18 Learning from examples

Defining learning

In which we describe agents that can improve their behaviour through diligent study of own experiences

An agent is learning if it can improve its performance on future tasks, through making observations in the world.

It can trivial e.g jotting a phone number down, to profound as exhibited by albert einstein who inferred a new theory of the universe.

We are concentrating on one class of learning problem: from a collection of input-output pairs, learn a function that predicts the output for new inputs.

Why would we want an agent to learn? If the design of the agent can be improved why wouldn't it be designed with that improvement to begin with?

Designers cannot predict all situations an agent would find itself in.

E.g a maze solving robot must learn the layout of each maze it encounters.

Designers also cannot anticipate changes over time, e.g a program designed to predict tomorrows stock market prices must learn to adapt when conditions change from boom to bust.

Third, sometimes human programmers have no idea how to program the solution themselves.

E.g most people are good at recognizing the faces of family members, but programmers are unable to program a computer to accomplish that, without using a learning algorithm.

18.1 Forms of learning

Any component of an agent can be improved by learning form 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 has.*
- What *representation* is used for the data and the component.
- What *feedback* is available to learn from.

Components to be learned

Earlier in the textbook several agent designs were described.

1. A direct mapping from conditions on the current state to actions.
    a. This refers to an agent that simply reacts to its current state by executing predefined actions. It's a very simple, reactive strategy where if certain conditions are met, specific actions are taken.
2. A means to infer relevant properties of the world from the percept sequence
    a. Agents often gather percepts (information about the world) over time. This component is about the agent being able to process and understand the sequence of these percepts to infer what's important or relevant for decision-making. For example, an agent might recognize that a series of events leads to a certain situation and adapt its actions accordingly.
3. Information about the way the world evolves and about the results of possible actions the agent can take.
    a. This is about understanding how the world changes over time and how the agent's actions affect the world. An agent needs to know what will happen if it takes a particular action, which helps it make informed decisions. For example, if an agent knows that opening a door will lead to a new room, it can decide whether this is beneficial or not.
4. Utility information indicating the desirability of the world states.
    a. This refers to the agent having a way to evaluate different states of the world based on how desirable they are. In simpler terms, it's about understanding how "good" or "bad" a particular situation is. The agent uses this utility information to choose actions that will lead to the most desirable outcomes.
5. Action-value information indicating the desirability of the worlds states.
    a. Similar to utility information, but this is specifically about evaluating how good or bad an action is in terms of its consequences. For example, an agent may know that a certain action has a high likelihood of achieving a desirable outcome, so it assigns that action a higher value.
6. Goals that describe classes of states whose achievement maximises the agents utility.
    a. Goals are the states that an agent aims to achieve. These are often the "targets" or "desired outcomes" for the agent. The agent tries to maximize its utility by taking actions that bring it closer to its goals

Inductive learning definition

Learning a (possibly incorrect) general function or rule from specific input-output pairs is called **inductive learning.**

Deductive learning

Is starting with a general rule and using logic to figure out conclusions. E.g if the agent knows if it rains the ground will be wet, and someone tells the agent it will rain tomorrow they can deduce that the ground will be wet tomorrow.

These refer to how knowledge is generated.

Whereas feedback is about how the agent interacts with its environment, and inductive and deductive describe how they use that feedback to build knowledge.

Three types of feedback

Unsupervised learning

The agent will learn patterns from the input even though no specific feedback is supplied. A good example of a common task is clustering. This refers to detecting useful clusters of input examples. For example a taxi agent might gradually develop a concept of good traffic day and bad traffic days without being given labeled examples of these.

Reinforcement learning

This refers to the agent receiving rewards or punishments. E.g the lack of a tip at the end of a journey would tell the taxi agent that it did something wrong. Two points for a win at the end of a chess game tells them it did something right. Its up to the agent to decide which actions were responsible for it.

Supervised learning

The agent will observe input output pairs and learns a function that maps from input to output. In reference to component 1 (a direct mapping from current state to actions). If a taxi agents teacher shouts break. Then the agent now knows a correct condition to break in. Another example relating to component 2 (A means to infer relevant properties of the world from the percept sequence), if the inputs are camera images and the teacher says that is a bus the agent will begin to learn what a bus looks like. In component 3 the theory of breaking is a function from sates and breaking actions to stopping distance in feet. IN this case the oupt value is available from what the agent perceives.

Semi-supervised learning

In practice the distinctions may not be as clear. In this type of learning we are given some labeled examples. The examples may not be oracular truths. For example if you were trying to train an agent to guess ages from a photo the values may not always be accurate since people can lie about their age. It's not just that there is noise in the data, but they are systematic inaccuracies to uncover that this is an unsupervised learning problem.

18.2 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 a unknown function $y = f(x)$,

Discover a function h that approximates to 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 a new example not known in the training set.

We can measure the accuracy of a hypothesis using a **test set.**

We can say that a hypothesis **generalizes** well if it correctly predicts the value of y for novcel examples. Sometimes the function of f is stochastic meaning it is not strictly a function of x, and what has to be learnt is a conditional probability distribution, P(Y | x).

