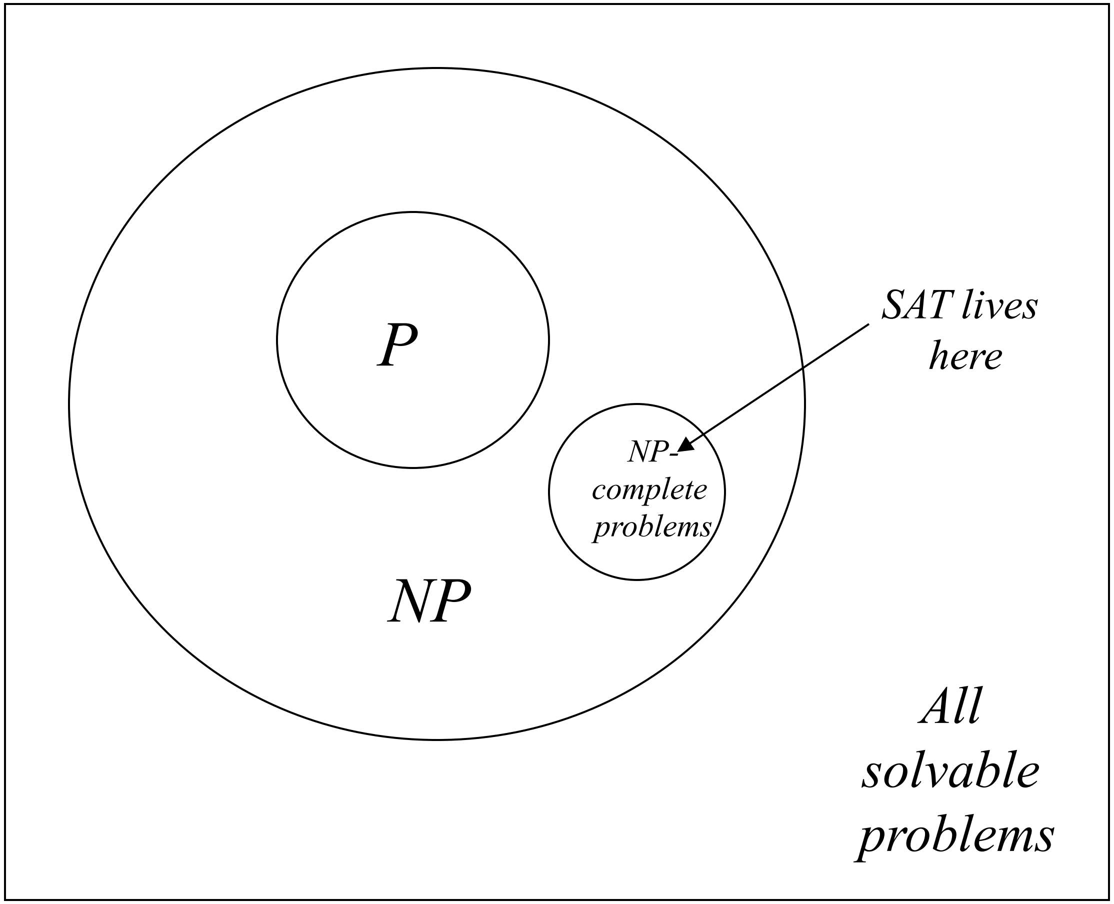
Traveling Salesman

NP-hard

Optimization Problem: Find a minimum weight Hamiltonian Path

Decision Problem: Given a graph and an integer k, is there a Hamiltonian Path with a total weight at most k ?

The picture so far…



Question:

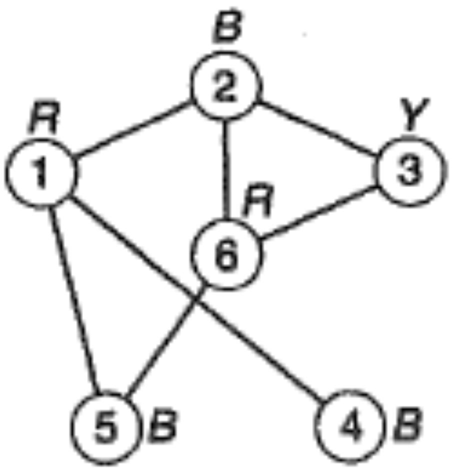
- Is there a clever way to change a non-deterministic polynomial time algorithm into a deterministic polynomial time algorithm, without an exponential increase in computation time?
- If we can solve an NP complete problem quickly then *all* NP problems are solvable in deterministic polynomial time.

Back to COLT

3-Term DNF concepts are not efficiently PAC learnable unless P=RP

- Randomized polynomial time (RP) is the complexity class of problems for which a Turing machine exists such that the
  - It always runs in polynomial time in the input size
  - If the correct answer is NO, it always returns NO
  - If the correct answer is YES, then it returns YES with probability at least ½ .
- Theorem: 3-term DNF concept class (disjunctions of at most 3 conjunctions) are not efficiently PAC-learnable using the *same* hypothesis class unless P=RP.
- Proof: By polynomial time reduction of graph 3-colorability (an NP-complete problem) to the problem of deciding whether a given set of labeled examples is consistent with some 3-term DNF formula.

Reduction



Graph G

S_G^+	S_G^-
<011111, 1>	<001111, 0>
<101111, 1>	<011011, 0>
<110111, 1>	<011101, 0>
<111011, 1>	<100111, 0>
<111101, 1>	<101110, 0>
<111110, 1>	<110110, 0>
	<111100, 0>

$T_R = x_2 \wedge x_3 \wedge x_4 \wedge x_5$

$T_B = x_1 \wedge x_3 \wedge x_6$

$T_Y = x_1 \wedge x_2 \wedge x_4 \wedge x_5 \wedge x_6$

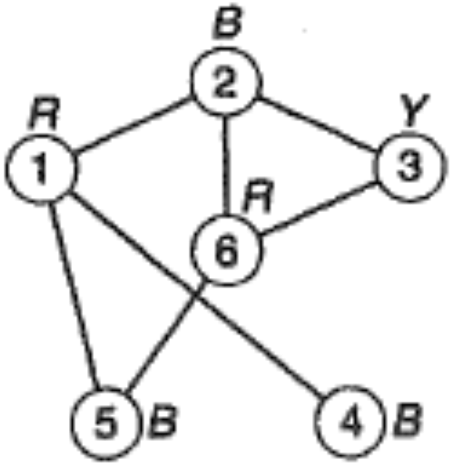
$$S_G = S_{G^+} \bigcup S_{G^-}$$

$$S_{G^+} = \left\{ \left(v_i, 1 \right) \mid v_i \text{ has 0 in the } i\text{th position and 1s everywhere else} \right\}$$

$$S_{G^-} = \left\{ \left(e_{ij}, 0 \right) \mid e_{ij} \text{ has 0 in the } i\text{th and } j\text{th position and 1s everywhere else} \right\}$$

- G is 3-colorable iff S_G is consistent with some 3-term DNF Formula

Reduction



Graph G

S_G^+	S_G^-
<011111, 1>	<001111, 0>
<101111, 1>	<011011, 0>
<110111, 1>	<011101, 0>
<111011, 1>	<100111, 0>
<111101, 1>	<101110, 0>
<111110, 1>	<110110, 0>
	<111100, 0>

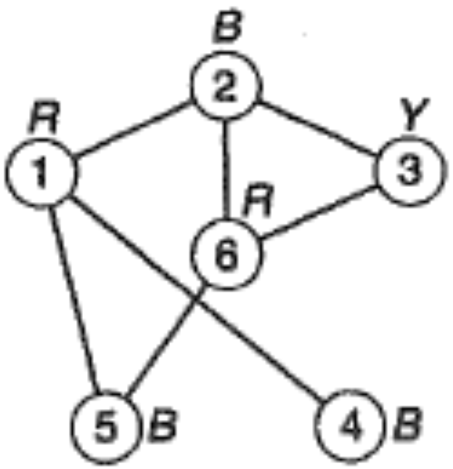
$$T_R = x_2 \wedge x_3 \wedge x_4 \wedge x_5$$

$$T_B = x_1 \wedge x_3 \wedge x_6$$

$$T_Y = x_1 \wedge x_2 \wedge x_4 \wedge x_5 \wedge x_6$$

- Suppose G is 3-colorable; Fix a 3-coloring of G.
- Let R be the set of red vertices. T_R the conjunction of literals whose indices don't appear in R; -> v_i must satisfy T_R
- No e_{ij} can satisfy T_R because no two adjacent vertices can be colored red and at least one of x_i and x_j should be in T_R

Reduction



Graph G

S_G^+	S_G^-
<011111, 1>	<001111, 0>
<101111, 1>	<011011, 0>
<110111, 1>	<011101, 0>
<111011, 1>	<100111, 0>
<111101, 1>	<101110, 0>
<111110, 1>	<110110, 0>
	<111100, 0>

$$T_R = x_2 \wedge x_3 \wedge x_4 \wedge x_5$$
$$T_B = x_1 \wedge x_3 \wedge x_6$$
$$T_Y = x_1 \wedge x_2 \wedge x_4 \wedge x_5 \wedge x_6$$

- Suppose T_R is consistent with the data
- Define coloring as follows: color of v_i is red if v_i satisfies $T_{R\,\ldots}$ similarly for other colors
- This yields a legal 3 coloring

Transforming Hard Problems to Easy ones

- Theorem: 3-term DNF concepts are efficiently PAC-learnable using 3-CNF (conjunction of disjunctions (clauses) with at most 3 literals per clause) hypothesis class.
- Proof: $3\text{-term DNF} \subseteq 3\text{-CNF}$
- Transform each example over N boolean variables into a corresponding example over N^3 variables (one for each possible clause in a 3-CNF formula).

$$T_1 \vee T_2 \vee T_3 = \bigwedge_{u \in T_1, v \in T_2, w \in T_3} (u \vee v \vee w)$$

- The problem reduces to learning a conjunctive concept over the transformed instance space.

Transforming Hard Problems to Easy ones

- Theorem For any $k \geq 2$ *k*-term DNF are efficiently PAC-learnable using the *k*-CNF hypothesis class.
- Remark: In this case, enlarging the search space by using a hypothesis class that is larger than strictly necessary, actually makes the problem easy!
- Remark: No, we have not proved that P=NP.
- Summary:

$$\begin{array}{ccccccc}\text{Conjunctive} & \subseteq & \text{k - term DNF} & \subseteq & \text{k - CNF} & \subseteq & \text{CNF} \\ \text{Easy} & & \text{Hard} & & \text{Easy} & & \text{Hard}\end{array}$$

Occam Learning Algorithm

- Definition: Let $\alpha \geq 0$ & $0 \leq \beta < 1$ be constants. A learning algorithm *L* is said to be an $\alpha - \beta$ Occam algorithm for a concept class *C* using a hypothesis class *H* if *L*, given a set *S* of *m* random examples of an unknown concept $c \in C$ outputs a hypothesis such that *h* is consistent with *S* and

$$h \in H$$

$$size(h) \leq \{Nsize(c)\}^{\alpha} m^{\beta}$$

Effective hypothesis space size $H_{nm} \leq 2^{(Nsize(c))^{\alpha} m^{\beta}}$

Occam Learning Algorithm outputs a succinct hypothesis

$$size(h) \leq \{N size(c)\}^{\alpha} m^{\beta}$$

$$\text{When } m >> N, \quad size(h) = O\left((size(c))^{\alpha} m^{\beta}\right)$$

Thus, *m* labels have to be compressed into $O(m)^{\beta}$ bits
-- a mild requirement because we can always obtain a
Consistent hypothesis that is *O*(*mn*) bits long (why?)

We have to allow *size*(*h*) to depend linearly on *size*(*c*)
in the event the shortest hypothesis in *H* may in fact be the
target concept *c*.

We allow a generous dependence on *m* – which often makes
It easier to find a consistent hypothesis – finding the shortest
hypothesis is often computationally intractable

Occam Learning Algorithm

- An Occam learning algorithm *L* for a concept class *C* is said to be an efficient $\alpha - \beta$ Occam learning algorithm for *C* if its running time is bounded by a polynomial in *n*, *m*, and *size*(*c*).
- The simple algorithm we considered for learning conjunctive concepts is an efficient Occam learning algorithm (Prove this!).

Sample complexity of an Occam Algorithm

- Theorem: An Occam algorithm is guaranteed to be PAC if the number of samples

$$m = O\left(\frac{1}{\varepsilon}\lg\frac{1}{\delta} + \left[\frac{\left(Nsize(c)\right)^{\alpha}}{\varepsilon}\right]^{\frac{1}{1-\beta}}\right)$$

- Proof: Left as an exercise.

k-decision lists

- *k* decision list over Boolean variables $x_1 \ldots x_N$ is an ordered sequence

$$l = \left((c_1, b_1) \ldots (c_l, b_l), b\right)$$

Where each c_i is a conjunction of at most k literals chosen from $x_1 \ldots x_N$ (and their negations) and each b_i and b is 0 or 1.

On a given *N*-bit input, *l* is evaluated like a nested if-then-else statement with *b* corresponding to the default output.

Occam algorithm is PAC for K-decision lists

- Theorem: For any fixed *k*, the concept class of *k*-decision lists is efficiently PAC-learnable using the same hypothesis class.
- Algorithm – Greedily find conjunctions of at most *k* literals that *cover* the largest subset of examples with the same class label.
- Remark: *k*-decision lists constitute the most expressive Boolean concept class over the Boolean instance space $\{0,1\}^N$ that are known to be efficiently PAC learnable.

Mistake and Loss Bound Models of Learning

- Outline
- Machine learning and theories of learning
- Mistake bound model of learning
- Mistake bound analysis of conjunctive concept learning
- Weighted majority and related multiplicative update algorithms
- Applications

Computational Models of Learning

- Model of the Learner: Computational capabilities, sensors, effectors, knowledge representation, inference mechanisms, prior knowledge, etc.
- Model of the Environment: Tasks to be learned, information sources (teacher, queries, experiments), performance measures
- Key questions: Can a learner with a certain structure learn a specified task in a particular environment? Can the learner do so efficiently? If so, how? If not, why not?

Models of Learning: What are they good for?

- To make explicit relevant aspects of the learner and the environment
- To identify easy and hard learning problems (and the precise conditions under which they are easy or hard)
- To guide the design of learning systems
- To shed light on natural learning systems
- To help analyze the performance of learning systems

Mistake Bound Analysis

Example – Learning Conjunctive Concepts

- Given an arbitrary, noise-free sequence of labeled examples $(X_1,C(X_1)),(X_2,C(X_2))\ldots(X_m,C(X_m))$ of an unknown binary conjunctive concept C over $\{0,1\}^N$, the learner's task is to predict $C(X)$ for a given X.

Theorem: Exact online learning of conjunctive concepts can be accomplished with at most $(N+1)$ prediction mistakes.

Online learning of conjunctive concepts

Algorithm A.1

- Initialize $L=\{X_1, \sim X_1, ... X_N \sim X_N\}$
- Predict according to match between an instance and the conjunction of literals in L
- Whenever a mistake is made on a positive example, drop the offending literals from L

Example

$(0111, 1)$ will result in L = $\{\sim X_1, X_2, X_3, X_4\}$

$(1110, 1)$ will yield $L = \{X_2, X_3\}$

Mistake bound analysis of conjunctive concept learning

Proof Sketch

- No literal in C is ever eliminated from L
- Each mistake eliminates at least one literal from L
- The first mistake eliminates N of the $2N$ literals
- Conjunctive concepts can be learned with at most $(N+1)$ mistakes

Conclusion

- Conjunctive concepts are *easy* to learn in the mistake bound model

Optimal Mistake Bound Learning Algorithms

Definition: An *optimal* mistake bound mbound(***C***) for a concept classs ***C*** is the *lowest possible* mistake bound in the *worst case* (considering *all* concepts in ***C***, and *all* possible sequences D of examples).

$$mbound(C) = \min_{learners\ L} \max_{c^* \in C} \max_{example\ sequences\ D} mistakes(c^*, L, D)$$

where mistakes (c^*,L,D) is the number of mistakes made by L in its attempt to learn c^* based on the sequence of examples provided.

Mistake Bounds and optimal mistake bounds

$$\left.\begin{array}{ll} mbound(\mathbf{C}) \le 2^{N} & \text{(why?)} \\ mbound(\mathbf{C}) \le \left|\mathbf{C}\right| - 1 & \text{(why?)} \end{array}\right\} \text{trivial bounds}$$

$$mbound(\mathbf{C}) \le \log\left(\left|\mathbf{C}\right|\right) \quad \left.\right\} \text{we will prove this}$$

Definition: An optimal learning algorithm for a concept class C (in the mistake bound framework) is one that is guaranteed to *exactly* learn *any* concept in C, using *any* noise-free example sequence, with at most O($mbound(C)$) mistakes.

Version space and Halving algorithm

$V_i = \{c \in \mathbf{C} \mid c \text{ is consistent with the first } i \text{ examples}\}$

$V_0 = \mathbf{C}$

$\xi_0(\mathbf{C}, X) = \{c \in \mathbf{C} : c(X) = 0\}$

$\xi_1(\mathbf{C}, X) = \{c \in \mathbf{C} : c(X) = 1\}$

For $i > 0$, $\quad V_i = \begin{cases} \xi_0(V_{i-1}, X_i) \text{ if } c^*(X_i) = 0 \\ \xi_1(V_{i-1}, X_i) \text{ if } c^*(X_i) = 1 \end{cases}$

Halving Algorithm : On input X_i, Predict 1 if

$|\xi_1(V_{i-1}, X_i)| \geq |\xi_0(V_{i-1}, X_i)|$

and 0 otherwise. Eliminate the concepts (majority or minority) that were wrong

Version space

Definition: The halving algorithm predicts according to the majority of concepts in the current version space and a mistake results in elimination of all the offending concepts from the version space.

$$V_i = \{c \in \mathbf{C} \mid c \text{ is consistent with the first } i \text{ examples}\}$$

$$V_0 = \mathbf{C}$$

$$\xi_0(\mathbf{C}, X) = \{c \in \mathbf{C} : c(X) = 0\}$$

$$\xi_1(\mathbf{C}, X) = \{c \in \mathbf{C} : c(X) = 1\}$$

$$\text{For } i > 0, \quad V_i = \begin{cases} \xi_0(V_{i-1}, X_i) \text{ if } c^*(X_i) = 0 \\ \xi_1(V_{i-1}, X_i) \text{ if } c^*(X_i) = 1 \end{cases}$$

The Halving Algorithm

- Theorem: $mbound(\mathbf{C}) \leq \log(|\mathbf{C}|)$

Proof: The halving algorithm predicts according to majority of concepts in the version space. Hence each mistake eliminates at least half of the candidate hypotheses in the version space.

The halving algorithm can be computationally feasible if there is a way to compactly represent and efficiently manipulate the version space. Otherwise it is not computationally feasible.

The Halving Algorithm

The halving algorithm is not optimal with respect to the number of mistakes. In order to minimize the number of mistakes, the learner has to guess according to the subset of the version space that is expected to yield the fewest mistakes.

The optimal mistake bound algorithm has to predict 1 if

$$mbound\left(\xi_1\left(V_{i-1}, X_i\right)\right) \geq mbound\left(\xi_0\left(V_{i-1}, X_i\right)\right)$$

and 0 otherwise. However this algorithm is even less efficient.

The Halving Algorithm

- Question: Are there any efficiently implementable learning algorithms with mistake bounds comparable to that of the halving algorithm?
- Answer: Littlestone's algorithm for learning *monotone disjunctions* of at most k of n literals using the hypothesis class of threshold functions with at most $(k \lg n)$ mistakes. More on this later.

