Bayesian Belief Networks:

The naïve Bayesian classifier makes the assumption of class conditional independence, that is, given the class label of a tuple, the values of the attributes are assumed to be conditionally independent of one another. This simplifies computation.

Bayesian belief networks specify joint conditional probability distributions. They allow class conditional independencies to be defined between subsets of variables. They provide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification. Bayesian belief networks are also known as belief networks, Bayesian networks, and probabilistic networks. For brevity, we will refer to them as belief networks.

A belief network is defined by two components—a *directed acyclic graph* and a set of *conditional probabability tables.*

The variables may be discrete or continuous-valued. They may correspond to actual attributes given in the data or to "hidden variables" believed to form a relationship (e.g., in the case of medical data, a hidden variable may indicate a syndrome, representing a number of symptoms that, together, characterize a specific disease). Each arc represents a probabilistic dependence. If an arc is drawn from a node *Y* to a node *Z*, then *Y* is a parent or immediate predecessor of *Z*, and *Z* is a descendant of *Y*. *Each variable is conditionally independent of its non descendants in the graph, given its parents.*

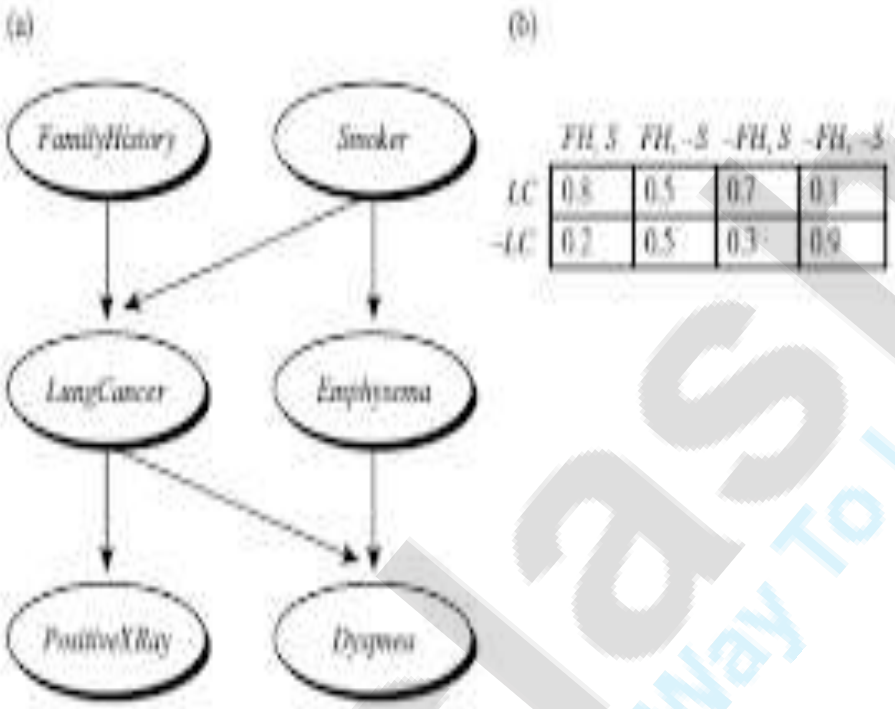


Figure 6.11 A simple Bayesian belief network: (a) A proposed causal model, represented by a directed acyclic graph. (b) The conditional probability table for the values of the variable *Lung Cancer (LC)* showing each possible combination of the values of its parent nodes, *Family History (FH)* and *Smoker (S)*.

A belief network has one conditional probability table (CPT) for each variable. The CPT for a variable *Y* specifies the conditional distribution *P*(*Y*|*Parents* (*Y*)), where *Parents*(*Y*) are the parents of *Y*.

$$P(LungCancer = yes \mid FamilyHistory = yes, Smoker = yes) = 0.8$$
$$P(LungCancer = no \mid FamilyHistory = no, Smoker = no) = 0.9$$

4.2.4.Training Bayesian Belief Networks :

The network topology (or "layout" of nodes and arcs) may be given in advance or inferred from the data. The network variables may be *observable*or *hidden* in all or some of the training tuples. The case of hidden data is also referred to as *missing values* or *incomplete data*.

If the network topology is known and the variables are observable, then training the network is straightforward. It consists of computing the CPT entries, as is similarly done when computing the probabilities involved in naive Bayesian classification. When the network topology is given and some of the variables are hidden, there are various methods to choose from for training the belief network.