Generalization definition - we say that the model (hypothesis) has generalised well if it has come up with a good underlying rule to get the correct output y for the input x.

Stochastic definition - sometimes the function may have some randomness this means that instead of learning a deterministic function it will learn a **probability distribution,** which tells the probability of each possible output y given an x input.

If the output is of a finite set of values e.g [sunny,cloudy,rainy] and the agent has to pick then this is called **classification,** it is known as **Boolean/ binary classification** if there are only **2 values.**

In cases where y is a number e.g tomorrow's weather the learning problem is called **regression,** technically solving this kinda of problem is finding the conditional expectation or average value of y, because the probability that we have found exactly the right real-valued number for y is 0. (due to temperature technically being an infinite number, since there can be infinite decimals to determine it).

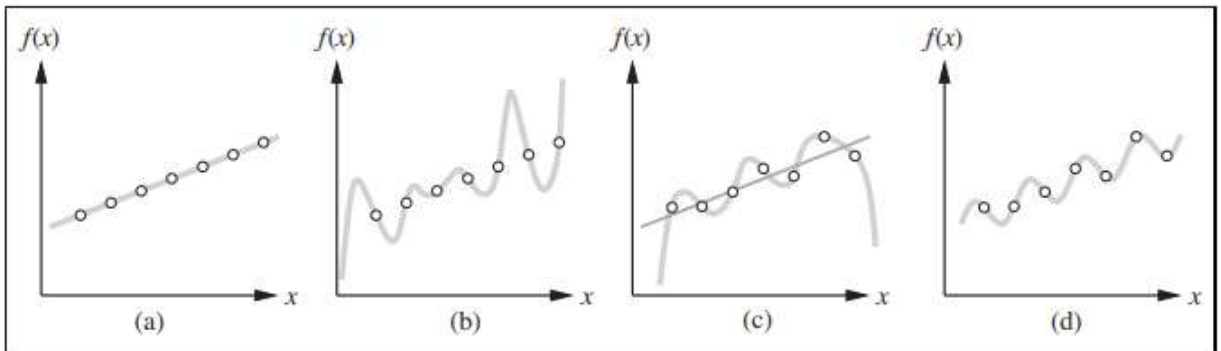


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data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

This image above shows a familiar problem, it involves mapping a function to positions on a graph, where hte input is x and the output is y. A agent presented with this problem will now know what the function is but will approximate it based on the past outputs of y. This function (h)

is selected from a **hypothesis space (H)****, in the context of this example we can say that the space consists of a set of polynomials, (functions which represent the graph). The first graph shows a straight line, this is called a **consistent hypothesis** since it agrees with all data presented to it. However b also fits with this same data. This represents one of the core problems in inductive learning, how do we choose from among multiple consistent hypotheses.

***It is helpful to understand that a **hypothesis space** is not referring to the actual creation of every option, it more refers to a hypothetical scope that the learning model sees as its option. In the context of a decision tree its **scope is all trees possible from the attributes,** this is the definition of a hypothesis space.*

One solution is to prefer the answer with the simplest hypothesis that is still consistent with the data thai is known as **Ockham's Razor**. It is clear that a degree 1 polynomial is better than a degree 7 polynomial and therefore a is preferred to b.

On the other hand c shows another data set without a consistent straight line, it requires a degree-6 polynomial for an exact fit. There are however 7 data points so a polynomial with 7 parameters does not seem to be finding a pattern in the data and it is not expected to generalize well, a straight line that is not consistent with any of the dat appoints, but might generalize well for unseen values of x, is also shown in ©. The key point here is that while it can become consistent with the training data, it possible for it to in a way memorize it rather than learning the underlying function.

In d the same function data set is presented but the hypothesis space is expanded to allow polynomials over both x and sin(x), when this is done we can find the the data can be fitted by a simple function ax+b+csin(x). This shows the importance of the hypothesis space.

We say that the 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.

In some cases an analyst is able to tell that a function may not be correct based on its complexity. We can say that the probability is high for a degree1 or 2 polynomial however it gets lower as the degree increases. This means that we can use these weird functions when they are needed but discourage them with the lower prior probability.

Why not make the hypothesis space massive? This is because there is a trade off between the complexity and the computation time. While having every possible turing machine inside of the hypothesis space would ensure a solution it would increase the time taken to compute an answer, whereas if you limit the hypothesis space the answers would be computed faster.

This is important because it’s valuable to balance speed with the ability to generalize well with new data.

18.3 Learning decision trees

Decision tree induction is one of the simplest and yet most successful form of machine learning. The representation will be described - the hypothesis space - and then show how to learn a good hypothesis.

18.3.1 The decision tree representation

Intro

A **decision tree** represents a function that takes an input of a vector of attribute values and it returns a “decision”, this is a single output value.

Types of inputs

Input definitions.

1. **Discrete** - refers to a set of inputs that have finite discrete values e.g true or false.
2. **Continuous** - refers to a set of inputs that can take any values between a range

Examples of Discrete Inputs

Categories/ Labels

Color of a car: {Red, Blue, Green, Black}
Type of animal: {Dog, Cat, Bird}
Education level: {High School, College, Graduate}:

Integer values

Number of children in a family: {0, 1, 2, 3, ...}
Rating of a movie: {1, 2, 3, 4, 5} (these are discrete because there are a limited number of possible values)

Specific States.

