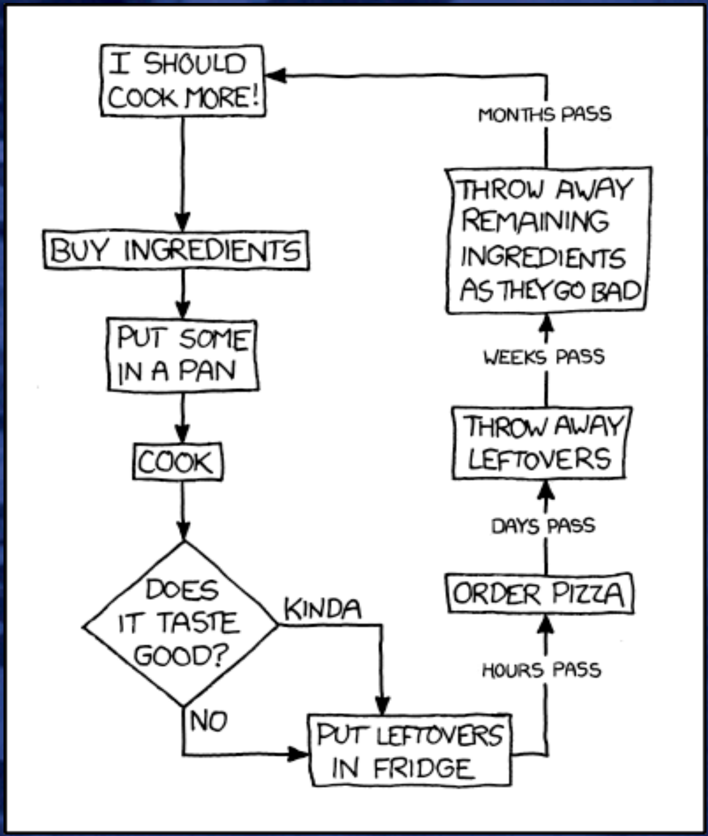
Learning Agents



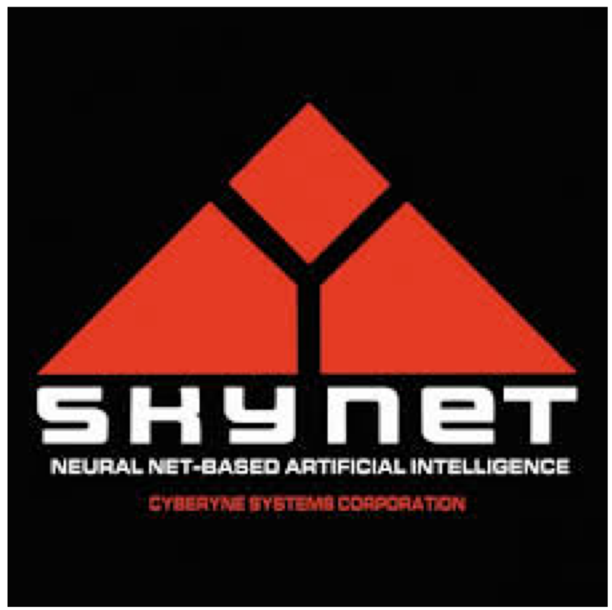
CITS3001 Algorithms, Agents and Artificial Intelligence

Tim French
Department of Computer Science and Software Engineering
The University of Western Australia

2021, Semester 2

Introduction

- We will discuss the basic structure of a learning agent
- We will discuss models of inductive learning
- We will discuss inferring decision trees as an example of a learning process
- We will discuss a methodology for assessing the performance of learning processes and the agents derived



Why do we want agents to learn

- In the agents we have described so far, all "intelligence" comes from the designer
  - From the algorithm design, and/or
  - From the heuristics used, and/or
  - From some other process used by the designer
- This has at least two significant disadvantages
  - It is time-consuming for the designer
  - It restricts the capabilities of the agent

- Learning agents can
  - Act autonomously
  - Adapt autonomously
  - Deal with unknown environments, outside their (and their designer's) experience
  - Handle complex data
  - Synthesise rules/patterns from large volumes of data
  - Improve their own performance
- But note it is *not* true that without learning, an agent can never outperform its designer
  - Computers can perform some kinds of processes far better than humans can!