Randomized Halving Algorithm

- The predictions made by the halving algorithm may not be based on any concept in C. There may not exist in C a concept that is consistent with the majority vote.
- The randomized halving algorithm due to Maass predicts according to a randomly selected concept $c \in C$
- All concepts in C that are inconsistent with the example are eliminated from further consideration.
- Theorem: The expected number of mistakes made by the randomized halving algorithm is at most $\log |C| + O(1)$

Randomized Halving Algorithm

- WLOG assume that the order of presentation of the examples is independent of the learner's actions
- Suppose the concepts in the version space are ordered by when they are going to be eliminated by examples.
- Let $c_1 \ldots. c_r$ be the order with $r=|V_i|$
- Let M_r be the expected number of mistakes
- The algorithm picks one of the r concepts at random with probability equal to $1/r$. If c_r is chosen, there are no further mistakes. One of the other concepts is chosen with probability $(r\text{-}1)/r$ in which case, there will be one mistake (at least) plus the expected number of mistakes for the remaining concepts

Randomized Halving Algorithm

$$M_1 = 0$$

$$M_r = \left(\frac{r-1}{r}\right)\left(1 + \frac{\left(\sum_{i=1}^{r-1} M_i\right)}{(r-1)}\right) = \left(\frac{r-1}{r}\right) + \left(\frac{\sum_{i=1}^{r-1} M_i}{r}\right)$$

$$rM_r = (r-1) + \sum_{i=1}^{r-1} M_i; \qquad (r-1)M_{r-1} = (r-2) + \sum_{i=1}^{r-2} M_i$$

$$r(M_r - M_{r-1}) + M_{r-1} = 1 + M_{r-1}$$

$$M_r = M_{r-1} + \frac{1}{r}$$

$$M_r = \sum_{i=1}^{r}\left(\frac{1}{r}\right) = \ln r + O(1)$$

Learning monotone disjunctions when irrelevant attributes abound

$$\mathbf{C} = \left\{ x_{i_1} \vee x_{i_2} \vee \ldots\ldots x_{i_k} \quad i_j \in \{1 \ldots N\}; j \in \{1 \ldots k\} \right\}$$

$$\left|\mathbf{C}\right| = \binom{N}{k} + \binom{N}{k-1} + \ldots \binom{N}{0}$$

$$\lg\left|\mathbf{C}\right| = \Theta(k \lg N)$$

This can be shown to be an optimal mistake bound

How can we design an algorithm that achieves this mistake bound?

Winnow Algorithm

- Idea – Use threshold neurons to learn monotone disjunctions
- (Monotone disjunctions are a subset of threshold functions)

$$\text{Initialize } \theta = \left(\frac{N}{2}\right); \ \mathbf{W} = (1.....1)$$

$$\text{Predict } y(\mathbf{X}) = 1 \text{ iff } \mathbf{W.X} > \theta \text{ otherwise predict } y(\mathbf{X}) = 0$$

$$\text{If } c(\mathbf{X}) = 1 \text{ but } y(\mathbf{X}) = 0, \text{ double all } w_i \text{ where } x_i = 1$$

$$\text{If } c(\mathbf{X}) = 0 \text{ but } y(\mathbf{X}) = 1, \text{ zero out all } w_i \text{ where } x_i = 1$$

Theorem – Winnow makes $\mathrm{O}(k \lg N)$ mistakes

Winnow Algorithm

u – number of times weights are doubled

v – number of times weights are zeroed out

$$0 \le \left(\sum_{i=1}^{N} w_i \right) \le \left(N + u\theta - v\theta \right)$$

Each weight doubling adds at most θ to the sum of weights and each zeroed out weight subtracts at least θ from the sum of weights

$$\forall i \quad w_i \le 2\theta$$

No weight that is greater than θ is ever doubled

$$\exists w_i \ \lg w_i \ge \left(\frac{u}{k} \right)$$

Each weight doubling has to affect at least one of the weights. Each weight doubling adds at least 1 to the logarithm of the weight that got doubled

$$\left(\frac{u}{k} \right) \le \left(\lg w_i \right) \le \left(\lg \theta + 1 \right)$$

$$u \le k\left(\lg \theta + 1 \right)$$

$$v \le \left(\frac{N}{\theta} \right) + k\left(\lg \theta + 1 \right)$$

$$\left(u + v \right) \le 2 + 2k \lg N$$

Theorem – Winnow makes $O(k \lg N)$ mistakes

Generalizations of Winnow

- Winnow algorithm and its variants and generalizations can be used to learn concepts from more expressive concept classes by preprocessing the input patterns – e.g. by transforming an n-bit pattern into an O(n^k) bit pattern that encodes all conjunctions of at most k literals (negated or un negated)
- Winnow algorithm and its variants can be made to generalize better by incorporating regularization

Weighted majority learning algorithm (WML)

Motivations

- Robust algorithm for learning monotone disjunctions
- Suppose we have a pool of predictors – features, experts, algorithms
- The optimal predictor – that is, the predictor that makes the fewest mistakes depends on the data and is not known a priori
- Basic idea – make predictions based on a weighted majority of predictions of all predictors in the pool; if a mistake is made, adjust the weights

Weighted majority learning algorithm (WML)

```
Initialize $\mathbf{W} = (1.....1);\quad \theta_0 = \theta_1 = 0$
For each training example $(\mathbf{X}, c(\mathbf{X}))$
    If $x_i = 0,\ \ \theta_0 \leftarrow \theta_0 + w_i$
    If $x_i = 1,\ \ \theta_1 \leftarrow \theta_1 + w_i$
    Predict $y(\mathbf{X}) = 1$ if $\theta_1 > \theta_0$
    Predict $y(\mathbf{X}) = 0$if $\theta_1 < \theta_0$
    Predict $y(\mathbf{X}) = Random(\{1,0\})$ if $\theta_1 = \theta_0$
    If $c(\mathbf{X}) \neq y(\mathbf{X}),\ \ w_i \leftarrow \beta w_i\quad (0 \leq \beta < 1)$
```

Weighted majority learning algorithm (WML)

Theorem: Let D be any sequence of training examples. Let A={A_1 … A_n} be any pool of n predictors. Let k be the number of mistakes made by the best predictor in the pool A on the sequence of examples D. Then

$$m = mbound_{WML} \leq \left(\frac{k \log \left(\frac{1}{\beta} \right) + \log \ n}{\log \left(\frac{2}{1+\beta} \right)} \right)$$

Weighted majority learning algorithm (WML)

Proof: The total weight associated with the n predictors at any time is $W = (\theta_0 + \theta_1)$ where θ_0 and θ_1 are as defined in WML

Consider an example on which a mistake is made. WLOG, assume that the prediction was 0. Then the total weight after the update is

$$W_{after} \leq (\theta_0 + \theta_1) - (1-\beta)\left(\frac{\theta_0 + \theta_1}{2}\right) \quad \text{(why?)}$$

$$W_{after} \leq \left(\frac{1+\beta}{2}\right)(\theta_0 + \theta_1) = \left(\frac{1+\beta}{2}\right)W_{before}$$

The predictors in the weighted majority must have held at least half the total weight. After update, this portion of the weight gets reduced by a factor (1-β)

Each mistake causes sum of weights after an update to be no more than $\left(\frac{1+\beta}{2}\right)$ times the value before the update

Weighted majority learning algorithm (WML)

$$W_{after} \leq \left(\frac{1+\beta}{2}\right) W_{before}$$

$$W_{final} \leq \left(\frac{1+\beta}{2}\right)^m W_{init} = \left(\frac{1+\beta}{2}\right)^m n$$

where m is the number of updates

The best predictor (say w_j) makes k mistakes and hence undergoes k weight updates. So

$$w_j^{final} = \beta^k$$

Clearly, the weight of the best predictor can be no greater than the sum of weights

$$\beta^k \leq \left(\frac{1+\beta}{2}\right)^m n$$

Weighted majority learning algorithm (WML)

$$\left\{\beta^k \le \left(\frac{1+\beta}{2}\right)^m n\right\} \Rightarrow \left\{\left(\frac{\beta^k}{n}\right) \le \left(\frac{1+\beta}{2}\right)^m\right\} \Rightarrow \left\{\left(\frac{n}{\beta^k}\right) \ge \left(\frac{2}{1+\beta}\right)^m\right\}$$

$$\Rightarrow \left\{\left(\log n - k\log\beta\right) \ge m\log\left(\frac{2}{1+\beta}\right)\right\} \Rightarrow \left\{\left(\log n + k\log\left(\frac{1}{\beta}\right)\right) \ge m\log\left(\frac{2}{1+\beta}\right)\right\}$$

$$\Rightarrow m \le \left(\frac{\log n + k\log\left(\frac{1}{\beta}\right)}{\log\left(\frac{2}{1+\beta}\right)}\right)$$

Implication: WML makes almost as few mistakes as the optimal learner in the pool – We can use weighted majority when it is unclear which learner in the pool is optimal

Some Variations on WML

Selection from countably infinite pool of predictors

- No algorithm (that halts) can obtain predictions from an infinite number of predictors
- However, we can modify WML so that it considers successively larger pools of predictors
- The modified algorithm behaves very much like WML with a degradation in mistake bound of the order of (log *i*) where *i*th predictor in the pool is the optimal predictor

Some Variations on WML

Randomized predictions

- Predict 1 with probability $\left(\frac{\theta_1}{\theta_1 + \theta_0}\right)$
- Predict 0 with probability $\left(\frac{\theta_0}{\theta_1 + \theta_0}\right)$
- The update equations can be modified so that the rate of mistakes approaches arbitrarily close to the rate of mistakes of the best predictor in the pool

Some Variations on WML – Balanced Winnow

Inputs and outputs are bipolar

$$\text{Balanced Winnow}\left(\mathbf{W}^{+},\mathbf{W}^{-},(\mathbf{X},y)\right)$$

$$0<\beta<1$$

$$\text{If } sign\left(\mathbf{W}^{+}\bullet\mathbf{X}-\mathbf{W}^{-}\bullet\mathbf{X}\right)\neq y$$

$$\text{If } y=1,\ \forall i\quad w_i^{+}\leftarrow w_i^{+}+\beta^{-x_i}w_i^{+};\, w_i^{-}\leftarrow w_i^{-}+\beta^{x_i}w_i^{-}$$

$$\text{If } y=-1,\ \forall i\quad w_i^{+}\leftarrow w_i^{+}+\beta^{x_i}w_i^{+};\, w_i^{-}\leftarrow w_i^{-}+\beta^{-x_i}w_i^{-}$$

Some Variations on WML – Balanced Winnow

Balanced Winnow is equivalent to keeping a scaled sum of updates as perceptron does but with a scale factor of

$$\eta = \log\left(\frac{1}{\beta}\right)$$

$$\text{Balanced Winnow}\left(\mathbf{W},\mathbf{Z},\left(\mathbf{X},y\right)\right)$$

$$0 < \beta < 1$$

$$\text{If } sign\left(\mathbf{W}\bullet\mathbf{X}\right) \neq y$$

$$\mathbf{Z} \leftarrow \mathbf{Z} + y\left(\log\frac{1}{\beta}\right)\mathbf{X}$$

$$\mathbf{W} \leftarrow 2\sinh\left(\mathbf{Z}\right) \text{ (That is, } \forall i \;\; w_i \leftarrow 2\sinh z_i = e^{z_i} - e^{-z_i}\text{)}$$

Generalized Perceptron algorithms

```
Generalized Perceptron $(\mathbf{W}, \mathbf{Z}, f, (\mathbf{X}, y))$
If $sign(\mathbf{W} \bullet \mathbf{X}) \neq y$
    $\mathbf{Z} \leftarrow \mathbf{Z} + \eta y \mathbf{X}$
    $\mathbf{W} \leftarrow \mathbf{f}(\mathbf{Z})$ (That is, $\forall i \;\; w_i \leftarrow f(z_i)$)
```

See Grove, Littlestone, Schuurmans

Quasi additive update algorithms for function approximation

$$\text{Generalized Gradient Descent}\left(\mathbf{W}, \mathbf{Z}, f, \left(\mathbf{X}, y\right)\right)$$

$$y \leftarrow \mathbf{W} \bullet \mathbf{X}$$

$$\mathbf{Z} \leftarrow \mathbf{Z} + \eta y \mathbf{X}$$

$$\mathbf{W} \leftarrow \mathbf{f}\left(\mathbf{Z}\right) \;\; (\text{That is, } \forall i \;\; w_i \leftarrow f(z_i))$$

By choosing the learning rate η and f appropriately, we can obtain gradient-based learning algorithms that work well in the presence of irrelevant attributes (See Kivinen and Warmuth)

Applications of Multiplicative Update Algorithms

Multiplicative update algorithms constitute an example of theoretical analysis of simple algorithms leading to a new powerful family of algorithms that are useful in practice

- Spelling correction
- Text processing (SPAM filters)
- Face recognition
- Portfolio selection
- Learning in game-theoretic settings

PAC Learnability of Infinite Concept Classes

- How many random examples does a learner need to draw before it has sufficient information to learn an unknown target concept chosen from a concept class *C* ?
- Sample complexity results derived previously answer this question for the case of finite concept classes.
- Are there any non-trivial infinite concept classes that are PAC learnable from a finite set of examples?
- Can we quantify the complexity of an infinite concept class? – yes, using Vapnik-Chervonenkis Dimension!

Vapnik-Chervonenkis (VC) Dimension

- Let C be a concept class over an instance space X.
- Both C and X may be infinite.
- We need a way to describe the behavior of C on a finite set of points $S \subseteq X$.

$$S = \{X_1, X_2 \ldots X_m\}$$

- For any concept class C over X, and any $S \subseteq X$,

$$\Pi_C(S) = \{c \cap S : c \in C\}$$

- Equivalently, with a little abuse of notation, we can write

$$\Pi_C(S) = \{(c(X_1) \ldots\ldots c(X_m)) : c \in C\}$$

$\Pi_C(S)$ is the set of all dichotomies or behaviors on S that are induced or realized by C

Vapnik-Chervonenkis (VC) Dimension

If $\Pi_C(S) = \{0,1\}^m$ where $|S| = m$, or equivalently,

$|\Pi_C(S)| = 2^m$ we say that S is shattered by C.

- A set S of instances is said to be *shattered* by a hypothesis class H if and only if for *every* dichotomy of S, there exists a hypothesis in H that is consistent with the dichotomy.

VC Dimension of a hypothesis class

- Definition: The VC-dimension $V(H)$, of a hypothesis class H defined over an instance space X is the cardinality d of the largest subset of X that is shattered by H. If arbitrarily large finite subsets of X can be shattered by H, $V(H)=\infty$

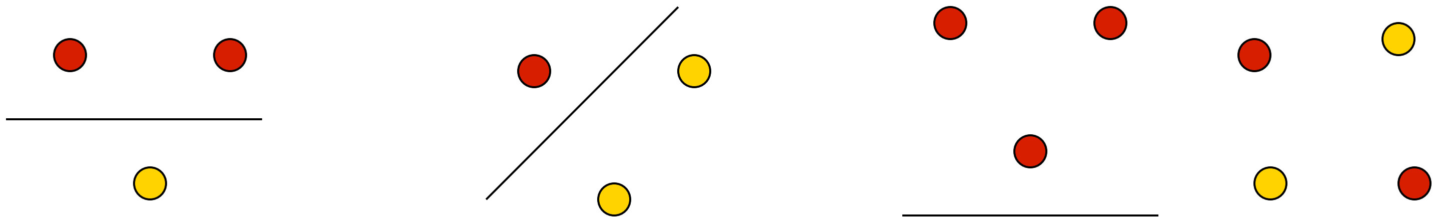
How can we show that $V(H)$ is at least d?

- Find a set of cardinality at least d that is shattered by H.

- How can we show that $V(H) = d$?
- Show that $V(H)$ is at least d and no set of cardinality $(d+1)$ can be shattered by H.

VC Dimension of a Hypothesis Class - Examples

- Example: Let the instance space X be the 2-dimensional Euclidian space. Let the hypothesis space H be the set of linear 1-dimensional hyperplanes in the 2-dimensional Euclidian space.



- Then $V(H)$=3 (a set of 3 points can be shattered by a hyperplane as long as they are not co-linear but a set of 4 points cannot be shattered). For the concept class of linear hyperplanes, VC dimension is n+1

VC Dimension and Sample complexity

- A concept class $C \subseteq 2^X$ is trivial if it contains a single concept or 2 disjoint concepts which partition X.

Theorem: Let C be a non trivial concept class.

Then C is PAC learnable if and only if $V(C)$ is finite.

If $V(C)=d$ and $d<\infty$, then the bounds on sample complexity of C are given by

$$m = O\left(\left(\frac{1}{\varepsilon}\right)\lg\left(\frac{1}{\delta}\right) + \left(\frac{d}{\varepsilon}\right)\lg\left(\frac{1}{\varepsilon}\right)\right)$$

$$m = \Omega\left(\frac{d}{\varepsilon}\right)$$

Proof: See Readings

Some Useful Properties of VC Dimension

$(C_1 \subseteq C_2) \Rightarrow V(C_1) \leq V(C_2)$

If C is a finite concept class, $V(C) \leq \lg|C|$

$\left(\overline{C} = \{X - c : c \in C\}\right) \Rightarrow V(C) = V(\overline{C})$

$(C = C_1 \cup C_2) \Rightarrow V(C) \leq V(C_1) + V(C_2) + 1$

If C_l is formed by a union or intersection of l

concepts from C, $V(C_l) = O(V(C) l \lg l)$

If $V(C) = d$, $\Pi_C(m) = \max\{|\Pi_C(S)| : |S| = m\}$,

$\Pi_C(m) \leq \Phi_d(m)$ where

$\Phi_d(m) = 2^m$ if $m \geq d$ and $\Phi_d(m) = O(m^d)$ if $m < d$

Proof: Left as an exercise

Sample complexity of a multilayer perceptron

- Acyclic, layered multi-layer networks of *s* threshold logic units, each with *r* inputs, has VC dimension

$$d = O(r+1)s\,\mathrm{lg}(s)$$

Hence, we have:

$$m = O\left(\left(\frac{1}{\varepsilon}\right)\lg\left(\frac{1}{\delta}\right)+\left(\frac{d}{\varepsilon}\right)\lg\left(\frac{1}{\varepsilon}\right)\right)$$

$$= O\left(\left(\frac{1}{\varepsilon}\right)\lg\left(\frac{1}{\delta}\right)+\left(\frac{r+1}{\varepsilon}\right)\left(s\lg\left(\frac{s}{\varepsilon}\right)\right)\right)$$

Occam Algorithm

- Theorem: An Occam algorithm is guaranteed to be PAC if the number of samples

$$m = O\left(\frac{1}{\varepsilon} \lg \frac{1}{\delta} + \left[\frac{\left(N size(c)\right)^{\alpha}}{\varepsilon} \right]^{\frac{1}{1-\beta}} \right)$$

Learning when the size of the target concept is unknown

- Results on efficient PAC learnability of concept classes are derived under the assumption that the size of the target concept is one of the inputs to the learning algorithm
- Can we guarantee efficient PAC learnability when the size of the target concept is unknown? – Yes, using the doubling trick and hypothesis testing

Aside – Hoffding Bounds

- Let $X_1 \ldots\ldots X_m$ be outcomes of independent Bernoulli trials each with probability of success p. Let