A gradient descent strategy is used to search for the *wijk* values that best model the data, based on the assumption that each possible setting of *wijk* is equally likely.

The gradient descent method performs greedy hill-climbing in that, at each iteration or step along the way, the algorithm moves toward what appears to be the best solution at the moment, without backtracking. The weights are updated at each iteration. Eventually, they converge to a local optimum solution.

1. **Compute the gradients: For each *i, j, k*, compute,**

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}} = \sum_{d=1}^{|D|} \frac{P(Y_i = y_{ij}, U_i = u_{ik} | X_d)}{w_{ijk}}.$$

2. **Take a small step in the direction of the gradient: The weights are updated by**

$$w_{ijk} \leftarrow w_{ijk} + (l)\frac{\partial \ln P_w(D)}{\partial w_{ijk}},$$

where *l* is the learning rate representing the step size and

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}}$$

is computed.

3. Renormalize the weights: Because the weights $w_{i,jk}$ are probability values, they must

be between 0.0 and 1.0 and

$$\sum_j w_{ijk}$$

must equal 1 for all *i, k*. Algorithms that follow this form of learning are called *Adaptive Probabilistic Networks*.

4.3. RULE-BASED CLASSIFICATION :

Rule-based classifiers, where the learned model is represented as a set of IF-THEN rules.

4.3.1. Using IF-THEN Rules for Classification :

Rules are a good way of representing information or bits of knowledge. A rule-based classifier uses a set of IF-THEN rules for classification. An IF-THEN rule is an expression of the form

IF *condition* THEN *conclusion*.

An example is rule *R*1,

R1: IF *age* = *youth* AND *student* = *yes* THEN *buys_computer* = *yes*.

The "IF"-part (or left-hand side)of a rule is known as the rule antecedent or precondition. The "THEN"-part (or right-hand side) is the rule consequent. In the rule antecedent, the condition consists of one or more *attribute tests*(such

as *age = youth*, and *student = yes*) that are logically ANDed. The rule's consequent contains a class prediction (in this case, we are predicting whether a customer will buy a computer). *R*1 can also be written as

$$\text{R1: } (age = youth) \wedge (student = yes) \Rightarrow (buys\_computer = yes).$$

If the condition (that is, all of the attribute tests) in a rule antecedent holds true for a given tuple,we say that the rule antecedent is satisfied (or simply, that the rule is satisfied) and that the rule covers the tuple.

A rule *R* can be assessed by its coverage and accuracy. Given a tuple,*X*, from a class labeled data set,*D*, let n_{covers} be the number of tuples covered by *R*; $n_{correct}$ be the number of tuples correctly classified by *R*; and |*D*| be the number of tuples in *D*. We can define the coverage and accuracy of *R* as

$$coverage(R) = \frac{n_{covers}}{|D|}$$

$$accuracy(R) = \frac{n_{correct}}{n_{covers}}.$$

That is, a rule's coverage is the percentage of tuples that are covered by the rule (i.e., whose attribute values hold true for the rule's antecedent). For a rule's accuracy, we look at the tuples that it covers and see what percentage of them the rule can correctly classify.

Let's see how we can use rule-based classification to predict the class label of a given tuple, *X*. If a rule is satisfied by *X*, the rule is said to betriggered. For example, suppose we have

$$X = (age = youth, income = medium, student = yes, credit\_rating = fair).$$

If *R*1 is the only rule satisfied, then the rule fires by returning the class prediction for *X*.

If more than one rule is triggered, we need a conflict resolution strategy to figure out which rule gets to fire and assign its class prediction to*X*. There are many possible strategies ,

✓ ***Size ordering*** **and**

✓ ***Rule ordering*****.**

The size ordering scheme assigns the highest priority to the triggering rule that has the "toughest" requirements, where toughness is measured by the rule antecedent *size*.That is, the triggering rule with the most attribute tests is fired.

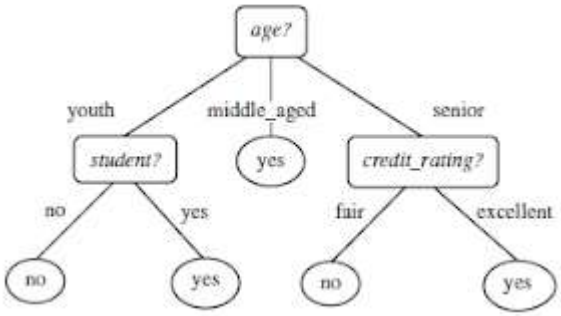
The rule ordering scheme prioritizes the rules beforehand. The ordering may be *class based* or *rule-based*.

