Learning

S. Sheerazuddin

May 5, 2013

An artificially intelligent agent is designed to look at the percepts (input) and decides the action to perform next. This decision is made using an agent function. The basic idea behind learning is that percepts can be used not only to decide the next action but also to modify (improve) the agent function. The simplest way to learn is via observations. The agent may observe its interaction with the environment and effects of its decision making process and learn (improve) its agent function. This is also known as **inductive learning**. As a simple case of inductive learning, we describe how to learn simple theories (formulas) in propositional logic. Later, we shall describe some advance learning techniques, including statistical learning and reinforcement learning.

1 Forms of Learning

The agent function of a learning agent is divided into two parts, a **performance element** which decides what action to take and a **learning element** that modifies the performance element so that it improves the decision-making of the agent. Had there been no learning in the agent, the performance element would encompass the whole agent function.

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

- **which components of the performance element are to be learnt**: The learning technique will depend on the type of the performance element. For example, in a reflex agent, the agent function is map from (conditions on) the current state to set of actions whereas a model-based reflex agent uses the information about how the world evolves and what are the results of possible actions to decide the next action to perform. Clearly, there should be two different way of learning these two different performance elements.

- **what feedback is available to learn these components**: Depending on the type of feedback available for learning, there are three kind of learning techniques
  - **Supervised Learning**: Learning a function from examples of input and output. Example, learning decision trees from examples (input output pairs).
  - **Unsupervised Learning**: Learning patterns in the input when no specific output values are supplied. Example, learning Bayesian networks from given data.
  - **Reinforcement Learning**: Learning from rewards and punishments (reinforcement).
- **what representation is used for the components**: There are different learning techniques for different representations of performance elements. For example, there are techniques to learn propositional and first order theories which are used by logical agents. Also, there are techniques to learn Bayesian networks which are used by probabilistic systems with uncertain knowledge (decision-theoretic agents).

2 Supervised and Unsupervised Learning

Applications in which the training data comprises examples of the input vectors along with their corresponding target vectors are known as supervised learning problems. Examples of supervised learning are: learning decision trees and explanation-based learning. There are two steps in supervised learning:

- **training step**: Learn classifier/regressor from training data and
- **the prediction step**: Assign class labels/functional values to test data.

Unsupervised learning refers to the problem of trying to find hidden structure in unlabelled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution. This distinguishes unsupervised learning from supervised learning and reinforcement learning.

Approaches to unsupervised learning include:

- clustering (e.g., k-means, mixture models, hierarchical clustering),

- blind signal separation using feature extraction techniques for dimensionality reduction (e.g., Principal component analysis, Independent component analysis, Non-negative matrix factorization, Singular value decomposition), and

- among neural network models, the self-organizing map (SOM) and adaptive resonance theory (ART) are commonly used unsupervised learning algorithms.

Supervised learning is the type of learning that takes place when the training instances are labelled with the correct result, which gives feedback about how learning is progressing. This is akin to having a supervisor who can tell the agent whether or not it was correct. In unsupervised learning, the goal is harder because there are no pre-determined categorizations.

2.1 Supervised Learning

Supervised learning is fairly common in classification problems because the goal is often to get the computer to learn a classification system that we have created. Digit recognition, once again, is a common example of classification learning. More generally, classification learning is appropriate for any problem where deducing a classification is useful and the classification is easy to determine. In some cases, it might not even be necessary to give pre-determined classifications to every instance of a problem if the agent can work out the classifications for itself. This would be an example of unsupervised learning in a classification context.

Supervised learning is the most common technique for training neural networks and decision trees. Both of these techniques are highly dependent on the information given by the pre-determined classifications. In the case of neural networks, the classification is used to determine the error of the network and then adjust the network to minimize it, and in decision trees, the classifications are used to determine what attributes provide the most information that can be used to solve the classification puzzle.

Speech recognition using hidden Markov models and Bayesian networks relies on some elements of supervision as well in order to adjust parameters to, as usual, minimize the error on the given inputs.

Notice something important here: in the classification problem, the goal of the learning algorithm is to minimize the error with respect to the given inputs. These inputs, often called the "training set", are the examples from which the agent tries to learn. But learning the training set well is not necessarily the best thing to do. For instance, if I tried to teach you exclusive-or,

but only showed you combinations consisting of one true and one false, but never both false or both true, you might learn the rule that the answer is always true. Similarly, with machine learning algorithms, a common problem is over-fitting the data and essentially memorizing the training set rather than learning a more general classification technique.

As you might imagine, not all training sets have the inputs classified correctly. This can lead to problems if the algorithm used is powerful enough to memorize even the apparently "special cases" that don't fit the more general principles. This, too, can lead to overfitting, and it is a challenge to find algorithms that are both powerful enough to learn complex functions and robust enough to produce generalizable results.

2.2 Unsupervised learning

Unsupervised learning seems much harder: the goal is to have the computer learn how to do something that we don't tell it how to do! There are actually two approaches to unsupervised learning. The first approach is to teach the agent not by giving explicit categorizations, but by using some sort of reward system to indicate success. Note that this type of training will generally fit into the decision problem framework because the goal is not to produce a classification but to make decisions that maximize rewards. This approach nicely generalizes to the real world, where agents might be rewarded for doing certain actions and punished for doing others.

Often, a form of reinforcement learning can be used for unsupervised learning, where the agent bases its actions on the previous rewards and punishments without necessarily even learning any information about the exact ways that its actions affect the world. In a way, all of this information is unnecessary because by learning a reward function, the agent simply knows what to do without any processing because it knows the exact reward it expects to achieve for each action it could take. This can be extremely beneficial in cases where calculating every possibility is very time consuming (even if all of the transition probabilities between world states were known). On the other hand, it can be very time consuming to learn by, essentially, trial and error.

But this kind of learning can be powerful because it assumes no pre-discovered classification of examples. In some cases, for example, our classifications may not be the best possible. One striking example is that the conventional wisdom about the game of backgammon was turned on its head when a series of computer programs (neuro-gammon and TD-gammon) that learned through unsupervised learning became stronger than the best hu-

man chess players merely by playing themselves over and over. These programs discovered some principles that surprised the backgammon experts and performed better than backgammon programs trained on pre-classified examples.

A second type of unsupervised learning is called clustering. In this type of learning, the goal is not to maximize a utility function, but simply to find similarities in the training data. The assumption is often that the clusters discovered will match reasonably well with an intuitive classification. For instance, clustering individuals based on demographics might result in a clustering of the wealthy in one group and the poor in another.

Although the algorithm won't have names to assign to these clusters, it can produce them and then use those clusters to assign new examples into one or the other of the clusters. This is a data-driven approach that can work well when there is sufficient data; for instance, social information filtering algorithms, such as those that Amazon.com use to recommend books, are based on the principle of finding similar groups of people and then assigning new users to groups. In some cases, such as with social information filtering, the information about other members of a cluster (such as what books they read) can be sufficient for the algorithm to produce meaningful results. In other cases, it may be the case that the clusters are merely a useful tool for a human analyst. Unfortunately, even unsupervised learning suffers from the problem of overfitting the training data. There's no silver bullet to avoiding the problem because any algorithm that can learn from its inputs needs to be quite powerful.

2.3 Summary

Unsupervised learning has produced many successes, such as world-champion calibre backgammon programs and even machines capable of driving cars! It can be a powerful technique when there is an easy way to assign values to actions. Clustering can be useful when there is enough data to form clusters (though this turns out to be difficult at times) and especially when additional data about members of a cluster can be used to produce further results due to dependencies in the data.

Classification learning is powerful when the classifications are known to be correct (for instance, when dealing with diseases, it's generally straightforward to determine the design after the fact by an autopsy), or when the classifications are simply arbitrary things that we would like the computer to be able to recognize for us. Classification learning is often necessary when the decisions made by the algorithm will be required as input somewhere

else. Otherwise, it wouldn't be easy for whoever requires that input to figure out what it means.