Traffic light color: {Red, Yellow, Green}
Day of the week: {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}

Examples of Continuous Inputs:

Height: A person could be 170.5 cm, 170.55 cm, 170.555 cm, etc.
Temperature: It can be 25.5°C, 25.55°C, or any value between, including decimal points.
Time: It can be 3.4 seconds, 3.45 seconds, 3.445 seconds, and so on.

Focus of these notes

We are focussing on problems where the inputs have discrete values, and an output with exactly two values, this is a boolean classification, each input will be classified as true or false.

How does a decision tree come to a conclusion?

It makes a decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one the input attributes, Ai.

The branches from the node are labeled with the possible values of the attribute, Ai = vik.

Each leaf node (node with no children) in the tree specifies a value to be returned by the function.

Example building of a decision tree.

As an example we will build a decision tree to decide whether or not to wait for a table at a restaurant.

The aim is to learn a definition for the goal predicate *WillWait*. First we list hte attributes that we 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 (none,some,full)
6. Price: the restaurant's price range ($,$$,$$$).
7. Raining: whether we made a reservation.
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, 10-30, 30-60 or 60 mins).

Every variable has a small set of possible values; the value of *WaitEstimate,* for example, is not an integer, it is one of four discrete values.

18.3.2 Expressiveness of decision trees.

A boolean decision tree is logically equivalent to the assertion that the goal is true if and only if the input attributes satisfy one of the paths leading to a leaf with a value true.

Writing this in propositional logic we get

$$Goal \Leftrightarrow (Path1 \vee Path2 \vee \ldots),$$

⇔ this represents logical equivalence aka biconditional.

The biconditional operator is used to show that two propositions have the same truth value.

Essentially meaning if the goal is true then the second statement must hold and vice versa.

The second part of the statement is true if any of the paths (path1, path2… pathN) is true. This means that the whole expression is equivalent to **disjunctive normal form** - meaning that it is a conjunction or ors e.g ((a and b) or (c and d) or …).

This means that **any function in propositional logic can be expressed as a decision tree.** This is because any function in propositional logic can be represented in disjunctive normal form.

For example in figure 18.2 is Path = (Patrons = Full and WaitEstimate = 0-10).

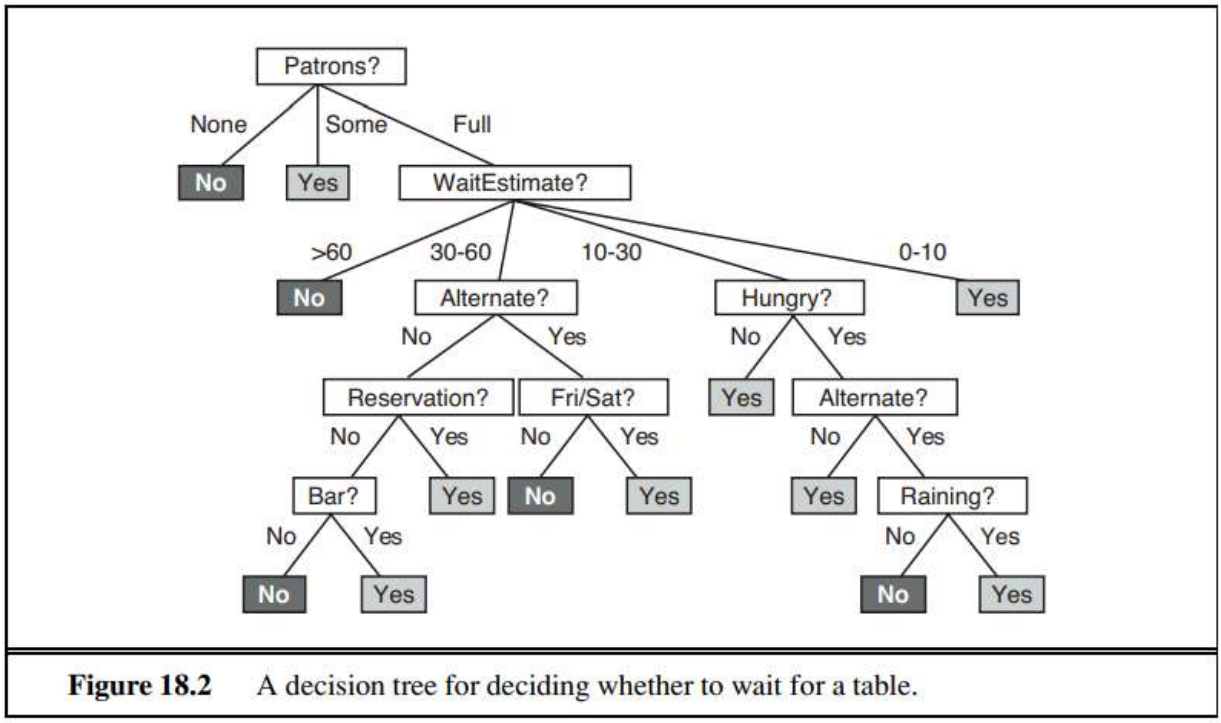


Figure 18.2 A decision tree for deciding whether to wait for a table.