$S = \sum_{i=1}^{m} X_i$ So $E(S) = pm$

$$\Pr(S \geq pm + t) \leq e^{-2mt^2}$$

$$\Pr(S \geq \alpha m) \leq e^{-2m(\alpha - p)^2} \text{ where } \alpha \geq p$$

$$\Pr(S \leq \alpha m) \leq e^{-2m(\alpha - p)^2} \text{ where } \alpha \leq p$$

Aside – Chernoff Bounds

Let $X_1 \ldots\ldots X_m$ be independent outcomes of independent Bernoulli trials each with probability of success p. Let

$$S = \sum_{i=1}^{m} X_i$$

and

$$LE(p,m,r) = \Pr(S \le r)$$
$$GE(p,m,r) = \Pr(S \ge r)$$

$$\left.\begin{array}{l} LE\left(p,m,(1-\alpha)pm\right) \le e^{-\alpha^2 mp/2} \\ GE\left(p,m,(1+\alpha)pm\right) \le e^{-\alpha^2 mp/3} \end{array}\right\} 0 \le \alpha \le 1$$

Chernoff Bounds are tighter than Hoffding Bounds when $p < 1/4$

How to determine if a hypothesis is ε-good

- We cannot distinguish with certainty between an ε-good hypothesis and one that has error slightly greater than ε by testing the hypotheses on a finite set of examples
- However, we can distinguish between an ($\varepsilon/2$)-good hypothesis and an ε-bad hypothesis with high confidence

How to determine if a hypothesis is ε-good

Algorithm *Test* (h,n,ε,δ)

1. Make $m = \left\lceil \left(\frac{32}{\varepsilon}\right)\left(n \ln 2 + \ln\left(\frac{2}{\delta}\right)\right)\right\rceil$ calls to $Example(c,D)$ (n is the size of the instances)
2. Accept h if it misclassifies at most $\left(\frac{3\varepsilon}{4}\right)m$ examples;

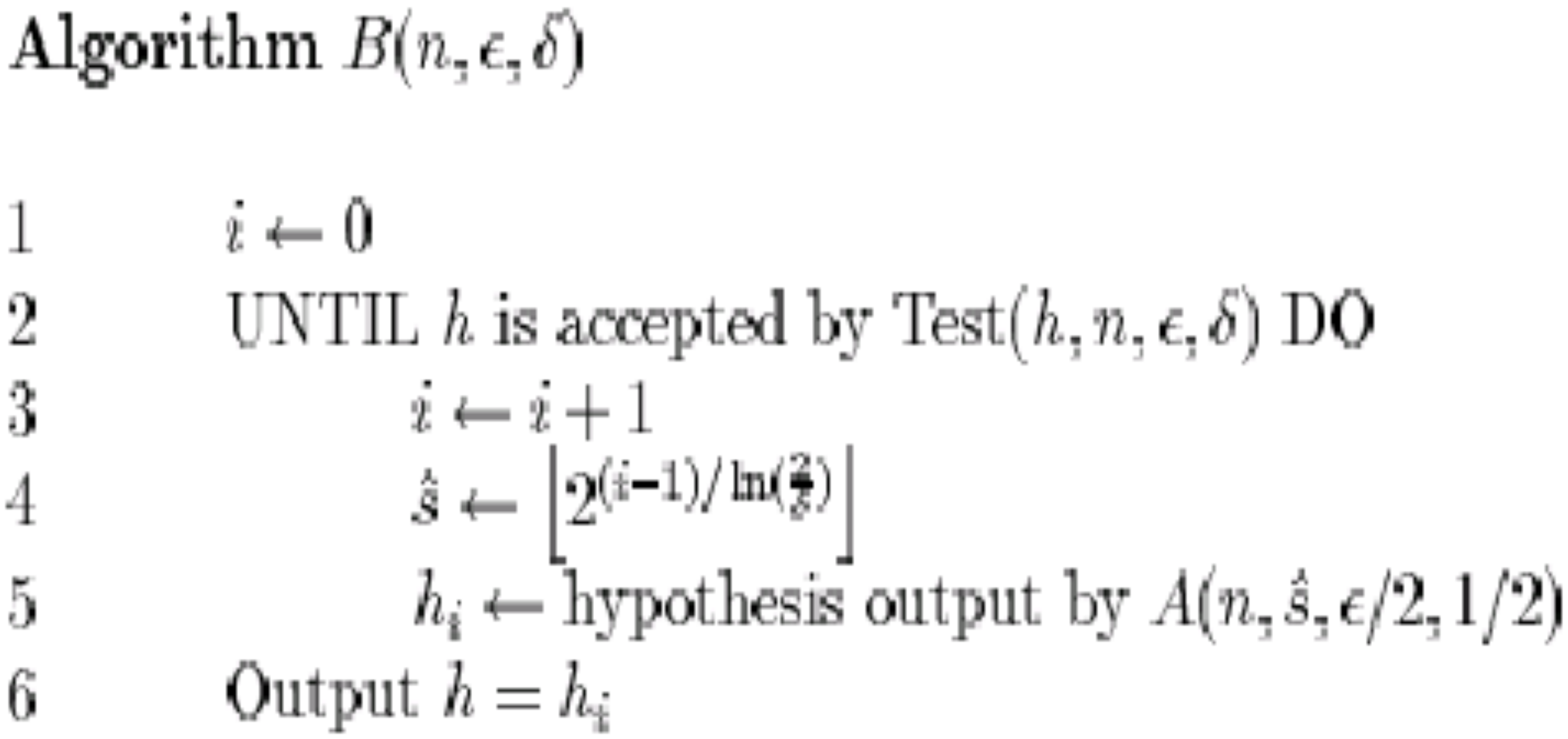
   Otherwise, reject h

$Test\ (h,n,\varepsilon,\delta)$ has the property:

If $error_{c,D}(h) \geq \varepsilon$, then $\Pr(h \text{ is accepted}) \leq \left(\frac{\delta}{2^{n+1}}\right)$

If $error_{c,D}(h) \leq \left(\frac{\varepsilon}{2}\right)$, then $\Pr(h \text{ is rejected}) \leq \left(\frac{\delta}{2^{n+1}}\right)$

Learning when the size of the target concept is unknown



Algorithm $B(n, \epsilon, \delta)$

```
$i \leftarrow 0$
UNTIL $h$ is accepted by Test$(h, n, \epsilon, \delta)$ DO
        $i \leftarrow i + 1$
        $\hat{s} \leftarrow \left\lfloor 2^{(i-1)/\ln(\frac{2}{\delta})} \right\rfloor$
        $h_i \leftarrow$ hypothesis output by $A(n, \hat{s}, \epsilon/2, 1/2)$
Output $h = h_i$
```

A – requires the target concept size as a parameter

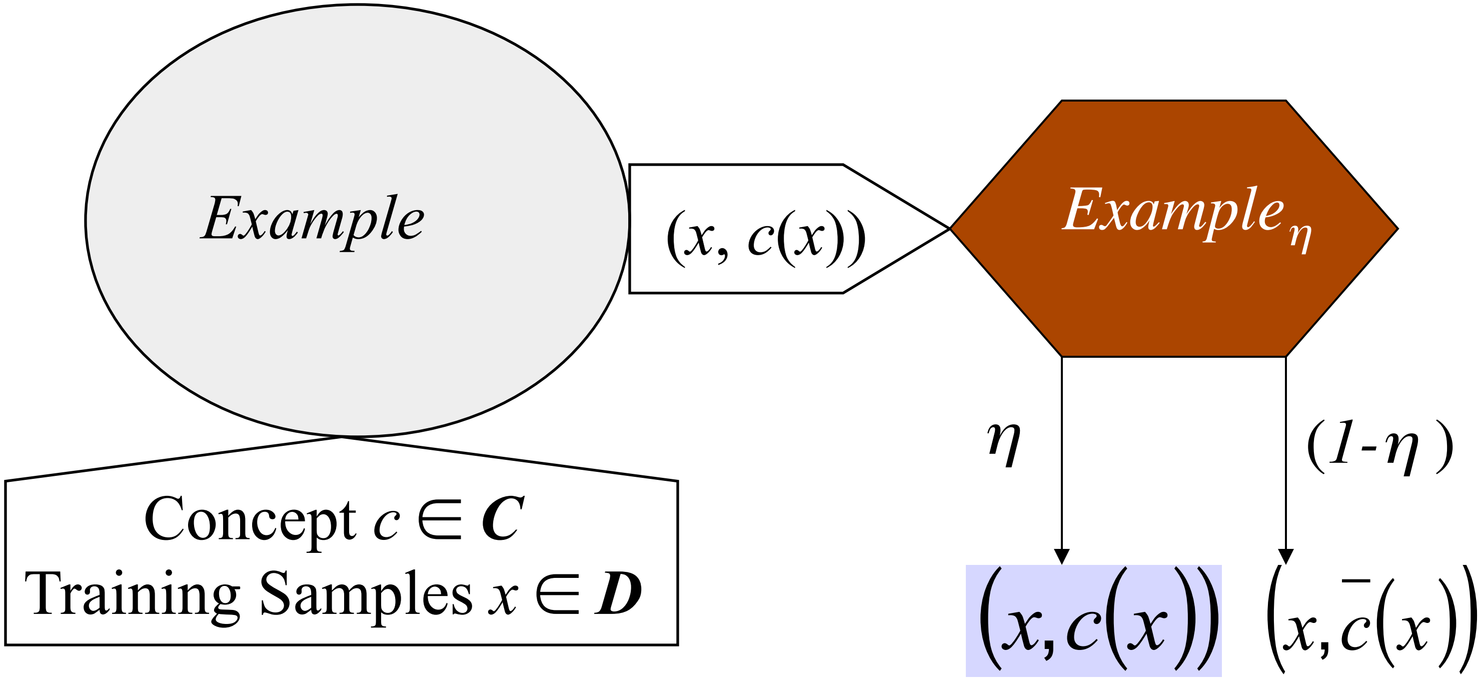
B – works for an unknown target concept size

Learning in the presence of noise

- Types of noise
- Random misclassification noise
- Random attribute noise – uniform, non uniform
- Malicious noise – examples selected and corrupted by an omnipotent adversary who may have access to the internal state of the learner
- ……

Learning in the presence of random misclassification noise

- Random misclassification noise – with probability η the instance is correctly labeled. With probability (1- η), the label is flipped


Example
$(x, c(x))$
$Example_\eta$
η
$(1-\eta\,)$
Concept $c \in \boldsymbol{C}$
Training Samples $x \in \boldsymbol{D}$
$\left(x, c\left(x\right)\right)$
$\left(x, \bar{c}\left(x\right)\right)$

Learning in the presence of random misclassification noise

- Assume WLOG that $0 \leq \eta \leq \eta_0 < 1/2$
- Draw $$m \geq \frac{2}{\varepsilon^2 (1-2\eta_0)^2} \ln\left(\frac{2|C|}{\delta}\right)$$ examples from $Example_\eta$

Output a hypothesis $h \in C$ that minimizes the training error

The method can be adapted to the case of unknown η_0
Minimizing error can be difficult in some cases
Alternative methods are available for specific concept classes

PAC learning using weak learners

Weak learner

Confidence lower than (1-δ)

→ Boost confidence

Error greater than ε

→ Boost accuracy

Error greater than ε and Confidence lower than (1-δ)

→ Boost accuracy and confidence

We can turn weak learners into strong (PAC) learners using accuracy and confidence boosting algorithms

Confidence Boosting

- Run the algorithm several times on independently drawn training sets to obtain a set of hypotheses – The number of independent runs is chosen to be large enough to ensure that the probability that at least one of the resulting hypothesis has error less than ε is at least (1-δ/2)
- Use hypothesis testing to select the best hypothesis in the pool with high confidence – alternatively use weighted majority classification

Accuracy Boosting

- Learn a sequence of hypotheses
- the first hypothesis is based on the training set
- each subsequent hypothesis is based on a sampling of the training set according to a distribution which assigns higher probability to training examples that were misclassified by the previously learned hypotheses and perhaps a different error parameter
- Classification is based on majority or weighted majority of the hypotheses

Accuracy Boosting Using Ensemble Classifiers

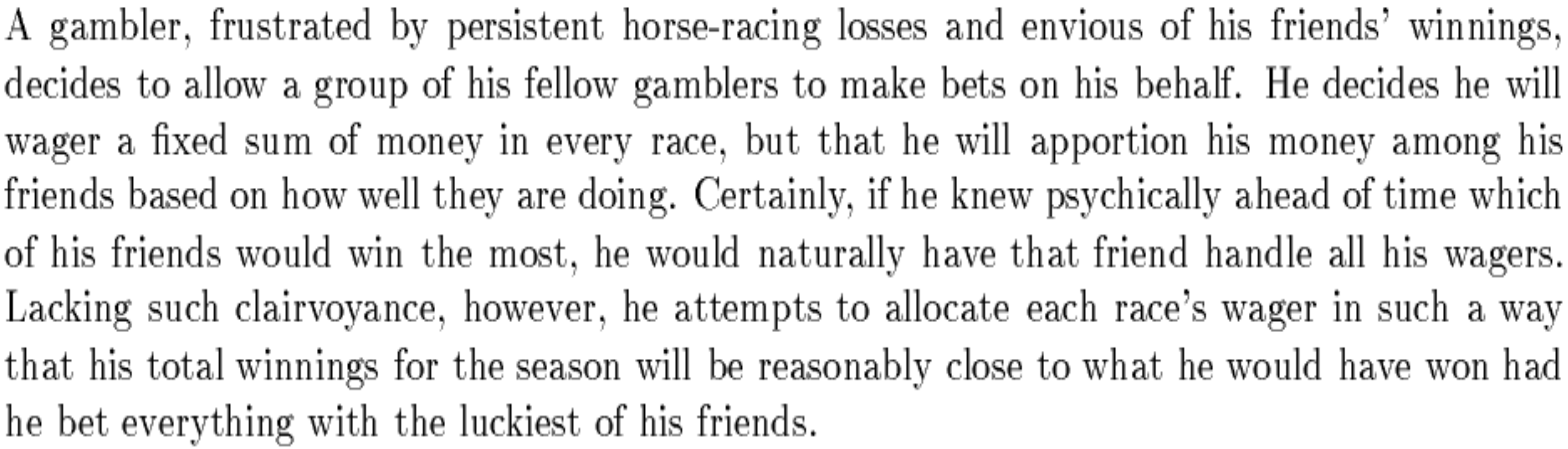
- Outline
- Ensemble methods
- Bagging
- Boosting
- Error-correcting output coding
- Why does ensemble learning work?

Readings

- Dietterich: Ensemble methods in machine learning (2000).
- Schapire: A brief introduction to boosting (1999).
- Schapire: The Boosting Approach to Machine Learning: An Overview (2002)

What is ensemble learning?

Ensemble learning refers to a collection of methods that learn a target function by training a number of individual learners and combining their predictions



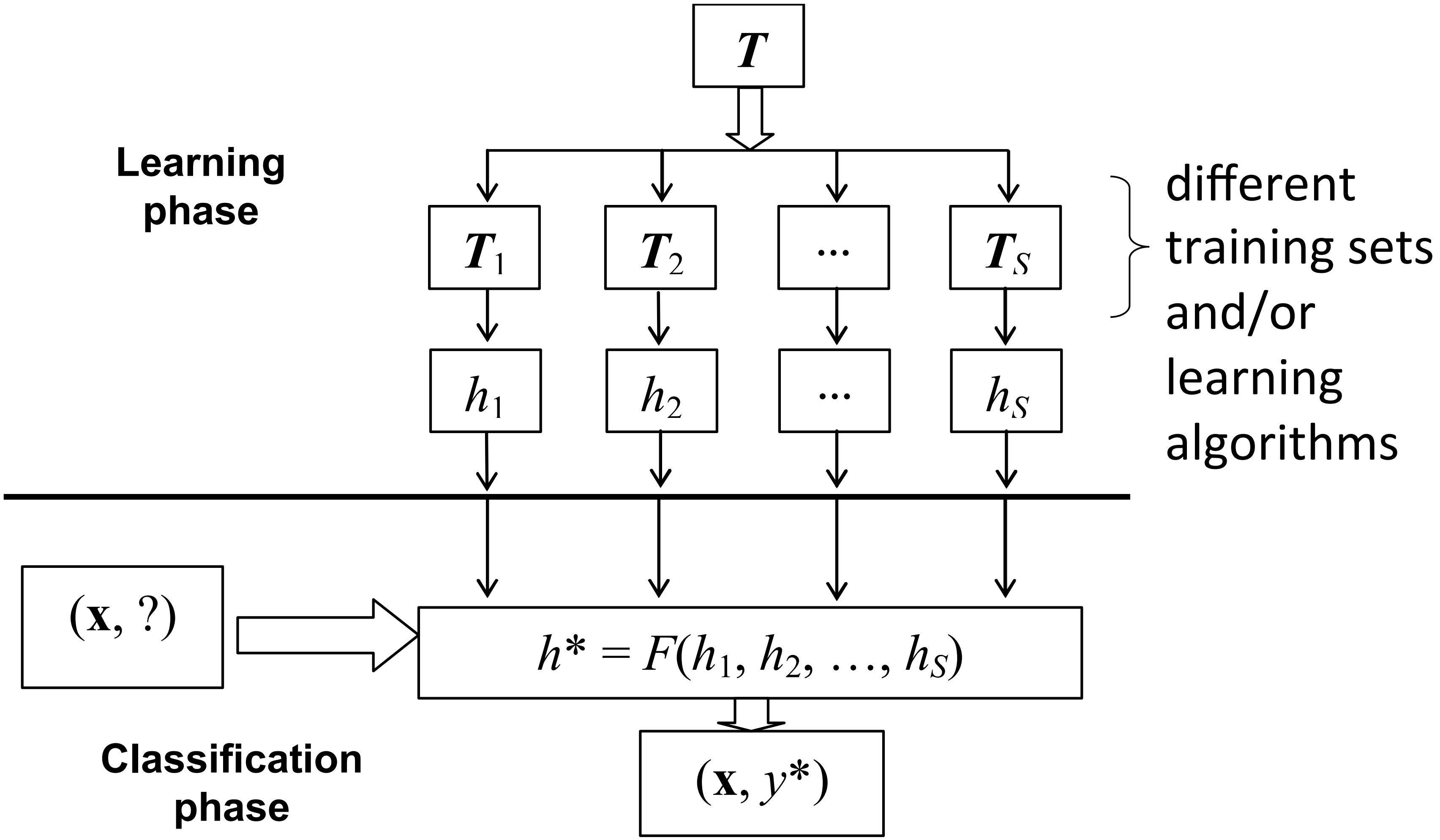
A gambler, frustrated by persistent horse-racing losses and envious of his friends' winnings, decides to allow a group of his fellow gamblers to make bets on his behalf. He decides he will wager a fixed sum of money in every race, but that he will apportion his money among his friends based on how well they are doing. Certainly, if he knew psychically ahead of time which of his friends would win the most, he would naturally have that friend handle all his wagers. Lacking such clairvoyance, however, he attempts to allocate each race's wager in such a way that his total winnings for the season will be reasonably close to what he would have won had he bet everything with the luckiest of his friends.

[Freund & Schapire, 1995]

Ensemble Learning

- Intuition: Combining Predictions of an ensemble is more accurate than a single classifier
- Justification:
  - It is easy to find quite correct “rules of thumb”
  - It is hard to find single highly accurate prediction rule
  - If the training examples are few and the hypothesis space is large then there are several equally accurate classifiers
  - Hypothesis space does not contain the true function, but it has several good approximations
  - Exhaustive global search in the hypothesis space is expensive so we can combine the predictions of several locally accurate classifiers

Ensemble learning



Learning phase

T

T_1 T_2 … T_S

h_1 h_2 … h_S

different training sets and/or learning algorithms

$(\mathbf{x}, ?)$

$h^* = F(h_1, h_2, \ldots, h_S)$

$(\mathbf{x}, y^*)$

Classification phase

How to make an effective ensemble?