Both techniques can be valuable and which one you choose should depend on the circumstances–what kind of problem is being solved, how much time is allotted to solving it (supervised learning or clustering is often faster than reinforcement learning techniques), and whether supervised learning is even possible.

3 Inductive Learning

An algorithm for deterministic supervised learning is given as input the correct value of the unknown function for particular inputs and must try to recover the unknown function or something close to it. More formally, we say that an **example** is a pair $(x, f(x))$, where x is the input and $f(x)$ is the output of the function applied to x. The task of **pure inductive inference** (or **induction**) is as follows:

Given a collection of examples of f, return a function h that approximates f.

The function h is called a **hypothesis**. A good hypothesis will **generalize** well——that is, will predict unseen examples correctly.

Consider the familiar problem of fitting a function of a single variable to some data points. The examples are $(x, f(x))$ pairs, where both x and $f(x)$ are real numbers. We can fix the **hypothesis space H**——the set of hypothesis we will consider——to be the set of polynomials of degree at most k. The polynomial which exactly fits the data is called a **consistent** hypothesis. In general there can me more than one hypothesis (in this case, polynomials) consistent with the data. The principle of **Ockham's razor** says that if there more than one hypotheses consistent with the data then choose that one which is the simplest. Defining simplicity is not easy, but it seems reasonable to say that a degree-1 polynomial is simpler than a degree-12 polynomial.

3.1 Learning Decision Trees

Decision tree induction is one of the simplest, and successful forms of learning algorithms. It serves as a good introduction to the area of inductive learning, and is easy to implement.

A **decision tree** takes as input an object or situation described by a set of **attributes** and returns a decision—the predicted value for the input. The input attributes can be discrete or continuous. For our discussion, we

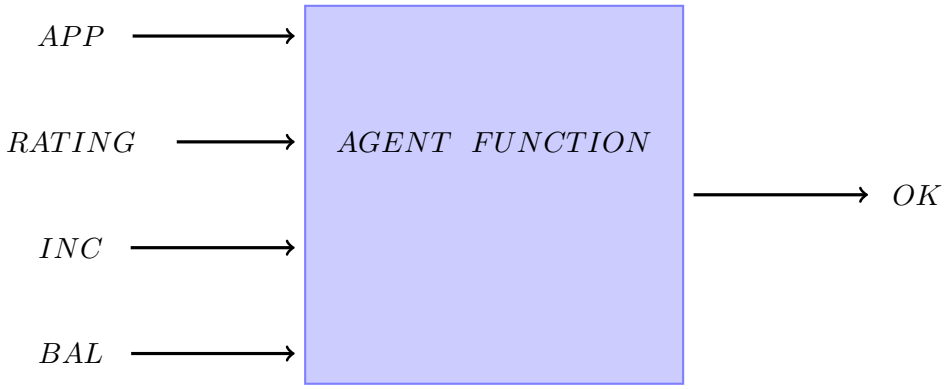

Figure 1: Rule-based System

assume boolean (discrete) inputs. The output attribute can also be discrete or continuous; learning a discrete-valued function is called **classification** learning; learning a continuous function is called **regression**. We will concentrate on *boolean* classification, wherein each example is classifies as true (**positive**) or false (**negative**).

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labelled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached.

It turns out that every propositional formula can be written, in an equivalent form, as a decision tree. The propositional formula $(p_1 \Rightarrow p_2) \wedge p_3$ can be represented as a decision tree given in the Figure 2. Note that every path in the decision tree of the formula $(p_1 \Rightarrow p_2) \wedge p_3$ corresponds uniquely to a row in the truth table of the formula $(p_1 \Rightarrow p_2) \wedge p_3$ and vice versa. Clearly, given the formula or its truth table we can easily construct the full decision tree of that formula. What if we are only provided a part of the truth table? Can we still construct a decision tree for the formula?

Consider the expert system given the Figure 1 which takes four boolean inputs APP (appraisal on the collateral greater than loan amount), RATING (good credit rating for the applicant), INC (income exceeds applicant's expenses) and BAL (applicant has excellent balance sheet) and decides whether to approve the loan application (OK). We are not given the rule (the corresponding propositional formula over APP, RATING, INC and BAL) but a part of the truth table given below. Can we construct a decision

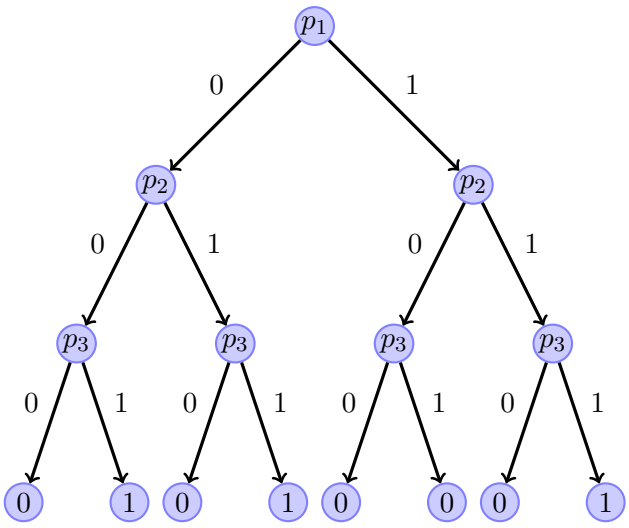

Figure 2: Decision tree for $(p_1 \Rightarrow p_2) \wedge p_3$

tree which **covers** all the given inputs and **predicts** those which are not given?

SNo	APP	RATING	INC	BAL	OK
X_1	1	0	0	1	0
X_2	0	0	1	0	0
X_3	1	1	0	1	1
X_4	0	1	1	1	1
X_5	0	1	1	0	0
X_6	1	1	1	0	1
X_7	1	1	1	1	1
X_8	1	0	1	0	0
X_9	1	1	0	0	0

A row in the partial truth table is called an **example** and the task is to **induce** a decision tree from a given set of examples. An example for a Boolean decision tree consists of a vector of input attributes, X, and a single Boolean output value y. A set of examples $(X_1, y_1)(X_2, y_2) \cdots (X_9, y_9)$ is shown in the table above. The **positive examples** are the ones in which the goal y (in this case OK) is true (X_3, X_4, X_6, X_7); the **negative examples** are the ones in which it is false $(X_1, X_2, X_5, X_8, X_9)$. The complete set of examples is called the **training set**.

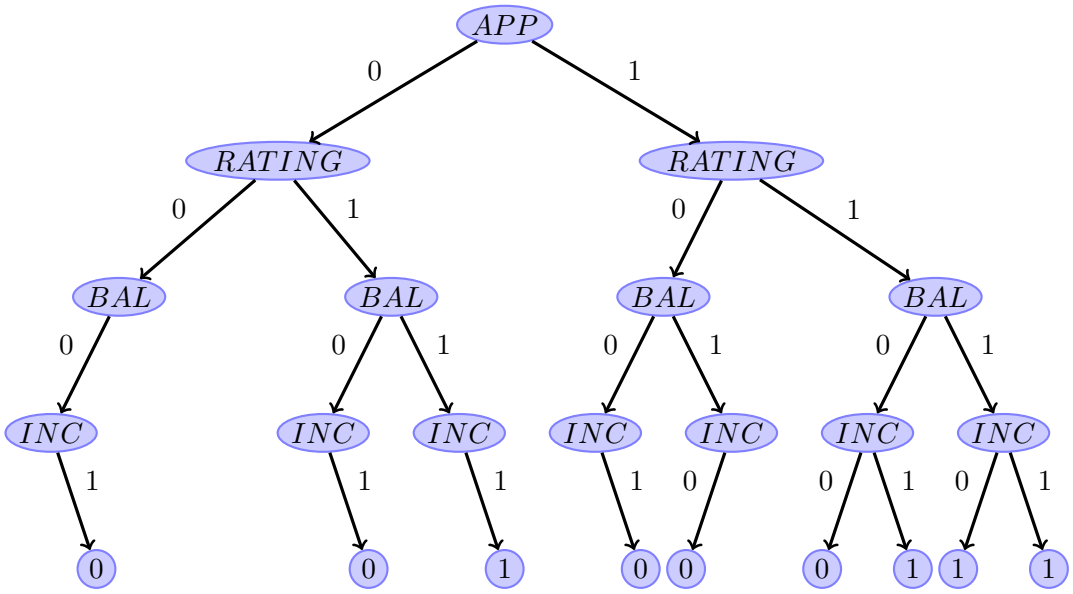

Figure 3: Decision tree that memorizes the input

The problem of finding a decision tree that agrees with the training set might seem difficult, but in fact there is a trivial solution, as given in the the Figure 3. We could simply construct a decision tree that has one path to a leaf for each example (row), where the path tests each attribute in turn and follows the value for the example and the leaf has the classification of the example. When given the same example again, the decision tree will come up with the right classification. Unfortunately, it will not have much to say about any other cases.

