UNIT-3 Continution

ASSOCIATION RULE:

An association rule is an implication expression of the form X—Y., where X and Y are disjoint itemsets, i.e., X ∩ Y = ∅. The strength of an association rule can be measured in terms of its support and confidence. Support determines how often a rule is applicable to a given data set, while confidence determines how frequently items in Y appear in transactions that contain X. The formal definition of these metrics are

$$\text{Support, } s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N}$$

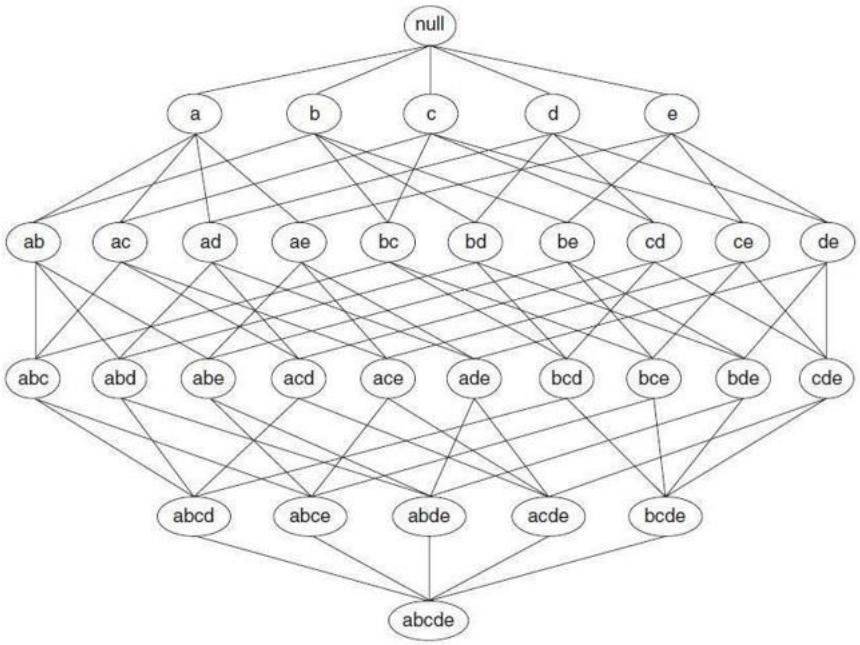
$$\text{Confidence, } c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$$

Why Use Support and Confidence? Support is an important measure because a rule that has very low support may occur simply by chance. A low support rule is also likely to be uninteresting from a business perspective because it may not be profitable to promote items that customers seldom buy together. For these reasons, support is often used to eliminate uninteresting rules.

Confidence, on the other hand, measures the reliability of the inference made by a rule. For a given rule $X \rightarrow Y$, the higher the confidence, the more likely it is for Y to be present in transactions that contain X. Confidence also provides an estimate of the conditional probability of Y given X.

Therefore, a common strategy adopted by many association rule mining algorithms is to decompose the problem into two major subtasks:

1. Frequent Itemset Generation, whose objective is to find all the item- sets that satisfy the *mins up* threshold. These item sets are called frequent item sets.
2. Rule Generation, whose objective is to extract all the high-confidence rules from the frequent item sets found in the previous step. These rules are called strong rules.



Frequent Itemset Generation:

A lattice structure can be used to enumerate the list of all possible item sets. Above Figure shows an itemset lattice for $I = \{a, b, c, d, e\}$. In general, a data set that contains k items can potentially generate up to $2^k - 1$ frequent item sets, excluding the null set. Because k can be very large in many practical applications, the search space of item sets that need to be explored is exponentially large.

To find frequent item sets we have two algorithms,

a) Apriori Algorithm
b) FP-Growth

a) APRIORI ALGORITHM:

Apriori is a seminal algorithm proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses *prior knowledge* of frequent itemset properties, as we shall see later. Apriori employs an iterative approach known as a *level-wise* search, where *k*- itemsets are used to explore

(k+1)-item sets.

First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. The resulting set is denoted by L1. Next, L1 is used to find L2, the set of frequent 2-itemsets, which is used to find L3, and so on, until no more frequent k-item sets can be found. The finding of each Lk requires one full scan of the database.

To improve the efficiency of the level-wise generation of frequent item sets, an important property called the Apriori property is used to reduce the search space.