A general model of learning agents

- The basic idea is that percepts are used not just for choosing actions, but also for improving future performance
- This requires four basic components

- A **performance element**
  - Responsible for choosing actions that are *known to offer good outcomes*
  - Corresponds to the agents discussed earlier
- A **learning element**
  - Responsible for improving the performance element
  - Requires *feedback* on how well the agent is doing
- A **critic element**
  - Responsible for providing feedback
  - Compares outcomes with some objective performance standard *from outside the agent*
- A **problem generator**
  - Responsible for generating new experience
  - Requires exploration – trying unknown actions *which may be sub-optimal*

Architecture

- Consider a uber driver agent

- *Performance element*: you want to go into Perth? Let's take Winthrop Avenue, it's worked well previously.
- *Problem generator*: nah, let's try Mounts Bay Road for a change – it may be better.
- *Critic element*: great, it was five minutes quicker, and what a nice view!
- *Learning element*: yeah, in future we'll take Mounts Bay Road.

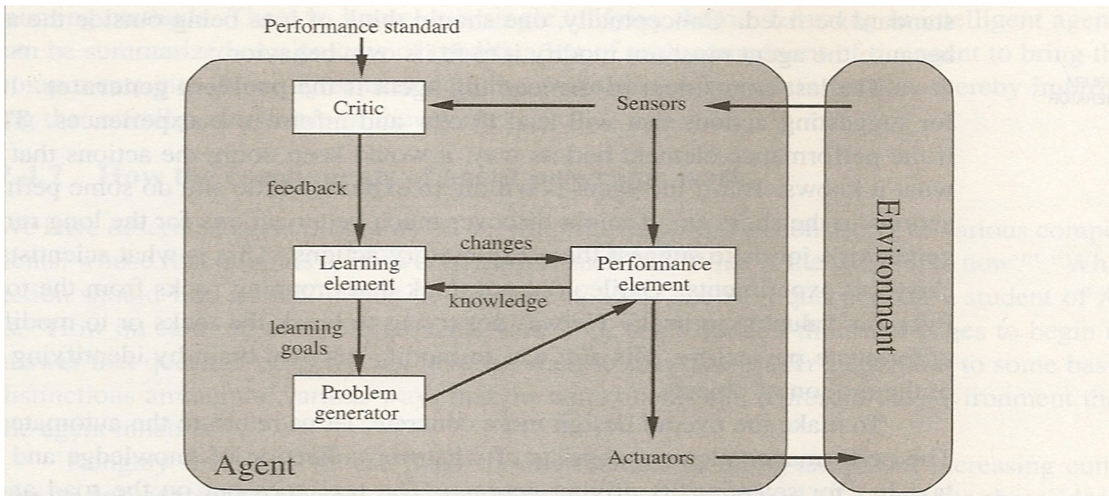
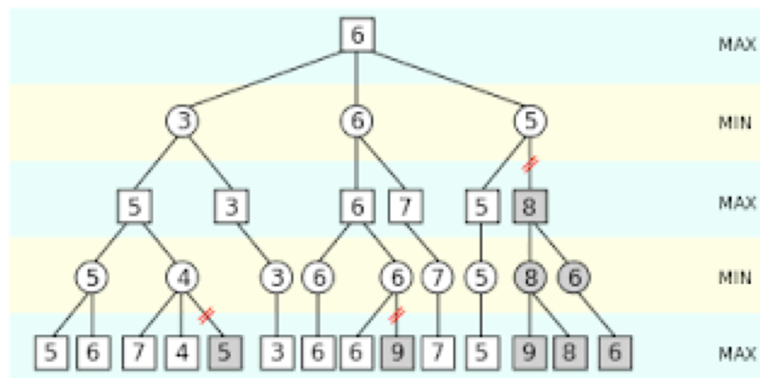


Figure 2.15 A general learning agent.

The learning element

- The learning element has two (separate) goals
  - Learning agents may focus on either or both, at any given time
- It wants to improve the outcome of the performance element
  - How good is the action chosen?
- Secondarily (usually), it wants to improve the time performance of the performance element
  - How fast does it operate?
  - This is called *speedup learning*
    - *e.g.* learning a good ordering for $\alpha\beta$
- The design of the learning element is affected by four main issues
  - The components of the performance element to be improved
  - The representation of those components
  - The feedback available, and its source
  - The prior information available



The performance element

THE UNIVERSITY OF WESTERN AUSTRALIA

- The performance element might have many components, *e.g.*
  - A mapping from states to actions
  - A means to infer information from percepts
  - Information about how the world evolves
  - Information about the effects of actions
  - Utility information about states
  - Goals whose achievement will increase utility