Two basic questions in designing ensembles:

- How to generate the base classifiers?

  *h*1, *h*2, …

- How to combine them?

  F(*h*1(x), *h*2(x), …)

How to combine classifiers

Usually take a weighted vote:

$$ensemble(x) = sign(\sum_{i} w_i\, h_i(x))$$

- w_i is the weight of hypothesis h_i
- $w_i > w_j$ means h_i is more reliable than h_j
- typically $w_i > 0$ (though could have $w_i < 0$ meaning h_i is more often wrong than right)

- Bayesian averaging is an example
- (Fancier schemes are possible but uncommon)

How to generate base classifiers

- A variety of approaches
- Bagging (Bootstrap aggregation)
- Boosting (Specifically, Adaboost – Adaptive Boosting algorithm)
- …

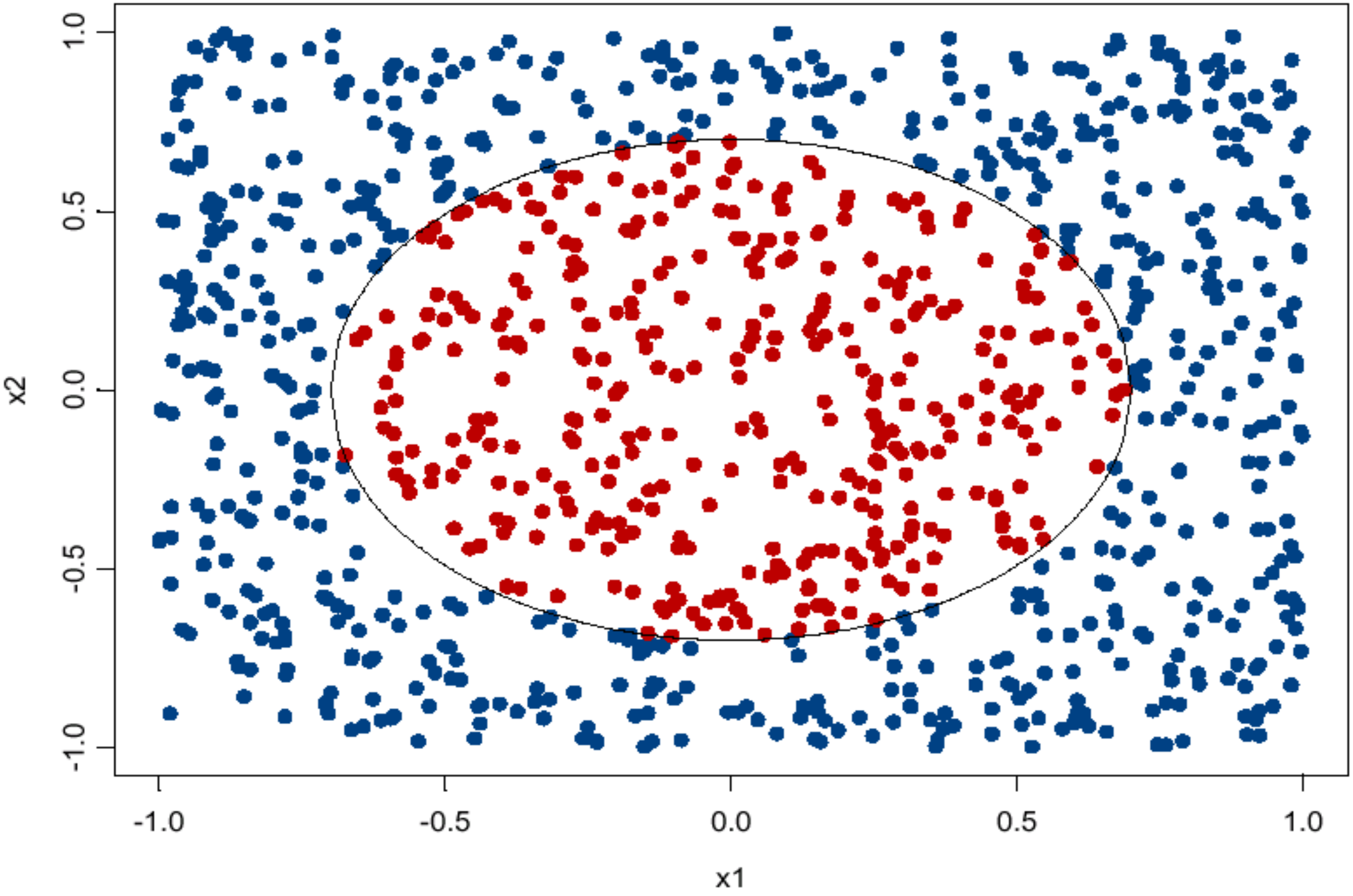
Bagging

- Generate a random sample from training set by selecting elements with replacement
- Repeat this sampling procedure, getting a sequence of *k* independent training sets
- A corresponding sequence of classifiers $C_1, C_2, \ldots, C_k$ is constructed from these training sets, by using the same classification algorithm
- To classify an unknown sample X, let each classifier predict
- The Bagged Classifier C* then combines the predictions of the individual classifiers to generate the final outcome. (sometimes combination is simple voting)

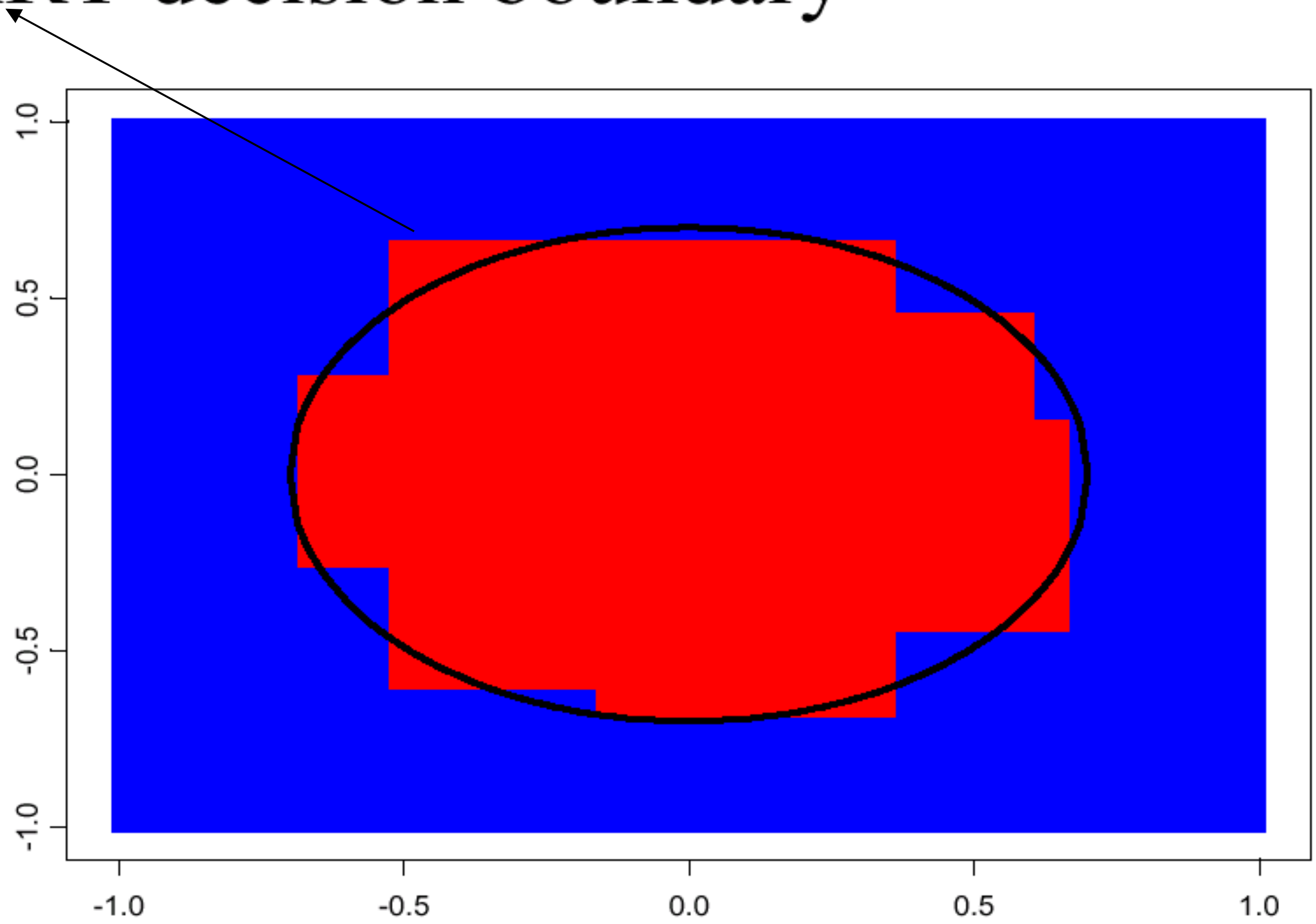
BAGGing = Bootstrap AGGregation (Breiman, 1996)

- for i = 1, 2, …, K :
  - T_i ← randomly select *M* training instances with replacement
  - h_i ← *learn* (T_i)
- Combine the T_i using uniform voting (w_i=1/K for all i)

Bagging Example

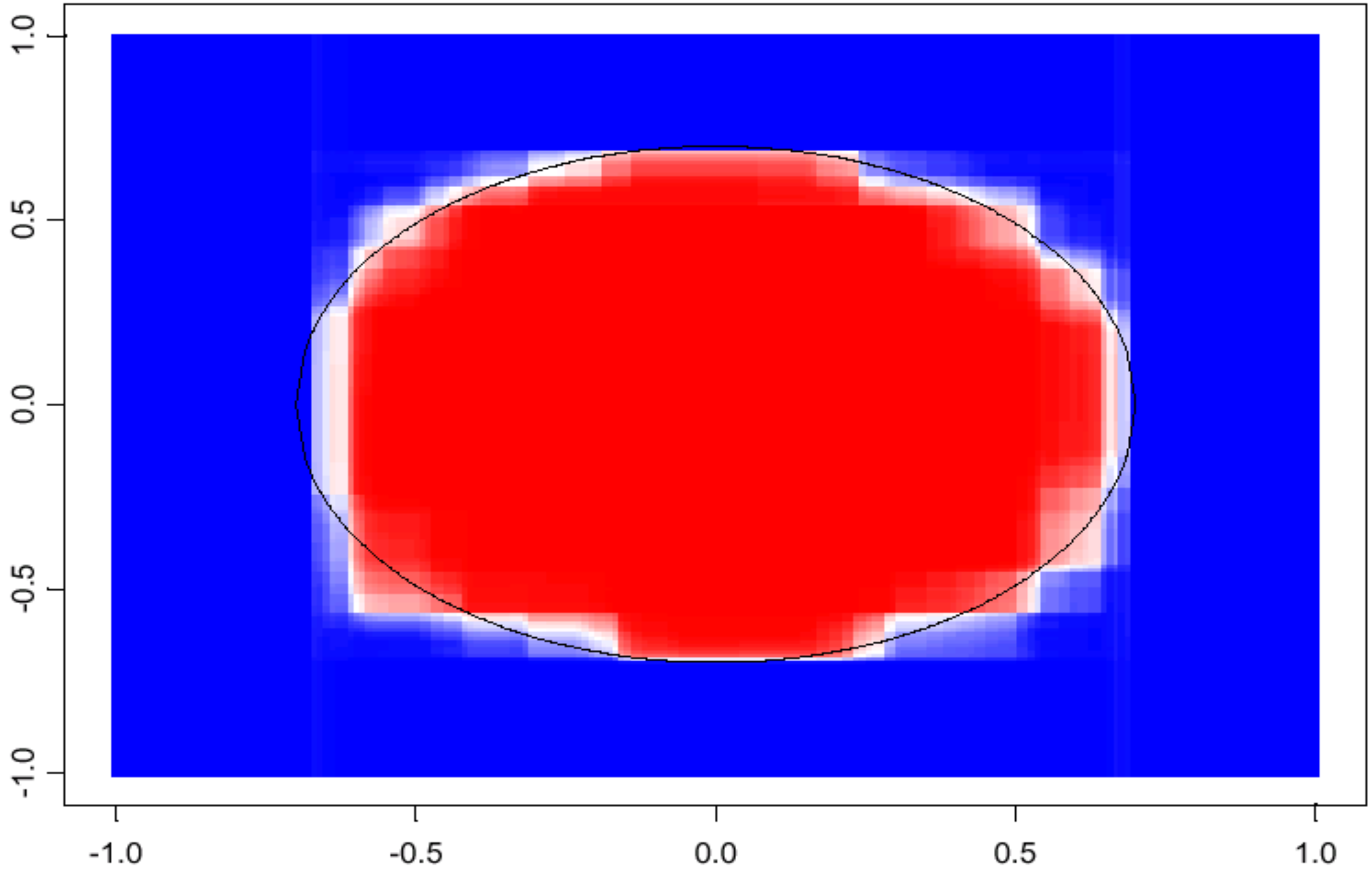


CART decision boundary



CART - decision tree learning algorithm similar to ID3

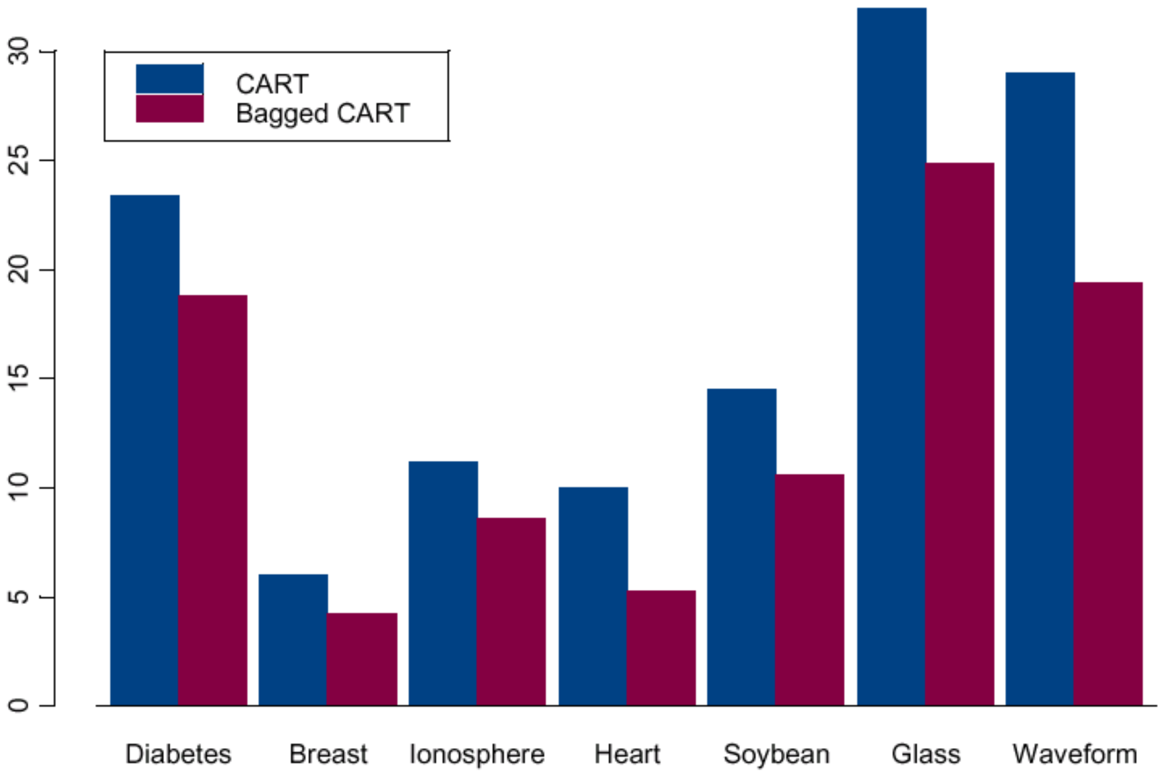
100 bagged trees



shades of blue/red indicate strength of vote for particular classification

Classification results

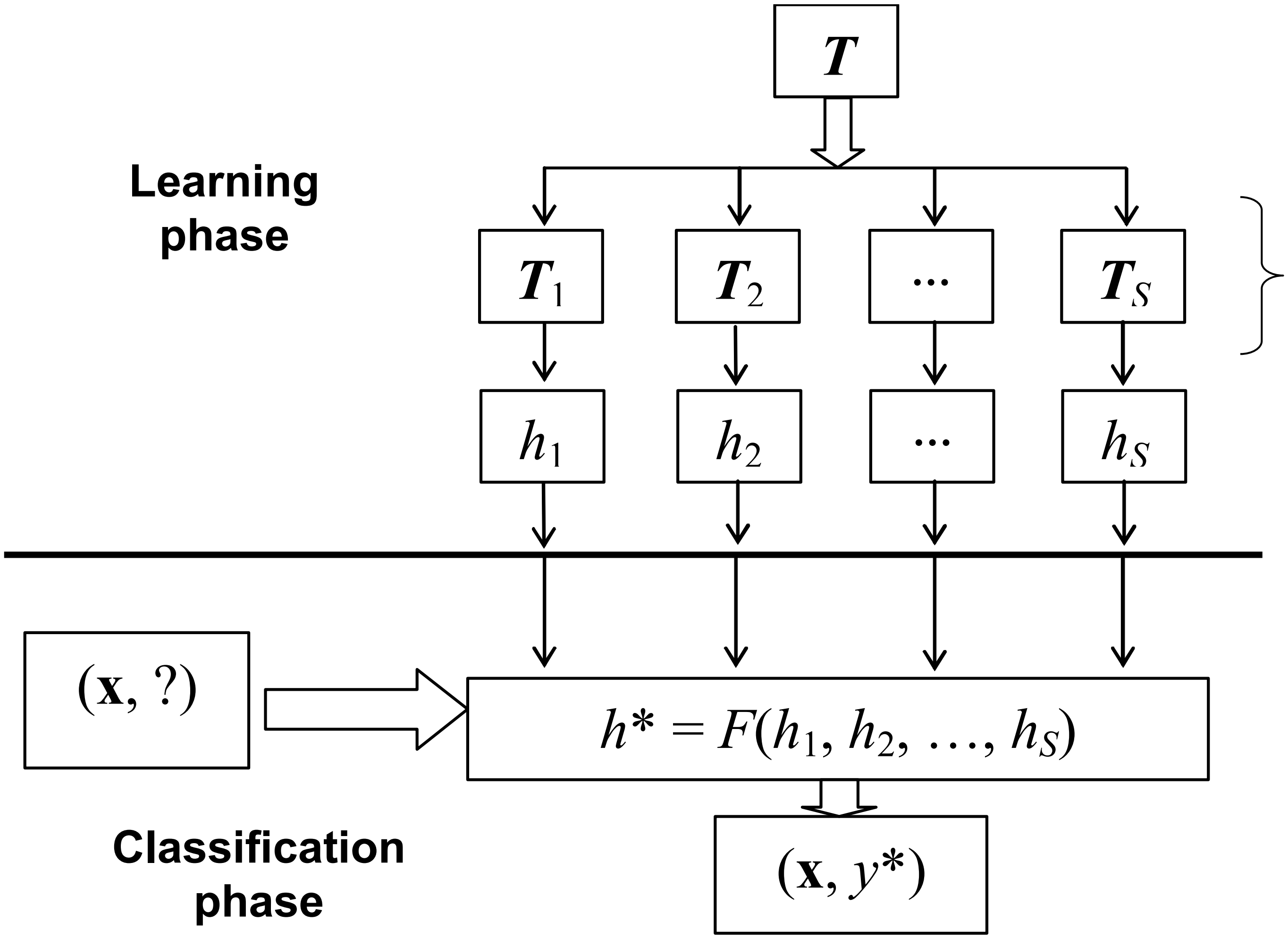
Misclassification rates



Bagging References

- Leo Breiman's homepage *www.stat.berkeley.edu/users/breiman/*
- Breiman, L. (1996) "Bagging Predictors," *Machine Learning*, 26:2, 123-140.
- Friedman, J. and P. Hall (1999) "On Bagging and Nonlinear Estimation" *www.stat.stanford.edu/~jhf*

