LECTURE NOTES–UNIT V

UNCERTAIN KNOWLEDGE & LEARNING

ECS302

ARTIFICIAL INTELLIGENCE

3/4 B.Tech [B3, B6]

R. PAVANI
DEPARTMENT OF CSE,GIT,GU

Module V Lecture Notes [*Uncertainty & Learning*]

Syllabus

Uncertain Knowledge: *Uncertainty: Acting under uncertainty, basic probability notation, the axioms of Probability, Inference using full joint distributions, independence, Baye's rule and its use, the wumpus world revisited.* ***Learning:*** *Learning from Observations: Forms of learning, Inductive learning, learning decision trees, ensemble learning. Why Learning Works: Computational learning theory.*

1. *Learning from observations*

Learning is a process that improves the knowledge of an AI program by making observations about its environment. It makes use of Percept's, not only for acting, but also for improving the agent's ability to act in the future.

1.1 *Forms of Learning:*

Learning agent contains a **performance element** that decides what actions to take and a **Learning element** that modifies the performance element so that it makes better decisions.

The design of a learning element is affected by three major issues:

i. *Which components of the performance element are to be learned?*

ii. *What feedback is available to learn these components?*

iii. *What representation is used for the components?*

Components of these agents include the following;

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves and about the results of possible actions the agent can take.

4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions.
6. Goals that describe classes of states whose achievement maximizes the agent's utility

Feedback:

- Each of the components can be learned from appropriate feedback.
- The type of feedback available for learning determines the nature of the learning problem that the agent faces.
- The field of machine learning usually distinguishes three cases: supervised, unsupervised, and reinforcement learning.

Unsupervised Learning:

It involves learning patterns in the input when no specific output values are supplied. For example, a taxi agent might gradually develop a concept of "good traffic days" and "bad traffic days" without ever being given labeled examples of each. A purely unsupervised learning agent cannot learn what to do, because it has no information as to what constitutes a correct action or a desirable state.

Supervised Learning:

It involves learning a function from examples of its inputs and outputs. Cases (I), (2), and (3) are all instances of supervised learning problems.

In (I), the agent learns condition-action rule for braking-this is a function from states to a Boolean output (to brake or not to brake),

In (2), the agent learns a function from images to a Boolean output (whether the image contains a bus).

In *(3),* the theory of braking is a function from states and braking actions to, say, stopping distance in feet. Notice that in cases (1) and (2), a teacher provided the correct output value of the examples;

in the third, the output value was available directly from the agent's percepts. For fully observable environments, it will always be the case that an agent can observe the effects of its actions and hence can use supervised learning methods to learn to predict them. For partially observable environments, the problem is more difficult, because the immediate effects might be invisible.

Reinforcement learning:

Rather than being told what to do by a teacher, a reinforcement learning agent must learn from reinforcement.

- The representation of learned information plays a role in determining how the learning algorithm must work. Any of the components of an agent can be represented using any of the representation schemes like linear weighted polynomials, propositional and first-order logical sentences, probabilistic descriptions etc.,
- Availability of prior knowledge also plays major role in design of learning system.

1.2 Inductive Learning:

Consider an example pair (x, f (x)), where x is the input and f(x) is the output of the function applied to x. The task of pure inductive inference is "Given a collection of examples of f, return a function h that approximates f". The function h is called a **hypothesis.**

- Inductive learning involves finding a consistent hypothesis that agrees with the examples.
- Figure: 1.2.1 shows a familiar example: fitting a function of a single variable to some data points. The examples are *(x,* f *(x))* pairs, where both x and f (x) are real numbers. We choose the **hypothesis space** H-the set of hypotheses we will consider-to be the set of polynomials of degree at most ***k.***
- Figure (a) shows some data with an exact fit by a straight line (a polynomial of degree 1). The line is called a **consistent** hypothesis because it agrees with all the data. Figure (b) shows a high-degree polynomial that is also consistent with the same data. This illustrates

the first issue in inductive learning: *how do we choose from among multiple consistent hypotheses?*