Apriori property: *All nonempty subsets of a frequent itemset must also be frequent.*

The Apriori property is based on the following observation. By definition, if an itemset *I* does not satisfy the minimum support threshold, *min sup*, then *I* is not frequent, that is, *P(I)< min sup*. If an item *A* is added to the itemset *I*, then the resulting itemset (i.e.,IUA) cannot occur more frequently than *I*. Therefore, IUA is not frequent either, that is, *P*(IUA) < *min sup*.

This property belongs to a special category of properties called **antimonotonicity** in the sense that *if a set cannot pass a test, all of its supersets will fail the same test as well*. It is called *antimonotonicity* because the property is monotonic in the context of failing a test.

A two-step process is followed, consisting of **join** and **prune** actions.

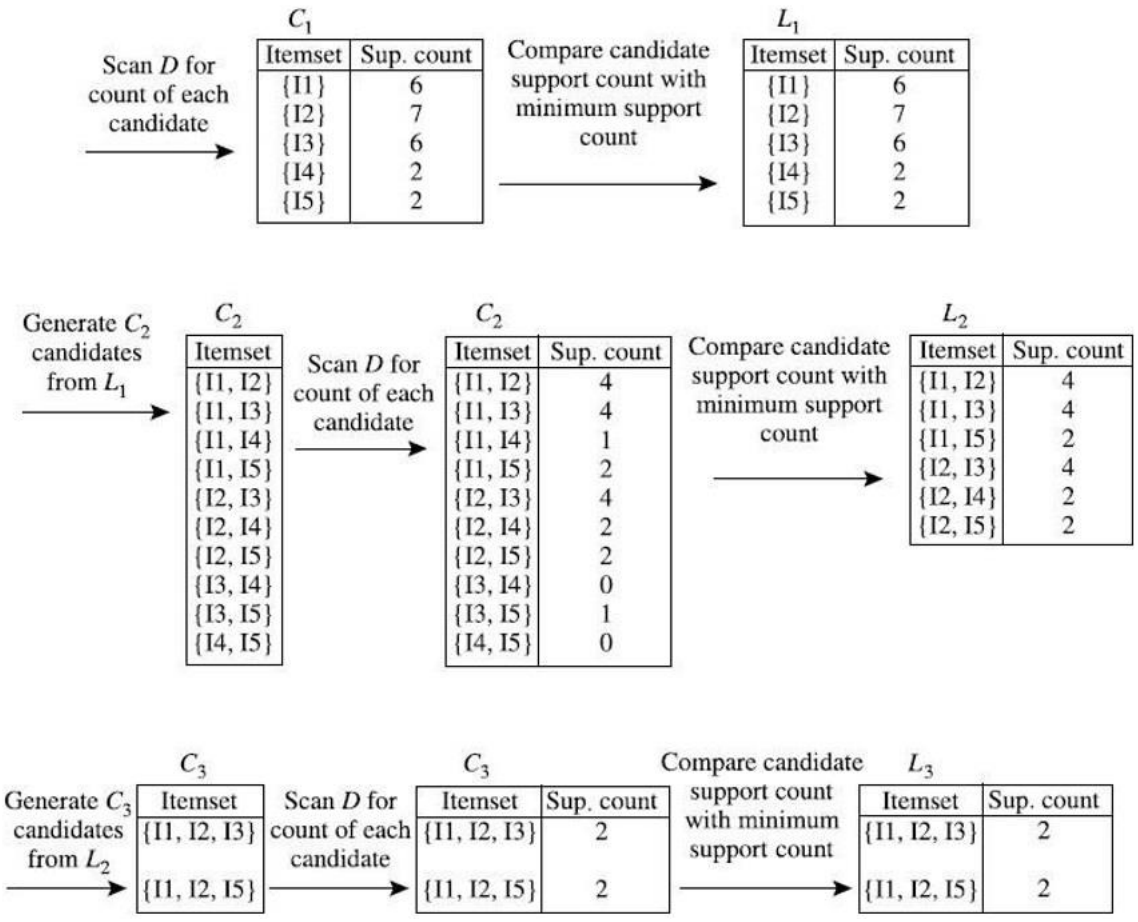
1. **The join step**: To find L_k, a set of **candidate** *k*-item sets is generated by joining L_{k-1} with itself. This set of candidates is denoted C_k.

The prune step: C_k is a superset of L_k, that is, its members may or may not be frequent, but all of the frequent *k*-item sets are included in C_k. A database scan to determine the count of each candidate in C_k would result in the determination of L_k.

