Decision Theory

We all take actions to accomplish our goals, and it's our *preferences* that have set those goals to begin with.

There are a variety of stepping stones that we take on the journey towards our goals, some that are more instrumental to accomplishing our task than others.

ά **Decision theory** is the art of designing rational agents that act based on their utility-based preferences and their perception of the world.

However, as opposed to some strong assumptions with planning that we made in past chapters, the consequences of our actions might not always be known.

In this case, where we're dealing with incomplete knowledge or stochastic outcomes, we have to take probabilities into account.

```
; In first order logic planning:
Action(Eat(Cake)
    Preconditions: Have(Cake)
    Effect: ¬Have(Cake) ∧ Eaten(Cake))

; ...where states:
s0 = Have(Cake)
s1 = ¬Have(Cake) ∧ Eaten(Cake)

; ...and so:
Result(s0, Eat(Cake)) = s1
; i.e., the result of being in state s0
; and eating your cake is that you *definitely*
; arrive in state s1
```

BUT... what if our cake is a Forney Industries Self-Replicating Cake, such that while eating it, it actually has a chance to make a copy of itself!

This means that eating it does *not* necessarily imply: ¬Have(Cake)

(OK, you can make the examples if you don't like that one)

Ḧ Instead of representing our action as a definite input-output function `Result(s0, a) = s1`, we'll instead represent it as a random variable: `Result(a) = s1` that has the *possibility* of arriving at some state s1 given an action we take and the evidence we observe about our environment.

```
; So now, instead of:
Result(s0, a) = s1

; ...we have:
Pr(Result(a) = s1 | a, e)

; ...where:
;   a  = the action we are taking (given because
;         we are interested in the posterior Pr that
;         we did indeed accomplish action a)
;   e  = the evidence about the environment
;         that we currently know about
;   s1 = state we are interested in the
;         *probability* of reaching given a, e

; [!] NOTE: In this new notation, the current state
; we are in, s0, is implicit, but could be made
; explicit by writing:
  Pr(Result(a) = s1 | a, e)
= Σ_s Pr(Result(s, a) = s1 | a) * Pr(s0 = s | e)

; i.e., the sum over all states of the probabilty of being
; in the initial state given the evidence (Pr(s0 = s | e))
; times the probability of reaching state s1 with action a
```

So now we have some notion of a distribution over possible states that we can reach from taking an action under the current evidence, we need to know *which* action we should take!

We do this by defining our agent's Utility function.

ά A **Utility Function** U(s) assigns a single number to express the desirability of a state (the higher the number, the more desirable).

ά The **Expected Utility** of an action given the evidence e, `EU(a | e)` is simply the average utility value of the outcomes weighted by the probability that the outcome occurs.

```
EU(a | e)
  = Σ_s Pr(Result(a) = s | a, e) * U(s)

; i.e., the expected utility of taking action a
; is the chance that action a will land us in state
; s with desirability of that state U(s)
```

Once we know the expected utility of taking some action, if we have multiple actions to consider, we simply take the one with the highest expected utility!

Wasn't it The Rolling Stones who said, "You can't always get what you want, but if you try sometimes you might find... maximizing your expected utility to be sufficient?"

á The **maximum expected utility (MEU)** criterion simply stipulates that an agent will choose an action (amongst all those that are possible) that has the highest expected utility of the choices.

```
action_choice = argmax EU(a | e)
                  a

; i.e., for all action choices a, choose
; the one who has the highest expected
; utility given evidence e
```

Example

t In the following examples, determine which action will be taken based on the maximum expected utility principle.

Problem 1: Andrew is deciding if he should use his day off to go to the beach, or stay inside and watch reruns of Golden Girls, because nothing else good is on TV... Going to the beach would be a lot of fun and staying in would just be OK, but forecasts gave a 60% chance of rain for the day, which would spoil a trip to the beach. Observe the following utility / probability table and decide what Andrew should do.

Action	Weather	U(Action, Weather)
AtHome	clear	2

AtHome	raining	3
AtBeach	clear	4
AtBeach	raining	1

à Click for solution.

Example
t Recompute the previous example given that we observed Weather = clear outside.

à Click for solution.

Preferences

So the next question you might be asking is: where do we get these utility values?

Do we need to have explicit numerical values mapped to states as indications of their desirability or can these numbers simply be relative and derived?

á **Preferences** are our agent's rankings of desirable states having considered their relative likelihoods of being reached.

We can use preference orderings to recover utility functions...

...but we also have to consider how desirable the state is *in concert with* the probability of reaching it.

In other words, two actions A1 and A2 might both be capable of reaching a desirable state, but if one of those two actions is more *likely* to reach that state, we will want to choose it over the other.

First, to formalize the possible outcomes of an action, we turn to the notion of a lottery:

ά A **lottery** is a representation of the possible outcome states from taking some action with the probabilities of reaching each outcome.

ά An **outcome** is either a primitive state or, recursively, another lottery

```
; For some action, a lottery L
; might look like:
L = [p1, S1; p2, S2; ...; pn Sn]

; ...where each:
;   p_i = the probability of reaching
;         S_i by taking that action
;   S_i = a possible state outcome OR
;         recursively, another lottery

; Example:
L1 = [0.5, S1; 0.25, S2; 0.25, L2]
L2 = [0.3, S3; 0.6, S1; 0.1, S5]
```

ά A lottery is called **complex** if it includes an outcome that is itself another lottery.

The term "lottery" is intuitive because taking an action associated with a lottery is like buying a Lotto ticket, and hoping that you "win" the state you desired.

Now that we know how we can model outcomes of taking a particular action, let's talk about preferences.

For any two lotteries A and B, we can use the following notation to describe preferences:

$A > B$	the agent prefers A over B
$A \sim B$	the agent is indifferent between A and B
$A \geq B$	the agent prefers A over B or is indifferent between them

The primary goal of utility theory is to determine how preferences between complex lotteries are related to preferences on the primitive states that compose them.

To do this, we can define some axioms on lotteries that, if violated by our intelligent systems, would lead to irrational behavior.

Axiom	Interpretation	Formalism
Orderability	An agent cannot avoid deciding between two actions (i.e., either one is preferred or they are equally preferable) and must assign one of the following relationships to lotteries A and B.	(A > B), (B > A), or (A ~ B)
Transitivity	If an agent prefers lottery A to lottery B, and also prefers lottery B to lottery C, then it also prefers lottery A to lottery C.	(A > B) ∧ (B > C) ⇒ (A > C)
Continuity	If A > B > C (i.e., some lottery B is between A and C in preference), then there is some probability p that we could find such that a certain outcome of B would be equally preferrable to an outcome of A with probability p or of C with probability (1 - p)	A > B > C ⇒ ∃p [p, A; 1 - p, C] ~ [1, B]
Substitutability	If two lotteries are equally preferrable, then you may substitute one in for the other in some other complex lottery.	A ~ B ⇒ [p A; 1 - p, C] ~ [p, B; 1 - p, C]
Monotonicity	If we prefer outcome A to outcome B, then we must also prefer lotteries that have a higher probability to reach A than B.	A > B ⇒ (p > q ⇔ [p, A; 1 - p, B] > [q, A; 1 - q, B])
Decomposability	We can reduce any number of complex lotteries down to a simpler one simply by the laws of probability.	L1 = [p, A; 1 - p, L2] L2 = [q, B; 1 - q, C] L1 ~ [p, A; (1 - p)q, B;

		(1 - p)(1 - q), C]

à Substitutability tells us that two equally preferrable lotteries can be substituted for one another in some other lottery. Does the following also hold?
`(A > B) ⇒ [p, A; 1 - p, C] > [p, B; 1 - p, C]`

Alright, so we have lotteries that abide by these axioms... what were we trying to do again?