- Ockham's razor suggests the *simplest* hypothesis consistent with the data.
- Figure (c) shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial (with 7 parameters) for an exact fit. There are just 7 data points, so the polynomial has as many parameters as there are data points: thus, it does not seem to be finding any pattern in the data and we do not expect it to generalize well. It might be better to fit a simple straight line that is not exactly consistent but might make reasonable predictions.
- Figure (d) shows that the data in (c) can be fit exactly by a simple function of the form ax + *b* + *c* sin x. This example shows the importance of the choice of hypothesis space. A hypothesis space consisting of polynomials of finite degree cannot represent sinusoidal functions accurately, so a learner using that hypothesis space will not be able to learn from sinusoidal data.
- A learning problem is realizable if the hypothesis space contains the true function; otherwise, it is unrealizable.

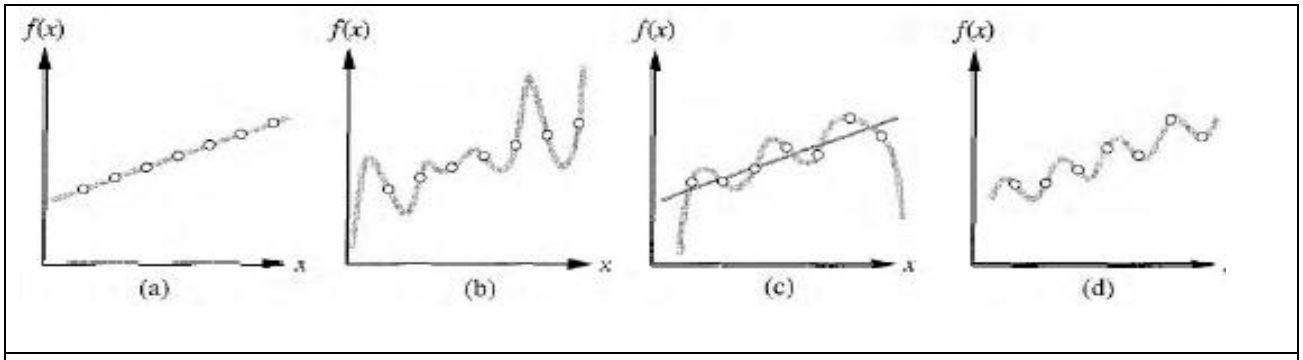


Figure 1.2.1 *(a) Example (x, f (x)) pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set that admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.*

1.3 Learning Decision Trees

A Decision tree takes as input an object or situation described by a set of attributes and returns a "decision" -the predicted output value for the input.

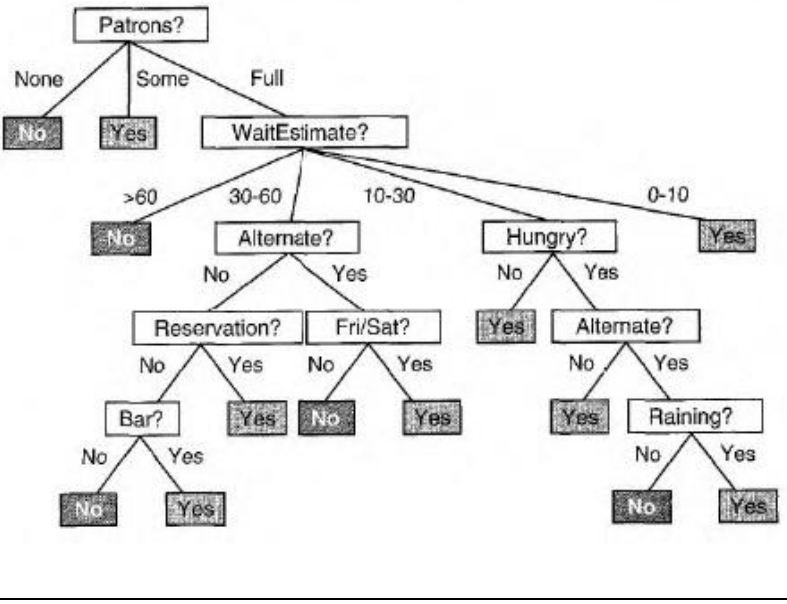
- The input attributes and the output values can be discrete or continuous.
- Learning discrete-valued function is called *"classification"* whereas continuous-valued functions are called *"regression".*
- A decision tree reaches its decision by performing a sequence of tests.
- Each internal node in the tree corresponds to a test of the value of one of the properties.
- The branches from the node are labelled with the possible values of the test.
- Each leaf node in the tree specifies the value to be returned if that leaf is reached.