Ensemble learning



different training sets and/or learning algorithms

Boosting

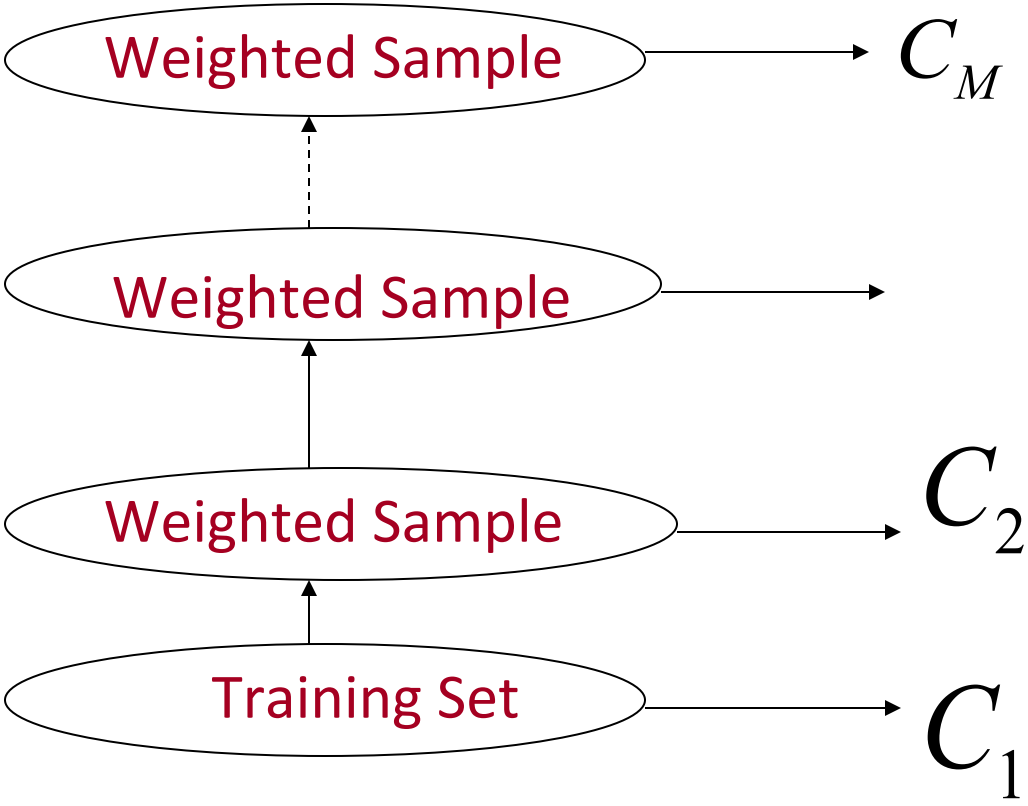
- Boosting, like bagging, is an ensemble method.
- The prediction generated by the classifier is a combination of the prediction of several predictors.
- What is different?
  - It is iterative
  - Boosting: Each successive classifier depends upon its predecessors unlike in the case of bagging where the individual classifiers were independent
  - Training Examples may have unequal weights
  - Look at errors from previous classifier step to decide how to focus on next iteration over data
  - Set weights to focus more on ‘hard’ examples. (the ones on which we committed mistakes in the previous iterations)

Boosting Algorithm

- $W(x)$ is the distribution of weights over the *N* training points $\sum W(x_i)=1$
- Initially assign uniform weights $W_0(x) = 1/N$ for all *x, step* $k=0$
- At each iteration k :
  - Find best weak classifier $C_k(x)$ using samples obtained using $W_k(x)$
    - Resulting error rate ε_k
    - The weight of the resulting classifier C_k is α_k
    - For each x_i, update weights based on ε_k to get $W_{k+1}(x_i)$
- $C_{FINAL}(x) = \text{sign} [\sum \alpha_i C_i (x)]$

Boosting (Algorithm)

$$C(x) = \sum_{j=1}^{M} \alpha_j C_j(x)$$



Boosting

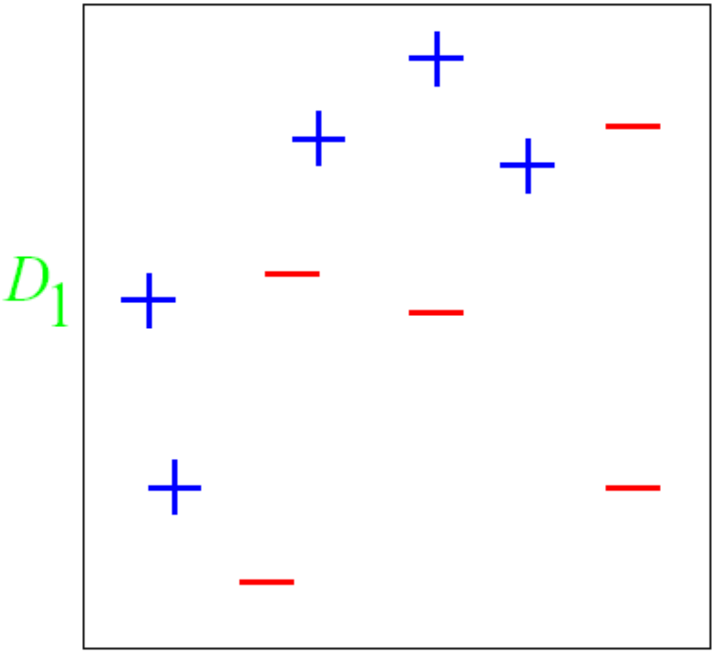
Basic Idea:

- assign a weight to every training set instance
- initally, all instances have the same weight
- as boosting proceeds, it adjusts weights based on how well we have predicted data points so far
  - data points correctly predicted →low weight
  - data points mispredicted → high weight
- Results: as learning proceeds, the learner is forced to focus on portions of data space not previously well predicted

AdaBoost Algorithm

- $W(x)$ is the distribution of weights over the *N* training points $\sum W(x_i)=1$
- Initially assign uniform weights $W_0(x) = 1/N$ for all x.
- At each iteration k :
  - Find best weak classifier $C_k(x)$ using weights $W_k(x)$
  - Compute ε_k the error rate as
    $\varepsilon_k = [\, \sum W(x_i) \cdot \mathrm{I}(y_i \neq C_k(x_i))\,]$ / $[\, \sum W(x_i)]$
  - weight the classifier C_k by α_k
    - $\alpha_k = \log((1 - \varepsilon_k)/\varepsilon_k)$
  - For each x_i, $W_{k+1}(x_i) = W_k(x_i) \cdot \exp[\alpha_k \cdot \mathrm{I}(y_i \neq C_k(x_i))]$
- $C_{FINAL}(x) = \mathrm{sign}\, [\, \sum \alpha_i C_i(x)\,]$