With class-based ordering, the classes are sorted in order of decreasing "importance," such as by decreasing *order of prevalence*.

With rule-based ordering, the rules are organized into one long priority list, according to some measure of rule quality such as accuracy, coverage, or size (number of attribute tests in the rule antecedent), or based on advice from domain experts.

4.3.2. Rule Extraction from a Decision Tree :

To extract rules from a decision tree, one rule is created for each path from the root to a leaf node. Each splitting criterion along a given path is logically ANDed to form the rule antecedent ("IF" part). The leaf node holds the class prediction, forming the rule consequent ("THEN" part).



R1: IF *age* = *youth* AND *student* = *no* THEN *buys_computer* = *no*
R2: IF *age* = *youth* AND *student* = *yes* THEN *buys_computer* = *yes*
R3: IF *age* = *middle_aged* THEN *buys_computer* = *yes*
R4: IF *age* = *senior* AND *credit_rating* = *excellent* THEN *buys_computer* = *yes*
R5: IF *age* = *senior* AND *credit_rating* = *fair* THEN *buys_computer* = *no*

A disjunction (logical OR) is implied between each of the extracted rules. Because the rules are extracted directly from the tree, they aremutually exclusive and exhaustive. By *mutually exclusive*, this means that we cannot have rule conflicts here because no two rules will be triggered for the same tuple. (We have one rule per leaf, and any tuple can map to only one leaf.) By *exhaustive*, there is one rule for each possible attribute-value combination, so that this set of rules does not require a default rule. Therefore, the order of the rules does not matter—they are *unordered*.

The training tuples and their associated class labels are used to estimate rule accuracy.

Other problems arise during rule pruning, however, as the rules *will no longer be* mutually exclusive and exhaustive. For conflict resolution, C4.5 adopts a class-based ordering scheme. It groups all rules for a single class together, and then determines a ranking of these class rule sets. Within a rule set, the rules are not ordered. C4.5 orders the class rule sets so as to minimize the number of *false-positive errors* (i.e., where a rule predicts a class, *C*, but the actual class is not *C*). The class rule set with the least number of false positives is examined first. Once pruning is complete, a final check is done to

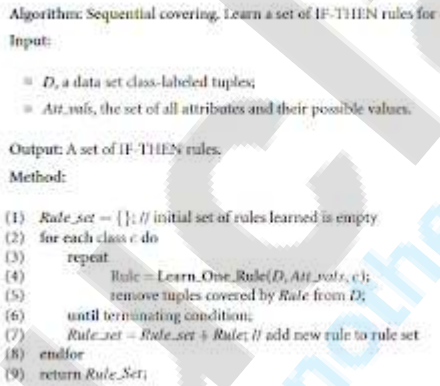
remove any duplicates.

4.3.3.Rule Induction Using a Sequential Covering Algorithm :

IF-THEN rules can be extracted directly from the training data (i.e., without having to generate a decision tree first) using a sequential covering algorithm.

Sequential covering algorithms are the most widely used approach to mining disjunctive sets of classification rules, and form the topic of this subsection.

A basic sequential covering algorithm is shown as ,



Algorithm: Sequential covering. Learn a set of IF-THEN rules for classification.

Input:

- *D*, a data set class-labeled tuples;
- *Att_vals*, the set of all attributes and their possible values.

Output: A set of IF-THEN rules.

Method:

```
(1)  Rule_set = { }; // initial set of rules learned is empty
(2)  for each class c do
(3)      repeat
(4)          Rule = Learn_One_Rule(D, Att_vals, c);
(5)          remove tuples covered by Rule from D;
(6)      until terminating condition;
(7)      Rule_set = Rule_set + Rule; // add new rule to rule set
(8)  endfor
(9)  return Rule_Set;
```

Here, rules are learned for one class at a time. Ideally, when learning a rule for a class, *Ci*, we would like the rule to cover all (or many) of the training tuples of class *C* and none (or few) of the tuples from other classes. In this way, the rules learned should be of high accuracy. The rules need not necessarily be of high coverage. This is because we can have more than one rule for a class, so that different rules may cover different tuples within the same class. The process continues until the terminating condition is met, such as when there are no more training tuples or the quality of a rule returned is below a user-specified threshold. The *Learn One Rule* procedure finds the "best" rule for the current class, given the current set of training tuples.