Consider a list of Attributes: [*Example-To wait for a table at restaurant?]*

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. *Bar*: whether the restaurant has a comfortable bar area to wait in.
3. *Fri/Sat*: true on Fridays and Saturdays.
4. *Hungry:* whether we are hungry.
5. Patrons: how many people are in the restaurant (values are None, Some, and Full).
6. Price: the restaurant's price range ($, $$, $$$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (French, Italian, Thai, or burger).
10. *WaitEstimate*: the wait estimated by the host (0–10 minutes, 10–30, 30–60, >60).

Decision tree:

Here, Attributes are processed by the tree starting at the root and following the appropriate branch until a leaf is reached. For instance, an example with Patrons = Full and Wait Estimate = 0-10 will be classified as positive (i.e., yes, we will wait for a table).



A decision tree for deciding whether to wait for a table.

Expressiveness of decision trees:

Any particular decision tree hypothesis for the Will Wait goal predicate can be seen as an assertion of the form

$$\forall s \;\; WillWait(s) \Leftrightarrow (P_1(s) \vee P_2(s) \vee \cdots \vee P_n(s)) \;,$$

Where each condition Pi(s) is a conjunction of tests corresponding to a path from the root of the tree to a leaf with a positive outcome.

Decision trees can express any function of the input attributes. For Boolean functions, truth table row gives path to leaf.

If the function is the **parity function,** which returns 1 if and only if an even number of inputs are 1, then an exponentially large decision tree will be needed. It is also difficult to use a decision tree to represent a **majority function,** which returns 1 if more than half of its inputs are 1.

The truth table has $2n$ rows, because each input case is described by n attributes. We can consider the "answer" column of the table as a 2n-bit number that defines the function.

Inducing Decision trees for examples:

An example for a Boolean decision tree consists of a vector of' input attributes, X, and a single Boolean output value y. A set of examples (X1, yl) . . . , (XI2, y12) is shown in following Figure.

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–40*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

Examples for the restaurant domain.

- The positive examples are the ones in which the goal Will Wait is true (X1, X3, . . .); the negative examples are the ones in which it is false (X2, X5,...)

- The complete set of examples is called the **"training set".**

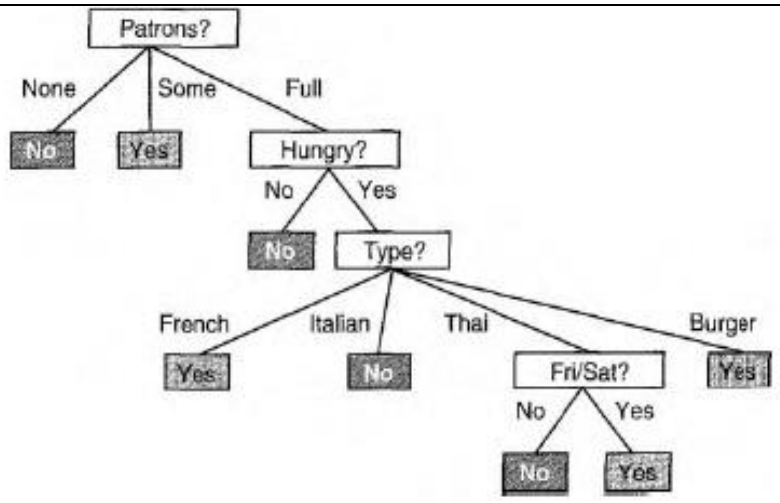
- Construct a decision tree that has one path to a leaf for each example, where the path tests each attribute in turn and follows the value for the example and the leaf has the classification of the example. When given the same example again, the decision tree will come up with the right classification.