- Each of these components might be improved by learning, *e.g.* for the uber driver agent
  - The driving instructor shouting “brake!”
  - Being taught to recognise an ambulance
  - Observing what effect rain has on road surfaces…
  - and how that affects braking
  - Observing how driving behaviour affects tips
  - Learning new routes and their effects on income

- Clearly the details are highly context-dependent

Representing the performance element

- Representations come in many forms, *e.g.*
  - Game-playing agents may use linear weighted polynomials
  - Reasoning agents may use logical sentences and inference engines
  - Belief networks may use probabilistic descriptions
  - *etc.*
- The scope for the learning element to improve the performance element will clearly depend on the representation used
- Again, the details will be context-dependent

The feedback available

- **Supervised learning** corresponds roughly to being taught by an expert
  - The agent is given a set of example input-output pairs, *i.e.* problems and correct answers
  - The agent learns a general rule that captures these examples as special instances
- **Reinforcement learning** corresponds roughly to learning from experience
  - *e.g.* from the result of a game, or the size of a tip
  - Try something new and see if it works better!
  - The agent experiments, and remembers what worked and what didn't
- **Unsupervised learning** happens (usually) in the absence of feedback
  - Basically means learning patterns in the input
  - The most common task is *clustering*
    - Partitioning input values into sets
  - *e.g.* a taxi driver may learn to distinguish "good traffic days" from "bad traffic days", or that the freeway is usually busy at 8am

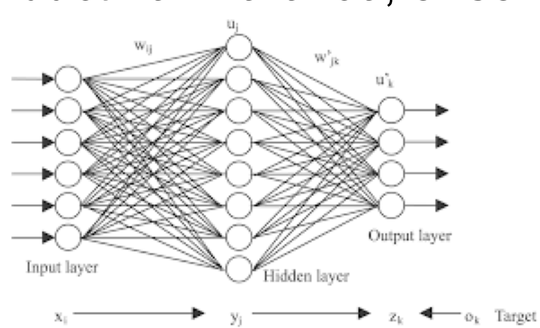
The prior knowledge available

- There are two “ends of the spectrum” in prior knowledge
- *tabula rasa*: the agent starts with an empty slate
  - And starts with only “basic skills”
  - Sometimes called *blue sky* or *green field* design
- The agent starts with a known good design
  - And tries to fine-tune it
- Obviously *tabula rasa* done well ends with fine-tuning…
- This distinction captures *exploration* vs. *exploitation*
  - Do we stick with (exploit) what we know, or do we try new things (explore) and hope they work better?
  - *cf.* teacher vs. student
- In practice, most situations fall somewhere in the middle
  - And learning is usually hard
  - Use background knowledge when available!
  - But relying too much on prior assumptions might mean that you get out only what you put in

Function approximation

- Mathematically, all components of the performance element can be described by a function
  - How the world evolves: *f: state → state*
  - Reaching a goal: *f: state → {0, 1}*
  - Optimising a utility: *f: state → [–∞, ∞]*
  - Evaluating an action: *f: (state, action) → [–∞, ∞]*
- Thus all learning can ultimately be seen as learning a function
  - All learning can be seen as *function approximation*
- Implementation details will vary dramatically…

- Given a set of data instances *(x, f(x))*, return a function *h* that approximates *f*
  - *h* is called a *hypothesis*
- This task is known as *pure inductive inference*, or sometimes just *induction*



Inductive learning

- In general, we have to decide
  - What mathematical operations are available for *h* (polynomials, exponentials, trigonometrics, *etc.*)
  - What trade-off we will tolerate between exactness and generalisability
  - Whether any of the data can be dismissed as *outliers*
- All sets of *n* pts fit exactly a *k*-degree polynomial, $k < n$!
- These decisions will determine both
  - The type of learning algorithm required
  - The overall tractability of the problem
- Another issue is the update policy when new data arrives
  - *Incremental learning* updates *h* with each new pair
  - *Reinforcement* relies on feedback from using *h*

