# Module 5

Learning from examples

# Introduction

- An agent is **learning** if it improves its performance on future tasks after making observations about the world.
- Why would we want an agent to learn? There are three main reasons.
- First, the designers cannot anticipate all possible situations that the agent might find itself in.
- Second, the designers cannot anticipate all changes over time; a program designed to predict tomorrow's stock market prices must learn to adapt when conditions change from boom to bust.
- Third, sometimes human programmers have no idea how to program a solution themselves.

# FORMS OF LEARNING

- Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:
  - Which *component* is to be improved
  - What *prior knowledge* the agent already has.
  - What *representation* is used for the data and the component.
  - What *feedback* is available to learn from.

**Components to be learned**

The components of these agents include:

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.

**Representation and prior knowledge**

- several examples of representations for agent components: propositional and first-order logical sentences for the components in a logical agent; Bayesian networks for the inferential components of a decision-theoretic agent, and so on.
- Effective learning algorithms have been devised for all of these representations.
- Learning a (possibly incorrect) general function or rule from specific input–output pairs is called **inductive learning**.

**Feedback to learn from**

- There are three *types of feedback* that determine the three main types of learning:
- In **unsupervised learning** the agent learns patterns in the input even though no explicit feedback is supplied. The most common unsupervised learning task is **clustering**: detecting potentially useful clusters of input examples. 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 by a teacher

- In **reinforcement learning** the agent learns from a series of reinforcements—rewards or punishments. For example, the lack of a tip at the end of the journey gives the taxi agent an indication that it did something wrong.
- In **supervised learning** the agent observes some example input–output pairs and learns a function that maps from input to output. In component 1 above, the inputs are percepts and the output are provided by a teacher who says “Brake!” or “Turn left.”
- In **semi-supervised learning** we are given a few labeled examples and must make what we can of a large collection of unlabeled examples.

# SUPERVISED LEARNING

- The task of supervised learning is this:
- Given a **training set** of N example input–output pairs

  (x1, y1), (x2, y2), . . . (xN, yN) ,
- where each yj was generated by an unknown function y = f(x),discover a function h that approximates the true function f.
- Here x and y can be any value; they need not be numbers. The function h is a **hypothesis**.
- Learning is a search through the space of possible hypotheses for one that will perform well,even on new examples beyond the training set.
- To measure the accuracy of a hypothesis we give it a **test set** of examples that are distinct from the training set.
- a hypothesis **generalizes** well if it correctly predicts the value of y for novel examples.

- When the output y is one of a finite set of values (such as *sunny, cloudy* or *rainy*), the learning problem is called **classification**, and is called Boolean or binary classification
- if there are only two values. When y is a number (such as tomorrow's temperature), the learning problem is called **regression**