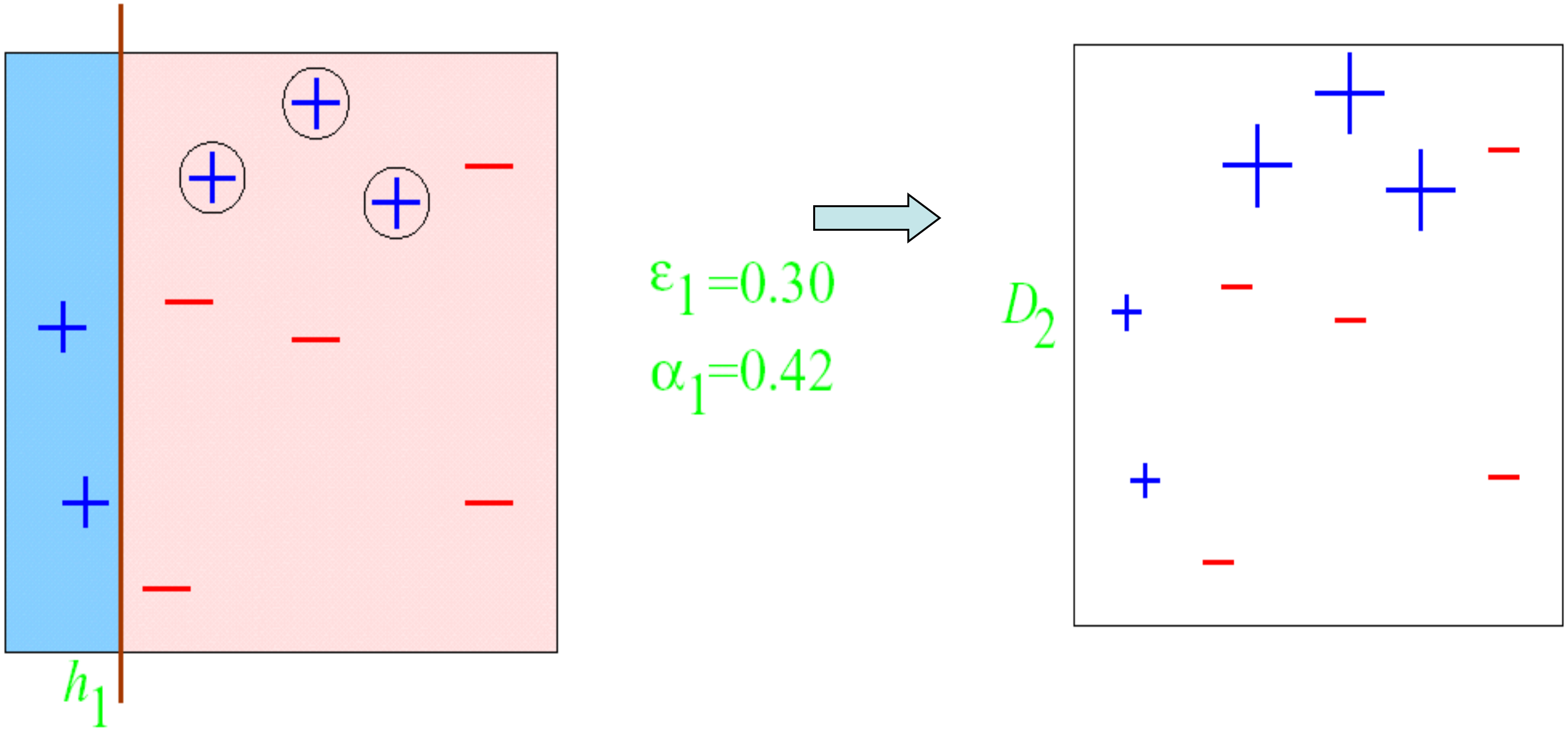
AdaBoost Example



Original Training set : Equal Weights to all training samples

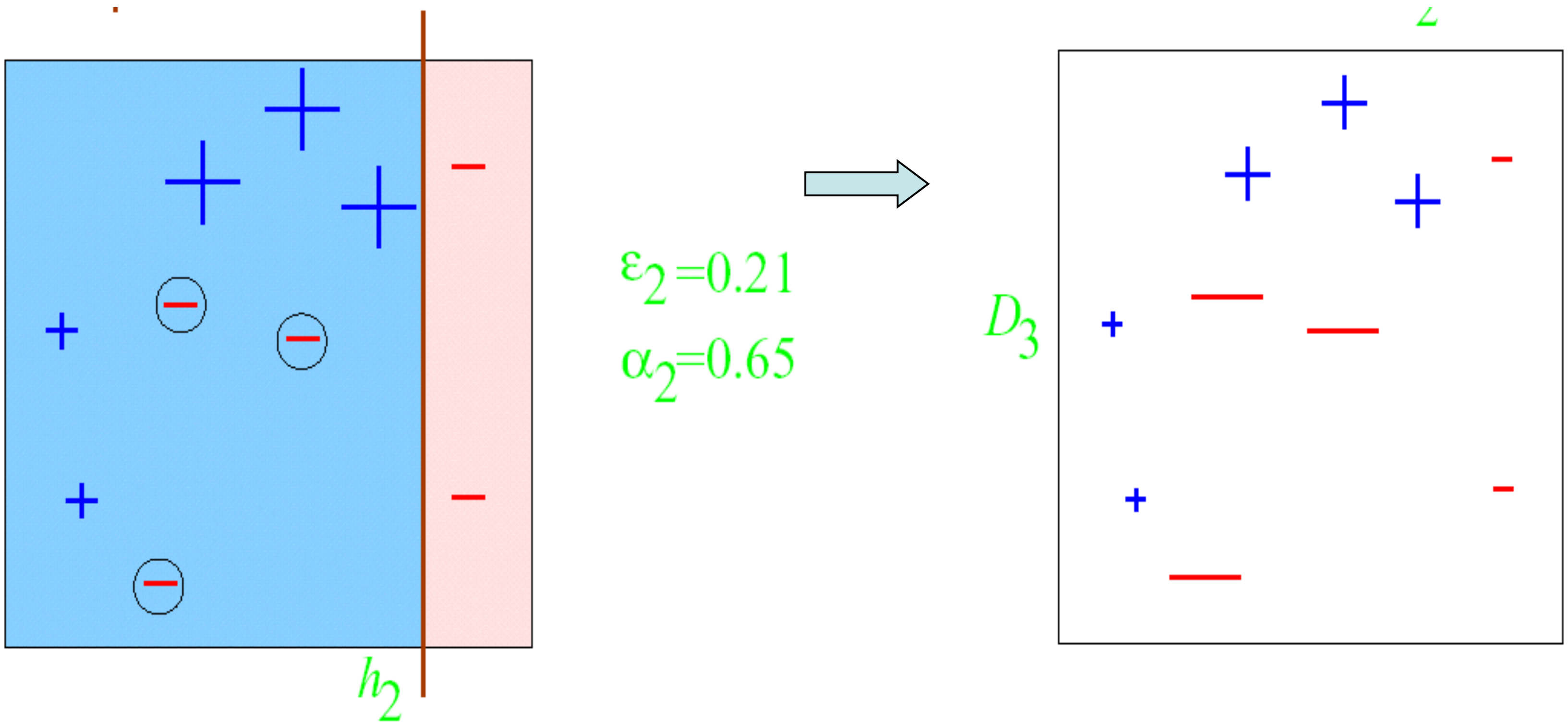
AdaBoost Example

ROUND 1



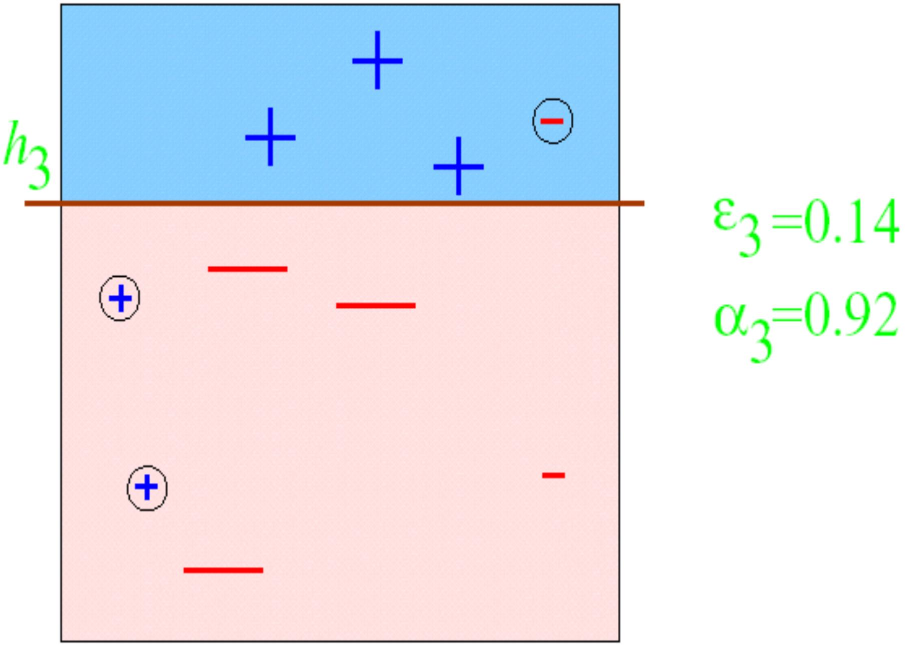
AdaBoost Example

ROUND 2

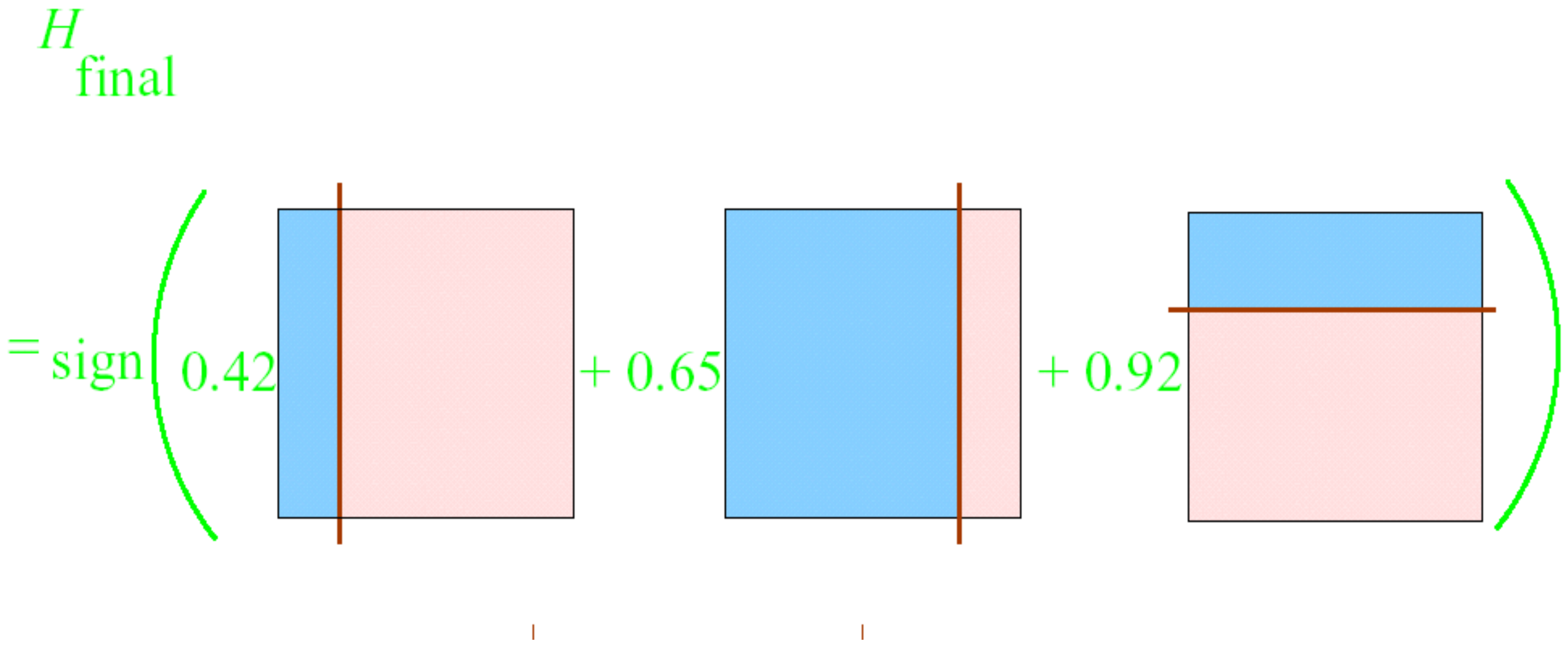


AdaBoost Example

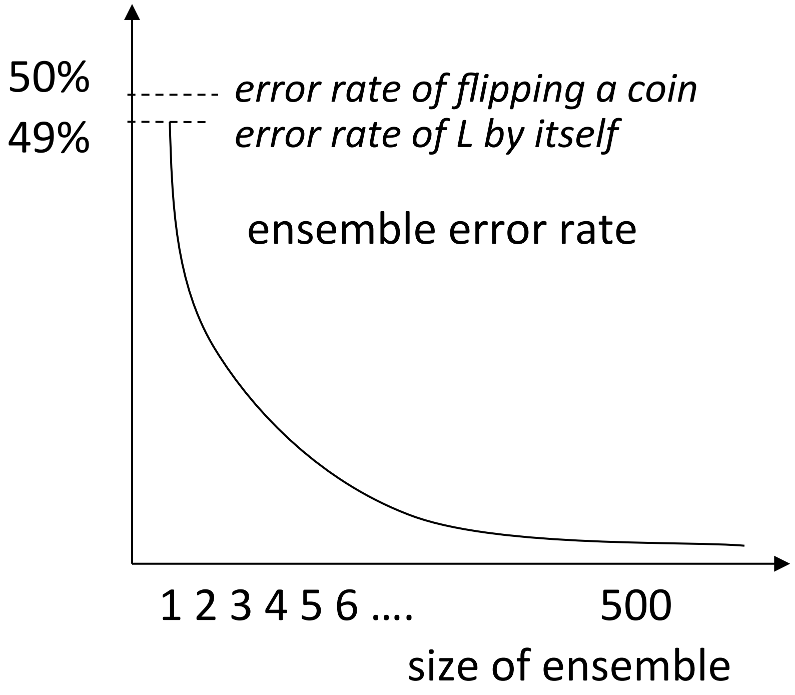
ROUND 3



AdaBoost Example



Boosting



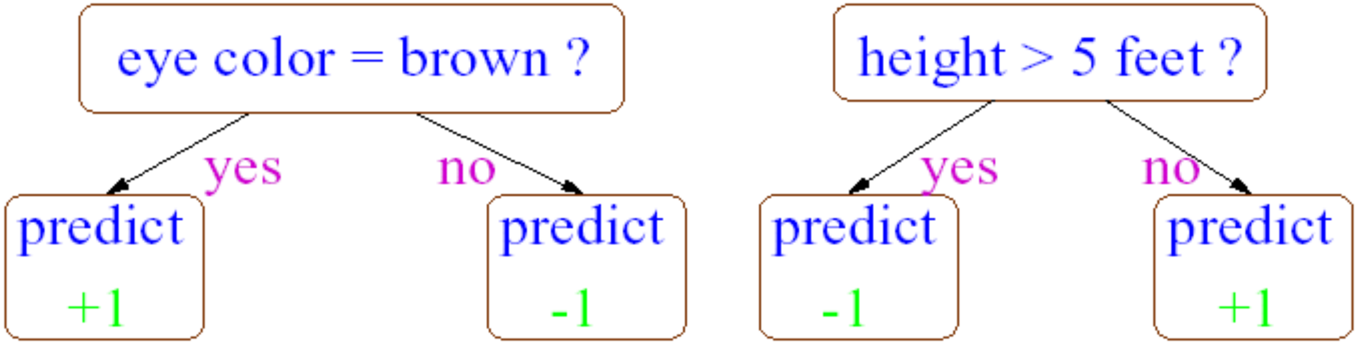
- Suppose *L* is a weak learner - one that can learn a hypothesis that is better than rolling a dice – but perhaps only a tiny bit better
  - <u>Theorem</u>: Boosting *L* yields an ensemble with arbitrarily low error on the training data!

Boosting performance

Decision stumps are very simple classifiers that test condition on a single attribute.

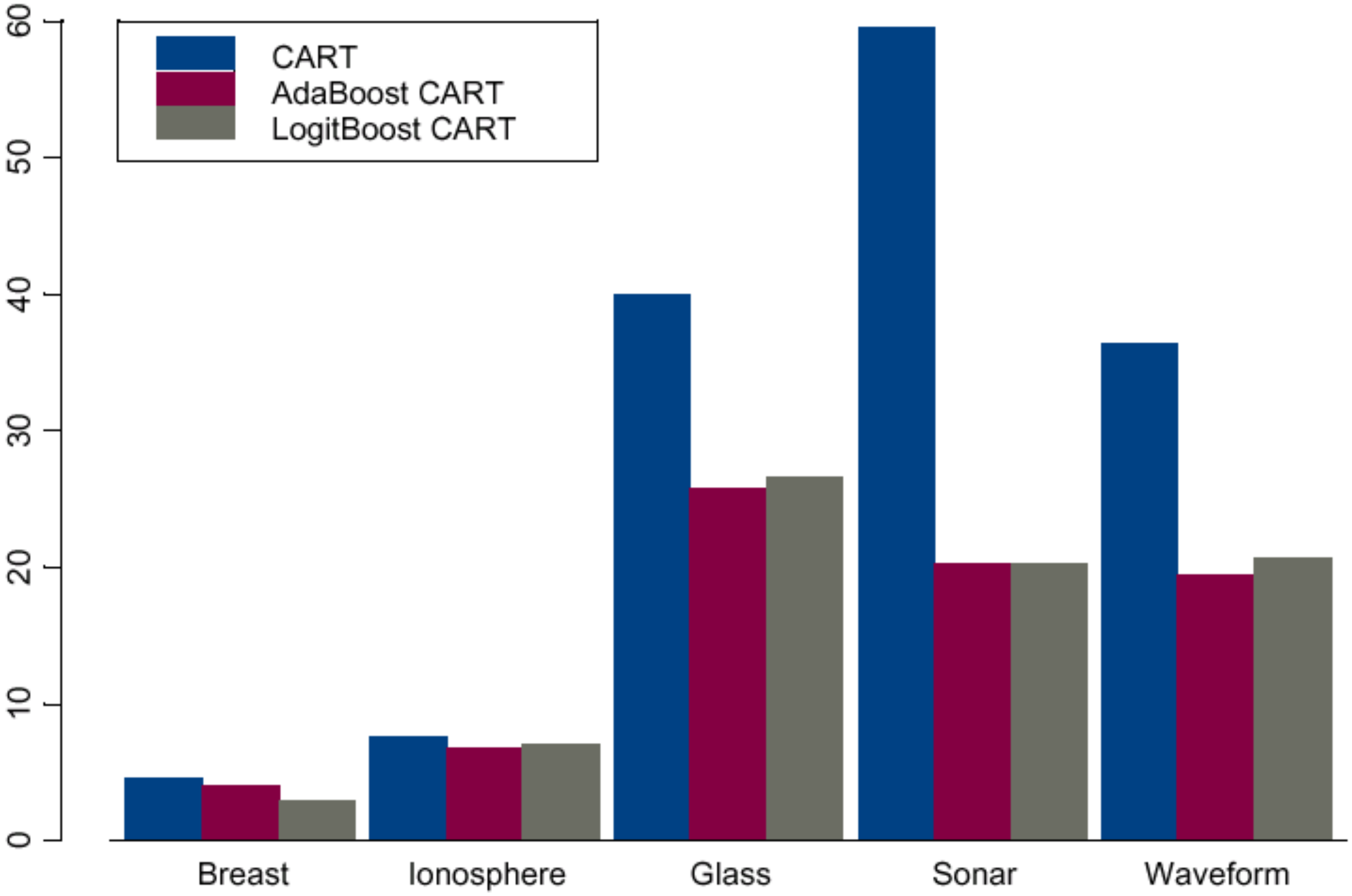
Suppose we use decision stumps as individual classifiers whose predictions were combined to generate the final prediction.

Suppose we plot the misclassification rate of the Boosting algorithm against the number of iterations performed.

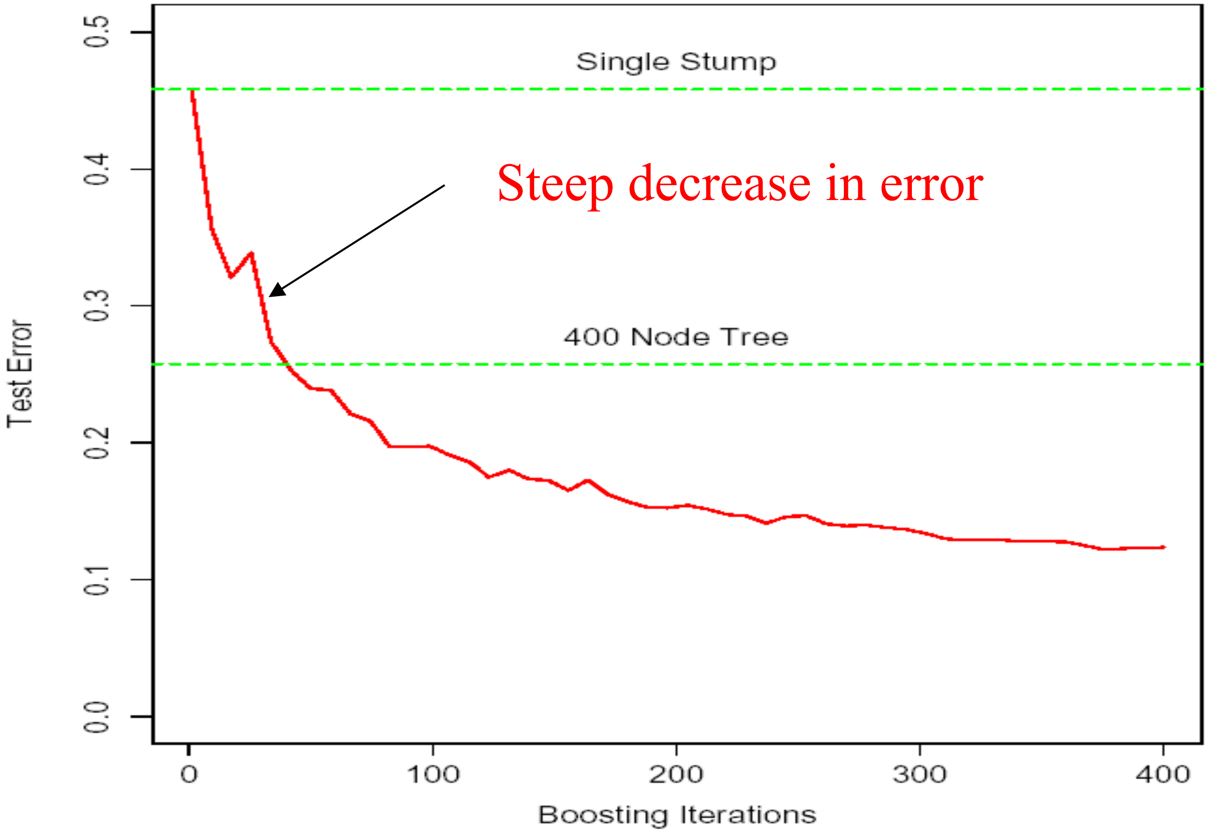


Misclassification rates

Friedman, Hastie, Tibshirani [1998]



Boosting performance



Steep decrease in error

Boosting performance

- Observations
  - First few (about 50) iterations increase the accuracy substantially.. Seen by the steep decrease in misclassification rate.
  - As iterations increase training error decreases
  - As iterations increase, generalization error decreases ?

Can Boosting do well if?

- Individual classifiers are not very accurate and have high variance (e.g., decision stumps) ?
  - It can if the individual classifiers have considerable mutual disagreement.
- Individual classifier is very accurate and has low variance (e.g., SVM with a good kernel function) ?
  - No..

Boosting as an Additive Model

- The final prediction in boosting *f(x)* can be expressed as an additive expansion of individual classifiers

$$f(\mathbf{x}) = \sum_{m=1}^{M} \beta_m b(\mathbf{x}; \boldsymbol{\gamma}_m)$$

- The process is iterative and can be expressed as follows.

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \beta_m b(\mathbf{x}; \boldsymbol{\gamma}_m)$$

- Typically we would try to minimize a loss function of the training examples

$$\min_{\{\beta_m, \boldsymbol{\gamma}_m\}_1^M} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(\mathbf{x}_i; \boldsymbol{\gamma}_m) \right)$$

Boosting as an Additive Model

- Simple case: Squared-error loss

$$L(y, f(\mathbf{x})) = \frac{1}{2}(y - f(\mathbf{x}))^2$$

- Forward stage-wise modeling amounts to just fitting the residuals from previous iteration.

$$L(y_i, f_{m-1}(\mathbf{x}_i) + \beta b(\mathbf{x}_i;\gamma))$$
$$= (y_i - f_{m-1}(\mathbf{x}_i) - \beta b(\mathbf{x}_i;)\boldsymbol{\gamma}_m)^2$$
$$= (r_{im} - \beta b(\mathbf{x}_i;\boldsymbol{\gamma}_m))^2$$

- Squared-error loss not robust for classification

Boosting as an Additive Model

- AdaBoost for Classification uses the exponential loss function:
  - L(*y*, *f* (*x*)) = exp(-*y* · *f* (*x*))

$$\arg\min_{f} \sum_{i=1}^{N} L(y_i, f(\mathbf{x}_i))$$

$$= \arg\min_{\beta, G_m} \sum_{i=1}^{N} \exp(-y_i \cdot [f_{m-1}(\mathbf{x}_i) + \beta \cdot G_m(\mathbf{x}_i)])$$

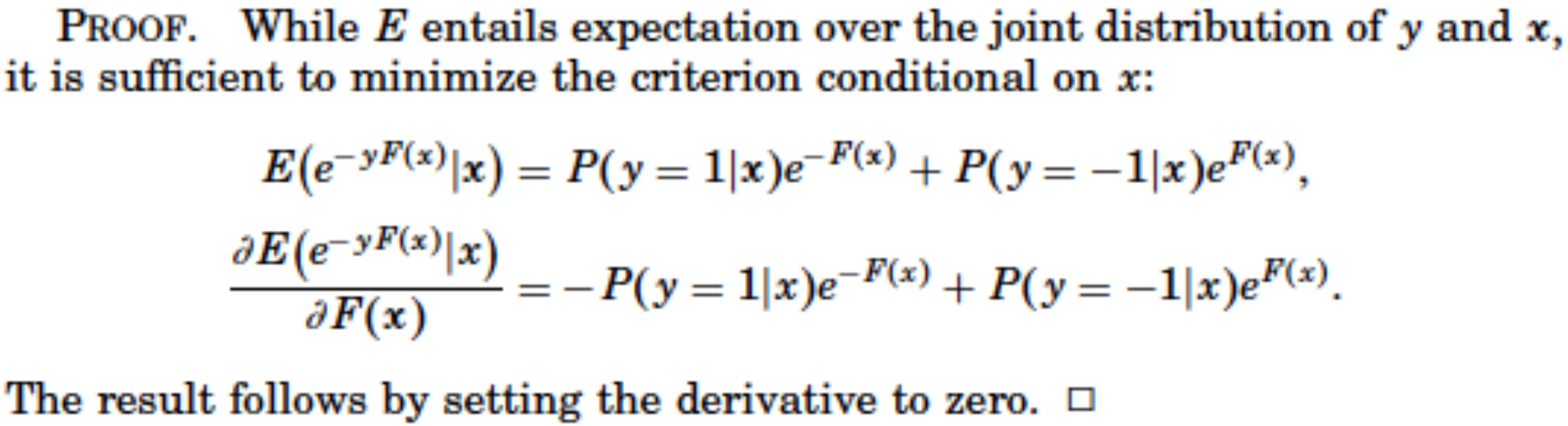
$$= \arg\min_{\beta, G_m} \sum_{i=1}^{N} \exp(-y_i \cdot f_{m-1}(\mathbf{x}_i)) \cdot \exp(-y_i \cdot \beta \cdot G_m(\mathbf{x}_i))$$

$E(e^{-yF(x)})$ *is minimized at*

$$F(x) = \frac{1}{2}\log\frac{P(y=1|x)}{P(y=-1|x)}.$$

$$P(y=1|x) = \frac{e^{F(x)}}{e^{-F(x)}+e^{F(x)}},$$

$$P(y=-1|x) = \frac{e^{-F(x)}}{e^{-F(x)}+e^{F(x)}}.$$



PROOF. While E entails expectation over the joint distribution of y and x, it is sufficient to minimize the criterion conditional on x:

$$E(e^{-yF(x)}|x) = P(y=1|x)e^{-F(x)} + P(y=-1|x)e^{F(x)},$$

$$\frac{\partial E(e^{-yF(x)}|x)}{\partial F(x)} = -P(y=1|x)e^{-F(x)} + P(y=-1|x)e^{F(x)}.$$

The result follows by setting the derivative to zero. □

Boosting as an Additive Model

First assume that β is constant, and minimize w.r.t. G_m:

$$\underset{\beta,G_m}{\operatorname{argmin}} \sum_{i=1}^{N} \exp(-y_i \cdot f_{m-1}(\mathbf{x}_i))\exp(-y_i \cdot \beta \cdot G_m(\mathbf{x}_i))$$

$$= \underset{\beta,G_m}{\operatorname{argmin}} \sum_{i=1}^{N} w_i^{(m)} \cdot \exp(-y_i \cdot \beta \cdot G_m(\mathbf{x}_i)),\ \ where\ w_i^{(m)} = \exp(-y_i \cdot f_{m-1}(\mathbf{x}_i))$$

$$= \underset{G_m}{\operatorname{argmin}} \sum_{y_i = G_m(x_i)}^{N} w_i^{(m)} \cdot e^{-\beta} + \sum_{y_i \neq G_m(x_i)}^{N} w_i^{(m)} \cdot e^{\beta}$$

$$= \underset{G_m}{\operatorname{argmin}}\ (e^{\beta} - e^{-\beta}) \sum_{i=1}^{N} [w_i^{(m)} \cdot I(y_i \neq G_m(\mathbf{x}_i))] + e^{-\beta} \sum_{i=1}^{N} w_i^{(m)}$$

$$= \underset{G_m}{\operatorname{argmin}}\ (e^{\beta} - e^{-\beta}) \frac{\sum_{i=1}^{N} [w_i^{(m)} \cdot I(y_i \neq G_m(\mathbf{x}_i))]}{\sum_{i=1}^{N} w_i^{(m)}} + e^{-\beta}$$

Boosting as an Additive Model

$$\arg\min_{G_m} (e^{\beta} - e^{-\beta}) \boxed{\frac{\sum_{i=1}^{N}[w_i^{(m)} \cdot I(y_i \neq G(\mathbf{x}_i))]}{\sum_{i=1}^{N} w_i^{(m)}}} + e^{-\beta}$$

$$= \arg\min_{G_m} (e^{\beta} - e^{-\beta}) \boxed{\cdot\, err_m} + e^{-\beta} = H(\beta)$$

err_m: the training error on the weighted samples

On each iteration we must find a classifier that minimizes the training error on the weighted samples!