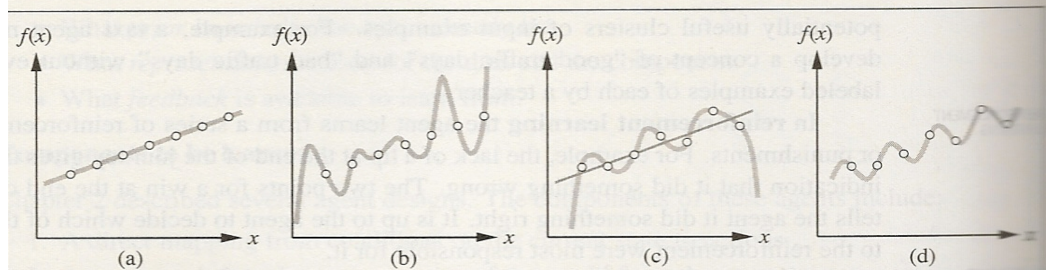


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data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

A concrete example – decision trees

- A decision tree is a representation of a Boolean function
  - *f: situation* → *{0, 1}*
  - Can also be thought of as defining a classification procedure, or a categorisation
  - Partitions the inputs into two subsets
- The input is a description of a situation
  - Abstracted by a set of *properties*, *attributes*, *features*, or *parameters*
- The output is *yes* or *no*
  - Identifies the situations with a positive response

- We will consider
  - Using decision trees in a performance element
  - Inducing decision trees in a learning element

Decision trees as performance elements

- Consider the question of deciding whether to wait for a table at a restaurant
- Our approach will be to formalise the question, and to build a decision tree that examines a situation and provides a yes/no answer
- The first (*crucial!*) step is to identify the relevant attributes of a situation that influence the decision, *e.g.*
  - Alternative nearby?
  - Bar?
  - Friday/Saturday?
  - Hungry?
  - Patrons?
  - Price?
  - Raining?
  - Reservation?
  - Type of food?
  - Estimated waiting time?

- Every attribute should be discretised so that it has only a small number of possible values
  - *e.g.* wait-time is discretised into four possibilities: < 10 minutes, 10–30, 30–60, > 60

Example decision tree

- The choice of attributes is crucial
  - Without examining the right attributes, it will be impossible to make a rational decision
  - “garbage in, garbage out”
- Sometimes this can be the hardest task!
  - *cf.* requirements analysis in software engineering