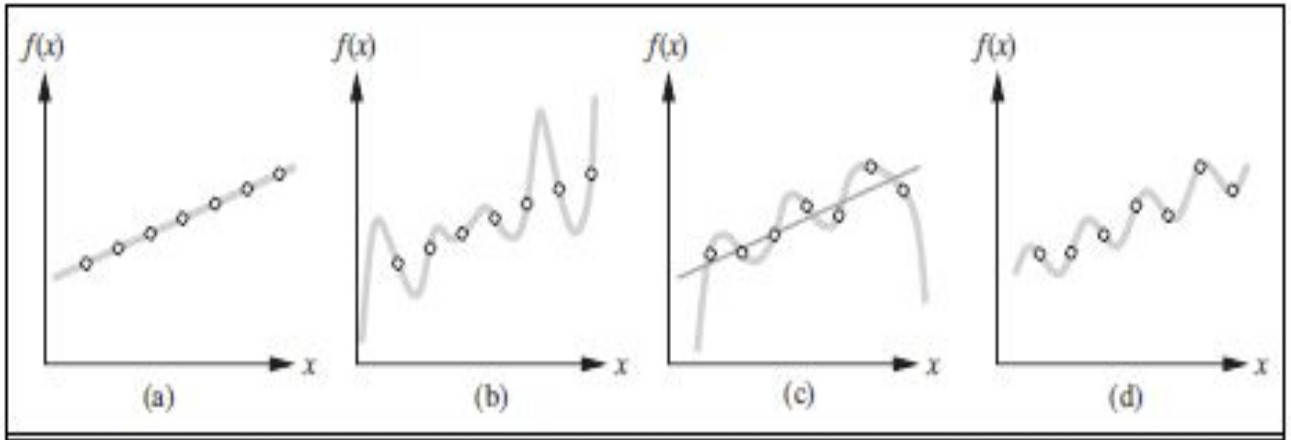


**Figure 18.1** (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

- Figure 18.1(a) shows some data with an exact fit by a straight line (the polynomial 0.4x + 3). The line is called a **consistent** hypothesis because it agrees with all the data.
- Figure 18.1(b) shows a highdegree polynomial that is also consistent with the same data.
- *how do we choose from among multiple consistent hypotheses?*
- One answer is to prefer the *simplest* hypothesis consistent with the data. This principle is called **Ockham's razor**,
- Figure 18.1(c) shows a second data set. There is no consistent straight line for this data set;
- *In general, there is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better.*
- a learning problem is **realizable** if the hypothesis space contains the true function. Unfortunately, we cannot always tell whether a given learning problem is realizable, because the true function is not known.

- Supervised learning can be done by choosing the hypothesis $h*$ that is most probable given the data:
- $h* = \underset{h \in H}{\text{argmax}}\, P(h|data)$
- By Bayes' rule this is equivalent to
- $h* = \underset{h \in H}{\text{argmax}}\, P(data|h)\, P(h)$ .
- Then we can say that the prior probability P(h) is high for a degree-1 or -2 polynomial, lower for a degree-7 polynomial,
- *There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding a good hypothesis within that space*

# LEARNING DECISION TREES

- Decision tree induction is one of the simplest and yet most successful forms of machine learning.

**The decision tree representation**

- A **decision tree** represents a function that takes as input a vector of attribute values and returns a "decision"—a single output value.
- The input and output values can be discrete or continuous. For now we will concentrate on problems where the inputs have discrete values and the output has exactly two possible values; this is Boolean classification, where each example input will be classified as true (a **positive** example) or false (a **negative** example).

- As an example, we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** WillWait .

the attributes that we will consider as part of the input:

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, or >60).