Boosting as an Additive Model

Now that we have found G, we minimize w.r.t. β:

$$H(\beta) = err_m \cdot (e^{\beta} - e^{-\beta}) + e^{-\beta}$$

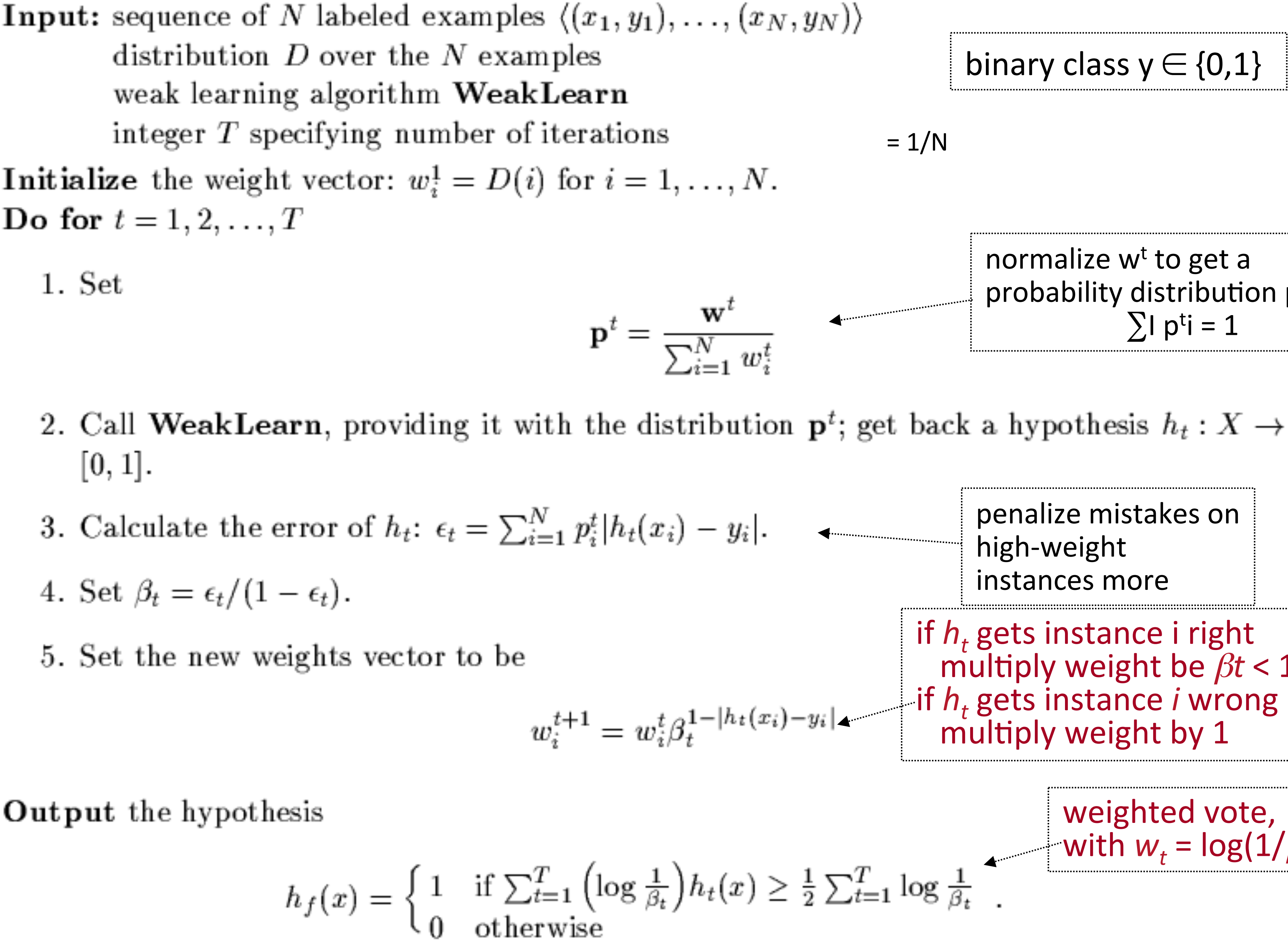
$$\frac{\partial H}{\partial \beta} = err_m \cdot (e^{\beta} + e^{-\beta}) - e^{-\beta} = 0$$

$$1 - e^{\beta} \cdot err_m (e^{\beta} + e^{-\beta}) = 0$$

$$1 - e^{2\beta} \cdot err_m - err_m = 0$$

$$\frac{1 - err_m}{err_m} = e^{2\beta}$$

$$\beta = \frac{1}{2} \log\left(\frac{1 - err_m}{err_m}\right)$$



Input: sequence of N labeled examples $\langle (x_1, y_1), \ldots, (x_N, y_N) \rangle$
distribution D over the N examples
weak learning algorithm **WeakLearn**
integer T specifying number of iterations

binary class y ∈ {0,1}

= 1/N

Initialize the weight vector: $w_i^1 = D(i)$ for $i = 1, \ldots, N$.

Do for $t = 1, 2, \ldots, T$

1. Set

$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^N w_i^t}$$

normalize w^t to get a probability distribution p^t
$\sum$I p^{t}i = 1

2. Call **WeakLearn**, providing it with the distribution $\mathbf{p}^t$; get back a hypothesis $h_t : X \to [0,1]$.

3. Calculate the error of h_t: $\epsilon_t = \sum_{i=1}^N p_i^t |h_t(x_i) - y_i|$.

penalize mistakes on high-weight instances more

4. Set $\beta_t = \epsilon_t / (1 - \epsilon_t)$.

5. Set the new weights vector to be

$$w_i^{t+1} = w_i^t \beta_t^{1 - |h_t(x_i) - y_i|}$$

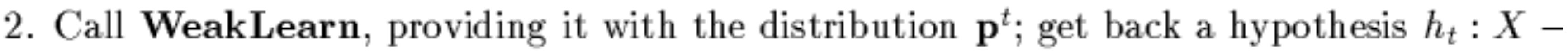
if h_t gets instance i right
multiply weight be βt < 1
if h_t gets instance i wrong
multiply weight by 1

Output the hypothesis

$$h_f(x) = \begin{cases} 1 & \text{if } \sum_{t=1}^T \left(\log \frac{1}{\beta_t}\right) h_t(x) \geq \frac{1}{2} \sum_{t=1}^T \log \frac{1}{\beta_t} \\ 0 & \text{otherwise} \end{cases}.$$

weighted vote,
with w_t = log(1/β t)

Learning from weighted instances?



2. Call **WeakLearn**, providing it with the distribution $\mathbf{p}^t$; get back a hypothesis $h_t : X -$ [0,1]

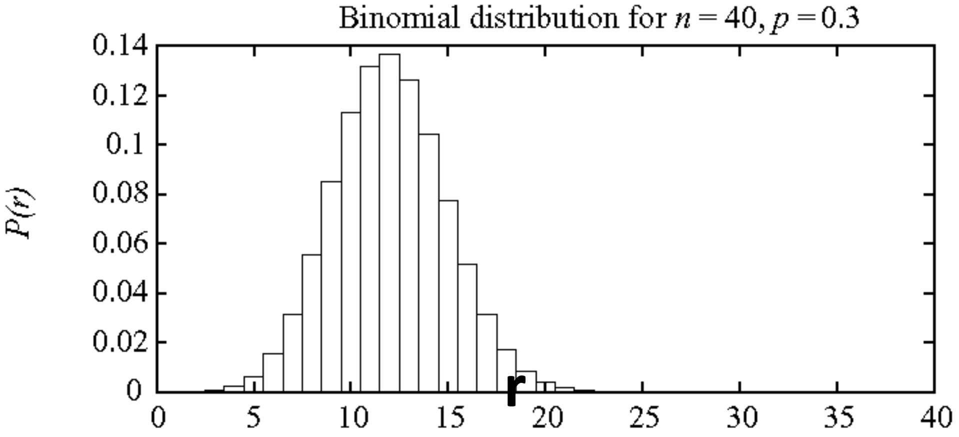
- The learning algorithms we have seen take as input a set of unweighted examples
- What if we have weighted examples instead?
- It is easy to modify most learning algorithms to deal with weighted instances:
  - For example, replace counts of examples that match some specified criterion by sum of weights of examples that match the specified criterion

AdaBoost (Characteristics)

- Why exponential loss function?
  - Computational
    - Simple modular re-weighting
    - Determining optimal parameters is relatively easy
  - Statistical
    - In a two label case it determines one half the log odds of $P(Y=1|x)$
    - We can use the sign as the classification rule
- Accuracy depends upon number of iterations

Why do ensembles work?

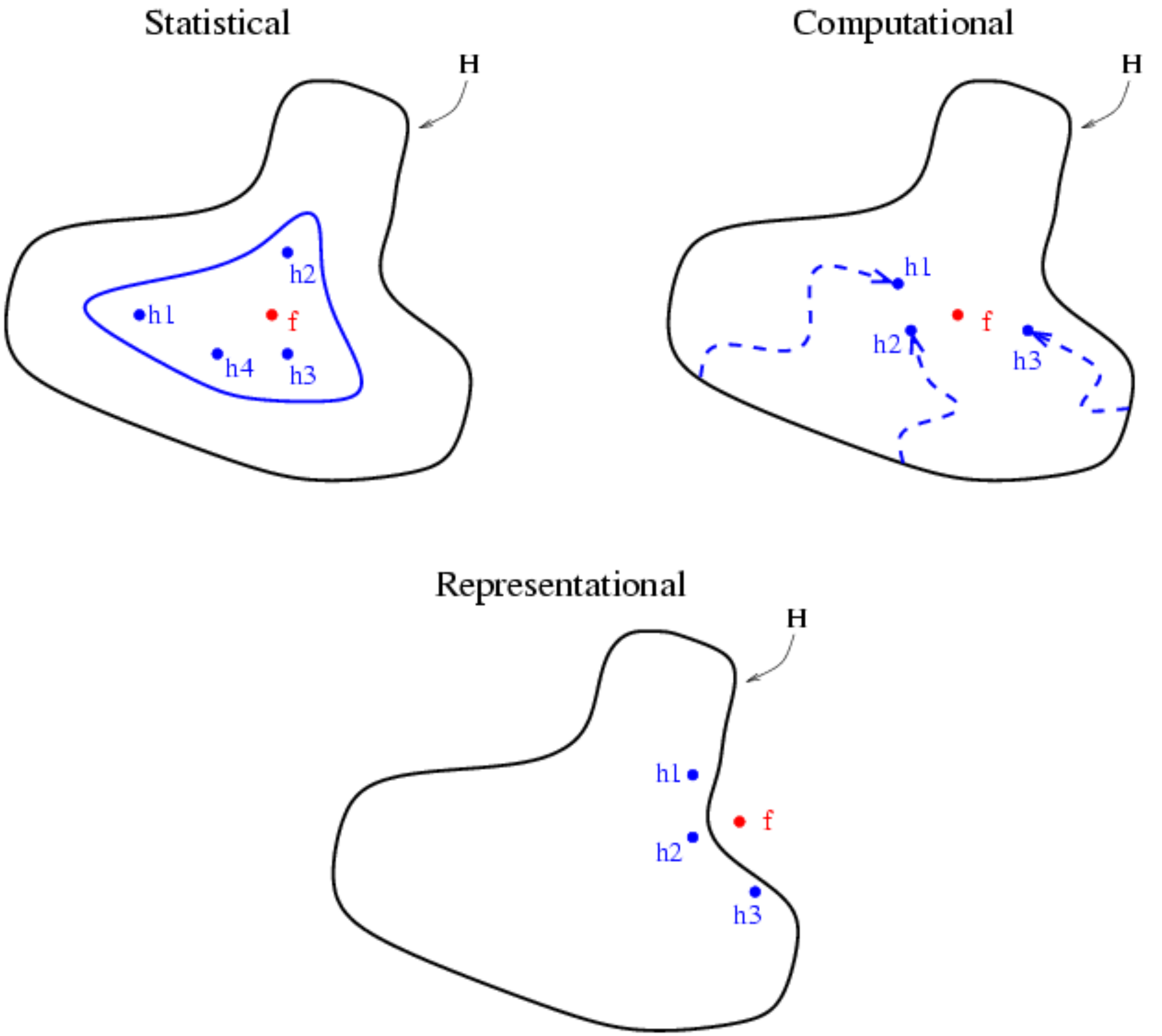
- Because uncorrelated errors of individual classifiers can be eliminated by averaging.
- Assume: 40 base classifiers, majority voting, each error rate 0.3
- Probability of getting ***r*** incorrect votes from 40 classifiers



$$P(r) = \frac{n!}{r!(n-r)!} error_D(h)^r (1 - error_D(h))^{n-r}$$

p(Ensemble is wrong) = p(>20 incorrect votes) = 0.01

Other explanations?



Statistical

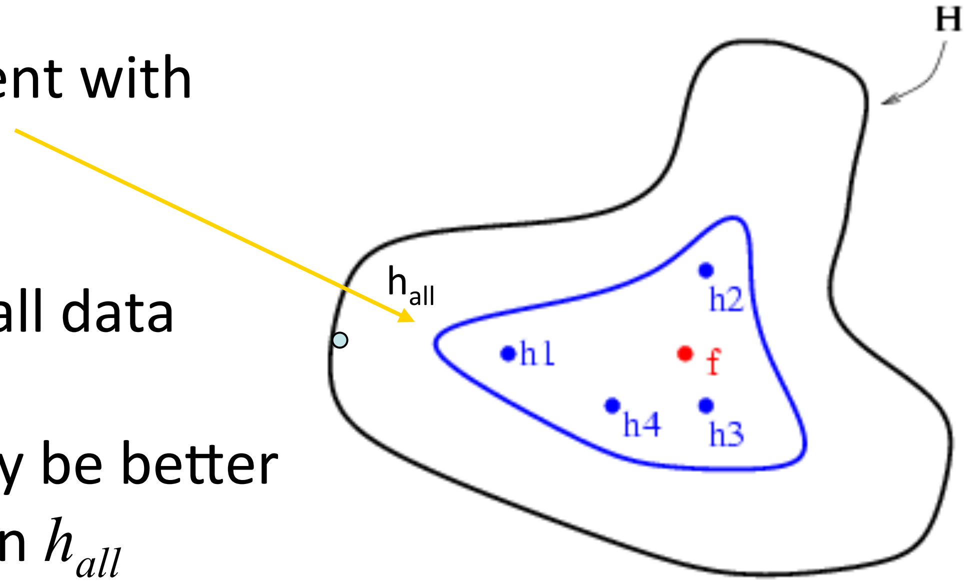
Computational

Representational

Statistical

- Given a finite amount of data, many hypothesis are typically equally good.
- How can the learning algorithm select among them?

hypothesis consistent with training data



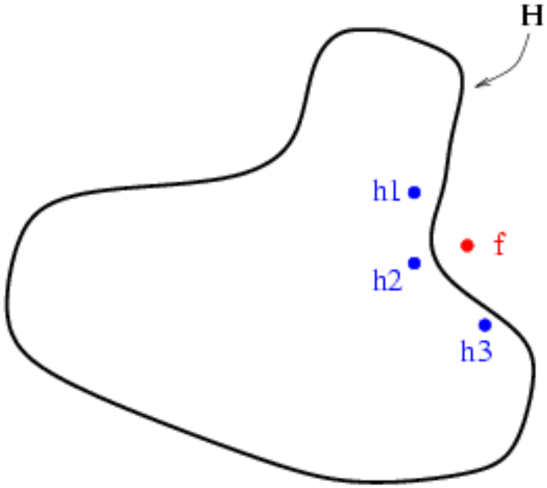
h_{all} = hypothesis from all data

averaged $h_1, h_2, \ldots$ may be better approximation to f than h_{all}

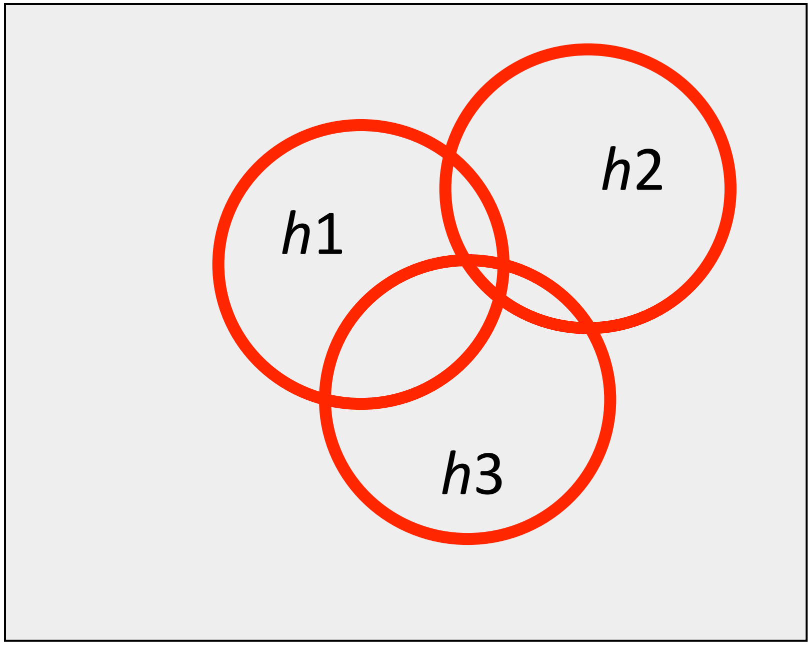
Representational

The desired target function may not be realizable using individual classifiers, but may be approximated by ensemble averaging

Consider a binary learning task over [0,1] x [0,1], and the hypothesis space H of "discs"

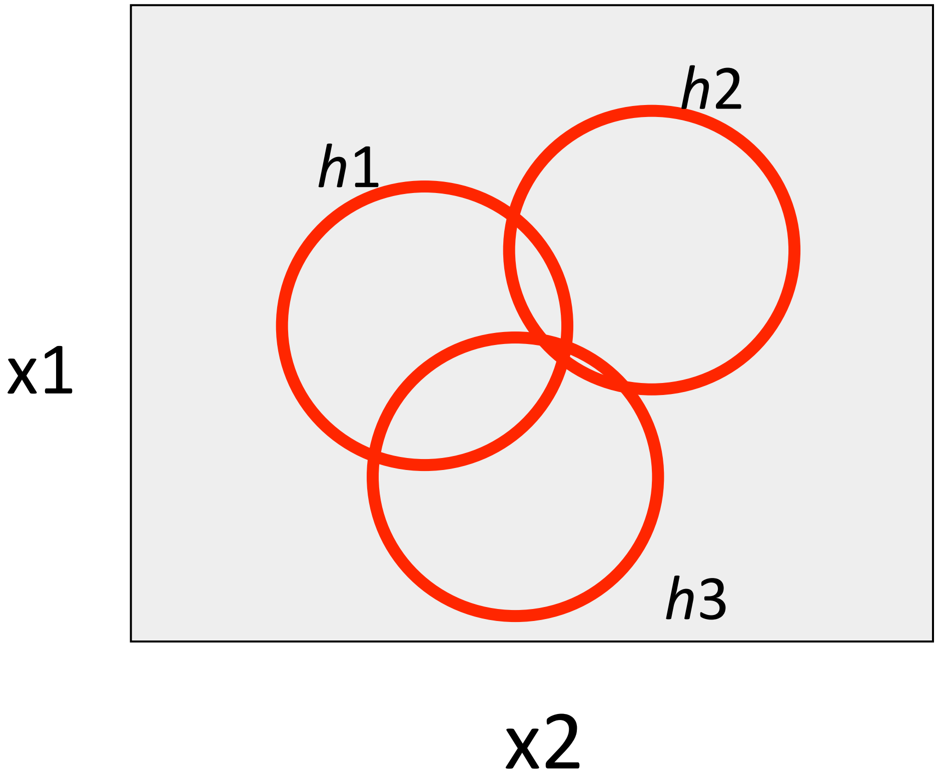


$h_1, h_2, h_3 \in H$



Representational (another example)