The problem with this tree is that it just memorizes the observations. It does not extract any patterns from the examples, we can not expect it to be able to extrapolate to examples it has not seen.

The basic idea behind the DECISION-TREE-LEARNING algorithm is to test the most important attribute first. By "most-important", we mean the one that makes most difference to the classification of examples. Thus, we are expected to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be small.

Figure 4 shows how the algorithms get started. We are given 9 training examples, which we classify into positive and negative sets. We then decide

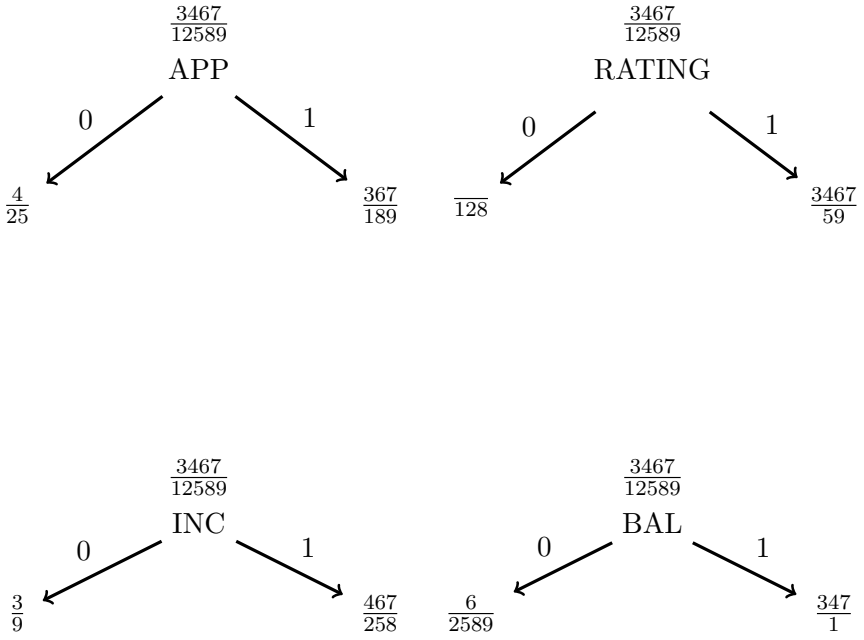

Figure 4: Classification of Examples by Attributes: Level 0

which attribute to use as the first test in the tree. Looking at the figure 4 we infer that INC is a poor attribute, because it leaves us with two outcomes each of which have the same number of positive and negative examples. On the other hand, RATING is a fairly important attribute , because if the value is 0, then we are left with an example set for which we can definitely say *No*. If the value is 1, we are left with a mixed set of examples. The other attributes BAL and APP are not as "important" as RATING as the splitting of training set remains mixed for both the values.

In general, after the first attribute splits up the examples, each outcome is a new decision tree learning problem in itself, with fewer examples and one fewer attributes. The application of DECISION-TREE-LEARNING algorithm on the given training set is explained pictorially in the Figures 5, 6, 7 and 8. There are four cases to consider for these recursive problems:

1. If there are some positive and negative examples, then choose the best attribute to split them.

2. If all the remaining examples are positive (or all negative), then we

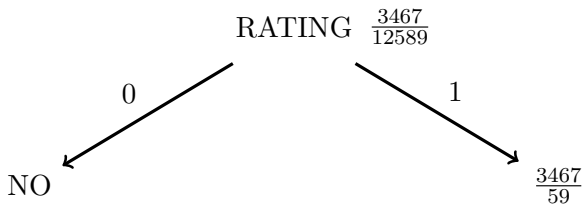

Figure 5: Partial Decision Tree: First Attribute

are done: we can answer *Yes* or *No*.

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

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

The final tree produced by the algorithm applied to the 9-example data set is shown in Figure 8. Note that this tree can be used to predict those combinations of attribute values which are not in the training set. Consider the vectors $(0, 0, 0, 0)$ and $(1, 0, 1, 1)$, the output for each case is 0 as the decision tree gives 0 when RATING has value 0.

Choosing Attribute Tests

The scheme used in decision tree learning is designed to minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. A perfect attribute divides the examples into sets that are all positive or all negative. A really useful attribute leaves the example sets with roughly the same proportion of positive and negative examples.

Therefore, we need a formal measure of "fairly good" and "really useless" in order to choose the appropriate attribute to split the rest of the examples