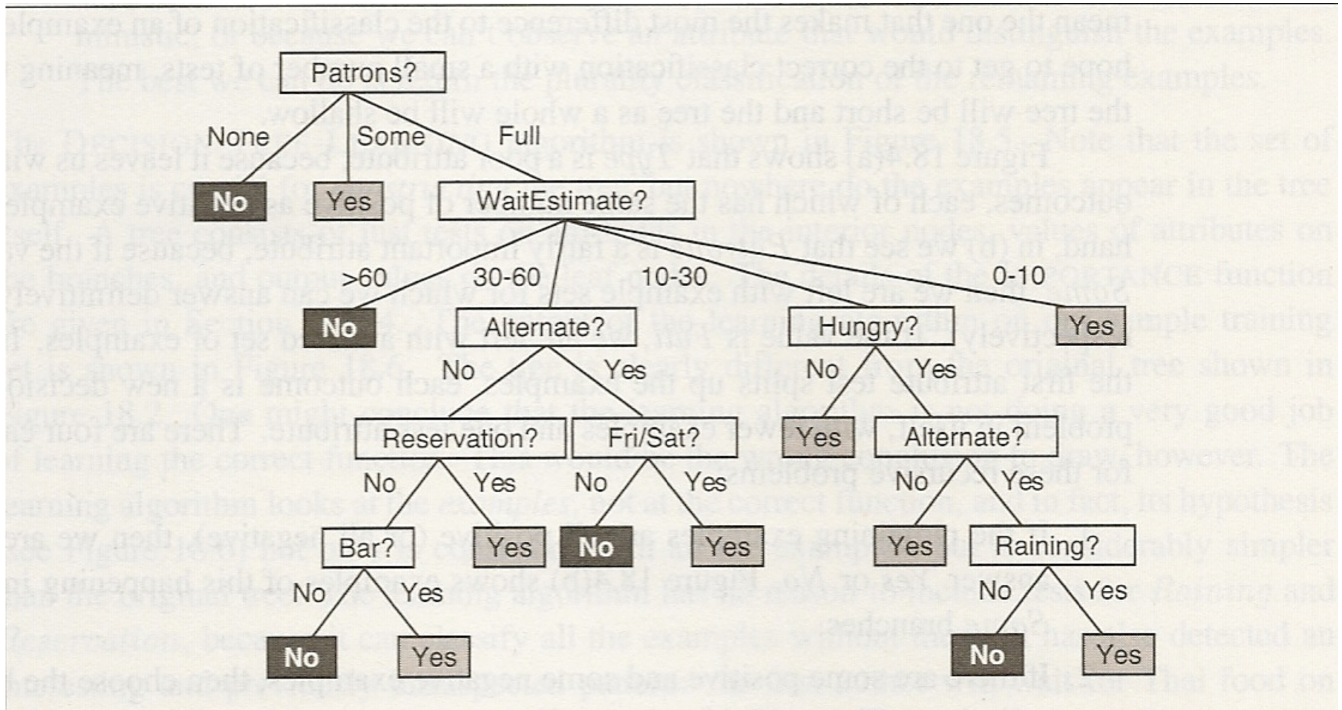


Figure 18.2 A decision tree for deciding whether to wait for a table.

Properties of decision trees as performance elements

- Limited inputs
  - Cannot handle continuous information
- Limited outputs
  - Can provide only yes/no answers
  - *e.g.* cannot choose amongst a set of restaurants
- Fully expressive wrt propositional problems
- *But* they can be huge

- Given *n* attributes:
  - There will be (at least) 2^n combinations of inputs
  - Hence (at least) 2^{2^n} possible functions
  - And many more possible trees!
- *e.g. 6* binary attributes implies $2^{2^6} \approx 10^{19}$ possible functions
- A non-trivial learning task!

Inducing decision trees

- We will use the following terminology
  - An *example* is a pair, with an input and an output
    - *({attributes}, value)*
  - A positive example is where *value* = *true*
  - A negative example is where *value* = *false*
  - A *training set* is a set of examples used for learning
- In 18.3, one row corresponds to one example
  - These come from exercising the tree in 18.2
- The goal of induction is to find a decision tree that
  - Agrees with all elements of the training set, and Is as small as possible

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.

A trivial induction algorithm

- Build a tree that branches on each attribute in turn, until you reach a distinct leaf for each example
- This approach has two principal problems
  - The tree will be much bigger than necessary
    - It does not search for *patterns* that summarise or simplify the training set
  - The tree will be unable to provide answers for examples that aren't in the training set
    - It cannot *generalise* from the training set
- These problems represent two sides of the same coin
  - They result from ignoring *Occam's Razor*
  - "the most likely hypothesis is the simplest one that is consistent with the data"
  - The tree has been *overfitted* to the data

A better induction algorithm

- Finding the (guaranteed) smallest tree is intractable
  - But we can use a greedy approach to find a “good” tree
- The basic idea is to always test the most important attribute first
  - This will give us a set of sub-problems that we can solve recursively, each with a subset of the data
- What do we mean by “the most important attribute”?
  - The one that “makes the most difference” to the example data
  - Note this implies that starting with different training examples will give a different tree
    - Is this a desirable feature of the approach?
- Usually aim to
  - make the whole tree as shallow as possible, or
  - make the average depth as small as possible, or
  - make the number of nodes as small as possible, or
  - …

Induction

- The text emphasises separating positive and negative examples as early as possible
  - Thus minimising the size of the tree
  - Thus *Patrons* is a good first attribute
- But an argument could also be made for *Type*
  - It minimises the size of the largest recursive sub-problem
  - Likely to minimise the depth of the tree
- This illustrates the heuristic nature of the approach