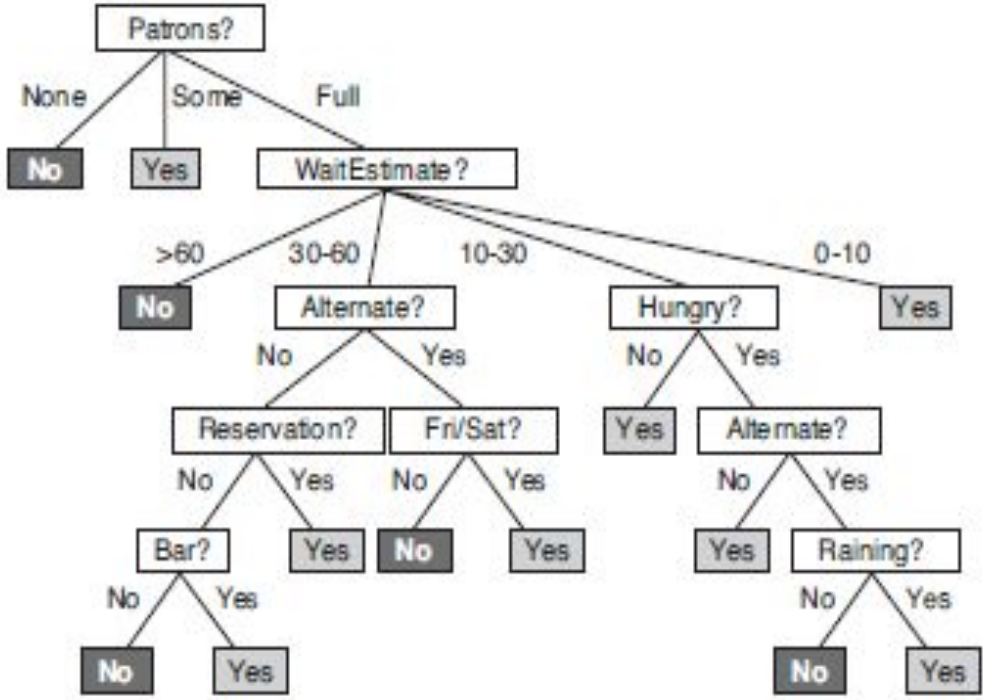


**Figure 18.2** A decision tree for deciding whether to wait for a table.

## Expressiveness of decision trees

- A Boolean decision tree is logically equivalent to the assertion that the goal attribute is true

if and only if the input attributes satisfy one of the paths leading to a leaf with value true.

- Writing this out in propositional logic, we have

  $\text{Goal} \Leftrightarrow (\text{Path1} \vee \text{Path2} \vee \cdots)$ ,

- where each Path is a conjunction of attribute-value tests required to follow that path.
- As an example, the rightmost path in Figure 18.2 is

  $\text{Path} = (\text{Patrons} = \text{Full} \wedge \text{WaitEstimate} = 0\text{–}10)$ .

# Inducing decision trees from examples

- An example for a Boolean decision tree consists of an (**x**, y) pair, where **x** is a vector of values for the input attributes, and y is a single Boolean output value. A training set of 12 examples

| Example | Input Attributes | | | | | | | | | | Goal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Alt* | *Bar* | *Fri* | *Hun* | *Pat* | *Price* | *Rain* | *Res* | *Type* | *Est* | *WillWait* |
| $\mathbf{x}_1$ | *Yes* | *No* | *No* | *Yes* | *Some* | *$$$* | *No* | *Yes* | *French* | *0–10* | $y_1 = Yes$ |
| $\mathbf{x}_2$ | *Yes* | *No* | *No* | *Yes* | *Full* | *$* | *No* | *No* | *Thai* | *30–60* | $y_2 = No$ |
| $\mathbf{x}_3$ | *No* | *Yes* | *No* | *No* | *Some* | *$* | *No* | *No* | *Burger* | *0–10* | $y_3 = Yes$ |
| $\mathbf{x}_4$ | *Yes* | *No* | *Yes* | *Yes* | *Full* | *$* | *Yes* | *No* | *Thai* | *10–30* | $y_4 = Yes$ |
| $\mathbf{x}_5$ | *Yes* | *No* | *Yes* | *No* | *Full* | *$$$* | *No* | *Yes* | *French* | *>60* | $y_5 = No$ |
| $\mathbf{x}_6$ | *No* | *Yes* | *No* | *Yes* | *Some* | *$$* | *Yes* | *Yes* | *Italian* | *0–10* | $y_6 = Yes$ |
| $\mathbf{x}_7$ | *No* | *Yes* | *No* | *No* | *None* | *$* | *Yes* | *No* | *Burger* | *0–10* | $y_7 = No$ |
| $\mathbf{x}_8$ | *No* | *No* | *No* | *Yes* | *Some* | *$$* | *Yes* | *Yes* | *Thai* | *0–10* | $y_8 = Yes$ |
| $\mathbf{x}_9$ | *No* | *Yes* | *Yes* | *No* | *Full* | *$* | *Yes* | *No* | *Burger* | *>60* | $y_9 = No$ |
| $\mathbf{x}_{10}$ | *Yes* | *Yes* | *Yes* | *Yes* | *Full* | *$$$* | *No* | *Yes* | *Italian* | *10–30* | $y_{10} = No$ |
| $\mathbf{x}_{11}$ | *No* | *No* | *No* | *No* | *None* | *$* | *No* | *No* | *Thai* | *0–10* | $y_{11} = No$ |
| $\mathbf{x}_{12}$ | *Yes* | *Yes* | *Yes* | *Yes* | *Full* | *$* | *No* | *No* | *Burger* | *30–60* | $y_{12} = Yes$ |