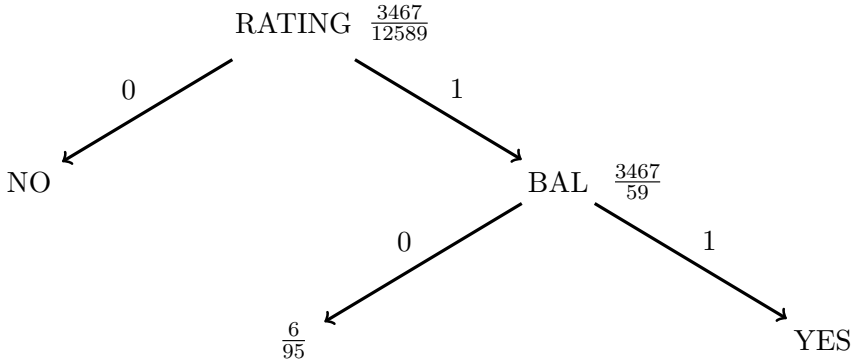

Figure 6: Partial Decision Tree: Second Attribute

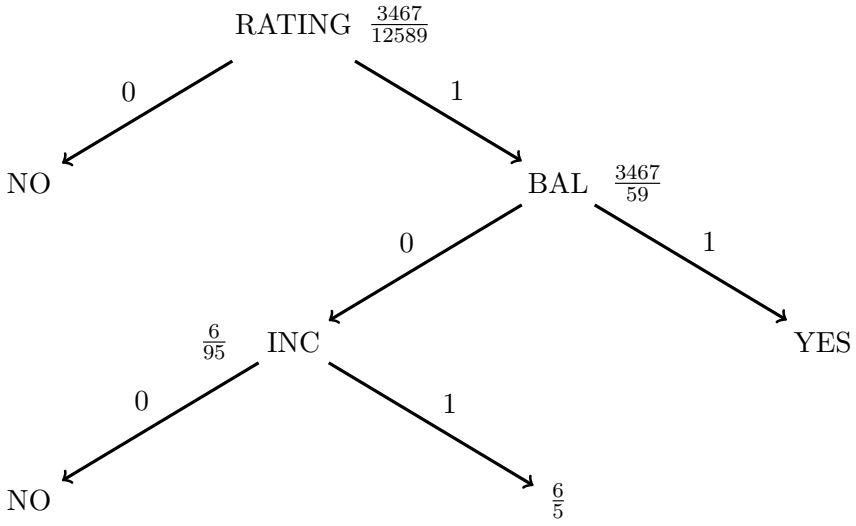

Figure 7: Partial Decision Tree: Third Attribute

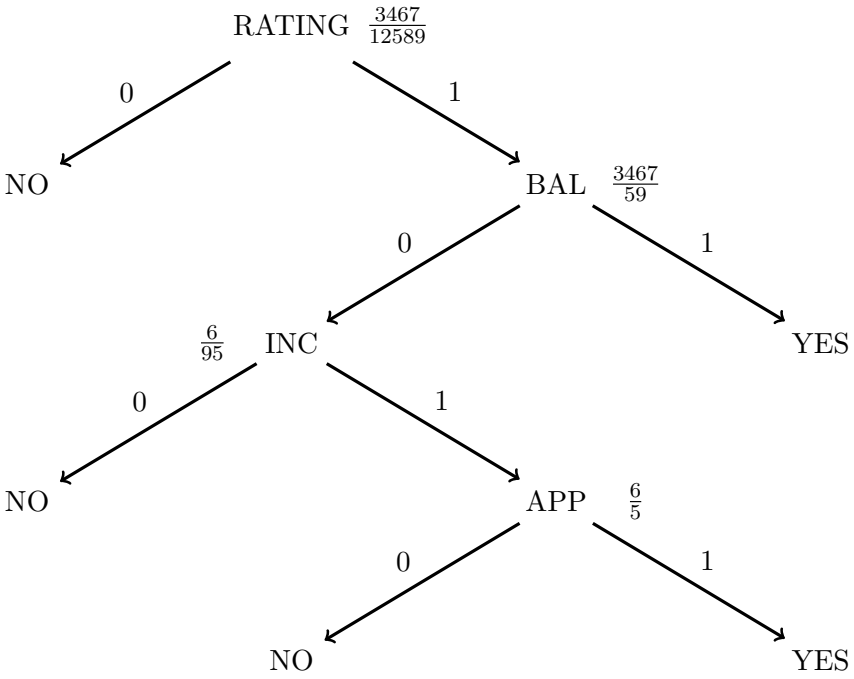

Figure 8: Partial Decision Tree: Fourth Attribute

in the decision tree construction. The measure should have its maximum value when the attribute is perfect and its minimum value when the attribute is of no use at all.

One suitable measure is the expected amount of **information** provided by the attribute, where we use the term in the mathematical sense first defined by Shannon and Weaver. To understand the notion of information, think about it as providing the answer to a question—-for example whether a coin will come up heads. The amount of information contained in the answer depends on one's prior knowledge. The less you know, the more information is provided.

Information theory measures information content in **bits**. One bit of information is enough to answer a yes/no question about which one has no idea, such as the flip of a fair coin. In general, if the possible answers v_i have probabilities $P(v_i)$, then the information content I of the actual answer is given by

$$I(P(v_1)\dots P(v_n)) = \sum_{i=1}^{n} -P(v_i)\cdot log_2 P(v_i).$$

To check this equation, for the tossing of a fair coin, we get

$$I(\frac{1}{2},\frac{1}{2}) = -\frac{1}{2}log_2\frac{1}{2} - \frac{1}{2}log_2\frac{1}{2} = 1\ bit.$$

If the coin is loaded to give 99% heads, we get $I(1/100, 99/100) = 0.08\ bits$, and as the probability of heads goes to 1, the information of the actual answer goes to 0.

For decision tree learning, the question that needs answering is; for a given example what is the correct classification? A correct decision tree will answer this question. An estimate of the probabilities of the possible answers before any of the attributes have been tested is given by the proportion of positive and negative examples in the training set. Suppose the training set contains p positive examples and n negative examples. Then an estimate of the information contained in a correct answer is

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

In our case, the training set has $p = 4$ positive examples and $n = 5$ negative examples which gives us .99 *bits*. Now a test on a single attribute will not usually tell us this much information, but it will give us part of it. We can measure exactly how much by looking at how much information we still need after the attribute test.

Any attribute A divides the training set E into E_1 and E_2 according to the two values of A. Each subset E_i has p_i positive values and n_i negative values, so if we go along that branch, we will need an additional $I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$ bits of information to answer the question. A random chosen example from the training set has the ith value for the attribute with probability $\frac{p_i+n_i}{p+n}$, so on average, after testing attribute A, we will need

$$Remainder(A) = \sum_{i=1}^{2} \frac{p_i + n_i}{p + n} I(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$$

bits of information to classify the example. The information gain form the attribute test is the difference between the original information requirement and the new requirement:

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

It turns out that, for our example, $Gain(RATING)$ is more that $Gain(BAL)$, $Gain(APP)$ and $Gain(INC)$ and therefore, RATING is chosen as the root by the decision-tree learning.

4 Explanation based Learning

We have already seen inductive learning where the learning routine takes a set of observations (**training set**) as input and constructs a hypothesis which **covers** the training set and correctly **predicts** the input in the **test set**. This scheme does not use any prior knowledge and every time the learning routine is given a training set as input it has to construct a hypothesis afresh without using any of its experience with similar training sets. We can depict inductive learning schematically as in Figure 9. We have also seen that this kind of learning turns out to be searching for an appropriate hypothesis in the hypothesis space.

Let $Descriptions$ denote the conjunction of all example descriptions in the training set, and let $Classifications$ denote the conjunction of all example classifications. Then, a hypothesis that "explains the observations" must satisfy the following semantic entailment constraint:

$$Hypothesis \wedge Descriptions \models Classifications$$

Pure inductive learning means solving this constraint, where $Hypothesis$ is drawn from some predefined hypothesis space.

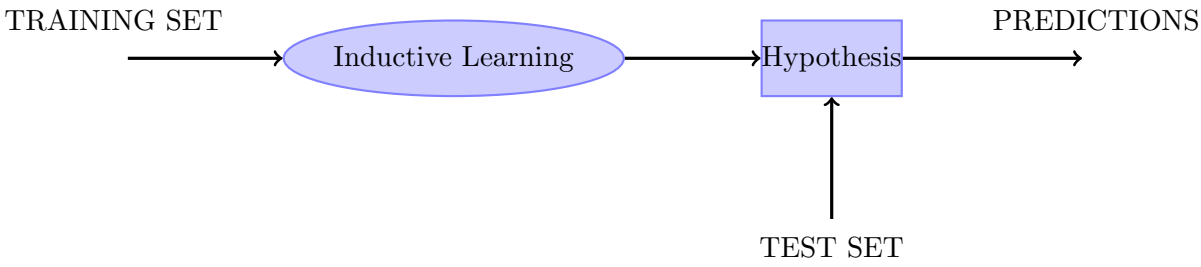

Figure 9: Pure Inductive Learning

The simple knowledge-free picture of inductive learning persisted till the 1980's. The modern approach is to design agents that already know something and are trying to learn more. The general idea is shown schematically in Figure 10. It turns out that learning with background knowledge is much faster than pure induction learning.

Explanation-based learning (EBL) is the simplest learning technique which uses **prior knowledge** to learn a **hypothesis** from a single **observation**. Consider a hypothetical situation of how cavemen learnt to cook small animals over a fire. Once, a really smart caveman, Zoag, came up with the idea of roasting his lizard at the end of a pointed stick. His compatriots generalized by **explaining** the success of the pointed stick: it supports the lizard while keeping the hand away from the fire. From this explanation, they can infer a general rule: that any long, rigid sharp object can be used to toast small, soft-bodied animals.

Notice that the general rule follows logically from the background knowledge possessed by the cavemen. Hence, EBL satisfies the following entailment constraint:

$$Background \models Hypothesis$$

Thereafter, the hypothesis can be used to classify descriptions in the test set.

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

Let us formally describe EBL with an example. Explanation-based learning takes three inputs, in some cases four:

- A **training example**, also called facts or situation description.
- A **goal concept**. It is an FO formula or sentence which gives a high level description of what the program is supposed to learn.