Transactional Data for an *AllElectronics* Branch

TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

1. In the first iteration of the algorithm, each item is a member of the set of candidates 1- item sets, $C1$. The algorithm simply scans all of the transactions to count the number of occurrences of each item.
2. Suppose that the minimum support count required is 2, that is, *min sup* = 2. (Here, we are referring to *absolute* support because we are using a support count. The corresponding relative support is 2/9 = 22%.) The set of frequent 1-itemsets, $L1$, can then be determined. It consists of the candidate 1-itemsets satisfying minimum support. In our example, all of the candidates in $C1$ satisfy minimum support.
3. To discover the set of frequent 2-itemsets, $L2$, the algorithm uses the join $L1 \bowtie L1$ to generate a candidate set of 2-itemsets, $C2$. $C2$ consists of 2-itemsets. Note that no candidates are removed from $C2$ during the prune step because each subset of the candidates is also frequent.
4. Next, the transactions in D are scanned and the support count of each candidate itemset in $C2$ is accumulated, as shown in the middle table of the second row in Figure
5. The set of frequent 2-itemsets, $L2$, is then determined, consisting of those candidates 2-item sets in $C2$ having minimum support.



6. The generation of the set of the candidate 3-itemsets, $C3$, is detailed in Figure From the join step, we first get $C3 = L2 \bowtie L2$ = {{I1, I2, I3}, {I1, I2, I5}, {I1, I3, I5}, {I2, I3, I4}, {I2, I3, I5}, {I2, I4, I5}} Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we can determine that the four latter candidates cannot possibly be frequent. We therefore remove them from $C3$, thereby saving the effort of unnecessarily obtaining their counts during the subsequent scan of D to determine $L3$.
7. The transactions in D are scanned to determine L3, consisting of those candidates 3-item sets in C3 having minimum support.
8. The algorithm uses L3 ⋈ L3 to generate a candidate set of 4-itemsets, C4. Although the join results in {I1, I2, I3, I5}, itemset {I1, I2, I3, I5} is pruned because its subset

{I2, I3, I5} is not frequent. Thus, $C4 \neq \emptyset$, and the algorithm terminates, having found all of the frequent item sets.

b) FP-GROWTH:

FP-growth (finding frequent item sets without candidate generation). We re-examine the mining of transaction database, *D*, of Table in previous Example using the frequent pattern growth approach.

Transactional Data for an *AllElectronics* Branch

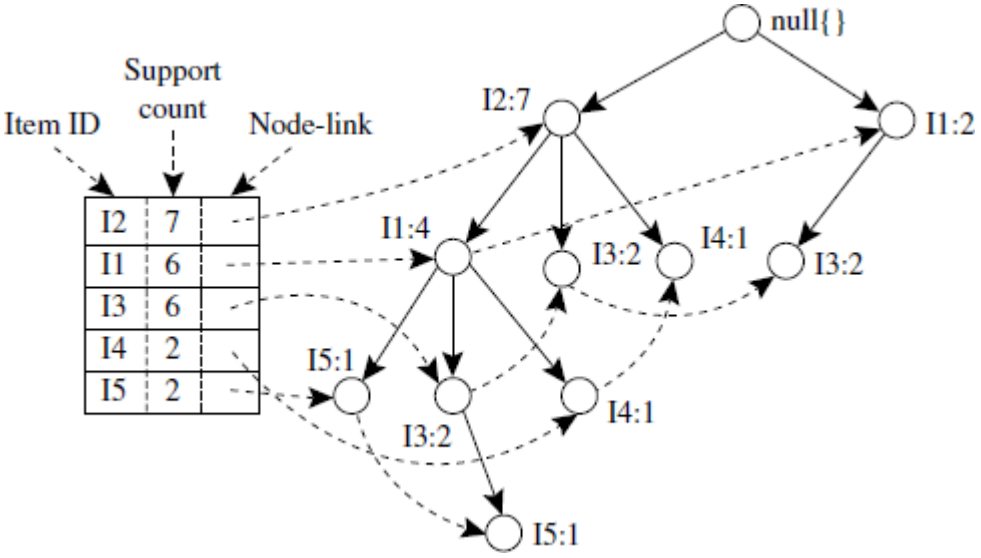
TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

The first scan of the database is the same as Apriori, which derives the set of frequent items (1-itemsets) and their support counts (frequencies). Let the minimum support count be

2. The set of frequent items is sorted in the order of descending support count. This resulting set or *list* is denoted by *L*. Thus, we have L = {{I2:7}, {I1:6}, {I3:6}, {I4:2}, {I5:2}}

An FP-tree is then constructed as follows. First, create the root of the tree, labeled with “null.” Scan database D a second time. The items in each transaction are processed in L

order (i.e., sorted according to descending support count), and a branch is created for each transaction.