For a wide variety of problems a decision tree format yields a nice concise result. But some functions cannot be represented concisely, for example the majority function which returns true if and only if more than half of the inputs are true, requires an exponentially large decision tree.

Decision trees are favourable for some functions and not for others.

Is there any representation that is good for all functions? No.

We can show this in a general way, consider a set of all Boolean functions on n attributes. How many different functions. How many different functions are in this set? This is just the number of different truth tables that we can write down, because the function is defined by its truth table.

A truth table over n attributes has 2^n rows, one for each combination of the values of the

attributes. We can consider the “answer” column as a 2^n bit number that defines the function. This means that there are 2^2^n different functions (and there will be more than that number of trees, since more than one tree can compute the same function). This is a scary number. For example with the ten boolean attributes of our restaurant there are 2^1024 or about 10^308 different functions. We will need some ingenious algorithms to find good hypotheses in such a large space.

18.3.3 Inducing decision trees from examples

An example for a Boolean decision tree consists of a (x,y) pair, where x is a vector of values of the input attributes, 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.

Is shown here.

The positive examples are ones in which the goal *WillWait* is true; the negative examples are the ones in which it is false.

We want a tree that is consistent with examples and as small as possible. Unfortunately, no matter how we measure size, it is an intractable problem to find the smallest consistent tree; there is no way to efficiently search through 2^2^n trees. With some simple heuristics, however we can find a good approximate solution: small but not smallest consistent tree.

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 sub problems that can then be solved recursively, by most important attribute we mean the one that makes the most difference to a classification of an input.

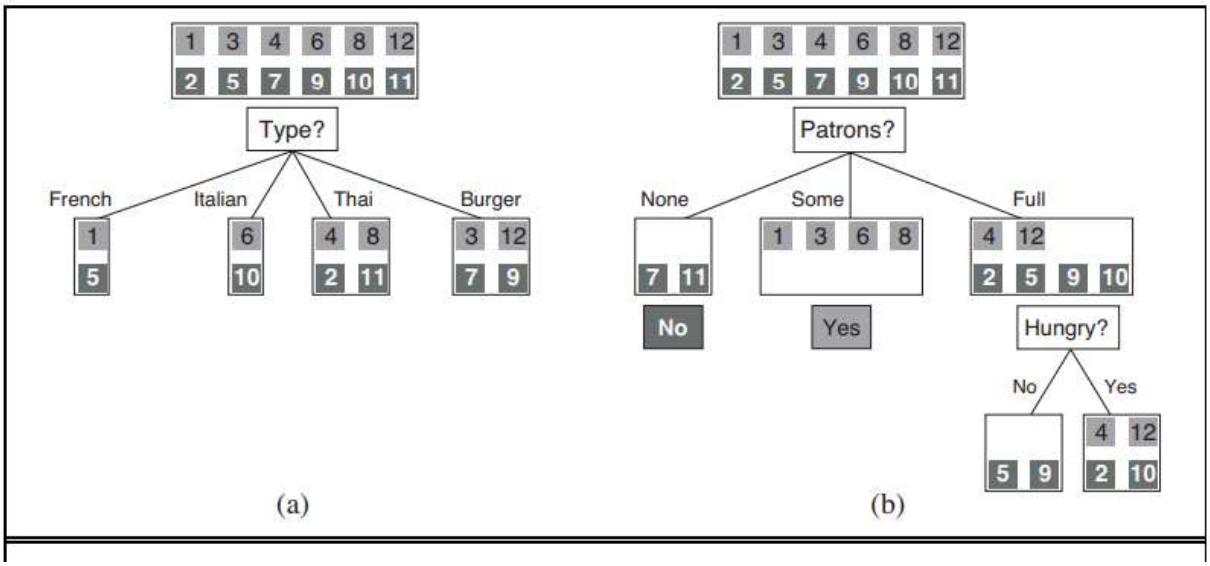


Figure 18.4 an example of splitting examples by testing on attributes.

When implementing this algorithm there are 4 cases to be considered:

1. If the remaining examples are all positive or all negative then we are done. This is shown by None and Some of the patrons predicate.
2. If there are some positive and negative attributes we must then split them by the best attribute. This is shown by full, where *Hungry* is used to split the remaining examples.
3. If there are no examples left then this means no example has been observed for this combination of values, so a default value calculated from the **plurality classification** - fancy way of saying the most likely result of the parent node.
4. If there are no attributes left but both positive and negative examples exist, it means that these examples have the same description but a different classification. This can happen due to error in the data; because the domain is nondeterministic; because we can observe an attribute that can distinguish them, in these cases we also return the plurality classification.