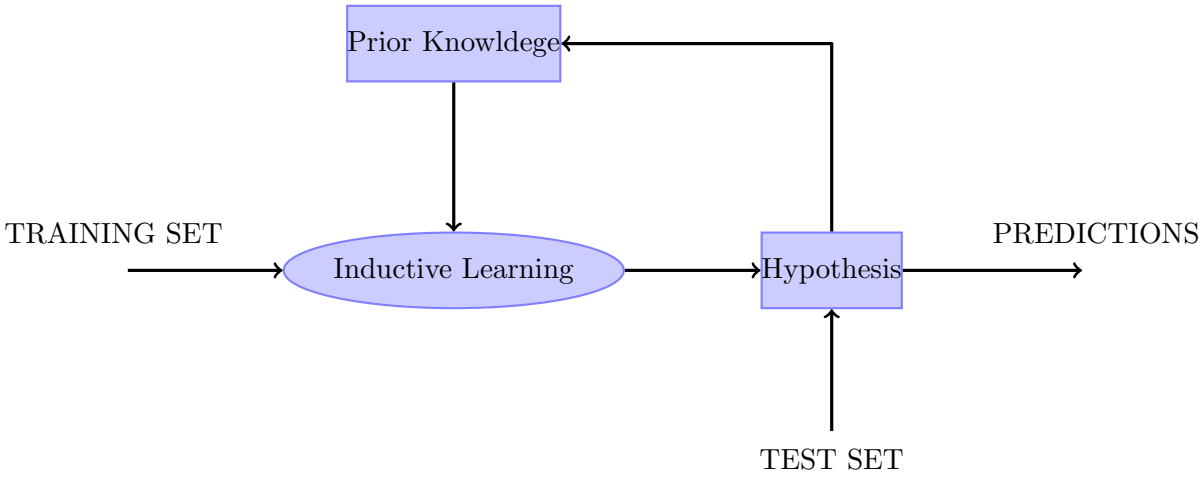

Figure 10: Knowledge-based Inductive Learning

- A **domain theory**. A set of inference rules that describe relationships between objects and actions in the domain.

The fourth (optional) input is an **operational criterion** which describes those concepts (terms) that are usable (can appear in the generalized used). We ignore this in our subsequent discussion.

Given the input, the EBL routine computes a generalization of the training example that

- describes the goal concept and
- satisfies the operation criteria.

This is done in two steps:

- **Explanation**: It is the justification of the goal in terms of the facts using the rules. Clearly, it is nothing but a proof of the goal from the facts. This step is depicted in the Figure 11.
- **Generalization**: The explanation is generalized by adding a rule $facts \Rightarrow goal$ to the set of rules. This step is depicted in the figure 12.

Consider the following EBL example with

$$facts = \{Human(John), Happy(John), Male(John)\}$$

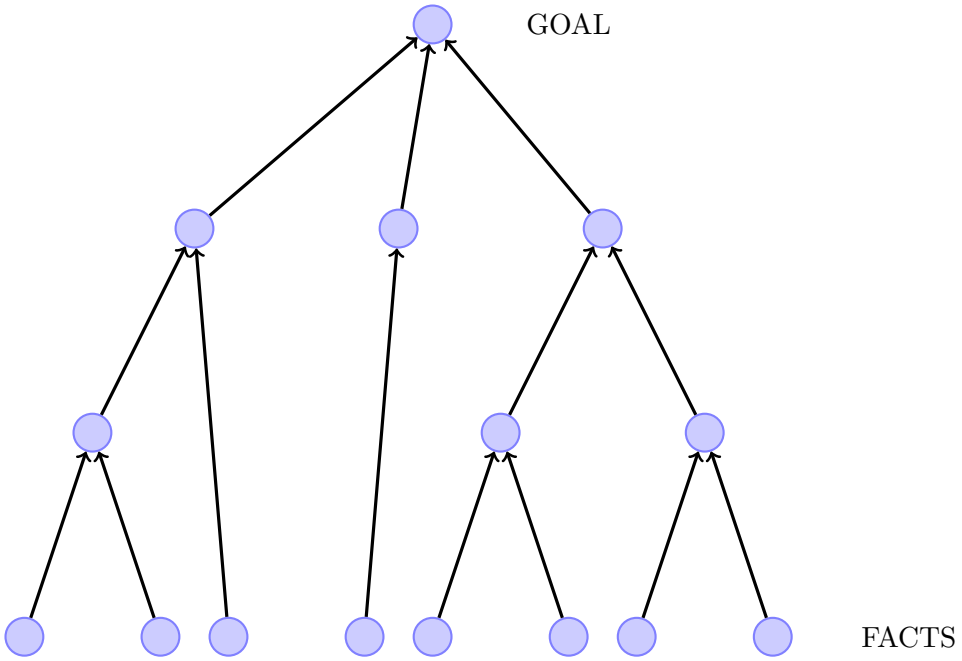

Figure 11: An Explanation of the Facts

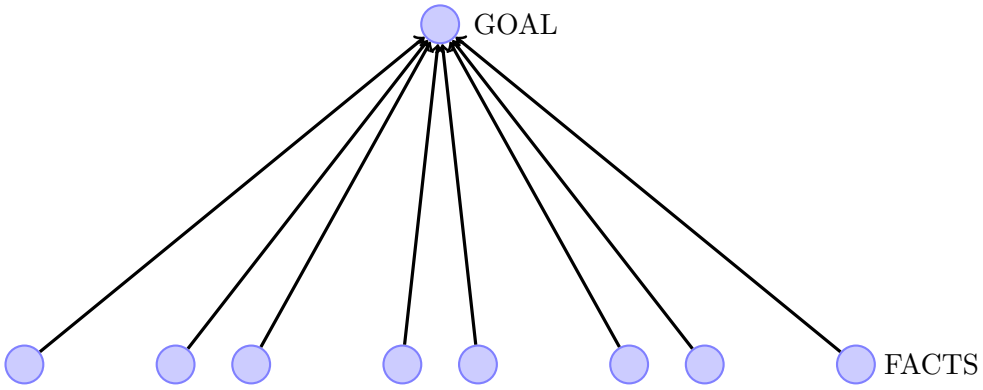

Figure 12: An Instance of Learning

$$goal\ concept = Likes(John, John)$$

and initial domain theory containing the following rules:

1. $Knows(x, y) \wedge Nice - person(y) \Rightarrow Likes(x, y)$
2. $Animate(z) \Rightarrow Knows(z, z)$
3. $Human(u) \Rightarrow Animate(u)$
4. $Friendly(v) \Rightarrow Nice - person(v)$
5. $Happy(w) \Rightarrow Nice - person(w)$

Note that there are implicit universal quantifiers in the rules, for example rule 1 is actually

$$(\forall x)(\forall y)\big(Knows(x, y) \wedge Nice - person(y) \Rightarrow Likes(x, y)\big)$$

which is true for every possible instances of x and y.

Now, the goal concept $Likes(John, John)$ can be explained from the facts $\{Human(John), Happy(John), Male(John)\}$ and we can draw a proof tree as given in the Figure 13. Notice that we have not used $Male(John)$ in our explanation. It turns out that any system of proof can used in this step, for example, resolution, resolution refutation, or Henkin style proofs which we have used here. For the given case we have liberally used **modus ponens** and the rule of **generalization** of FOL.

After we have explained the goal from the facts, we notice that this explanation is quite general and shall work even if we replace $John$ by $Mike$ or $Joseph$. Clearly, it means that if $Human(John) \wedge Happy(John) \Rightarrow Likes(John, John)$ holds then $(\forall x)\big(Human(x) \wedge Happy(x) \Rightarrow Likes(x, x)\big)$ holds too. Therefore, we can add $\big(Human(x) \wedge Happy(x) \Rightarrow Likes(x, x)\big)$ to the domain theory.