The FP-tree is mined as follows. Start from each frequent length-1 pattern (as an initial **suffix pattern**), construct its **conditional pattern base** (a "sub-database," which consists of the set of *prefix paths* in the FP-tree co-occurring with the suffix pattern), then construct its (*conditional*) FP-tree, and perform mining recursively on the tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Mining the FP-Tree by Creating Conditional (Sub-)Pattern Bases

Item	*Conditional Pattern Base*	*Conditional FP-tree*	*Frequent Patterns Generated*
I5	{{I2, I1: 1}, {I2, I1, I3: 1}}	⟨I2: 2, I1: 2⟩	{I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}
I4	{{I2, I1: 1}, {I2: 1}}	⟨I2: 2⟩	{I2, I4: 2}
I3	{{I2, I1: 2}, {I2: 2}, {I1: 2}}	⟨I2: 4, I1: 2⟩, ⟨I1: 2⟩	{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}
I1	{{I2: 4}}	⟨I2: 4⟩	{I2, I1: 4}

Finally, we can conclude that frequent item sets are {I2, I1, I5} and {I2, I1, I3}.

GENERATING ASSOCIATION RULES FROM FREQUENT ITEMSETS

Once the frequent item sets from transactions in a database *D* have been found, it is straightforward to generate strong association rules from them (where *strong* association rules satisfy both minimum support and minimum confidence). This can be done using Eq. for confidence, which we show again here for completeness:

$$confidence(A \Rightarrow B) = P(B|A) = \frac{support\_count(A \cup B)}{support\_count(A)}.$$

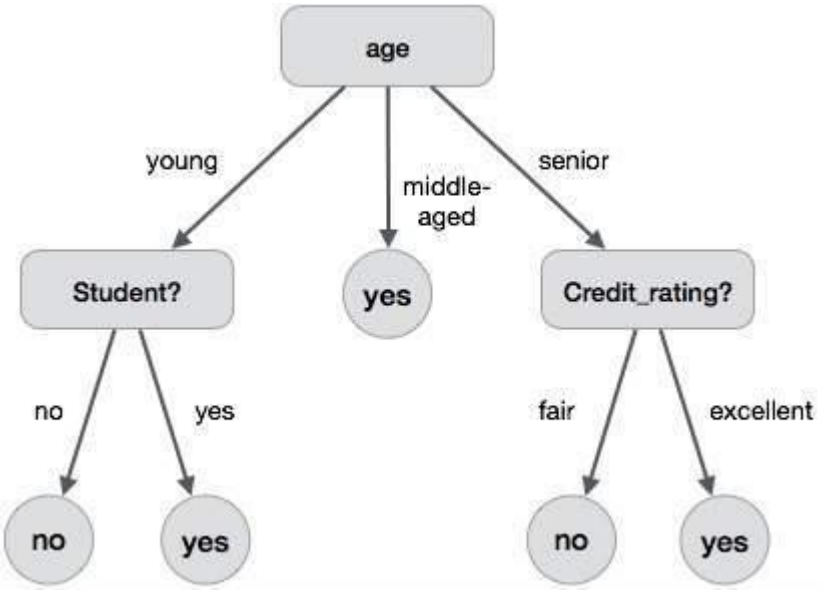
UNIT-4

Classification

1. Decision Tree induction:

A decision tree is a structure that includes a root node, branches, and leaf nodes. Each internal node denotes a test on an attribute, each branch denotes the outcome of a test, and each leaf node holds a class label. The topmost node in the tree is the root node.

The following decision tree is for the concept buy_computer that indicates whether a customer at a company is likely to buy a computer or not. Each internal node represents a test on an attribute. Each leaf node represents a class.



The benefits of having a decision tree are as follows −

- It does not require any domain knowledge.
- It is easy to comprehend.
- The learning and classification steps of a decision tree are simple and fast.

Decision Tree Induction Algorithm

A decision tree algorithm known as ID3 (Iterative Dichotomiser). Later C4.5, which was the successor of ID3. ID3 and C4.5 adopt a greedy approach. In this algorithm, there is no backtracking; the trees are constructed in a top-down recursive divide-and-conquer manner. Generating a decision tree form training tuples of data partition D

Algorithm : Generate_decision_tree

Input:

Data partition, D, which is a set of training tuples and their associated class labels.

attribute_list, the set of candidate attributes. Attribute selection method, a procedure to determine the splitting criterion that best partitions that the data tuples into individual classes. This criterion includes a splitting_attribute and either a splitting point or splitting subset.