Algorithm:

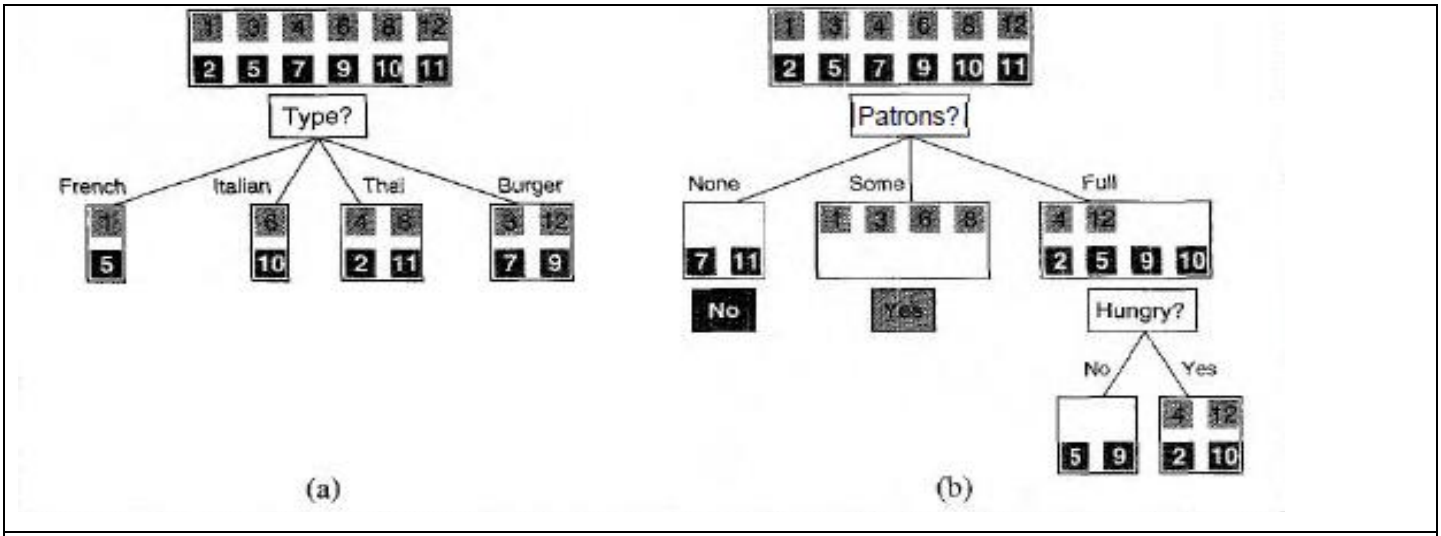
```
function DECISION-TREE-LEARNING(examples, attribs, default) returns a decision tree
  inputs: examples, set of examples
          attrzbs, set of attributes
          default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attrzbs is empty then return MAJORITY-VALUE(examples)
  else
      best ← CHOOSE-ATTRIBUTE(attribs, examples)
      tree ← a new decision tree with root test best
      m ← MAJORITY-VALUE(examples)
      for each value v_i of best do
          examples_i ← {elements of examples with best = v_i}
          subtree ← DECISION-TREE-LEARNING(examples_i, attribs – best, m)
          add a branch to tree with label v_i and subtree subtree
      return tree
```

The Decision tree Learning Algorithm



Decision tree induced from 12-example training set



Splitting the examples by testing on attributes. (a) Splitting on Type brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on Patrons does a good job of separating positive and negative examples? After splitting on patrons, Hungry is a fairly good second test.

There are four cases to consider for these recursive problems:

1. If there are some positive and some negative examples, then choose the best attribute to split them. Figure (b) shows Hungry being used to split the remaining examples.