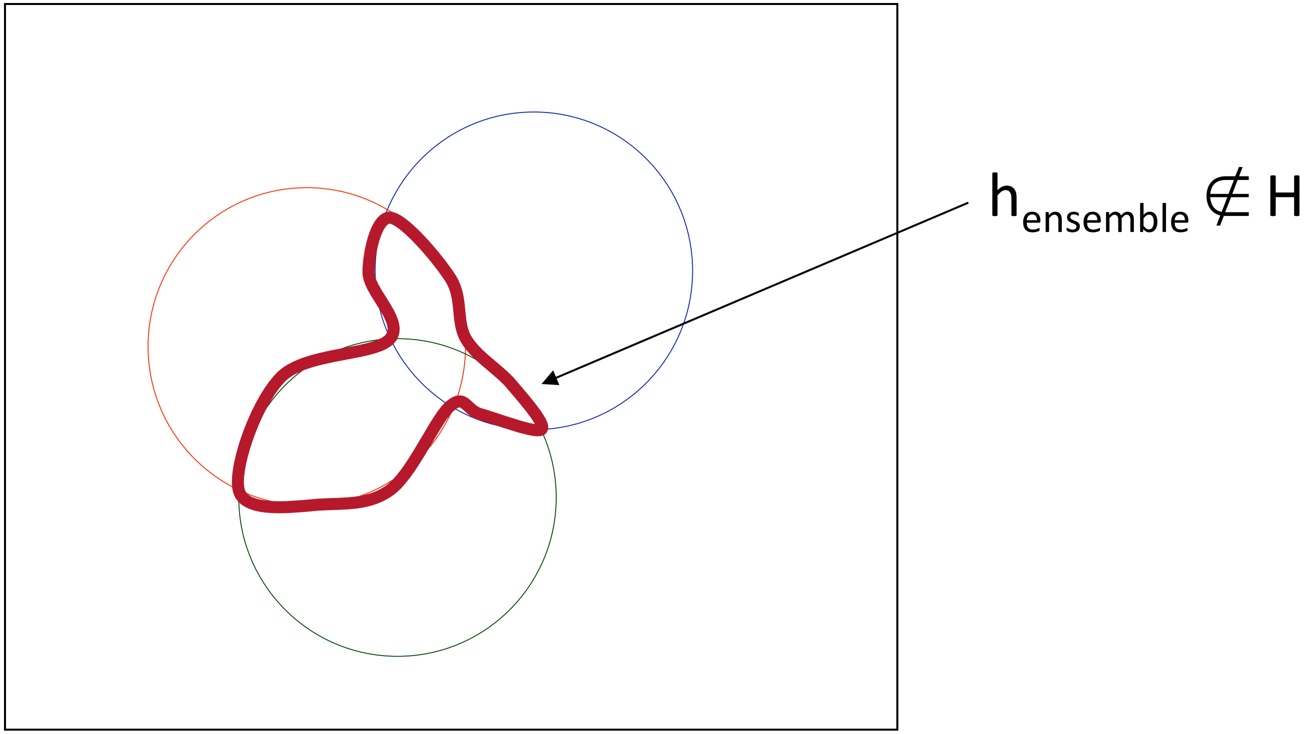
- Consider a binary learning task over [0,1] x [0,1], and the hypothesis space H of “discs”



$$h1, h2, h3 \in H$$

Representational (another example)

- H_{ensemble} = vote together h_1, h_2, h_3

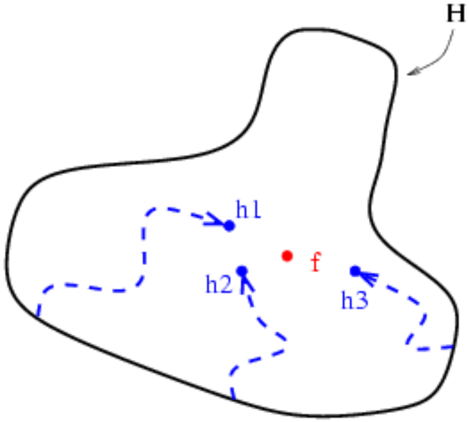


- Even if target concept $\notin$ H, a mixture of hypothesis $\in$ H might be highly accurate

Computational

- All learning algorithms do some sort of search through some space of hypotheses to find one that is "good enough" for the given training data
- Since interesting hypothesis spaces are huge/infinite, heuristic search is essential (e.g. decision tree learner does a greedy search in space of possible decision trees)
- So the learner might get stuck in a local minimum
- One strategy for avoiding local minima: repeat the search many times with random restarts

  ➔ bagging

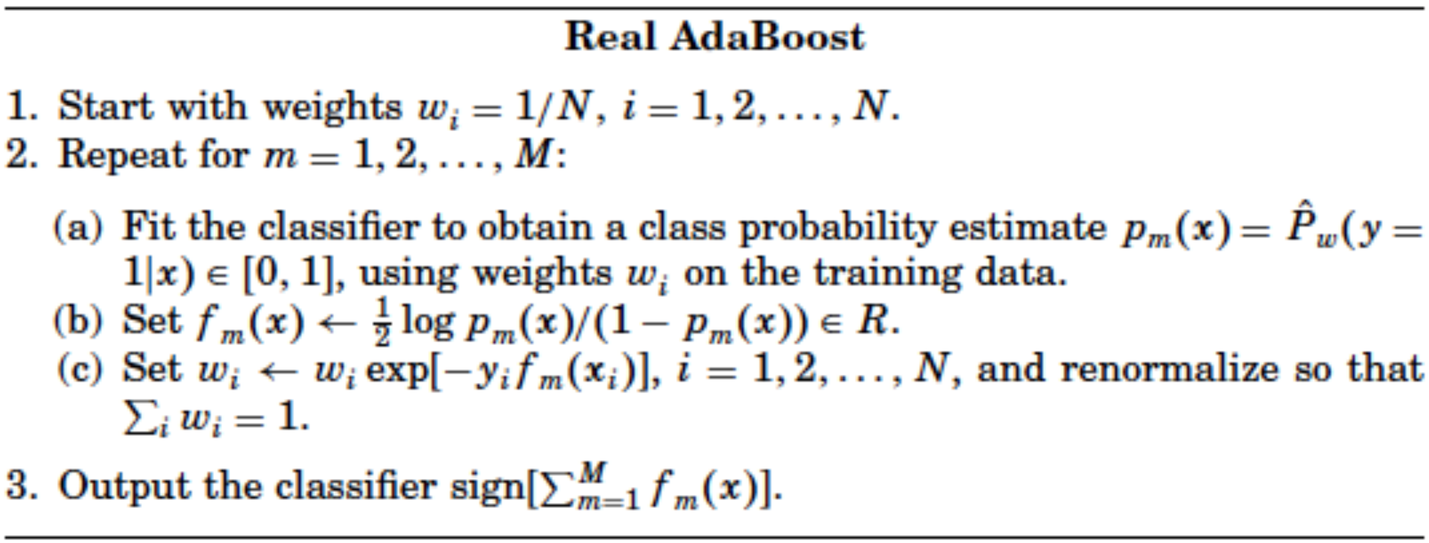


Boosting - Summary

- Basic motivation – creating a committee of experts is typically more effective than trying to derive a single super-genius
- Boosting provides a simple and powerful method for turning weak learners into strong learners

Boosting - Variations

- The simple algorithm described here has been extended to:
  - classifiers that produce confidences associated with class predictions (e.g., posterior probabilities as opposed to class assignments)



Real AdaBoost

```
1. Start with weights $w_i = 1/N$, $i = 1, 2, \dots, N$.
2. Repeat for $m = 1, 2, \dots, M$:
    (a) Fit the classifier to obtain a class probability estimate $p_m(x) = \hat{P}_w(y = 1|x) \in [0, 1]$, using weights $w_i$ on the training data.
    (b) Set $f_m(x) \leftarrow \frac{1}{2} \log p_m(x)/(1 - p_m(x)) \in R$.
    (c) Set $w_i \leftarrow w_i \exp[-y_i f_m(x_i)]$, $i = 1, 2, \dots, N$, and renormalize so that $\sum_i w_i = 1$.
3. Output the classifier $\text{sign}[\sum_{m=1}^{M} f_m(x)]$.
```

Boosting - Summary

- Basic motivation – creating a committee of experts is typically more effective than trying to derive a single super-genius
- Boosting provides a simple and powerful method for turning weak learners into strong learners
- The simple algorithm described here has been extended to:
  - multi-class classification problems
  - classifiers that produce confidences associated with class predictions (e.g., posterior probabilities as opposed to class assignments)
  - Weak classifiers trained on subsets of attributes
  - Recent theoretical results have shown deep connections between boosting and maximizing margin of separation (similar to SVM)

Boosting - Variations

- The simple algorithm described here has been extended to:
  - Ensembles of multi-class classifiers
  - Ensemble classifiers trained on subsets of attributes
- Recent theoretical results have shown deep connections between boosting and maximizing margin of separation (similar to SVM)

Error correcting output codes (ECOC)

- So far, we've been building the ensemble by tweaking the distribution of of training instances
- ECOC involves tweaking the output (class) to be learned

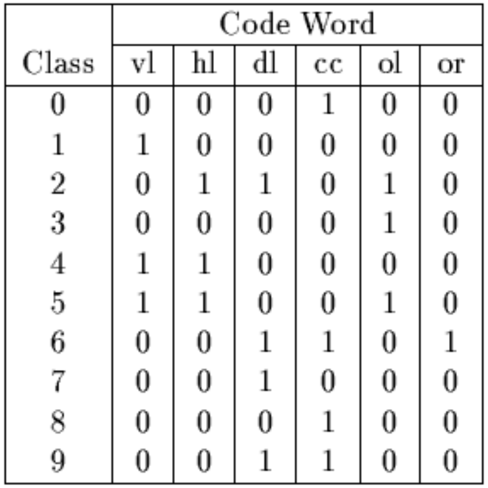
Example: Handwritten number recognition

→ 7, 4, 3, 5, 2

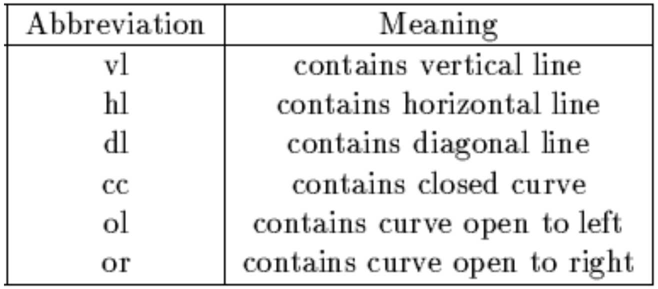
“obvious” approach: learn function: Scribble → {0,1,2,…,9}

→ doesn’t work very well (too hard!)

What if we “decompose” the learning task into six “subproblems”?



Class	Code Word					
	vl	hl	dl	cc	ol	or
0	0	0	0	1	0	0
1	1	0	0	0	0	0
2	0	1	1	0	1	0
3	0	0	0	0	1	0
4	1	1	0	0	0	0
5	1	1	0	0	1	0
6	0	0	1	1	0	1
7	0	0	1	0	0	0
8	0	0	0	1	0	0
9	0	0	1	1	0	0



Abbreviation	Meaning
vl	contains vertical line
hl	contains horizontal line
dl	contains diagonal line
cc	contains closed curve
ol	contains curve open to left
or	contains curve open to right

1. learn an ensemble of classifiers, one specialized to each of the 6 “sub-problems”
2. to classify a new scribble, invoke each ensemble member. then predict the class whose code-word is closest (Hamming distance) to the predicted code

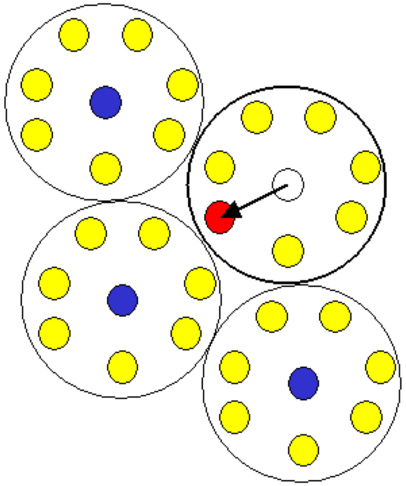
Error-correcting codes

Suppose we want to send *n*-bit messages through a noisy channel.

To ensure robustness to noise, we can map each *n*-bit message into an *m*-bit code (*m*>*n*) – note |codes| >> |messages|

When receive a code, translate it to message corresponding to the "nearest" (Hamming distance) code

Key to robustness: assign the codes so that each *n*-bit "clean" message is surrounded by a "buffer zone" of similar *m*-bit codes to which no other *n*-bit message is mapped.



The corrupted word still lies in its original unit sphere. The center of this sphere is the corrected word.

blue = message (*n* bits)
yellow = code (*m* bits)

white = intended message
red = received code

ISBN

- The International Standard Book Number (ISBN) system identifies every book with a ten-digit number, such as 0-226-53420-0.
- The first nine digits are the actual number but the tenth is added according to a mathematical formula based on the first nine.
- If a single one of the digits is changed, as in a misprint when ordering a book, a simple check verifies that something is wrong.

Designing code-words for ECOC learning

Coding: *k* labels → *m* bit codewords

Good coding:

- row separation: want "assigned" codes to be well-separated by lots of "unassigned" codes
- column separation: each bit *i* of the codes should be uncorrelated with all other bits *j*

Selecting good codes is hard!

class	1	2	3	4	5	6	7	8
Monday	0	0	1	0	0	0	1	0
Tuesday	0	0	1	1	1	0	0	1
Wednesday	0	0	1	0	0	0	1	0
Thursday	0	0	0	1	0	1	1	0
Friday	0	1	1	1	1	0	0	0
Saturday	1	1	1	1	0	0	0	1
Sunday	1	1	1	1	0	0	1	1

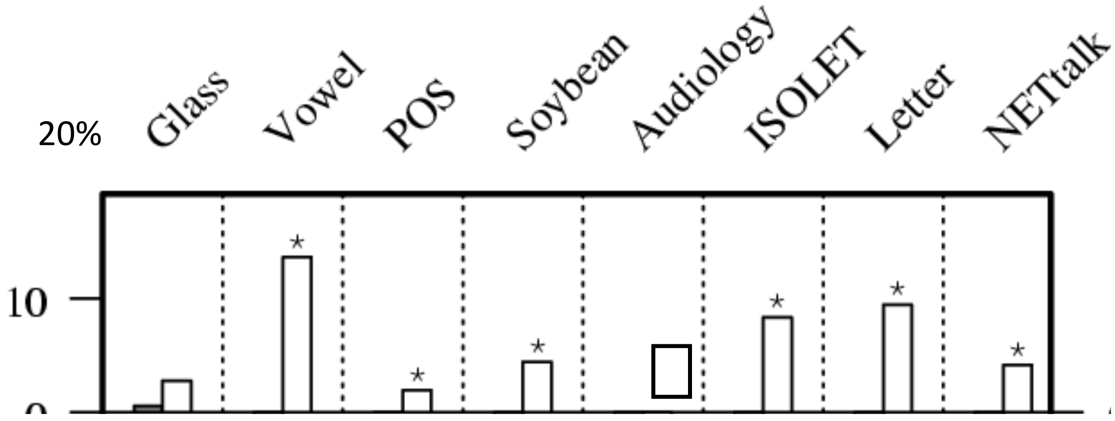
Bad codes

class	1	2	3	4	5	6	7	8
Monday	0	0	1	0	0	0	1	0
Tuesday	0	0	1	1	1	0	0	1
Wednesday	0	0	1	0	0	0	1	0
Thursday	0	0	0	1	0	1	1	0
Friday	0	1	1	1	1	0	0	0
Saturday	1	1	1	1	0	0	0	1
Sunday	1	1	1	1	0	0	1	1

correlated
rows ➔ bad

correlated
columns ➔ bad

Performance of ECOC



% decrease in error of ECOC over an ID3-like learning algorithm

Summary…

- Ensembles: basic motivation – creating a committee of experts is typically more effective than trying to derive a single super-genius
- Key issues:
  - Generation of base models
  - Integration of base models
- Popular ensemble techniques
  - manipulate training data: bagging and boosting (ensemble of "experts", each specializing on different portions of the instance space)
  - Manipulate input feature space
  - manipulate output values: error-correcting output coding (ensemble of "experts", each predicting 1 bit of the multibit full class label)

Learning under helpful distributions

- PAC Learning requires success under all possible probability distributions
- Some concept classes are hard to learn under all distributions – e.g., regular languages or deterministic finite state automata (DFA), yet they are readily learned by humans
- Question – can natural settings be modeled by more benign or helpful distributions? E.g., can DFA be learned under helpful distributions?
- What precisely are helpful distributions?

Digression – Kolmogorov Complexity

- Kolmogorov complexity *K* (*x*) is a machine independent i.e. universal measure of the complexity of description of an object
- *K* (*x*) = the number of bits in the *shortest* universal Turing machine program for *x*

- Object – 01010101010101...0101010101010101 = $(01)^{500}$
- Program – Print "01" 500 times
- Object – 11001101011111... 100101110111 (random string)
- Program –Print "11001101 ... 01110111"
- Simple objects have low Kolmogorov complexity

Universal distribution

We fix a universal Turing machine U

$$K(\alpha) = \min_{\pi} \left\{ length(\pi) \mid U(\pi) = \alpha \right\}$$

$$K(\alpha \mid \beta) = \min_{\pi} \{ length(\pi) \mid U(\pi, \beta) = \alpha \}$$

$$K(\alpha \mid \beta) \leq K(\alpha)$$

Universal distribution M assigns higher probabilities to simpler objects

$$M(x) \propto 2^{-K(x)}$$

$$M(x \mid \alpha) \propto 2^{-K(x \mid \alpha)}$$

Learning Under Universal distribution

Universal distribution M multiplicatively dominates all Enumerable distributions including finite precision Poisson, Gaussian, and many other distributions

Theorem: A concept class is Probably approximately Learnable under each enumerable distribution iff it is Probably approximately learnable under the universal distribution assuming during learning examples are drawn according to $M\ (x)$

Learning under the universal distribution

- Li and Vitanyi (1991) showed that log *n* -term *DNF* are learnable under *M* (*x* | *c*) where *c* is the target concept
- Parekh and Honavar (1999, 2001) showed that
    - Simple DFA (with encoding of size O (log *N*) where *N* is the number of states) are efficiently learnable under the universal distribution *M* (*x*)
    - DFA are efficiently learnable with a helpful teacher – examples are drawn according to *M* (*x* | *c*) where *c* is the target concept
- Denis (2001) showed that DFA are efficiently learnable from positive examples alone under *M* (*x* | *c*)
- Tu and Honavar (2012) showed the benefits of ordering examples according to M(x|c)

Additional Possibilities

- PAC learning model assumes that target concepts are selected uniformly at random from C
- Benign teacher – How about if target concepts are selected according to universal distribution over the concept class, namely $M(c)$?
- Occam Learner – Impose a preference bias over the set of consistent hypotheses – Select hypothesis h according to $M(h)$
- Bayesian learner – Assume priors given by $M(h)$

Summary of Distribution-Independent Learning Theory

- PAC-Easy learning problems lend themselves to a variety of efficient algorithms.
- PAC-Hard learning problems can often be made PAC-easy through appropriate instance transformation and choice of hypothesis space
- Occam's razor often helps
- Weak learning algorithms can often be used for PAC learning through accuracy and confidence boosting
- Learning under restricted classes of instance distributions (e.g., universal distribution) offers new possibilities