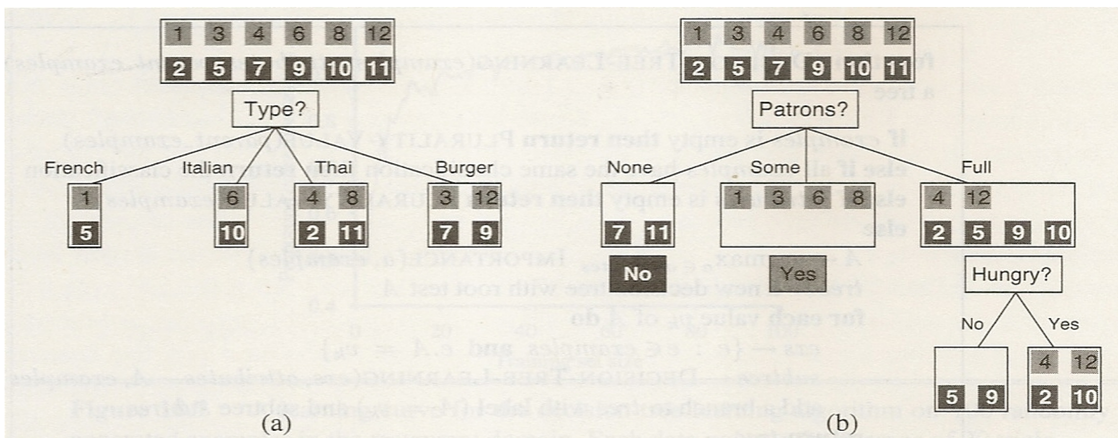


Figure 18.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

A recursive algorithm

- There are three possible base cases
- The remaining examples are all positive or all negative
  - *e.g.* for all Indian restaurants, we wait!
  - Stop and label the leaf either *yes* or *no*
- There are no examples left
  - *e.g.* there are no Indian restaurants in the data
  - No relevant examples are in the training set, so use the “majority vote” from the parent node
- There are no attributes left
  - *i.e.* there are identical rows with conflicting answers
  - The data is inconsistent, so the attributes originally chosen were inadequate
  - Either start again, or use majority vote

- There is one recursive case

- There are (still) both positive and negative examples
  - Choose the next attribute to discriminate on, create a node and divide up the set, and recurse

The derived tree

- Note that this tree is different to the original tree (18.2)
  - Despite using examples derived from the original!
- So is it wrong?
  - No – wrt the training set
  - Probably – wrt unseen examples
- But it is more concise, and it highlights new patterns
  - *e.g.* if there's no table available and you aren't hungry, leave!
- This process is akin to *data mining*
  - Identifying previously unseen patterns in the data

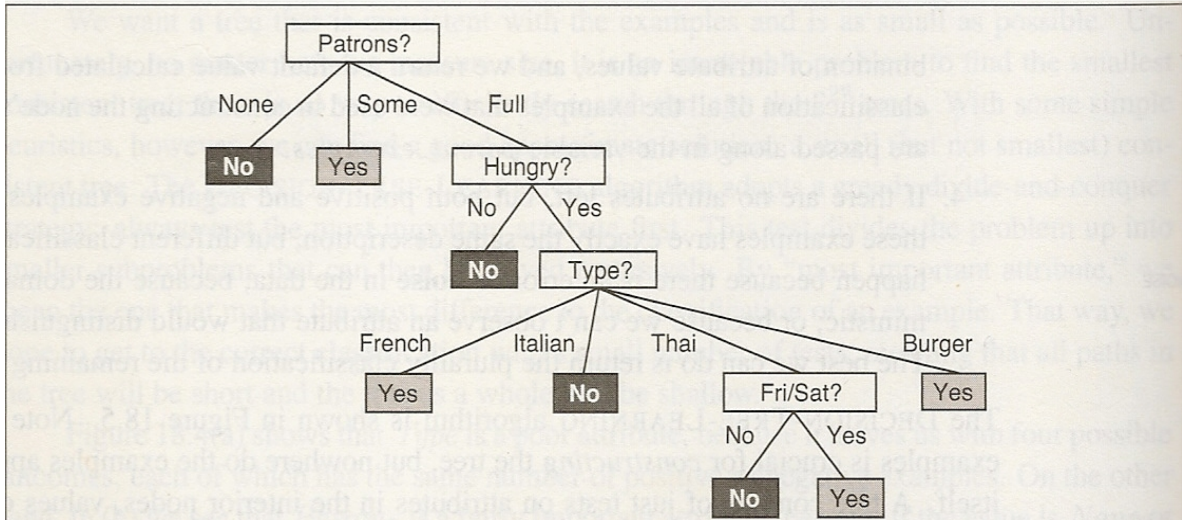


Figure 18.6 The decision tree induced from the 12-example training set.

Assessing performance

THE UNIVERSITY OF WESTERN AUSTRALIA

- We have seen that the derived tree
  - Fits with the seen data
  - Predicts the classifications of unseen data
- So to test whether it is a “good tree”, we need unseen examples to exercise it with
  - But of course we need to know the answers for those unseen examples
- The usual methodology is to
  - Collect a large set of examples
  - Divide them into a *training set* and a *test set*
  - Use the training set in the learning process
  - Then use the test set to assess the resulting agent
- One question is – how do we split the data?
  - More training data is good
  - But more test data is also good!
  - So try it out with different splits…

The happy graph

- Correctness on test set increases with size of training set
  - Zags at the end result from lack of test data
  - A common approach is 90% training, 10% test
- Basically, the shape of the happy graphs tells us that
  - There is a pattern
  - And the algorithm has identified it!