When it comes to constructing the tree using this algorithm a set of examples is **crucial**, however the examples themselves are not shown in the tree. The tree consists of just tests on attributes - which one of the attributes is it, the values of the attributes on the branches and the return value of a node leaf - the true or false return value.

```
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 function defined.

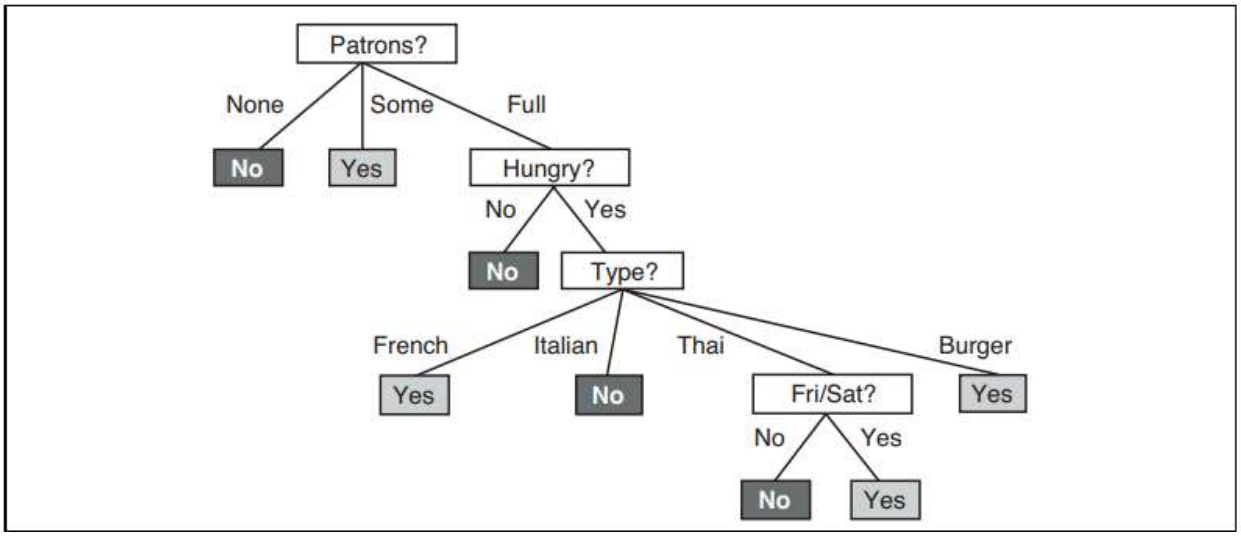


Figure 18.6 The decision tree induced from figure 18.3 training set.

It would be an easy mistake to make saying the Figure 18.6 appears to not be doing a very good job, however, the decision tree learning algorithm looks only at examples. And its hypothesis is consistent with all the examples and simpler. It has no reason to include tests for training and reservations, since it can classify all examples without them. It also has detected an interesting and previously unsuspected pattern: the first author - an example will wait for Thai food on weekends. Also worth noting that the tree is bound to make mistakes for cases where it has seen no examples e.g it has never seen a case where the wait is 0-10 minutes but the restaurant is full.

In that case the decision tree would say NO do not wait. However, many people would happily wait 0-10 minutes to be seated at a restaurant if they were hungry. With a more extensive training set the program could correct this mistake.

It is also important to note that there is a risk of **over-interpreting** (attributing more significance to something than it actually has) the tree that the algorithm selects. When multiple attributes has a similar importance, a small change in the test data could change the entire tree. The function computed is still similar but the structure can vary widely. When testing a decision tree we can evaluate the accuracy with a learning curve.

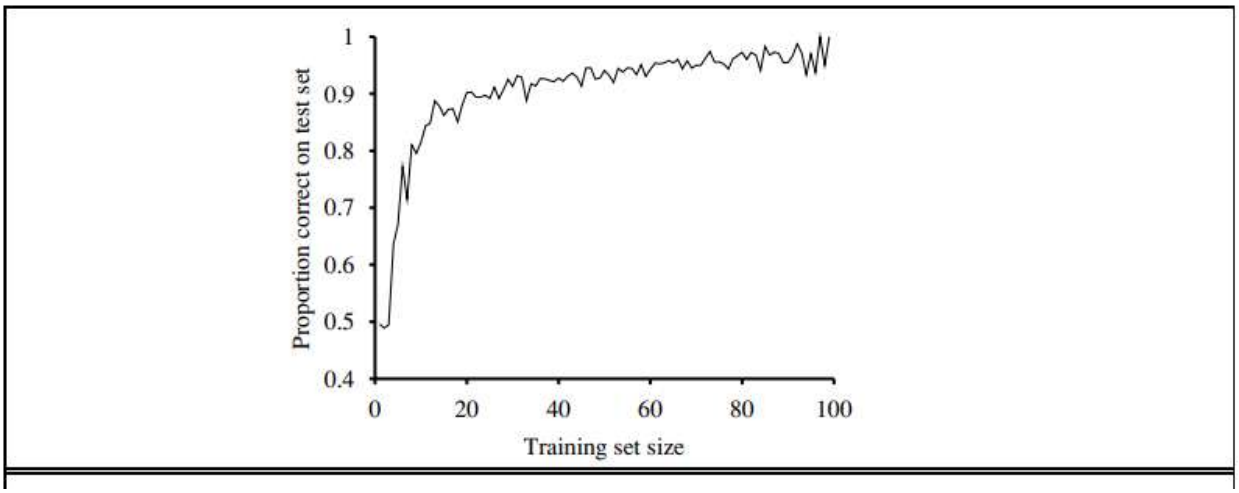


Figure 18.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain.

We can see that as the training set size increases, correctness on the test set also increases.

To conduct a test the example data e.g (100), is randomly split for the first test a random example is selected for the training and the rest is used as testing. This is then repeated 20 times, and the average is what is marked on the graph.

As the testing data increases the accuracy increases. (For this reason learning curves are also called happy graphs).

18.3.4 Choosing attribute tests

Choosing an attribute test refers to how to select which attribute is better than the other, this idea was discussed briefly earlier.

A perfect attribute is one that perfectly splits into positive and negative values for the given values, these become leaves of the tree.

For example the patron attribute is **not perfect** but it is fairly good as ⅔ of the results only contain either true or false values.

A bad one is one that makes very little difference in the proportion of true and false in the values than the original examples.

For example Type, leaves the sets with roughly the same proportion of positive and negative values.

What we need is a formal way to measure how good an attribute is, so we can implement the aforementioned IMPORTANCE function shown in Fig 18.5.

To do this we can use the notion of information gain, which is defined in terms of **entropy** (a measure of the uncertainty of results of a particular attribute).

An example of high entropy is *Type* since it doesn't draw us closer to an answer. On the other hand an example of low entropy is *Patron* since this leaves us with more certainty as to what the result is from the attribute.

An acquisition of information results in a reduction in entropy.

A random variable with 1 value e.g a coin that always shows heads - has no uncertainty and this its entropy is defined as 0.

A fair coin is equally likely to come up heads or tails so this counts as 1 bit of entropy.

The roll of a four sided dice has 2 bits of entropy, this is because it takes two bits to describe 4 equally probable choices.

An unfair coin that shows heads 99% of the time has less uncertainty than the far coin - if we guess heads we'll be wrong only 1% of the time - so we would like it to have an entropy measure that is close to zero, but is positive.

Entropy formula

The entropy of a random variable (V), with the values vk, each with a probability P(vk), 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 1 bit.

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

If the coin is 99% heads we get:

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

It can help to define B(q), as the entropy of a Boolean random variable that is true with probability q. This is because it simplifies the calculation when working with boolean.

Simply its an easier bit of maths which gets the same result when working with Boolean.

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

$$\text{Thus, } H(Loaded) = B(0.99) \approx 0.08.$$

If a training set has p (postive examples) and n (negative examples), then the entropy can be defined as:

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

The training set in Fig 18.3 has an even distribution of true or false final results (p = n = 6), from this we know the entropy is B(0.5) or 1 bit.

When testing a single attribute (A) it may only give us part of this 1 bit. We can measure exactly how much by looking at the entropy after the attribute test.

Calculating expected remaining entropy

Calculating the expected entropy of a single attribute A can somewhat be broken down into steps.

I will first define the values that we will use throughout this explanation:

A = attribute - e.g *Patrons* in the tree.
d = distinct values - how many unique values are in that attribute e.g *none, some, empty.*
E = training set - the training set.
Ek = one of the subsets of the training set.
pk = positive examples from each subset
nk = negative examples from each subset

Calculating the entropy can be done as follows.

1. Take the attribute.

: $\sum_{k=1}^{d}$

2. For each of the distinct values create a subset Ek this is represent as
3. For each of these subsets calculate the weight that this subset holds this is done using the equation $\frac{p_k+n_k}{p+n}$
4. Then we multiply that by the entropy of the subset, how much certainty it offers: $B(\frac{p_k}{p_k+n_k})$. we pass it pk/pk+nk because this gives us the probability of it coming out positive. This represents the amount of necessary bits we would need to answer the question. B returns the entropy of this particular distinct value.
5. Once this has been done for each value and they have all been added together the Remainder is the entropy of all the distinct sets put together.

The full equation looks like this:

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

Remember the higher the entropy the more uncertain.

Calculating information gain.
The higher the information gained, from an attribute, the better the attribute is.

The gain is easy to calculate,

You simply take the entropy of the whole set, represented by $B(\frac{p}{p+n})$

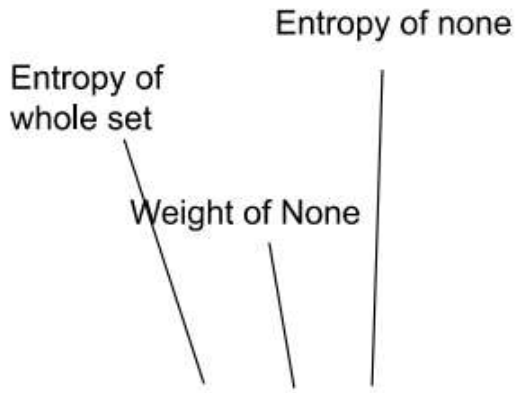
And reduce it by the calculation remainder(A).
The full equation looks like this:

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

The higher the information gain the better because it means picking that attribute will result in a smaller tree because more of the training set is sorted faster.

If you did these calculations for both *Patrons* and *Type.*

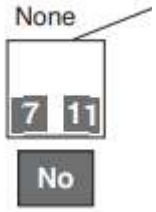
This is what the respective equations would look like:



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

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

I have annotated this to better illustrate what it looks like, the entropy of the whole set is 1 because there are equal amounts of positive and negative outcomes, this is reduced by the sum of each distinct value for the attribute *Patrons.*



For the case of *None* this picture, above for clarity, has 2 of the overall 12 test rows. This means that the attribute holds a weight of 2/12s. This is then multiplied by the entropy. We recall that this is calculated by pk/pk+nk there are no positive results and 2 negative subsisting we get 0/0+2, the 0 on the bottom has no impact so it becomes 0/2 and finally this is passed into B(0/2), to return the entropy which in this case is 0, since there is no uncertainty about what the set contains. They are then multiplied together which returns 0 similarly for the second which represents some, they are all true meaning the entropy returns 0 and the multiplication does as well. So far we have 0+0

Finally *Full* is considered, in this case it has half of the weight containing 6/12 and the entropy calculation B(2/6) returns 0.459 bits. Reducing this from the entropy of the whole set we get 0.541.

18.3.5 Generalising and overfitting

Defining the issue

On some problems the DECISION LEARNING TREE algorithm will create a large tree when there is no pattern found.

Imagine the problem of trying to know whether a dice **will roll 6 or not.**