Oh yeah... get some notion of what to use for utility functions...

Well, we have the following two consequences of preferential axioms that can allow us to solve for some (non-unique) utility function:

Consequence	**Interpretation**	**Formalism**
Existence of Utility Function	If an agent's preferences abide by the above axioms, then there exists a utility function such that: `U(A) > U(B)` if and only if A is preferred to B, and `U(A) = U(B)` if and only if the agent is indifferent between A and B.	U(A) > U(B) ⇔ A > B U(A) = U(B) ⇔ A ~ B
Expected Utility	The utility of a lottery is the sum of the probability of each outcome times the utility of that outcome.	U([p1, S1; ...; pn, Sn]) = Σ_i p_i * U(S_i)

From these two consequences, we see that, while it *does* matter what numbers we choose for our utility functions (dependent upon the scenario and how much more state s0 is desirable compared to state s1), there exists a utility function capable of respecting our preference ordering.

Example
t Read the following preferences, observe the action lotteries, and then decide which action our intelligent system would choose based on the MEU criterion.

```
; Our agent has the following preferences,
; which abide by the 6 axioms above:
1. A > B
2. B ~ C
3. C ≥ D

; Utilities:
      A  B  C  D
U(S)  3  2  2  1

; Action 1 corresponds to lottery:
L1 = [0.2, A; 0.3, B; 0.5, L3]

; Action 2 corresponds to lottery:
L2 = [0.5, B; 0.3, C; 0.2, L3]

L3 = [0.6, D; 0.4, B]
```

à Click for solution.

Multi-attribute Utility

Previously, we've dealt with states that have had an atomic utility value assigned to them, for example:

```
; Whole state of being at home and it
; raining out gets assigned, statically,
; the utility value of 3
U(AtHome, rainy) = 3
```

BUT, what if I wanted to assess *components* of states and treat them with different utility contributions?

Take the book example for... example:

> Siting (i.e., determining where to build based on analyzed factors) an airport requires a variety of considerations. If we're choosing between some number of land plots on which to build our airport, then we can analyze the putative values of some variables of interest based on our possible choices / actions. Let's say we were on the city planning committee of such and determined the following variables would comprise our decision criteria:

- **Cost:** the price of the land required to build upon, plus construction costs, plus the price of legal fees and processing, etc.
- **Noise:** the amount of noise generated by the airport.
- **Deaths:** possible deaths from construction hazards and other airport-related risks.

If we're considering a bunch of construction sites, it might not be feasible to sit down and assign a utility value to each one individually...

BUT, we might have some data on our variables of interest based on projections from studies and other sources of information, so perhaps we can construct our utility values from these facts.

That said, we often don't have deterministic information available (e.g., we don't know for certain that construction on site A will incur exactly 3 deaths), so we need to use estimation techniques.

Additionally, we don't always treat all of our variables of interest equally--some might need to be weighted as more important than others.

ά For some vector of variables **X** = {X1, X2, ..., Xn} and their value functions (f1, f2, ..., fn), where each f_i corresponds to a variable X_i, the utility of a state with variable instantiation: x = {x1, x2, ..., xn} is given by:

```
U(x1, x2, ..., xn) = F[f1(x1), f2(x2), ..., fn(xn)]
```

ά When our metrics of interest exibit **mutual preferential independence**, it means that a change in the value of one variable will not necessarily cause a change in the value of another variable.

Ħ If all of our metrics of interest exibit mutual preferential independence, then our agent's choice boils down to a simple maximimization of:

```
F[f1(x1), f2(x2), ..., fn(xn)] = Σ_i f_i(x_i)
```

Here, F is a simple function like addition that would aggregate all of the individual variable-weighting **value functions** f1, f2, ..., fn for variable values x1, x2, ..., xn.

The idea is that we want to condense all of the variable information into a single utility value based on the importance weights of each variable.

Example
t For our airport siting example, let's consider the following two sites and the value functions that provide the proper variable weighting. We'll assume our variables of interest exibit mutual preferential independence. Determine, using the mutual preferential independence utility function above with F being simple summation, which is the superior site.

Variable	**Value Function**	**Site 1 Value**	**Site 2 Value**
Cost	f_cost (x) = -x	$10,000,000	$15,000,000
Noise	f_noise (x) = -x * 1000	200dB	180dB
Deaths	f_deaths (x) = -x * 10^9	2	1

à Click for solution

á **Strict dominance** is the case where a state is superior on all metrics.

We see that in our example above, Site2 is not strictly dominant compared to Site1 because Site1 actually has the lower monetary cost.

It turns out, however, that the utility cost of our estimated number of deaths far outweighs the monetary cost.

Decision Networks

Often, however, our decisions have factors that are not necessarily the effect of anything we can control.

To model this uncertainty, we can combine Bayesian Networks, which model our knowledge of the way the world works, with the preference model we've been discussing to assess a utility value for possible actions under consideration.

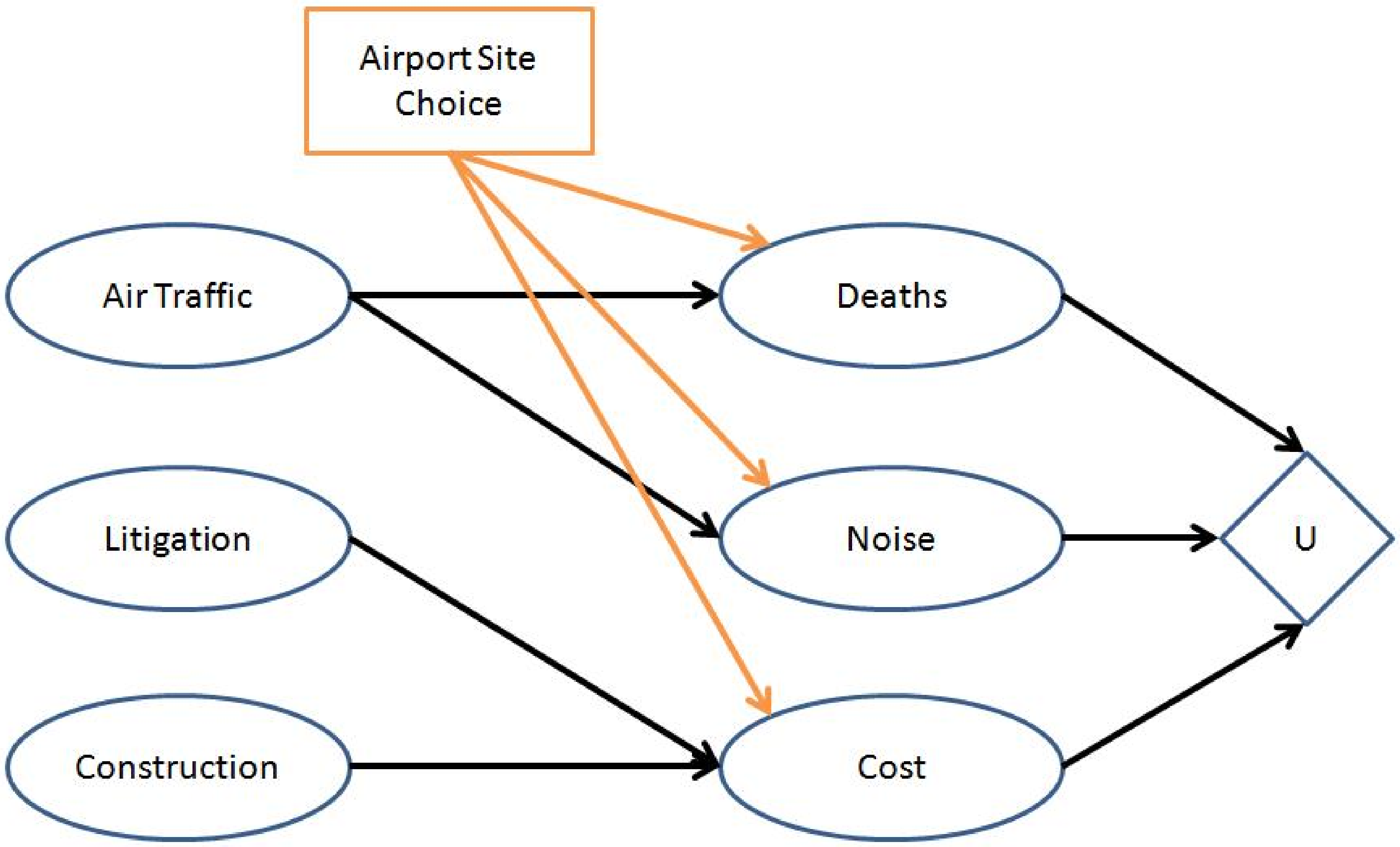
A **decision network** unifies the stochastic modelling capacities of a Bayesian network with the action-choice and utility concepts of preference models by adding two additional nodes to the Bayesian network model.