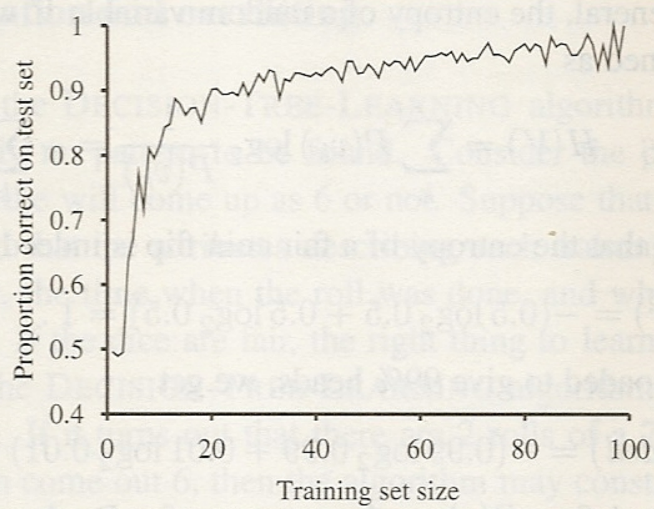


Figure 18.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

Practical instances of decision tree learing: GASOIL

- Michie, BP, deployed 1986
- Designed complex gas-oil separation systems
  for offshore oil platforms
- Attributes included
  - Relative proportions of gas, oil, and water
  - Flow rate
  - Pressure
  - Density
  - Viscosity
  - Temperature
  - Susceptibility to waxing
- World's largest commercial expert system in its day
  - Approx. 2,500 rules
- Building by hand would have taken 10 person-years
- Decision-tree learning was applied to a database of existing designs
  - System was developed in 100 person-days
- Outperformed human experts
  - More systematic, thinks "outside the box"
  - Said to have saved BP many millions of dollars

Practical instances of decision tree learning: C4.5

THE UNIVERSITY OF WESTERN AUSTRALIA

- Sammut *et al.*, 1992
- Learned to fly a Cessna light plane on a flight simulator
  - Learned a state-action mapping (a policy)
- Training was provided by three skilled human pilots
  - Each pilot flew an assigned flight plan 30 times
  - 90 flights, approx. 1,000 actions/flight
- Twenty attributes were used
  - *e.g.* wind, altitude, throttle, ailerons, angle, *etc.*
  - *i.e.* over $2^{1,000,000}$ possible functions!
- The generated decision tree was fed back into the simulator
  - Tree flew better than its teachers
  - Using the generalisation process “cleans out” “mistakes” by the teachers

Learning Under Uncertainty

THE UNIVERSITY OF WESTERN AUSTRALIA

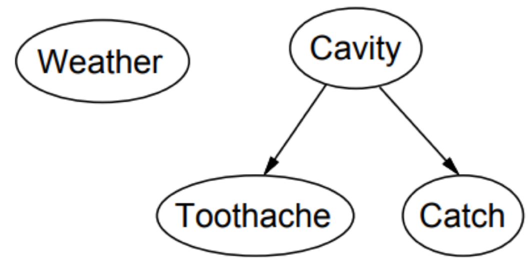
- Often we are required to learn in uncertain domains, where we do not have an oracle providing the correct class for a given observation.
- A variety of approaches exist, like fuzzy logic or belief functions, but probabilistic reasoning is the most widely used.
- Probabilities are given for events. E.g. *X* is "I will pass CITS3001", may have a probability P(*X*)=0.95 (95%) (the *prior probability*)
- We write ¬*X* for "not X", *X* ∨ *Y* for "X or Y", and *X* ∧ *Y* for "X and Y"
- Probabilities for different events are related: If *Y* is "I study for the CITS3001 exam" then we have the probability of *X given Y,* P(*X* | *Y*)=0.99 (the *conditional probability*).
- Conditional probabilities are defined by Bayes' Rule
- Probabilities must obey the Kolmogorov axioms:
  - $0 \leq P(X) \leq 1$
  - P(*true*) = 1, P(*false*) = 0
  - $P(X \vee Y) = P(X) + P(Y) - P(X \wedge Y)$