5 Statistical Learning

In the case of inductive learning, we learn a hypothesis (propositional theories as a special case) from a given set of specification examples (training set) which correctly classifies the examples in the test set. Inductive learning is essentially searching for an appropriate hypothesis in the hypothesis space. If the specification examples (data) are defined over n propositional variables then the size of hypothesis space is of the order of 2^{2^n}.

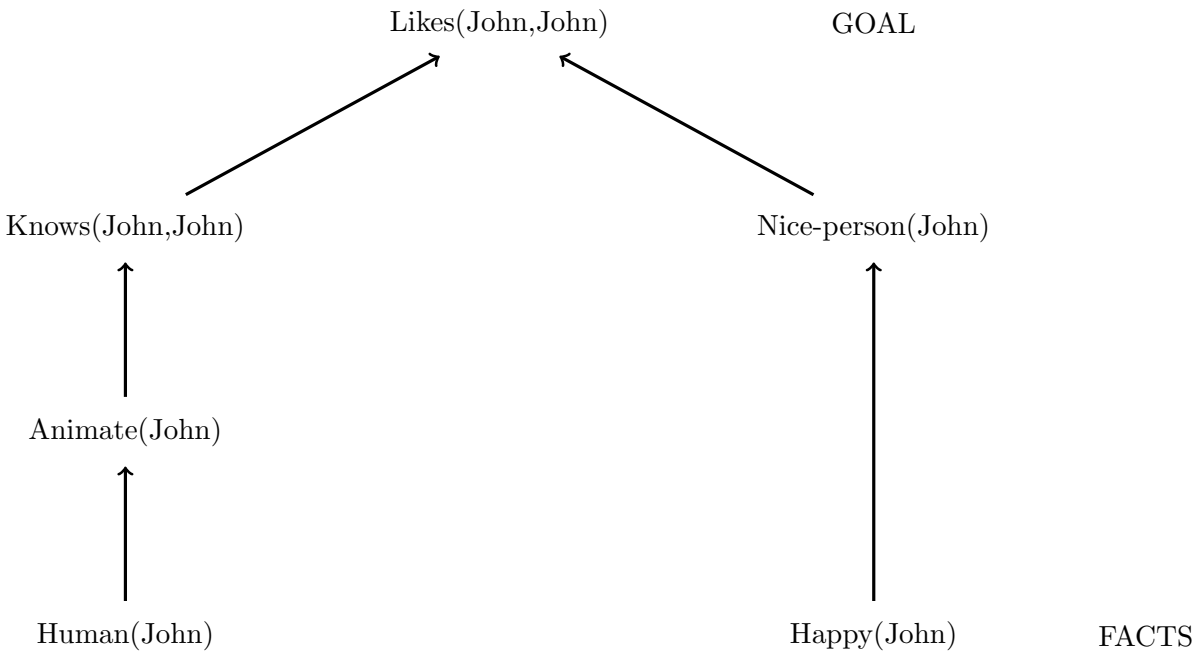

Figure 13: An Example proof (explanation)

In the world of uncertain knowledge, statistical learning is defined as finding posterior probabilities of the hypotheses in the hypotheses space given their prior probabilities and the likelihood of data for each hypothesis.

Consider a very simple example where have Surprise candies which come in two **flavours**: cherry and lime. Each candy is wrapped in some opaque wrapper regardless of the flavour. The candy is sold in **very** large bags, of which there are known to be of five kinds indistinguishable from outside:

1. h_1 : 100% cherry
2. h_2 : 75% cherry and 25% lime
3. h_3 : 50% cherry and 50% lime
4. h_4 : 25% cherry and 75% lime
5. h_5 : 100% lime

In general, there can be infinitely many types of bags depending on the proportions of cherry and lime but we consider only five them for the purpose of illustration. The bags denote **hypotheses** and candies **data**.

Given a new bag of candy, the random variable H denotes the type of the bag, with possible values h_1 through h_5. Note that H is not directly observable. The bags are too large and we can never find exactly the proportion of cherry and lime candies. We can guess (predict) the value of H by sampling the candies in the bag.

We open the pieces of candy and reveal the data——$D_1, D_2, \cdots, D_N$, where each D_i is a random variable with possible values *cherry* and *lime*. The basic task is to predict the flavour of the next candy picked. This is accomplished if we can calculate the posterior probability of each hypothesis, given the data, $\mathbf{P}(H|\mathbf{d})$, where $\mathbf{d} = d_1 d_2 \cdots d_N$ and for all i, $d_i \in \{cherry, lime\}$. Therefore, in **Bayesian learning**, given the prior probability distribution $\mathbf{P}(H)$ and likelihood of data under each hypothesis $\mathbf{P}(\mathbf{d}|H)$, we are asked to compute $\mathbf{P}(H|\mathbf{d})$.

For most cases, in particular the given candy example, we assume that the observations for computation of the likelihood of data are **iid** (independent and identically distributed). That is,

$$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i).$$

Using Bayes theorem,

$$\mathbf{P}(H|\mathbf{d}) = \alpha \mathbf{P}(\mathbf{d}|H)\mathbf{P}(H)$$

where $\alpha = P(\mathbf{d})$ is the normalization constant. For each hypothesis h_i, the formula is

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i).$$

Now, suppose we want to make a prediction about an unknown quantity X. In the case Surprise candy, X may correspond to the flavour of the next candy after seeing N of them. The probabilities $\mathbf{P}(X|\mathbf{d})$ can be computed as follows:

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)\mathbf{P}(h_i|\mathbf{d}) \quad : \text{conditioning over } h_i\text{'s.}$$

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)\mathbf{P}(h_i|\mathbf{d}) \quad : X \text{ and } \mathbf{d} \text{ are cond. ind. wrt } h_i.$$

Note that the predictions about X are weighted averages over the predictions of individual hypotheses. That is, the hypotheses are *intermediaries* between X and data $\mathbf{d}$. Clearly, the size of summation depends on the size of the hypothesis space. In the present case there are only five of them in the hypothesis space $\{h_1, h_2, h_3, h_4, h_5\}$ but, in general, there can be infinitely many hypotheses in the set which will make the above summation unwieldy. Therefore, the question is, can we approximate the above summation? The answer is yes, here is how.

Maximum A Posteriori Hypothesis

Maximal a posteriori (MAP) hypothesis is the one that **maximizes** $\mathbf{P}(H|\mathbf{d})$. This is a common approximation which usually adopted in science.

$$h_{MAP} = \underset{h \in Dom(H)}{\mathbf{argmax}}\ P(h|\mathbf{d})$$

$$h_{MAP} = \underset{h \in Dom(H)}{\mathbf{argmax}}\ \alpha P(\mathbf{d}|h)P(h)$$

$$h_{MAP} = \underset{h \in Dom(H)}{\mathbf{argmax}}\ P(\mathbf{d}|h)P(h)$$

It turns out that predictions made according to the MAP hypothesis are approximately Bayesian in nature. That is,

$$\mathbf{P}(X|\mathbf{d}) \approx \mathbf{P}(X|h_{MAP})$$

We can be see that finding MAP hypothesis is much easier than Bayesian learning because it requires solving an optimization problem instead of a large summation (or integration) problem.

Minimum Description Length Hypothesis

Another way to look at MAP learning is as follows. Suppose we take the logarithm of the following equation:

$$\mathbf{P}(H|\mathbf{d}) = \alpha \mathbf{P}(\mathbf{d}|H)\mathbf{P}(H)$$

$$log(\mathbf{P}(H|\mathbf{d})) = log\alpha + log\mathbf{P}(\mathbf{d}|H) + log\mathbf{P}(H)$$

Clearly choosing h_{MAP} to maximize $\mathbf{P}(\mathbf{d}|H)\mathbf{P}(H)$ is same choosing h_{MAP} to maximize $log\mathbf{P}(\mathbf{d}|H)+log\mathbf{P}(H)$. This is equivalent to minimizing $-log\mathbf{P}(\mathbf{d}|H)-log\mathbf{P}(H)$. Using the connection between information encoding (content) and probability, as given by Shannon, we can say the following

- $-log\mathbf{P}(H)$ is the number of bits required to specify the hypothesis h_i
- $-log\mathbf{P}(\mathbf{d}|H)$ is the number of additional bits required to specify the data given the hypothesis.