Decision network **chance nodes** (ovals) represent random variables, and illustrate uncertainty about the values of some of our variables (same way that Bayesian networks handled this: conditional probability tables).

Decision network **decision nodes** (rectangles) represent points where the agent has a choice of actions.

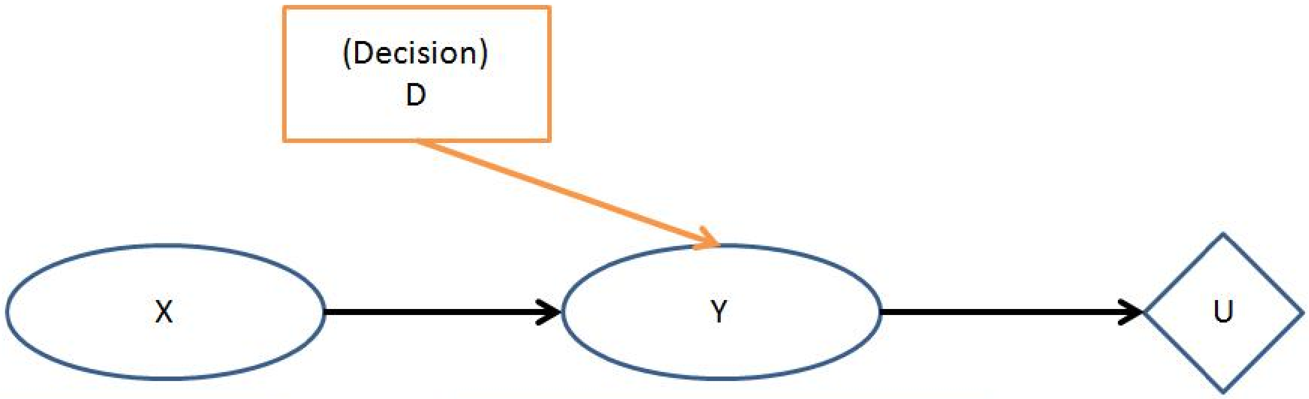
Decision network **utility nodes** (diamonds) represent the agent's utility function for a given instantiation of decisions and inference.

Here is the book's example for our airport siting problem, with some additional chance nodes added in to represent our knowledge of the world:



BUT, that example uses continuous variables, which we're not used to seeing... let's look at a quick example using discrete variables:

(here our decision node might have some nondeterministic action consequence)

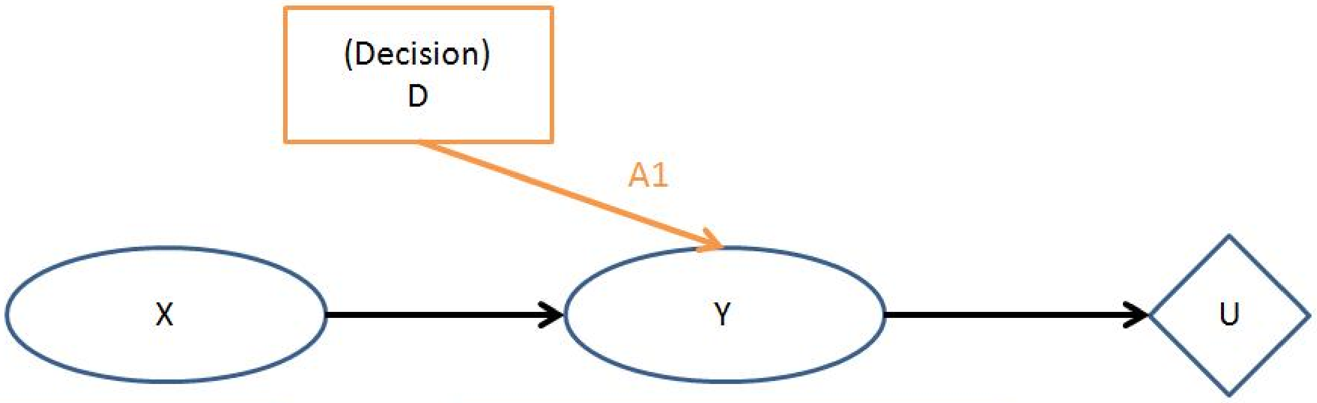


X	Pr(X)
0	0.4
1	0.6

D	X	Y	Pr(Y | X, D)
A1	0	0	0.2
A1	0	1	0.8
A1	1	0	0.7
A1	1	1	0.3
A2	0	0	0.5
A2	0	1	0.5
A2	1	0	0.4
A2	1	1	0.6

$U(Y = 0) = 3$
$U(Y = 1) = 1$

So, we can ask what the utility of a given action is by setting the action, which causes the decision node to act like a "given" chance node:



X	Pr(X)
0	0.4
1	0.6

D	X	Y	Pr(Y | X, A1)
A1	0	0	0.2
A1	0	1	0.8
A1	1	0	0.7
A1	1	1	0.3
~~A2~~	~~0~~	~~0~~	~~0.5~~
~~A2~~	~~0~~	~~1~~	~~0.5~~
~~A2~~	~~1~~	~~0~~	~~0.4~~
~~A2~~	~~1~~	~~1~~	~~0.6~~

$U(Y = 0) = 3$
$U(Y = 1) = 1$

à What is the expected utility of A1 as listed above? (click for solution)

Example
t Compute the EU(A2) using the above example.

So here are some observations to make:

- **Purpose of decision networks:** judge which actions from our decision (rectangle) nodes produce the highest utility.
- **Data formats:**

    - Chance nodes are CPTs just like in Bayesian networks, except parents of chance nodes can be decision nodes as well (which are always given because we're evaluating between action choices)
    - Decision nodes for a given action simply become given chance nodes.
    - Utility nodes are an evaluation of its direct causes (parents) based on supplied value functions. In our example above, U = F(f_death(Deaths), f_noise(Noise), f_cost(Cost))
- **Inference:** based on the choice for our decision nodes, we can use an inference algorithm (like variable elimination from last week) that gives us the utility node's parents' posterior probabilities to determine not only the weighted utility of a given variable value, but also accounting for the chance that it will occur based on any evidence.

And that's a decision network in a nutshell!

Of course, all of this relies on us as humans knowing what data and variables to incorporate into our decision networks...

Our next topic will be the ability to learn how to construct a decision mechanism from raw data alone!

Learning

The art of machine learning is little different from how we as humans acquire new information, and has particularly close analogy to how we teach children.