2. If all the remaining examples are positive (or all. negative), then we are done: we can answer *Yes* or No. Figure (b) shows examples of this in the none and some cases.

3. If there are no examples left, it means that no such example has been observed, and we return a default value calculated from the majority classification at the node's parent.

4. If there are no attributes left, but both positive and negative examples, we have a problem. It means that these examples have exactly the same description, but different classifications. This happens when some of the data are incorrect; we say there is **noise** in the data. It also happens either when the attributes do not give enough information to describe the situation fully, or when the domain is truly nondeterministic. One simple way out of the problem is to use a majority vote.

Choosing Attribute Sets:

- One suitable measure is the expected amount of **information provided** by the attribute.
- It can think of as giving answer to question. The amount of information contained in the answer depends on one's prior knowledge.
- If the possible answers *vi have probabilities P(vi),* then the information content I of the actual answer is given b.

$$I(P(v_1), \ldots, P(v_n)) = \sum_{i=1} -P(v_i) \log_2 P(v_i)$$

- Suppose the training set contains p positive examples and n negative examples. Then an estimate of the information contained in a correct answer is

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n} .$$

- After testing attribute A, we will need

$$Remainder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

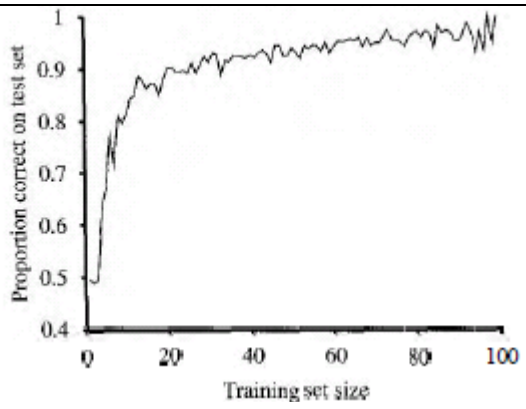
- The **information gain** from the attribute test is the difference between the original information requirement and the new requirement

$$Gain(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - Remainder(A)$$

Assessing the performance of the learning algorithm:

- A learning algorithm is good if it produces hypotheses that do a good job of predicting the classifications of unseen examples.
- If we train on all our available examples, then we will have to go out and get some more to test on
- Collect a large set of examples.
- Divide it into two disjoint sets: the **training set and the test set.**
- **Apply the learning algorithm to the training set, generating a hypothesis h.**
- Measure the percentage of examples in the test set that are correctly classified by h.

- Repeat steps 2 to 4 for different sizes of training sets and different randomly selected training sets of each size.



A learning curve for the decision tree algorithm on 100 randomly generated examples in the restaurant domain. The graph summarizes 20 trials.

Noise and over fitting:

Whenever there is a large set of possible hypotheses, one has to be careful not to use the resulting freedom to find meaningless "regularity" in the data. This problem is called ***over fitting***. A very general phenomenon, over fitting occurs even when the target function is not at all random. It afflicts every kind of learning algorithm, not just decision trees.

Decision tree pruning is here to deal with the problem. ***Pruning*** works by preventing recursive splitting on attributes that are not clearly relevant, even when the data at that node in the tree are not uniformly classified.

Applicability of decision trees:

-Missing data

-Multi valued attributes

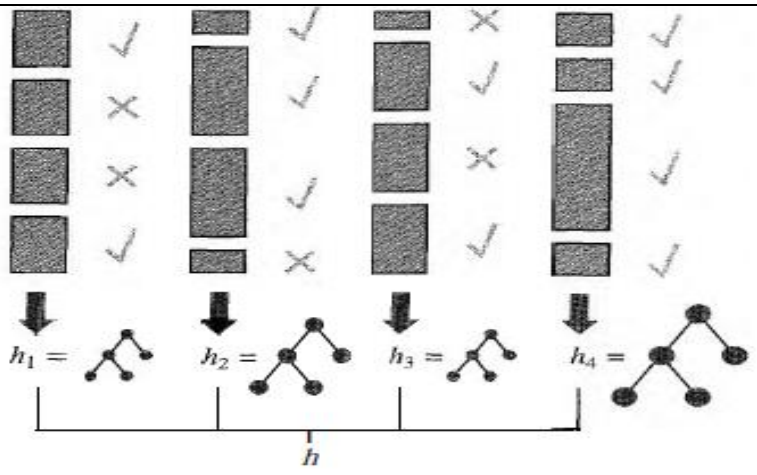
-Continuous and integer-valued input attributes