Hence, MAP learning turns out to be task of choosing the hypothesis that provides maximum **compression** of the data. This is known as **minimum description length** learning which attempts to minimize the size of the hypothesis and data encodings rather than work with probabilities.

Maximal Likelihood Hypothesis

Suppose we assume a uniform prior probability over the space of the hypothesis, i.e.,

$$\forall h \in Dom(H), P(H = h) = 1/n \text{ where } n \text{ is the size of } Dom(H).$$

Then, MAP learning reduces to choosing the hypothesis h_{ML} that maximizes $P(\mathbf{d}|H)$. This is called maximum likelihood learning which is very common in statistics. ML learning is a reasonable approach when there is no reason to prefer one hypothesis over the other *a priori*. ML learning provides good approximation to Bayesian and MAP learning when the data set is large, i.e.,

$$\mathbf{P}(X|\mathbf{d}) \approx \mathbf{P}(X|h_{MAP}) \approx \mathbf{P}(X|h_{ML})$$

5.1 Statistical Learning with Complete Data

The simplest task in statistical learning methods is **parameter learning** with **complete data**. This task involves finding the numerical parameters

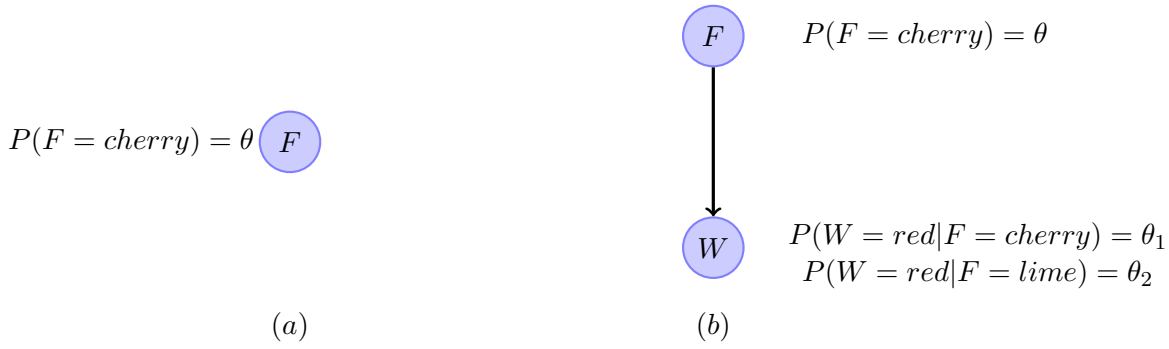

Figure 14: Bayesian Networks for Surprise Candy Example

for a probability model (like Bayesian Network) whose structure is fixed. This means learning the conditional probabilities in a Bayesian network with a given structure.

Consider a simple Bayesian network with one node corresponding to the random variable F (Flavour of Surprise candy) whose prior probability $P(F = cherry) = \theta$ we need to find. This Bayesian network maps to a bag of lime and cherry candies whose proportion θ of cherry and lime candies we don't know. Note that the proportion θ can be anywhere between 0 and 1. We are told that when we sample candies from this bag, say N of them then c of these have cherry flavour whereas $l = N - c$ are lime. Using this information we are asked to find the most suitable value of θ.

The parameter in this case is θ and the hypothesis is h_θ. The likelihood of the data, given h_θ is a function of θ given as follows:

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^c \cdot (1-\theta)^l.$$

We assume that hypothesis prior is same for all θ. That is, for any θ_1, θ_2, $P(h_{\theta_1}) = P(h_{\theta_2})$. With such an assumption the most suitable hypothesis which approximates Bayesian learning is maximal likelihood hypothesis h_{ML}. h_{ML} is given by the value of θ that maximizes $P(\mathbf{d}|h_\theta)$. This is computed as follows using log likelihood:

$$logP(\mathbf{d}|h_\theta) = c \cdot log\theta + l \cdot log(1-\theta)$$

$$L = c \cdot log\theta + l \cdot log(1-\theta), \text{ abbreviating } logP(\mathbf{d}|h_\theta) \text{ by } L$$

In order to find the maximum we differentiate L wrt θ

$$\frac{dL}{d\theta} = \frac{c}{\theta} - \frac{l}{(1-\theta)}$$

L is maximized when $\frac{c}{\theta} - \frac{l}{(1-\theta)}$ is 0. So L is maximum when $\theta = \frac{c}{(l+c)} = \frac{c}{N}$. Therefore, according to h_{ML} the actual proportion of cherries in the bag is equal to the observed proportion of cherries in the candies unwrapped so far.

This gives us the standard method for **maximum likelihood parameter learning**.

1. Write down an expression for the likelihood of the data $P(\mathbf{d}|h_\theta)$ as a function of the parameter θ.

2. Write down the derivative of the log likelihood with respect to the parameter.

3. Find the parameter value such that the derivative is zero.

There is a **significant problem with maximal-likelihood learning**: when the data set is small enough that some events have not yet been observed then the maximum likelihood hypothesis assigns zero probability to those events. For example, in the case of the given surprise candy bag where θ is not known it may be possible that proportion of cherry candies may be significantly high but in the first N picks we may not observe any cherries, that is c would be 0. As already seen, h_{ML} will approximate θ to be zero which is patently wrong.

It may be possible that there are more than one parameters to learn, say $\theta_1, \cdots, \theta_k$. In that case we write the expression for $P(\mathbf{d}|h_{\theta_1,\cdots,\theta_k})$ which may be a function of all these parameters. We take partial derivatives of log likelihood with respect to each such θ_i and then equate it to zero to find θ_i.

Consider the extended example where this new candy manufacturer wants to give a little hint to the customer and uses candy wrappers coloured red and green. The *wrapper* for each candy is selected probabilistically, according to some unknown conditional distribution depending on the flavour. This corresponds to the Bayesian network where there are two nodes, one each for flavour F and wrapper W. The node F has no parents whereas the node W has F as its parent. The conditional probabilities in the network have three unknowns: θ, θ_1 and θ_2 where $P(F = cherry) = \theta$, $P(W = red|F = cherry) = \theta_1$ and $(W = red|F = lime) = \theta_2$. The other conditional probabilities can be obtained from these three.

That is, these results can be extended to any Bayesian network where conditional probabilities are represented as tables. With complete data the maximum likelihood learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter.

5.2 Learning as Inference

The first surprise candy example in the previous section has one parameter θ. θ is the proportion of cherry candies in a given bag of surprise candy. In a straightforward way θ corresponds to the probability that a randomly selected candy from a surprise candy bag is cherry flavoured. Clearly, θ can take any value between 0 and 1. In the Bayesian view, θ is the unknown value is of a random variable Θ. Clearly, Θ is a continuous random variable and $\mathbf{P}(\Theta)$ is nothing but the hypothesis prior for the surprise candy example.

We know about a number of continuous density functions. It turns out that uniform density function $P(\theta) = U[0,1](\theta)$ is a suitable candidate for the random variable Θ. Uniform density is a member of the family of **beta** distributions where each beta distribution is defined by two **hyper parameters** a & b

$$beta[a,b](\theta) = \alpha \cdot \theta^{a-1} \cdot (1-\theta)^{b-1},$$

θ is in the range $[0,1]$ and α is a function of a and b. If Θ has a prior $beta[a,b]$ distribution then after a data point is observed, the posterior distribution is also a beta distribution.