> ά **Machine learning** is the process by which programmers equip intelligent systems to modify their behavior based on training, observations, and other features available to the program during operation that might not have been available or convenient for the programmer.

So one of the first questions you might ask is: why give computers this adaptability at all? Shouldn't our programmers have all the tools necessary to empower their agents when they sit down to program them in the first place?

As it turns out, the answer is: not always; there are a variety of cases in which it is not possible for a programmer to explicitly define an agent's behavior, including:

- Prediction engines that assess changes over time cannot account for all possible future changes at the time of programming and must be adaptive.

- Navigational systems cannot be programmed with all possible obstacles, mazes, and terrain features at the time of programming and must be adaptive.
- Programmers may not *know* exactly how to define certain behavior, but can instead pose learning problems to their systems to get a gist for the intended behavior.

Learning algorithms generally suit one of a few different flavors:

Learning Class	Description	Example
Unsupervised Learning	Unsupervised learning is a type of pattern extrapolation engine whose typical task is **clustering**, the act of finding similarities between various input items (like images, text, etc.) and lumping them into some sort of group. In other words, it attempts to find hidden structures in unlabeled data.	Image Classification: given lots of images of faces and lots of images of computers, an unsupervised learning algorithm should be capable of lumping the faces together apart from the computers.
Reinforcement Learning	Just like operant conditioning (for humans), reinforcement learning provides an input for our program, on which it will make some sort of decision and be rewarded (if that was the right decision) or punished (if that was the wrong decision), figuratively speaking.	Animat Modeling: simulation study of animat populations where taking some action (like eating poison berries) has a harmful consequence that discourages that action in the future.
Supervised Learning	Just like having an instructor or parent correct your mistakes or reward your triumphs, supervised learning has some "oracle" that will tell you the correct answer on some problem during training so that you'll know what to change when you're wrong and what to keep when you're right. Additionally, labeled input data.	Object Recognition: given lots of images of objects with corresponding labels, e.g. a picture of a chair with label "chair", object recognition will be able to find other chairs in the future based on the label given *and* know that the particular object represents a chair (unlike unsupervised learning, which can only cluster)
Semi-supervised Learning	Almost exactly like supervised learning except that the labels or corrections given to us by our, now imperfect, oracle may not be entirely trustworthy. This creates extra noise that sets semi-supervised learning	User Input Classification: determining age based on images of people and their self-reported age (on which they might have lied).

	apart from its accurately supervised counterpart.	

So, the first task of machine learning is to choose a model class that best fits the data that we have available.

What exactly constitutes our definition of "best" depends on the task at hand...

Two tools we'll discuss for some applications are decision trees and Bayesian networks.

Classification

One of those most common machine learning tasks involves classifying different items evaluated on their variable settings and sorting them into the proper group.

Classification attempts to learn a **model** from a collection of examples that will predict a classification given the observable attributes.

Just what model we should use is dependent on the question we're asking, i.e., the classification we're trying to make, and the data that we have available.

So, let's look at some example classification problems, and the models appropriate for them!

Decision Trees

One type of classification is making a decision:

Given some input features about a situation, we have to evaluate whether to make one decision or another.

To model this task, we might consider a decision tree... but first...

Looking at a simplified version of the book's example:

Example
Dining Dilemma

Forney Industries has ~~recently contracted with the NSA~~ decided to branch out into the phone app business, and is debuting with its first app that learns your dining habits and can predict whether or not you will wait for a seat at a restaurant based on a number of attributes. The app will therefore generate a "yes" or "no" conclusion of whether or not you are going to wait based on the following inputs:

- **Patrons?** A measure of how many diners are currently at the restaurant, can be: {none, some, many}
- **Hungry?** Whether or not you are currently on the brink of starvation (ok, maybe that's too dramatic... you just haven't eaten since lunch), and can be: {yes, no}
- **Type?** The type of food the restaurant serves, which can be: {French, Italian, Thai, or Burgers} (you know... from the country of Burger?)
- **Fri / Sat?** Whether or not it is a peek-serving day of Friday or Saturday, can be: {yes, no}.

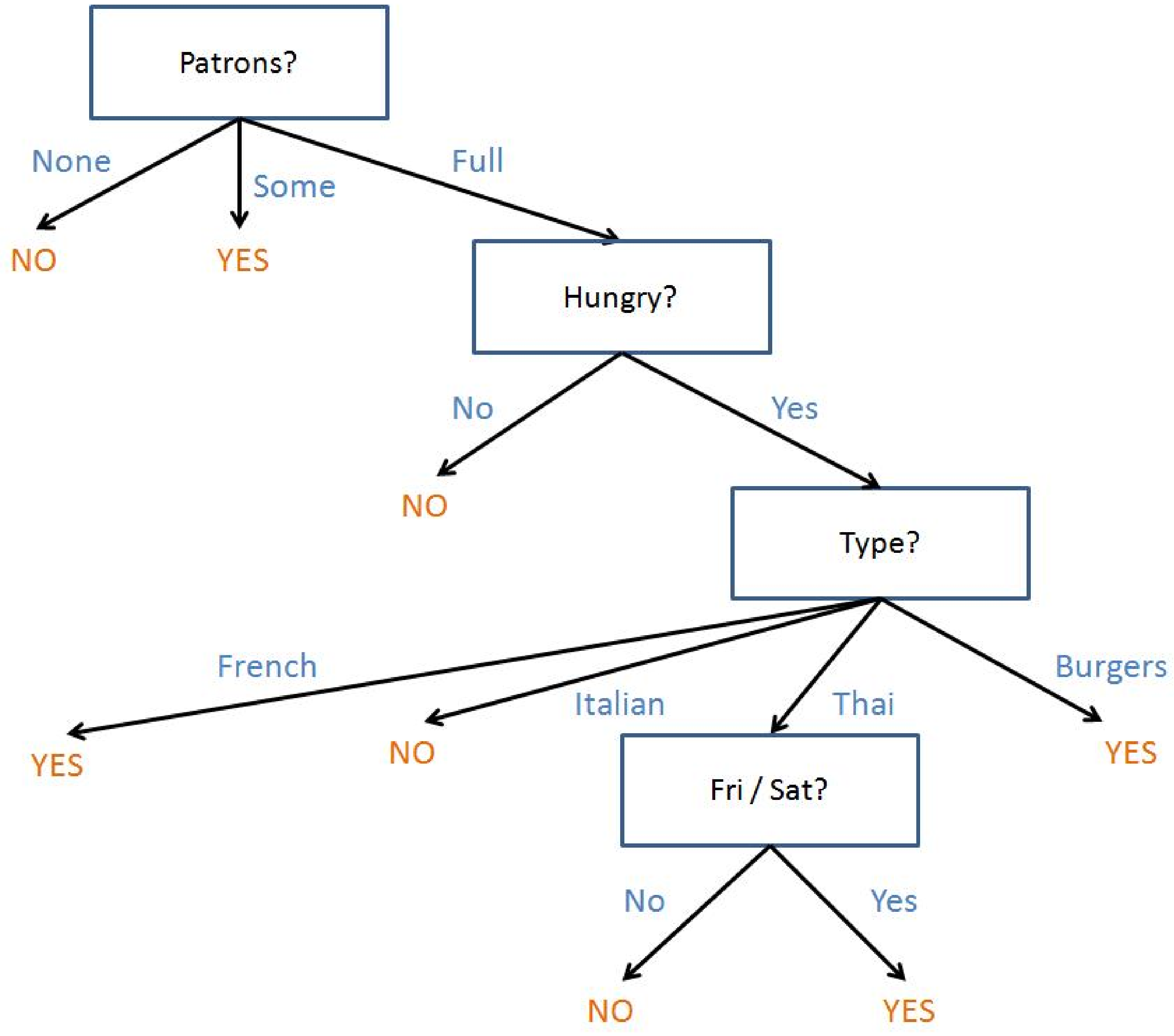
Our app has been learning your habits for awhile now and made the following determinations (expressed here just semantically):

- If there are no patrons you don't trust the restaurant quality and so will not wait, but if it's full, you'll only wait based on other attributes:
- If you're not really hungry, you're not going to wait, otherwise:
- You love French food and Burgers, so you'll wait at this point if it's either of those... but if it's Thai...
- You'll only wait if it's not a peek dining day of Friday or Saturday.