Now suppose that the nice had attributes like color, weight and time. If the dice are fair since it is impossible to predict what it will do the tree should have a single node “no”. However the algorithm will find any pattern it can in the data. If it turns out that there are 2 rolls of a 7-gram blue die with fingers crossed and they both came out 6 the algorithm would draw correlation. This is known as over fitting

Overfitting becomes more common as the hypothesis space and the number of inputs grows, and less likely as we increase the training examples. A large hypothesis space a large number potential answers means that the model is morel likely to choose a solution that simply fits that and is not well generalized.

The larger the input the more likely because the model is more likely to draw patterns that don’t exist inside of the data.

Increasing the training size reduces it because the model has more opportunities to learn the underlying patterns.

Decision tree pruning

This is a method of combating overfitting by removing nodes that are not clearly relevant. We take the tree algorithm found. We then look at a test node that has only leaf nodes as descendents, if this test appears to be irrelevant detecting only noise in the data we eliminate it by replacing it with a leaf node. This process is repeated for each test with only leaf descendants, until one has either been pruned or accepted as is.

How do we test that a node is testing an irrelevant attribute?

In order to test this we can look at what the example split is, if they are roughly the same proportion of positive to negative as the original set itself then we can assume it is relevant. The lack of information gain is a good clue to irrelevance. How large a gain should be required to split on a particular attribute? We can do this using a **significance test.**

Significance test

This test begins by assuming there is no underlying pattern. Then the actual data is anaylzed to calculate to what extent it deviates from a perfect absence of a pattern. IF it is statistically unlikely (5% or less), then that is considered to be good evidence for the presence of a significant pattern in the data.

19 Knowledge in learning

19.1 A logical formulation of learning

19.1.1 Examples and hypotheses

The aim of inductive learning in general is to find a hypothesis that classified the examples well and generalises well new examples.

Here we are concerned with hypothesis expressed in logic; each hypothesis hj will be represented by a **logical schema** ∀xGoal(x) ⇔ Cj(X),

Cj(x) is a candidate definition, some expression involving the attributes predicates.

In our example of will wait the tree can be represented as

$$\begin{aligned}\forall\, r\ \mathit{WillWait}(r) \leftrightarrow\ & (\mathit{Patrons}(r, \mathit{Some})) \\ & \vee\ (\mathit{Patrons}(r, \mathit{Full}) \wedge \mathit{Hungry}(r) \wedge \mathit{Type}(r, \mathit{French})) \\ & \vee\ (\mathit{Patrons}(r, \mathit{Full}) \wedge \mathit{Hungry}(r) \wedge \mathit{Type}(r, \mathit{Thai}) \wedge \mathit{Fri/Sat}(r)) \\ & \vee\ (\mathit{Patrons}(r, \mathit{Full}) \wedge \mathit{Hungry}(r) \wedge \mathit{Type}(r, \mathit{Burger})).\end{aligned}$$

Which is the DNF of all the paths. Meaning its a series of Ands connected by ors.

Each of these are a hypothesis, each hypothesis predicts a certain set of examples, the ones that satisfy the candidate definition (the second part). Will be examples of the goal predicate.

This set is known as an extension of the predicate.

The hypothesis space H is a set of all hypothesis {h1, …,h2} that a learning algorithm is designed to consider.

A decision tree learning algorithms hypothesis space contains all of the decision trees that agree with the attributes it was provided with, meaning it could come up with any of them however when an example set of data is passed in it then finds the one that matches that.

This means that the algorithm believes that one of these is correct h1 V h2 V h3 V … V hn.

Each h being a potential decision tree.

19.2 Knowledge in learning

When working with decision trees we didn't make use of prior knowledge. The advantage of prior knowledge is that it narrows down the hypothesis space.

A hypothesis that explains the observations from the training set must satisfy this property. .

$$Hypothesis \wedge Descriptions \models Classifications \ .$$

Hypothesis - the solution the learning model has created, e.g a decision tree.
Descriptions - the training data attributes and their corresponding value