Typically, rules are grown in a *general-to-specific* manner, The classifying attribute is *loan decision*, which indicates whether a loan is accepted (considered

safe) or rejected (considered risky). To learn a rule for the class "accept," we start off with the most general rule possible, that is, the condition of the rule antecedent is empty. The rule is:

$$\text{IF} \quad \text{THEN } loan\_decision = accept.$$

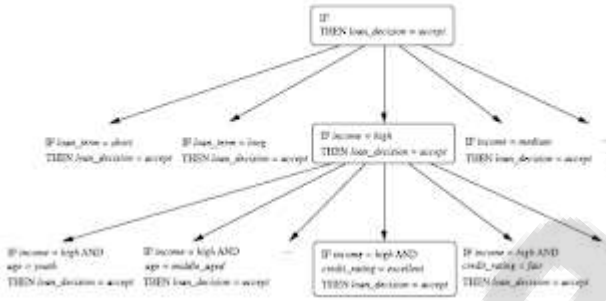


Figure shows a general-to-specific search through rule space.

IF *income = high* THEN *loan decision = accept.*

Each time we add an attribute test to a rule, the resulting rule should cover more of the "accept" tuples. During the next iteration, we again consider the possible attribute tests and end up selecting *credit rating = excellent*. Our current rule grows to become

$$\text{IF } income = high \text{ AND } credit\_rating = excellent \text{ THEN } loan\_decision = accept.$$

The process repeats, where at each step, we continue to greedily grow rules until the

resulting rule meets an acceptable quality level.

4.3.4.Rule Quality Measures :

***Learn One Rule* needs a measure of rule quality. Every time it considers an attribute test,**

it must check to see if appending such a test to the current rule's condition will result in an improved rule.

Choosing between two rules based on accuracy. Consider the two rules as illustrated in Figure 6.14. Both are for the class *loan decision = accept*. We use "*a*" to represent the tuples of class "*accept*" and "*r*" for the tuples of class "*reject.*" Rule *R*1 correctly classifies 38 of the 40 tuples it covers. Rule *R*2 covers only two tuples, which it correctly classifies. Their respective accuracies are 95% and 100%. Thus, *R*2 has greater accuracy than *R*1, but it is not the better rule because of its small coverage.

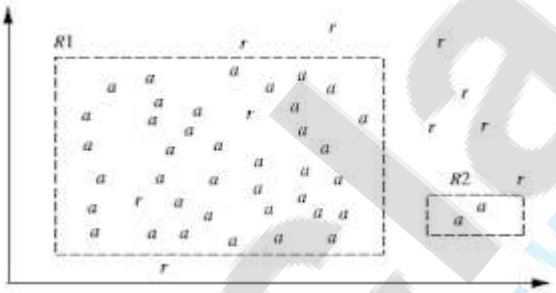


Figure shows Rules for the class *loan decision = accept*, showing *accept* (*a*) and *reject* (*r*) tuples.

Another measure is based on information gain and was proposed inFOIL (First Order Inductive Learner), a sequential covering algorithm that learns first-order logic rules.

Learning first-order rules is more complex because such rules contain variables, whereas the rules we are concerned with in this section are propositional.

FOIL assesses the information gained by extending *condition* as

$$FOIL\_Gain = pos' \times \left(\log_2 \frac{pos'}{pos' + neg'} - log_2 \frac{pos}{pos + neg} \right).$$

4.3.5.Rule Pruning :

The rules may perform well on the training data, but less well on subsequent data. To compensate for this, we can prune the rules. A rule is pruned by removing a conjunct (attribute test). We choose to prune a rule, *R*, if the pruned version of *R* has greater quality, as assessed on an independent set of tuples. FOIL uses a simple yet effective method. Given a rule, *R*,

$$FOIL\_Prune(R) = \frac{pos - neg}{pos + neg},$$

where *pos* and *neg* are the number of positive and negative tuples covered by*R*, respectively.

This value will increase with the accuracy of *R* on a pruning set. Therefore, if the *FOIL Prune* value is higher for the pruned version of *R*, then we prune *R*.