So if that's the plain-English description of our habits, how does an intelligent system predict whether or not we'll wait?

ά A **decision tree (d-tree)** is a tree structure with nodes as attributes and leaves as deterministic classification outcomes.

The d-tree of our above observations might look like:



It's easy to see what a decision tree will classify for our decision; we simply start at the root and trace the edges until we end up at a leaf!

Example

t What will our above decision tree give us for the following samples?

Patrons?	**Hungry?**	**Type?**	**Cost?**	**Fri / Sat?**	**Wait?**
some	yes	burgers	$$	yes	???

full	yes	italian	$$$	no	???
full	yes	thai	$	yes	???

You might notice: there was another attribute included in our sample data (Cost?) that was never used by our d-tree. That's fine, and sometimes preferrable, as we'll soon find out...

So now that we see how a d-tree works... let's talk about how to learn them in the first place!

Learning a D-Tree

To talk about learning d-trees, we imagine we have some large amount of data on which we can extract patterns to perform the learning aspect.

A **training set** is a list of input / output pairings where, given the input characteristics of a particular sample, we tell our learning system what the expected outcome should be (under the assumption that it will be able to formulate a classification function from lots of examples).

So let's take a look at an arbitrary training set (Example credit to Evan Lloyd; it was too beautiful not to use)

Imagine that we have arbitrary attributes A, B, C, and D. We must make a yes / no decision for some classification X.

A	B	C	D	X
grn	sml	1	0	yes
grn	sml	0	3	yes
red	med	0	5	yes
blu	med	0	5	no
grn	med	1	4	no
grn	lrg	1	1	yes
red	lrg	0	4	yes
blu	med	0	2	no
blu	lrg	1	4	no
blu	med	0	3	no
red	med	0	3	yes
grn	lrg	0	5	yes
grn	med	1	1	no
red	sml	1	2	yes
grn	lrg	1	3	yes

A	B	C	D	X
red	sml	0	1	yes
blu	sml	1	0	no
grn	lrg	1	2	yes
blu	med	1	4	no
grn	med	1	5	no
grn	med	0	2	no
red	sml	0	1	yes
grn	med	0	5	no
blu	med	1	1	no
red	sml	1	0	yes
blu	med	1	2	no
grn	lrg	1	1	yes
grn	lrg	1	1	yes
blu	lrg	1	0	no
grn	med	0	3	no

à What form of learning is this? Unsupervised, semi-supervised, or supervised?

Now, we want to find some decision tree that accurately gives the correct classification based on attributes A, B, C, and D.

But which one do we choose?

H A **single-node split** on a training set tries to maximize the classification accuracy based on a single attribute's values.

When we split on a value of a variable, e.g., A = red, we look at how many samples got classified to X = yes and X = no in the training set and try to make the classification that is most accurate based only on A = red.

So... let's try splitting on A first. This would give us:

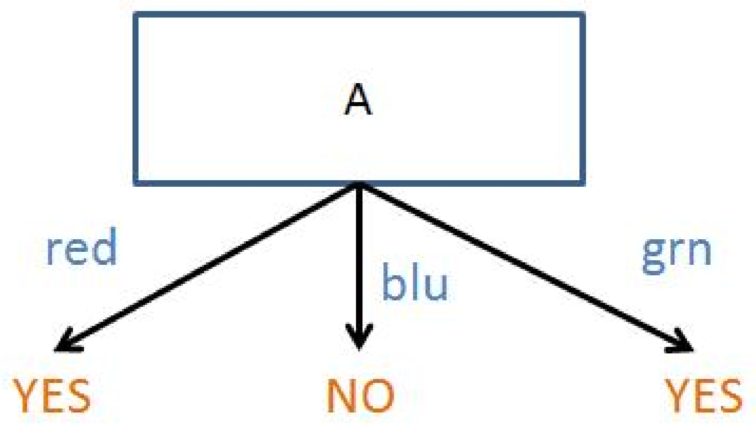
	A = red	**A = blu**	**A = grn**
X = yes	7	0	8
X = no	0	9	6

à Given this break down, what would be the most accurate classification to give the cases where A = red?

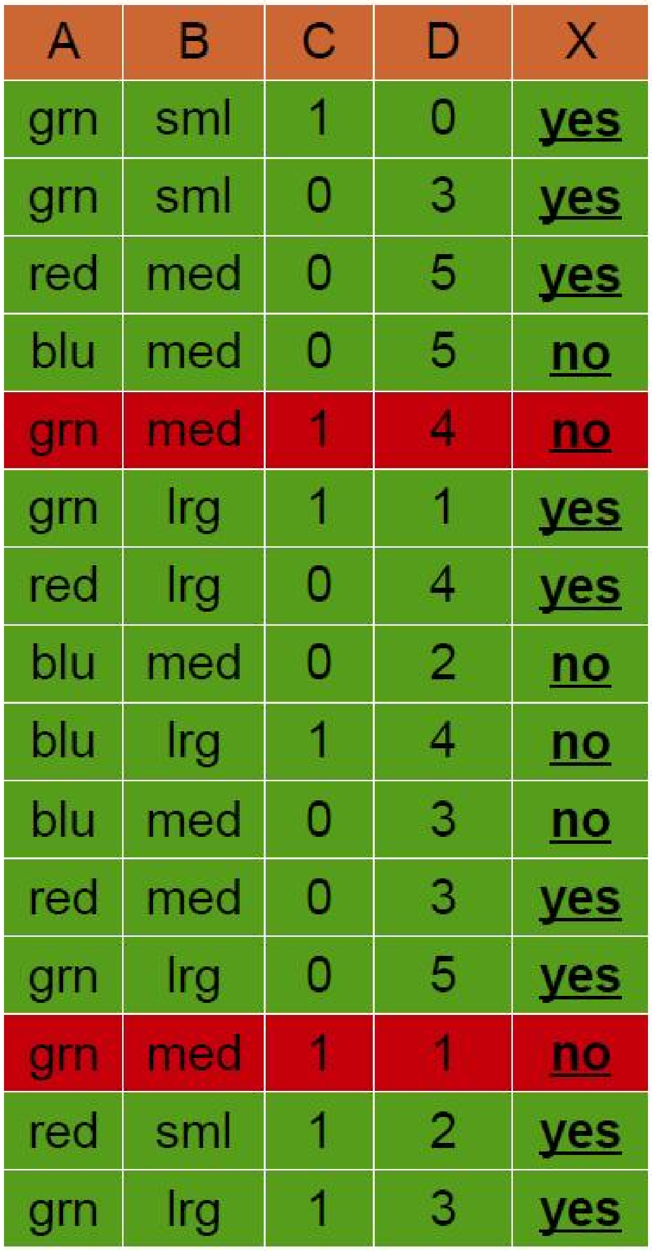
à Given this break down, what would be the most accurate classification to give the cases where A = blu?

à Given this break down, what would be the most accurate classification to give the cases where A = grn?