**Figure 18.3** Examples for the restaurant domain.

- The positive examples are the ones in which the goal WillWait is true (**x**1, **x**3, . . .); the negative examples are the ones in which it is false (**x**2, **x**5, . . .).
- We want a tree that is consistent with the examples and is as small as possible.
- The DECISION-TREE-LEARNING algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first. This test divides the problem up into smaller subproblems that can then be solved recursively. By "most important attribute," we mean the one that makes the most difference to the classification of an example.

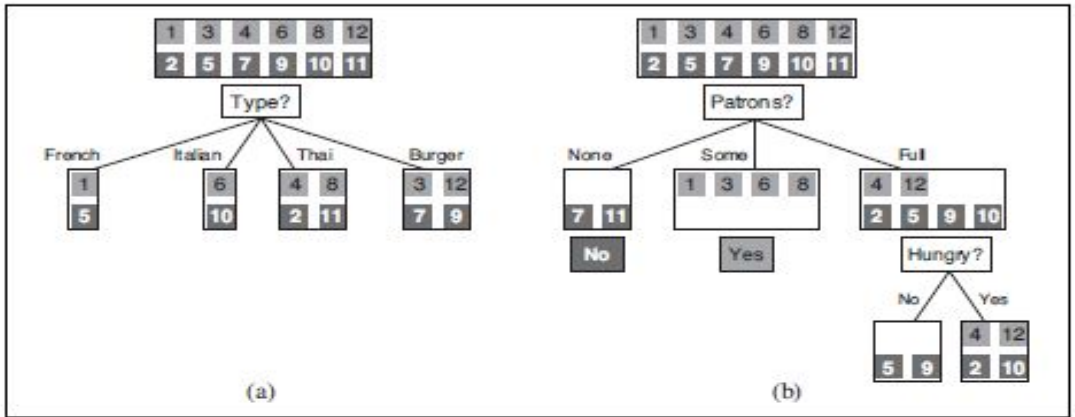


**Figure 18.4** Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job 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 the remaining examples are all positive (or all negative), then we are done: we can answer Yes or No. Figure 18.4(b) shows examples of this happening in the None and Some branches.

2. If there are some positive and some negative examples, then choose the best attribute to split them. Figure 18.4(b) shows Hungry being used to split the remaining examples.

3. If there are no examples left, it means that no example has been observed for this combination of attribute values, and we return a default value calculated from the plurality classification of all the examples that were used in constructing the node's parent. These are passed along in the variable parent examples.

4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can happen because there is an error or **noise** in the data; because the domain is nondeterministic;or because we can't observe an attribute that would distinguish the examples.