Suppose, we observe a cherry candy in the first sampling of the surprise candy bag. The posterior probability is given as follows:

$$P(\theta|D_1 = cherry) = \alpha \cdot P(D_1 = cherry|\theta)P(\theta)$$

$$P(\theta|D_1 = cherry) = \alpha \cdot \theta \cdot P(\theta)$$

$$P(\theta|D_1 = cherry) = \alpha \cdot \theta \cdot beta[a,b](\theta)$$

$$P(\theta|D_1 = cherry) = \alpha' \cdot \theta \cdot \theta^{a-1} \cdot (1-\theta)^{b-1}$$

$$P(\theta|D_1 = cherry) = \alpha' \cdot \theta^{a} \cdot (1-\theta)^{b-1}$$

$$P(\theta|D_1 = cherry) = beta[a+1,b](\theta)$$

Similarly,

$$P(\theta|D_1 = lime) = beta[a,b+1](\theta)$$

Thus, after seeing a cherry candy we simply increment the 'a' parameter to get the posterior distribution and similarly, after seeing a lime candy, we

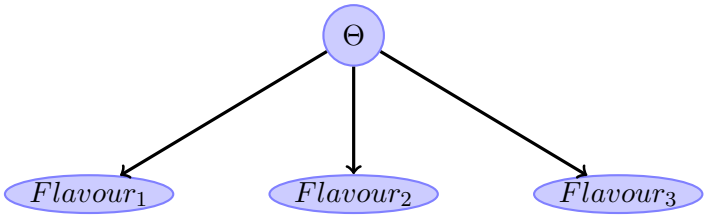

Figure 15: Bayesian Network for Learning parameter θ

increment the 'b' parameter to get the posterior distribution. This is how Bayesian learning is accomplished when the parameter θ is assumed to have beta distribution.

The second surprise candy example has three parameters θ, θ_1 and θ_2 where $P(F = cherry) = \theta$, $P(W = red|F = cherry) = \theta_1$ and $P(W = red|F = lime) = \theta_2$. We would discuss Bayesian learning for this extended example. First, the hypothesis prior $\mathbf{P}(\Theta, \Theta_1, \Theta_2)$. We assume that these parameters are independent of each other, which means

$$\mathbf{P}(\Theta, \Theta_1, \Theta_2) = \mathbf{P}(\Theta) \cdot \mathbf{P}(\Theta_1) \cdot \mathbf{P}(\Theta_2).$$

In such a scenario, each parameter can have its own beta distribution that is updated separately as data arrive. Now, we can incorporate the random variables representing unknown parameters into a larger Bayesian network. We also need to make copies of the variables describing each instance.

For example, suppose we have observed three candies then we need the following variables apart from $\Theta, \Theta_1, \Theta_2$ in the Bayesian network,

$$Flavour_1, Flavour_2, Flavour_3 \text{ and } Wrapper_1, Wrapper_2, Wrapper_3$$

Now, the entire Bayesian learning process can be formulated as an inference problem in a suitably constructed Bayesian network as shown in the figure 16. Prediction for a new instance is done simply by adding new instance variables to the network.

6 Reinforcement Learning

Reinforcement learning is different from other learning methods where we learn functions and probability models from given data (example, training set), as in supervised learning, or we learn patterns in the data, as in

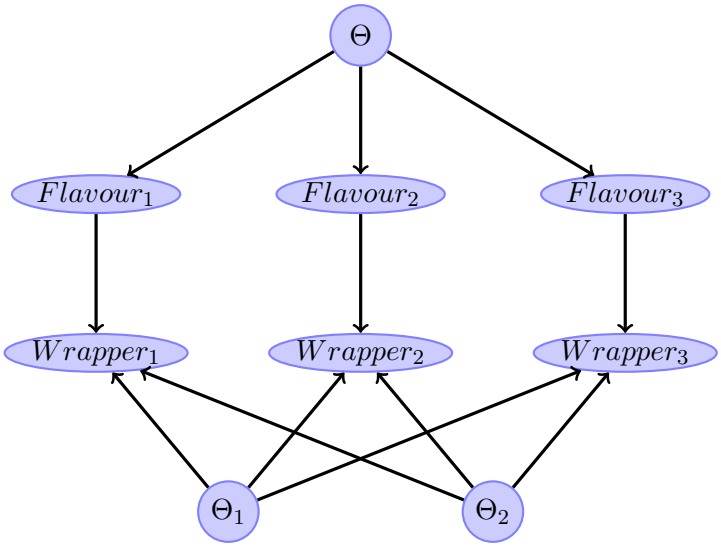

Figure 16: Bayesian Network for Learning parameters $\theta, \theta_1, \theta_2$

unsupervised learning. Reinforcement learning is like playing a new game whose rules you don't know, after a hundred or so moves, your opponent announces, "You lose" (or "You win"). Clearly, after playing the game for a number of rounds you are expected to learn the rules of the game.

Consider an agent trying to learn to play chess. By trying random moves, the agent can eventually build a predictive model of its environment: a) what the board will be like after it makes a move and b) how the opponent is likely to reply in a given situation. Still the agent needs to have some feedback about what are good (set of) moves and what are bad (set of) moves. This kind of feedback is called **reward** or **reinforcement**.

In games like chess, the reinforcement is received only at the end of the game. In other environments, the rewards may come more frequently. For example, in table tennis, each point scored can be considered as a reward.

In most of the cases agent regards the reward as part of the input percept, but they must be able to differentiate between rewards and other sensory inputs. For example, animals seem to be hardwired to recognize pain and hunger as negative rewards and pleasure and food intake as positive rewards, which are different from other sensory inputs.

The task of reinforcement learning is to use observed rewards to learn an optimal policy for the environment.

Apart from the agent and the environment, there are four main subelements of a reinforcement learning system:

1. a **policy**,
2. a **reward function**,
3. a **value function**, and, optionally,
4. a **model of the environment**.

A **policy** defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic.

A **reward function** defines the goal in a reinforcement learning problem. Roughly speaking, it maps each perceived state (or state-action pair) of the environment to a single number, a reward, indicating the intrinsic desirability of that state. A reinforcement learning agent's sole objective is to maximize the total reward it receives in the long run. The reward function defines what are the good and bad events for the agent. In a biological system, it would not be inappropriate to identify rewards with pleasure and pain. They are the immediate and defining features of the problem faced by the agent. As such, the reward function must necessarily be unalterable by the agent. It may, however, serve as a basis for altering the policy. For example, if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. In general, reward functions may be stochastic.

Whereas a reward function indicates what is good in an immediate sense, a **value function** specifies what is good in the long run. Roughly speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those states. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards. Or the reverse could be true. To make a human analogy, rewards are like pleasure (if high) and pain (if low), whereas values correspond to a

more refined and farsighted judgment of how pleased or displeased we are that our environment is in a particular state.

The fourth and final element of some reinforcement learning systems is a **model of the environment**. This is something that mimics the behavior of the environment. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced.

There are three fundamental classes of methods for solving the reinforcement learning problem:

1. **Dynamic programming**,
2. **Monte Carlo methods**, and
3. **Temporal-difference learning**.

All of these methods solve the full version of the problem, including delayed rewards (as in chess playing).

Each class of methods has its strengths and weaknesses. Dynamic programming methods are well developed mathematically, but require a complete and accurate model of the environment. Monte Carlo methods don't require a model and are conceptually simple, but are not suited for step-by-step incremental computation. Finally, temporal-difference methods require no model and are fully incremental, but are more complex to analyze. The methods also differ in several ways with respect to their efficiency and speed of convergence.

To conclude, reinforcement learning is a computational approach for understanding and automating goal-directed learning and decision-making. It is distinguished from other computational approaches by its emphasis on learning by the individual from direct interaction with its environment, without relying on exemplary supervision or complete models of the environment.

Extra Reading

1. For **decision-tree learning** I have omitted the algorithm given in page 658, Figure 18.5. Please make a note of it. If there is a 8 (or more) marks question on decision-tree learning, remember to include the above mentioned algorithm.

2. I discussed logical formulation of learning, page 678 chapter 19, in the class even though it is not include in these notes. You can read it from the book if you want, it is not in the syllabus though.