-Continuous-valued output attributes

1.4 Ensemble Learning:

The idea of ensemble learning methods is to select a whole collection, or ensemble, of hypotheses from the hypothesis space and combine their predictions. Using majority voting any example is classified using ensemble learning.

- The most widely used ensemble method is called **boosting.**
- Weighted training set is used where each example has given some weight which indicates its importance**.**
- Boosting starts with w = 1 for all the examples from this set, it generates the first hypothesis, *h1 that classifies examples correctly and incorrectly.*
- To generate next hypothesis better on the misclassified examples, so we increase their weights while decreasing the weights of the correctly classified examples



How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The ticks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.

Weak Learning:

There are many variants of the basic boosting idea with different ways of adjusting the weights and combining the hypotheses. One specific algorithm, called ADABOOST. ADABOOST does have a very important property: if the input learning algorithm L is a **weak learning** algorithm-

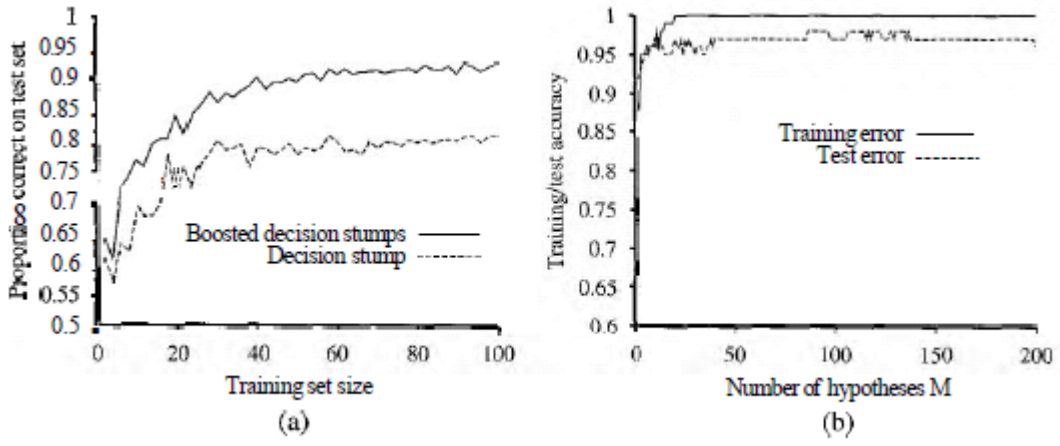
which means that L always returns a hypothesis with weighted error on the training set that is slightly better than random guessing.

```
function ADABOOST(examples, L, M) returns a weighted-majority hypothesis
  inputs: examples, set of N labelled examples (x1, y1), ..., (xN, yN)
          L, a learning algorithm
          M, the number of hypotheses in the ensemble
  local variables: w, a vector of N example weights, initially 1/N
                   h, a vector of M hypotheses
                   z, a vector of M hypothesis weights

  for m = 1 to M do
    h[m] ← L(examples, w)
    error ← 0
    for j = 1 to N do
      if h[m](xj) ≠ yj then error ← error + w[j]
    for j = 1 to N do
      if h[m](xj) = yj then w[j] ← w[j] . error/(1 − error)
    w ← NORMALIZE(w)
    z[m] ← log (1 − error)/ error
  return WEIGHTED-MAJORITY(~z)
```

Decision Stumps:

Let us see how well boosting does on the restaurant data. We will choose as our original hypothesis space the class of **decision stumps,** which are decision trees with just one test at the root. The lower curve in Below Figure (a) shows that un boosted decision stumps are not very effective for this data set, reaching a prediction performance of only 8 1 % on 100 training examples. When boosting is applied (with M = 5), the performance is better, reaching 93% after 100 examples. An interesting thing happens as the ensemble size M increases. Figure (b) shows the training set performance (on 100 examples) as a function of M. Notice that the error reaches zero (as the boosting theorem tells us) when M is 20; that is, a weighted-majority combination of 20 decision stumps suffices to fit the 100 examples exactly. As more stumps are added to the ensemble, the error remains at zero. The graph also shows that *the test set performance continues to increase long after the training set error has reached zero.* At M = 20, the test performance is 0.95 (or 0.05 error), and the performance increases to 0.98 as late as M = 137, before gradually dropping to 0.95.



(a) Graph showing the performance of boosted decision stumps with M = 5 versus decision stumps on the restaurant data. (b) The proposition correct on the training set and the test set as a function of M, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training, accuracy reaches 1, i.e., after the ensemble fits the data exactly.

1.5 Computational Learning Theory:

The underlying principle is the following: *any hypothesis that is seriously wrong will almost certainly be "found out" with high probability after a small number of examples, because it will make an incorrect prediction. Thus, any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong: that is, it must be probably approximately correct.* Any learning algorithm that returns hypotheses that are probably approximately correct is called a "**PAC-learning"** algorithm.

Stationarity:

There are some subtleties in the preceding argument. The main question is the connection between the training and the test examples; after all, we want the hypothesis to be approximately correct on the test set, not just on the training set. The key assumption is that the training and test sets are drawn randomly and independently from the same population of examples with the *same probability distribution.* This is called the **stationarity** assumption.

How many examples are needed?

In order to put these insights into practice, we will need some notation:

Let X be the set of all possible examples.

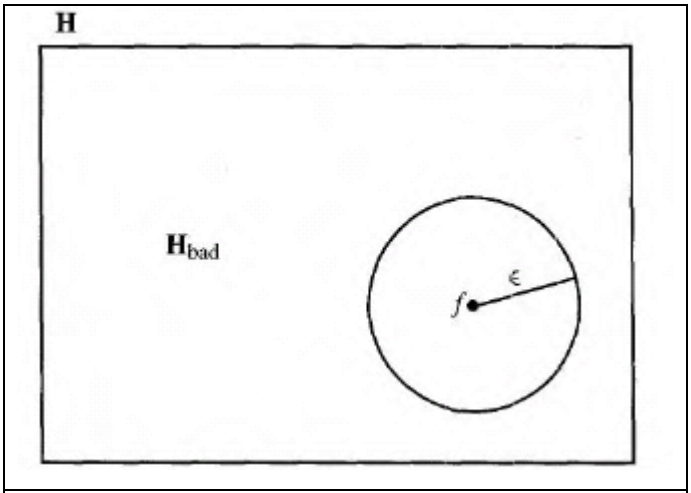
Let D be the distribution from which examples are drawn.

Let H be the set of possible hypotheses.

Let *N* be the number of examples in the training set.

Initially, we will assume that the true function *f* is a member of H. Now we can define the **error** of a hypothesis h with respect to the true function *f* given a distribution *L)* over the examples as the probability that h is different from *f* on an example: error(h) = P(h(x) # *f* (x) (*x* drawn from D) .

A hypothesis h is called **approximately correct** if error(h) <= **€,** where € is a small constant. The plan of attack is to show that after seeing N examples, with high probability, all consistent hypotheses will be approximately correct. One can think of an approximately correct hypothesis as being "close" to the true function in hypothesis space: it lies inside what is called the **€-ball** around the true function *f* .



Schematic diagram of hypothesis space, showing the "€ ball" around the true function f.