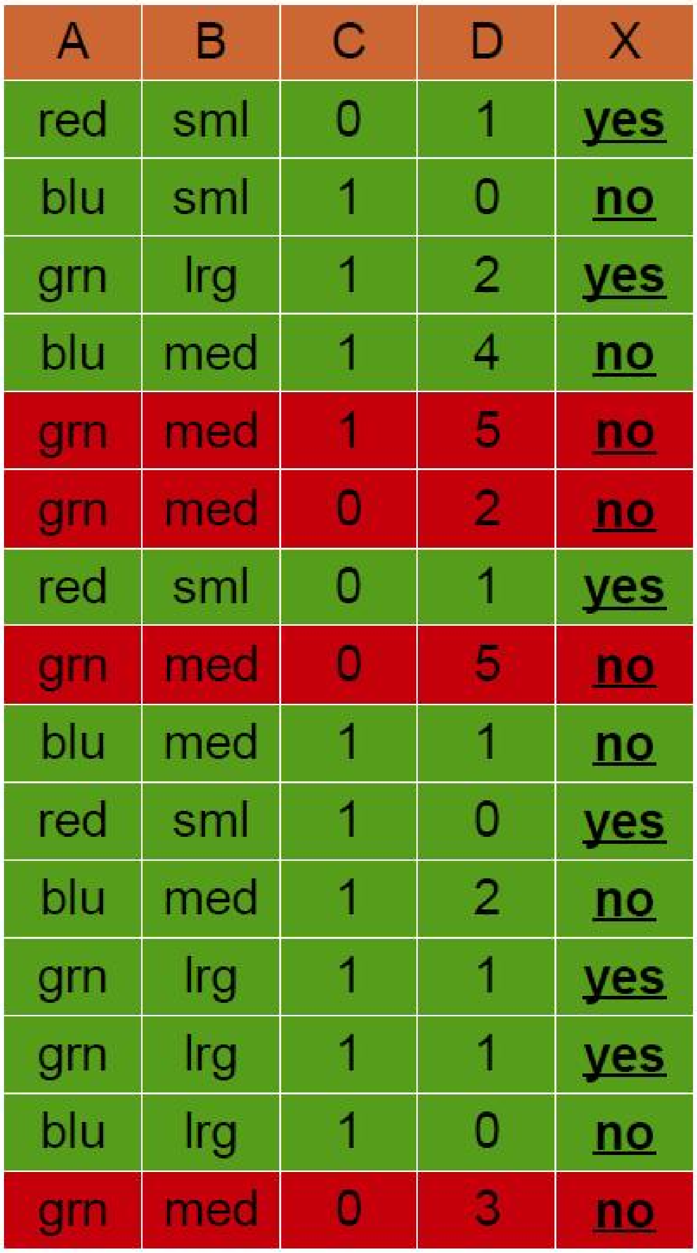
This would give us a single node d-tree looking like:



So, using this single node split, we would achieve an 80% accuracy since 6 / 30 samples are misclassified by saying that X = yes whenever A = grn



A	B	C	D	X
grn	sml	1	0	yes
grn	sml	0	3	yes
red	med	0	5	yes
blu	med	0	5	no
grn	med	1	4	no
grn	lrg	1	1	yes
red	lrg	0	4	yes
blu	med	0	2	no
blu	lrg	1	4	no
blu	med	0	3	no
red	med	0	3	yes
grn	lrg	0	5	yes
grn	med	1	1	no
red	sml	1	2	yes
grn	lrg	1	3	yes



A	B	C	D	X
red	sml	0	1	yes
blu	sml	1	0	no
grn	lrg	1	2	yes
blu	med	1	4	no
grn	med	1	5	no
grn	med	0	2	no
red	sml	0	1	yes
grn	med	0	5	no
blu	med	1	1	no
red	sml	1	0	yes
blu	med	1	2	no
grn	lrg	1	1	yes
grn	lrg	1	1	yes
blu	lrg	1	0	no
grn	med	0	3	no

Alright, so let's see if doing a single node split on another attribute can do better than 80%.

Shall we try to split on B?

	B = sml	**B = med**	**B = lrg**
X = yes	6	2	7
X = no	1	12	2

We actually do a bit better with the single var split on B if we choose to again use the plurality rule of classification, giving us only 5 / 30 misclassifications:

A	B	C	D	X
grn	sml	1	0	yes
grn	sml	0	3	yes
red	med	0	5	yes
blu	med	0	5	no
grn	med	1	4	no
grn	lrg	1	1	yes
red	lrg	0	4	yes
blu	med	0	2	no
blu	lrg	1	4	no
blu	med	0	3	no
red	med	0	3	yes
grn	lrg	0	5	yes
grn	med	1	1	no
red	sml	1	2	yes
grn	lrg	1	3	yes

A	B	C	D	X
red	sml	0	1	yes
blu	sml	1	0	no
grn	lrg	1	2	yes
blu	med	1	4	no
grn	med	1	5	no
grn	med	0	2	no
red	sml	0	1	yes
grn	med	0	5	no
blu	med	1	1	no
red	sml	1	0	yes
blu	med	1	2	no
grn	lrg	1	1	yes
grn	lrg	1	1	yes
blu	lrg	1	0	no
grn	med	0	3	no

But, as we've seen with our restaurant waiting example, we're not always interested in a single-node split on an attribute, but a multi-node one... why waste all that extra info?!

Now, of course, we have the question of which node to split on first... it's clear that order matters in deciding the most accurate classification!

ά **Entropy** is a measure of uncertainty of a random variable and is the fundamental unit of information theory.

Some things to note about entropy:

- It's measured in the bits it would require in order to represent all possible equally likely outcomes (manifest in the log_2 in the equation, to follow)
- A higher entropy means it's closer in value to a uniform distribution, like a coin flip (i.e., the higher the value, the more random)
- The lower the entropy, the more certain the outcome is.

```
 ; The equation for entropy is given by:
 H(V) = - Σ Pr(v_k) * log_2 [Pr(v_k)]
          k

; In other words, it's the sum, for all values
; v ∈ V, of the probability of seeing value
; v_k times log_2 of that probability
```

ὰ What is the entropy of a coin flip?

So, we see that a coin flip has 1 bit of entropy, since it only takes 1 bit (i.e., a 0 or 1 outcome uniformly distributed) to model the uncertainty.

Our goal in building decision trees, however, is to MINIMIZE uncertainty.

Minimizing the remaining uncertainty after a split is equivalent to maximizing **information gained** from that split.

ά **Information gain** is the expected reduction in entropy from splitting on some attribute.

We can model this in terms of **expected entropy,** which provides a metric for the entropy reducing contributions of each variable value.

```
; For value v of variable V, the
; positive classification cases p for that
; value v, as well as the negative classification
; cases n, then:

E_v(p, n) = -[p / (p + n)] * log_2[p / (p + n)]
          -  [n / (p + n)] * log_2[n / (p + n)]

; And the expected entropy is defined as:
Σ Pr(v) * E_v(p, n)
v ∈ V

; For all variable values v in variable V, sum
; up their entropies split on positive and negative
; classifications
```

Example

t Compute the expected entropy of splitting on variable A in our example from before (table replicated below for ease):

n = 30	A = red	A = blu	A = grn
X = yes	7	0	8
X = no	0	9	6

```
; Expected entropy of A is:
  Σ Pr(a) * E_a(p, n)
  a ∈ A

= Pr(A = red) * E_red(p, n) +
  Pr(A = blu) * E_blu(p, n) +
  Pr(A = grn) * E_grn(p, n)

; Step One:
; We know from the table that there are 30
; samples, so we can find the probabilities:

Pr(A = red) = 7 / 30
Pr(A = blu) = 9 / 30
Pr(A = grn) = 14 / 30
```

```
; Step Two:
; We can compute each of the intermediary
; values next:

E_red(p, n) = E_red(7, 0)
            = -[7 / (7 + 0)] * log_2[7 / (7 + 0)]
            -  [0 / (7 + 0)] * log_2[0 / (7 + 0)]
            = 0

E_blu(p, n) = E_blu(0, 9)
            = -[0 / (0 + 9)] * log_2[0 / (0 + 9)]
            -  [9 / (0 + 9)] * log_2[9 / (0 + 9)]
            = 0

E_grn(p, n) = E_grn(8, 6)
            = -[8 / (8 + 6)] * log_2[8 / (8 + 6)]
            -  [6 / (8 + 6)] * log_2[6 / (8 + 6)]
            = 0.9852
```

```
; Step Three
; Now, to find expected entropy, we plug back in:
  Pr(A = red) * E_red(p, n) +
  Pr(A = blu) * E_blu(p, n) +
  Pr(A = grn) * E_grn(p, n)

= (7 / 30) * 0 +
  (9 / 30) * 0 +
  (14 / 30) * 0.9852

= 0.4598 bits
```

Whew! What a pain! I hope I never have to do that again...