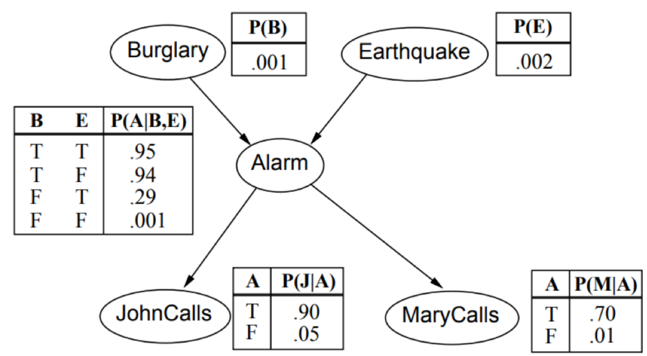
$$\Rightarrow \text{Bayes' rule } P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

Dependence

- Reasoning under uncertainty comes down to learning the probabilities of events, and how the probabilities are related.
- Given a set of events, the *joint probability distribution* is the probability for combinations of events occuring.
- For n events, there are 2^n different combinations to learn. However, many of these events may be *independent* (so $P(X \wedge Y) = P(X).P(Y)$) or *conditionally independent* so X and Y may whave a common cause, but are otherwise independent.
- Independence is a strong assumption, that makes computing probabilities much simpler.
- Bayesian Networks organise represent events in a directed acyclic graph, where events are only dependent on their parents, and otherwise conditionally idependent.
- We then just need to know the *joint* for nodes and their parents.

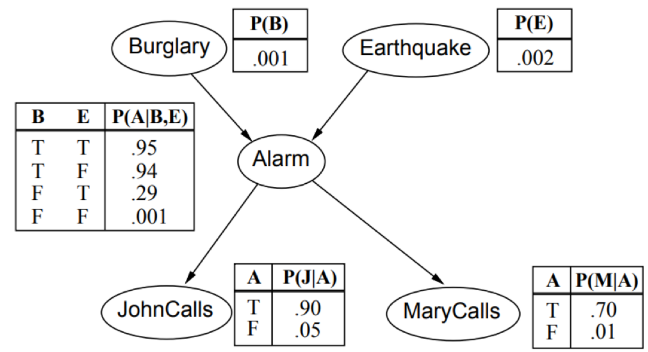
	toothache		*¬ toothache*	
	catch	*¬ catch*	*catch*	*¬ catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ cavity	**.016**	**.064**	**.144**	**.576**

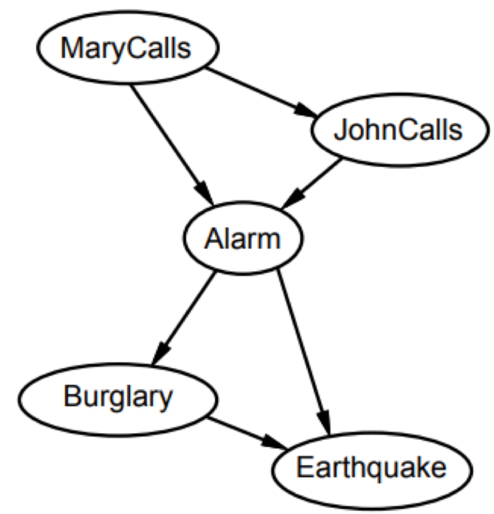




Bayesian Networks

- Bayesian Networks organise represent events in a directed acyclic graph, where events are only dependent on their parents, and otherwise conditionally idependent.
- We then just need to know the *joint* for nodes and their parents.
- Applying Bayes’ Rule we can represent the same information in networks with a different topology, but the complexity will not be the same.
- In general, computing the best topology for a Bayesian Network, or computing conditional probabilities from a Bayesian Network are NP-Hard.
- However, could approximations of probabilities can be approximated by using sampling algorithms, such as Gibbs sampling, or Markov Chain Monte Carlo methods.





Example: Car Diagnosis

- Bayesian Networks are a good method to take prior knowledge and assumptions, and compute conditional probabilities to support rational decisions.
- They can be generalized to handle continuous variables, and dynamic information.
- Bayesian Networks are used extensively in medical applications for diagnosis, but often still rely on expert guidance.

Initial evidence: car won't start

Testable variables (green), "broken, so fix it" variables (orange)

Hidden variables (gray) ensure sparse structure, reduce parameters