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns a tree

  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
      A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← a new decision tree with root test A
      for each value v_k of A do
          exs ← {e : e ∈ examples and e.A = v_k}
          subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
          add a branch to tree with label (A = v_k) and subtree subtree
      return tree
```

**Figure 18.5** The decision-tree learning algorithm. The function IMPORTANCE is described in Section 18.3.4. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

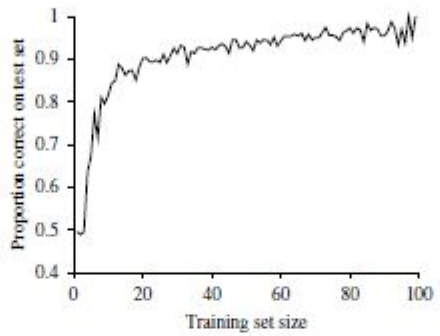


**Figure 18.7** A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

## Choosing attribute tests

- Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy.

positive. In general, the entropy of a random variable $V$ with values $v_k$, each with probability $P(v_k)$, is defined as

$$\text{Entropy:} \quad H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k) .$$

We can check that the entropy of a fair coin flip is indeed 1 bit:

$$H(Fair) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1 .$$

If the coin is loaded to give 99% heads, we get

$$H(Loaded) = -(0.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits.}$$

It will help to define $B(q)$ as the entropy of a Boolean random variable that is true with probability $q$:

$$B(q) = -(q \log_2 q + (1-q) \log_2 (1-q)) .$$

Thus, $H(Loaded) = B(0.99) \approx 0.08$. Now let's get back to decision tree learning. If a training set contains $p$ positive examples and $n$ negative examples, then the entropy of the goal attribute on the whole set is

$$H(Goal) = B\left(\frac{p}{p+n}\right) .$$

can measure exactly how much by looking at the entropy remaining after the attribute test.

An attribute $A$ with $d$ distinct values divides the training set $E$ into subsets $E_1, \ldots, E_d$. Each subset $E_k$ has $p_k$ positive examples and $n_k$ negative examples, so if we go along that branch, we will need an additional $B(p_k/(p_k + n_k))$ bits of information to answer the question. A randomly chosen example from the training set has the $k$th value for the attribute with probability $(p_k + n_k)/(p + n)$, so the expected entropy remaining after testing attribute $A$ is

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k+n_k}{p+n} B(\frac{p_k}{p_k+n_k}) .$$

INFORMATION GAIN

The **information gain** from the attribute test on $A$ is the expected reduction in entropy:

$$Gain(A) = B(\frac{p}{p+n}) - Remainder(A) .$$

In fact $Gain(A)$ is just what we need to implement the IMPORTANCE function. Returning to the attributes considered in Figure 18.4, we have

$$Gain(Patrons) = 1 - \left[\frac{2}{12}B(\frac{0}{2}) + \frac{4}{12}B(\frac{4}{4}) + \frac{6}{12}B(\frac{2}{6})\right] \approx 0.541 \text{ bits},$$

$$Gain(Type) = 1 - \left[\frac{2}{12}B(\frac{1}{2}) + \frac{2}{12}B(\frac{1}{2}) + \frac{4}{12}B(\frac{2}{4}) + \frac{4}{12}B(\frac{2}{4})\right] = 0 \text{ bits},$$

confirming our intuition that *Patrons* is a better attribute to split on. In fact, *Patrons* has the maximum gain of any of the attributes and would be chosen by the decision-tree learning algorithm as the root.

## Generalization and overfitting

- On some problems, the DECISION-TREE-LEARNING algorithm will generate a large tree when there is actually no pattern to be found. Consider the problem of trying to predict whether the roll of a die will come up as 6 or not.
- In Figure 18.1(b) and (c), we saw polynomial functions overfitting the data. Overfitting becomes more likely as the hypothesis space and the number of input attributes grows, and less likely as we increase the number of training examples.
- For decision trees, a technique called **decision tree pruning** combats overfitting. Pruning works by eliminating nodes that are not clearly relevant.
- the information gain is a good clue to irrelevance. how large a gain should we require in order to split on a particular attribute?
- We can answer this question by using a statistical **significance test**. Such a test begins by assuming that there is no underlying pattern (the so-called **null hypothesis**). Then the actual data are analyzed to calculate the extent to which they deviate from a perfect absence of pattern.

each subset, $p_k$ and $n_k$, with the expected numbers, $\hat{p}_k$ and $\hat{n}_k$, assuming true irrelevance:

$$\hat{p}_k = p \times \frac{p_k + n_k}{p + n} \qquad \hat{n}_k = n \times \frac{p_k + n_k}{p + n}.$$

A convenient measure of the total deviation is given by

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k}$$

**early stopping**—have the decision tree algorithm stop generating nodes when there is no good attribute to split on, rather than going to all the trouble of generating nodes and then pruning them away. The problem with early stopping is that it stops us from recognizing situations where there is no one good attribute, but there are combinations of attributes that are informative.

# REGRESSION AND CLASSIFICATION WITH LINEAR MODELS

- **Consider linear functions** of continuous-valued inputs.

**Univariate linear regression**