Example

t What is the expected entropy for splitting on attribute B in our example? Table replicated below for convenience.

n = 30	B = sml	B = med	B = lrg
X = yes	6	2	7
X = no	1	12	2

à Click for brief solution

SO, we see that the expected entropy of splitting on A is actually *less than* that of splitting on B, even though it's the less accurate choice by itself...

BUT, what if we then split its innacurate component (i.e., A = grn) on B? How would we decide to do that?

H The **decision tree algorithm** provides us a mean of automating the attribute splitting to minimize expected entropy.

Its basic steps are as follows:

1. Pick an attribute A (via the expected entropy, like we did above), and set that as the root
2. For each **value** of A, if there are attributes remaining on which to split, and classification is not perfect, then recurse on the subtree with A removed.
3. Otherwise, there were no more attributes on which to split, so simply maximize your accuracy by classifying with a plurality vote (i.e., satisfy the most positives or negatives even though you won't be perfect).

Applying this algorithm to our example, we would set variable A as the root.

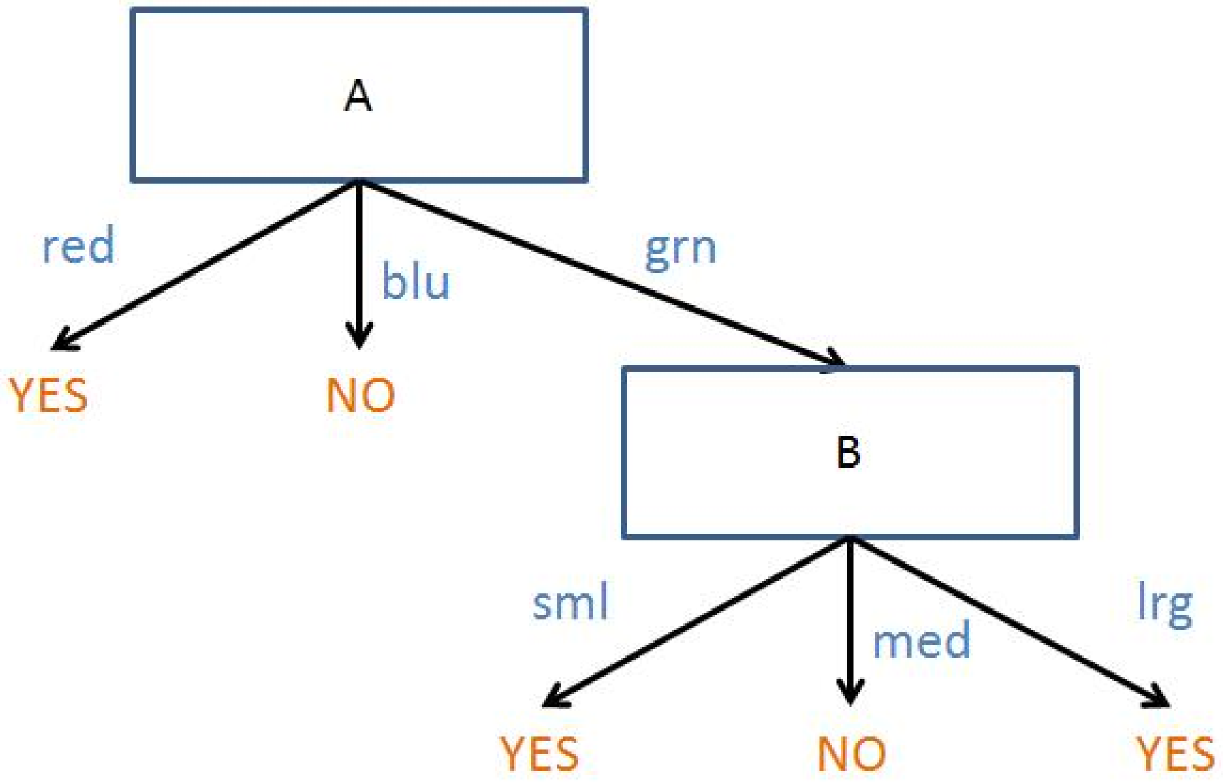
We note that A = red and A = blu are perfect accuracy, but A = grn is NOT.

So, we can recurse on the A = grn subtree (i.e., the cases in our sample in which A = grn) and try splitting on another attribute in the same manner... let's try B:

A	B	C	D	X
grn	sml	1	0	yes
grn	sml	0	3	yes
red	med	0	5	yes
blu	med	0	5	no
grn	med	1	4	no
grn	lrg	1	1	yes
red	lrg	0	4	yes
blu	med	0	2	no
blu	lrg	1	4	no
blu	med	0	3	no
red	med	0	3	yes
grn	lrg	0	5	yes
grn	med	1	1	no
red	sml	1	2	yes
grn	lrg	1	3	yes

A	B	C	D	X
red	sml	0	1	yes
blu	sml	1	0	no
grn	lrg	1	2	yes
blu	med	1	4	no
grn	med	1	5	no
grn	med	0	2	no
red	sml	0	1	yes
grn	med	0	5	no
blu	med	1	1	no
red	sml	1	0	yes
blu	med	1	2	no
grn	lrg	1	1	yes
grn	lrg	1	1	yes
blu	lrg	1	0	no
grn	med	0	3	no

On the 14 remaining cases where A = grn, splitting on B actually gives us a perfect fit! Let's see the decision tree that results:



And that's how you do decision trees!

PAC

Sometimes, answers are not so clean-cut, and a descent down a decision tree is insufficient for classification.

So, we'll return to our old friend the Bayesian network in order to help with classification; let's look at our motivating example:

Example
t GreeterBot 4000

Forney Industries is developing its most ambitious robotic project yet: GreeterBot 4000. GreeterBot 4000 needs to be able to walk around department stores greeting and answering questions of the customers. You've been tasked with programming GreeterBot 4000's ~~KillAllHumans~~ IdentifyHuman protocol, which must determine if some object in its environment is a human or not.

The main issue is that there are a variety of mannequins throughout the store that, although resembling humans, should not be greeted.

That said, GreeterBot 4000 is already equipped with input sensors that can assess the following attributes:

- **Movement (M):** we can assess whether an object is moving at, say, 3 discrete velocities: {not, slow, fast}
- **Height (H):** we can assess the height of an object at, say, 3 discrete heights relative to humans: {sml, med, lrg}
- **Speaking (S):** we have audio sensors capable of determining if some object is speaking a language: {0, 1}
- **Form (F):** we have some scoring algorithm that returns a score from 0 - 5 based on whether an object has human parts like a head, torso, arms, etc.