We can calculate the probability that a "seriously wrong" hypothesis *hb* € Hbad is consistent with the first N examples as follows. We know that error (hb) > **€.** Thus, the probability that it agrees with a given example is at least 1 - **€.** The bound for examples is

$$P(h_b \text{ agrees with N examples}) \leq (1-\epsilon)^N$$

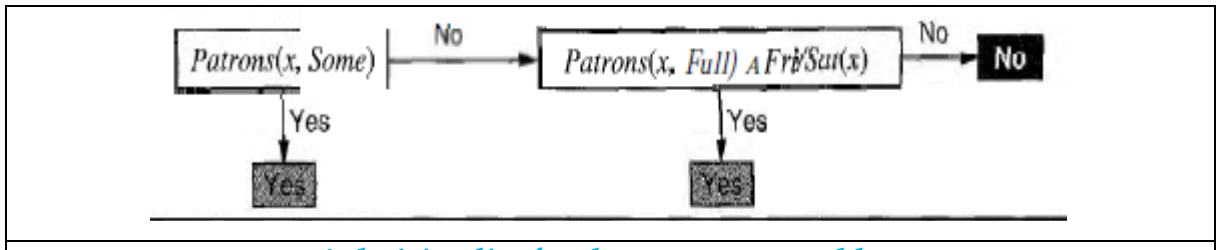
The probability that Hbad contains at least one consistent hypothesis is bounded by the sum of the individual probabilities:

$$P(\mathbf{H}_{\text{bad}} \text{ contains a consistent hypothesis}) \leq |\mathbf{H}_{\text{bad}}|(1-\epsilon)^N \leq |\mathbf{H}|(1-\epsilon)^N$$

Learning decision lists:

A **decision list** is a logical expression of a restricted form. It consists of a series of tests, each of which is a conjunction of literals. If a test succeeds when applied to an example description, the decision list specifies the value to be returned. If the test fails, processing continues with the next test in the list. Below Figure shows a decision list that represents the following hypothesis:

$$\forall x \;\; WillWait(x) \Leftrightarrow Patrons(x, Some) \vee (Patrons(x, Full) \wedge Fri/Sat(x))$$



A decision list for the restaurant problem

Algorithm:

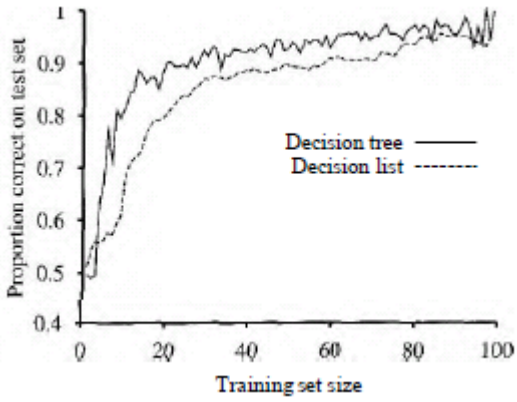
The next task is to find an efficient algorithm that returns a consistent decision list. We will use a greedy algorithm called DECISION-LIST-LEARNING that repeatedly finds a test that agrees exactly with some subset of the training set. Once it finds such a test, it adds it to the decision list under construction and removes the corresponding examples. It then constructs the remainder of the decision list, using just the remaining examples. This is repeated until there are no examples left.

```
function DECISION-LIST-LEARNING(examples) returns a decision list, or failure

    if examples is empty then return the trivial decision list No
    t ← a test that matches a nonempty subset examples_t of examples
        such that the members of examples_t are all positive or all negative
    if there is no such t then return failure
    if the examples in examples_t are positive then o ← Yes else o ← No
    return a decision list with initial test t and outcome o and remaining tests given by
        DECISION-LIST-LEARNING(examples − examples_t)
```

An algorithm for learning decision lists

It is reasonable to prefer small tests that match large sets of uniformly classified examples, so that the overall decision list will be as compact as possible. The simplest strategy is to find the smallest test *t* that matches any uniformly classified subset, regardless of the size of the subset. It is shown in below graph;



Learning curve for decision-list-learning algorithm on the restaurant data. The curve for decision-tree-learning is shown for comparison.