Classifications - the classifications refer to what the result was from the descriptions in the training set.

Logical Entailment - there is now row where the former is true but the later isn't e.g

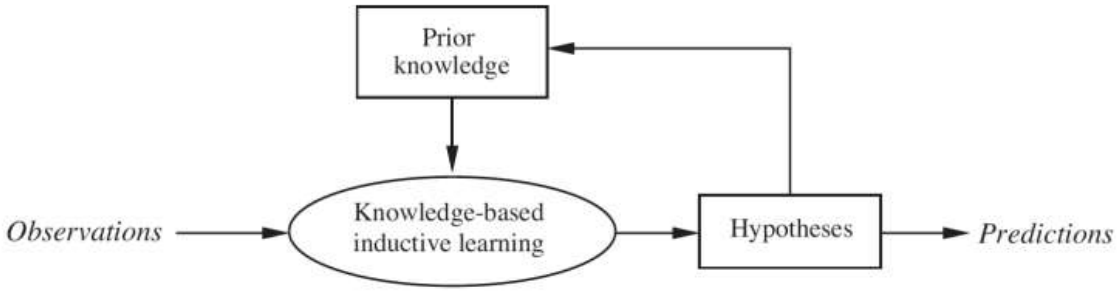
A (Raining)	**B (Wet ground)**		**A ⊨ B?**
T	T	✅	
F	T	✅	
F	F	✅	
T	F	❌ ← contradiction! (this would break entailment).	

A	**B**	**A ⊨ B**
T	T	✅
F	T	✅
F	F	✅

This holds entailment.

This property is known as the **entailment constraint of pure inductive learning** here the *Hypothesis* is unknown.

Pure knowledge free inductive learning can be replaced with a new approach:



This is designing agents the already know something and are trying to learn more. The graph shows that the hypothesis, successful or incorrect, can be stored to increase the prior knowledge of the agent.

Generalised Entailment Constraint

We can represent the entailment constraint for this as:

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications$$

Background - prior knowledge.

The background and the hypothesis combine to explain the classifications.

Algorithms that satisfy these constraints are called **knowledge based inductive learning** (KBIL), algorithms.

They have mainly been studied in the field of inductive learning.

19.5 Inductive Logic Programming

19.5.1 An example

This refers to a rigorous approach to knowledge based inductive learning problems. By rigorous we mean it is based on logical foundations.

ILP allows us to create methods to infer information from prior knowledge.
This prior knowledge is represented in first order logic, meaning we can represent more complex relationships where a decision tree would fail.

Example:
parent(alice, bob).

parent(bob, charlie).
parent(bob, diana).
parent(carol, edward).
parent(edward, frank).

Decision trees fail to represent this as they can only look at attributes not relationships between different entities.

It also allows us to create rules surrounding these objects, rather than only their attributes.

Example:
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

A DTL algorithm would have to create pairs of these objects essential examples such as

Grandparent(alice, diana)
The descriptions (the node decision making) would be hard to represent e.g

FirstElementIsMotherOfEllizabeth.

This means that the rule for grandparent is a large disjunction which make sit hard to generalize.

Advantage of ILP is it can apply to relational predicates so it can cover more problems

This has benefits as it makes them more readable to humans.

Additional background knowledge can be used to obtain more consensus hypthosies.

Suppose we know:

$$\text{Parent}(x,y) \leftrightarrow [\text{Mother}(x,y) \vee \text{Father}(x,y)]$$

We can make our previous grandparent predicate.

$$\text{Grandparent}(x, y) \leftrightarrow [\exists z\ \text{Parent}(x,z) \wedge \text{Parent}(z,y)]$$

This means x is a grandparent of y if and only if there exists some parent of z such that x is a parent of z and z is a parent of y.

This is a property of ILP algs that allow us to create new predicates based on current

knowledge. This is called constructive induction.

19.5.2 Top-down learning methods

In this section we discuss how FOIL the first ILP program worked.

We grow a hypothesis from a very general rule, and instead of using a tree, we can use **horn clauses with negation as failure.** Its called horn cause thats the guy who made it.

We can generate more specialized causes by adding conditions to the rule.

Literates (a logical expression that we whether want to prove true or false). Can be added using predicates, which can take only variables as their arguments.

This literals must include a variable that already appears in the rule.

These conditions can include equality/inequality constraints and arithmetic comparisons.

This causes a large branching factor, meaning there are many possibilities at each stage, which is computational expensive.

However its possible to add more information to reduce this.

Heuristic for a choice of information gain, is a good practice.

Also hypothesis that are longer than the length of examples should be removed as it can cause overfitting.

To write the most optimal hypothesis follow these steps:

1. Split examples into negative and positive by hand.
2. Start with the most general rule mark all as true.
    a. consider all the hypothesis needed to uphold the rule
    b. Use each of these and whichever classifies the most correctly should be the preferred predicate.
3. Repeat these steps with the new information gained from the optimal predicate.