The goal, then, is to determine for each object we encounter assessed with the above 4 attributes, whether or not that object is a human.

(ignoring, of course, that this is just a re-labelling of the previous example with different semantics...)

The problem is that our classification might not be so simple as splitting on a single attribute, some confounding cases might be:

- A loudspeaker might announce the latest deals and be perceived as "speaking" even though it's not a human.
- A mannequin might have a good human form, and the right height, even though it's not human.
- A true human might be any height, have good human form (if our visual sensors aren't occluded), and simply not be speaking at a given moment.

So, needless to say, there are a variety of different cases that need to be handled, but different attributes confer some amount of human-ness.

Because of the amount of noise implicit in this problem, however, we may not be able to get away with clean classification divisions like with decision trees.

ά **Probably Approximately Correct (PAC)** classification strategies are those that, given a sufficiently large training set, can make a decision based on the most likely outcome of any witnessed evidence.

Frankly, I think the term "Probably Approximately Correct" sounds like a rationalizing scientist who isn't quite sure of his invention...

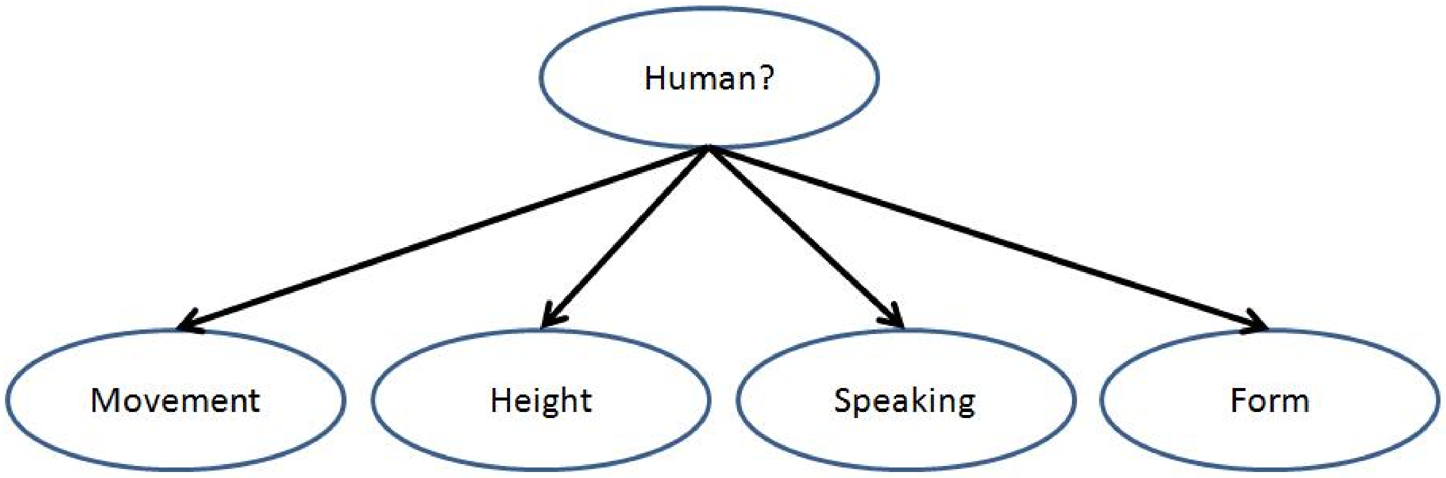
PAC strategies are good when we don't need a completely accurate answer where all cases are considered, but we have a good gist for when to say "Yes" and when to say "No"

ά The **Naive Bayes Classifier** is a PAC classification structure where we consider some class C to be the cause of witnessing some number of identifiers, F_i.

The Naive Bayes Classifier is a Bayesian network where we have a single cause (the class) being the reason for certain indications of that class.

It's called "Naive" because given the class, we assume that all of the indicators are conditionally independent.

For our example, the Naive Bayes structure might look like this:



This classifier says, "If you are a human, then you'll probably exibit some movement, a certain height distribution, the propensity for speech, and some amount of human form."

Furthermore, if we know you're human, then the indicators are all conditionally independent of one another because knowing that a human can speak tells us nothing more about a human's ability to move.

Being a Bayesian network, these indicating propensities are all implicit within the network's conditional probability tables.

Naive Bayes classification then simply asks: having observed some object with indicators F1, F2, ..., Fn, which classification is most likely?

Let's look at the probability statements of interest:

```
; We are interested in determining which
; classification in some set of class
; variable (C) values is most likely, having
; witnessed some object with indicators F

; So, we are interested in the table:
Pr(C | f1, f2, ..., fn)

; ...to see which value of C is most probable.
```

So, for our example, let's say we encountered an object exibiting the following:

Movement	Height	Speaking	Form

slow	med	1	4

Here we have some object that is moving slowly, of moderate height, that's speaking, and closely resembles the human form.

Chances are good that it's a human!

```
; In this sense, f1, f2, ..., fn is our evidence
; so we can simply use a familiar theorem to
; find our answer:

  Pr(C | f1, f2, ..., fn)
= [Pr(f1, f2, ..., fn | C) * Pr(C)] / Pr(f1, f2, ..., fn)

; Notice that Pr(C) is the prior on C, and since
; our evidence is conditionally independent given C,
; we can write

= [Pr(f1 | C) * Pr(f2 | C) * ... * Pr(fn | C) * Pr(C)]
  / Pr(f1, f2, ..., fn)
```

So, in our example, we would have each indicator f_i be a different attribute about our witnessed object, so our quantity of interest is:

```
  Pr(Human? | Movement = slow,
              Height = med,
              Speech = 1,
              Form = 4)

= [Pr(Movement = slow | Human?) *
   Pr(Height = med | Human?) *
   Pr(Speech = 1 | Human?) *
   Pr(Form = 4 | Human?) *
   Pr(Human?)]
 / Pr(Movement = slow, Height = med, Speech = 1, Form = 4)
```

Almost there! Now we just need one more element to complete our classification:

```
; Now all we need to find out is the Pr(f1, f2, ..., fn)
; We have a strategy to do this: Case analysis!

Pr(f1, f2, ..., fn)
  = Σ_c∈C Pr(f1, f2, ..., fn | c) * Pr(c)
  = Σ_c∈C Pr(f1 | c) * Pr(f2 | c) * ... * Pr(fn | c) * Pr(c)

; And so, substituting back into our original equation:
[Pr(f1 | c) * Pr(f2 | c) * ... * Pr(fn | c) * Pr(C)]
  / Pr(f1, f2, ..., fn)

= [Pr(f1 | C) * Pr(f2 | C) * ... * Pr(fn | C) * Pr(C)]
  / [Σ_c∈C Pr(f1 | c) * Pr(f2 | c) * ... * Pr(fn | c) * Pr(c)]
```

Notice: This gives us a *table* since we have the class variable C in the numerator; we really want to determine *which* value of C is most likely.

So, we simply compare the different values of C and classify based on whichever is most probable.

This outcome is defined by:

```
; The most likely class for all classes in C will
; be represented by:
argmax [Pr(f1 | c_i) * Pr(f2 | c_i) * ... * Pr(fn | c_i) * Pr(c_i)]
c_i∈C   / [Σ_c∈C Pr(f1 | c) * Pr(f2 | c) * ... * Pr(fn | c) * Pr(c)]

; Noting that the division is simply the normalizing constant:
∝ argmax [Pr(f1 | c_i) * Pr(f2 | c_i) * ... * Pr(fn | c_i) * Pr(c_i)]
   c_i∈C
```

...which looks intimidating, but just says, "Choose the value of class variable C that has the highest probability given the witnessed indicator values for the object"

In our example, we would be choosing between: C = Human? = {yes, no}