Output:

A Decision Tree

Method

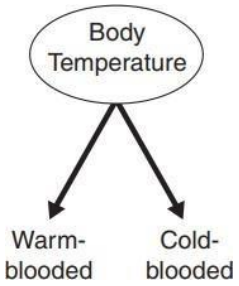
```
create a node N;

if tuples in D are all of the same class, C then
   return N as leaf node labeled with class C;

if attribute_list is empty then
   return N as leaf node with labeled
   with majority class in D;|| majority voting

apply attribute_selection_method(D, attribute_list)
to find the best splitting_criterion;
label node N with splitting_criterion;

if splitting_attribute is discrete-valued and
   multiway splits allowed then  // no restricted to binary trees

attribute_list = splitting attribute; // remove splitting attribute
for each outcome j of splitting criterion
// partition the tuples and grow subtrees for each partition
   let Dj be the set of data tuples in D satisfying outcome j; // a partition

   if Dj is empty then
      attach a leaf labeled with the majority
      class in D to node N;
   else
```

attach the node returned by Generate
decision tree(Dj, attribute list) to node N;
end for
return N;

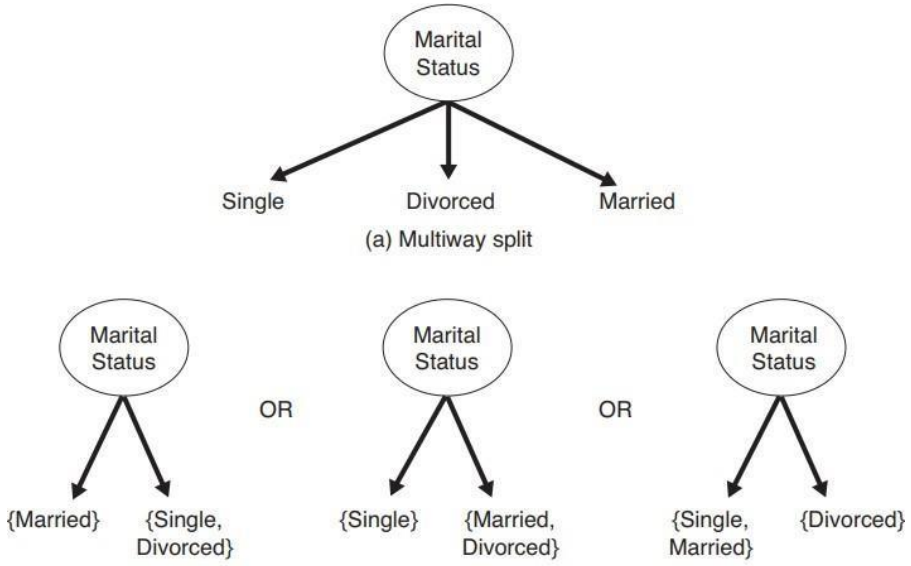
Methods for selecting best test conditions

Decision tree induction algorithms must provide a method for expressing an attribute test condition and its corresponding outcomes for different attribute types.

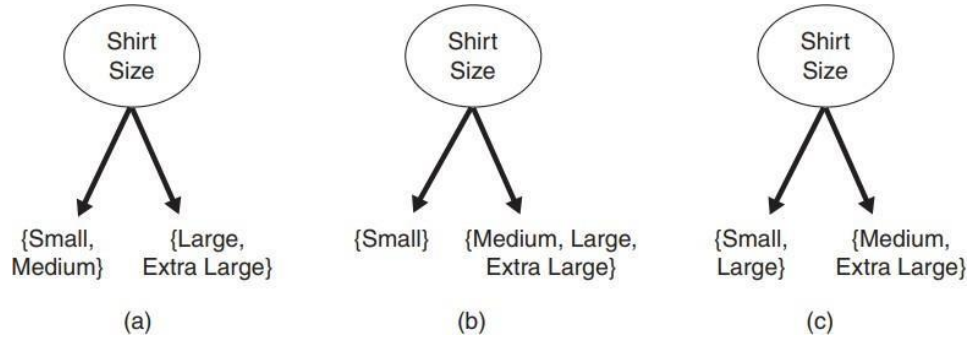
Binary Attributes: The test condition for a binary attribute generates two potential outcomes.

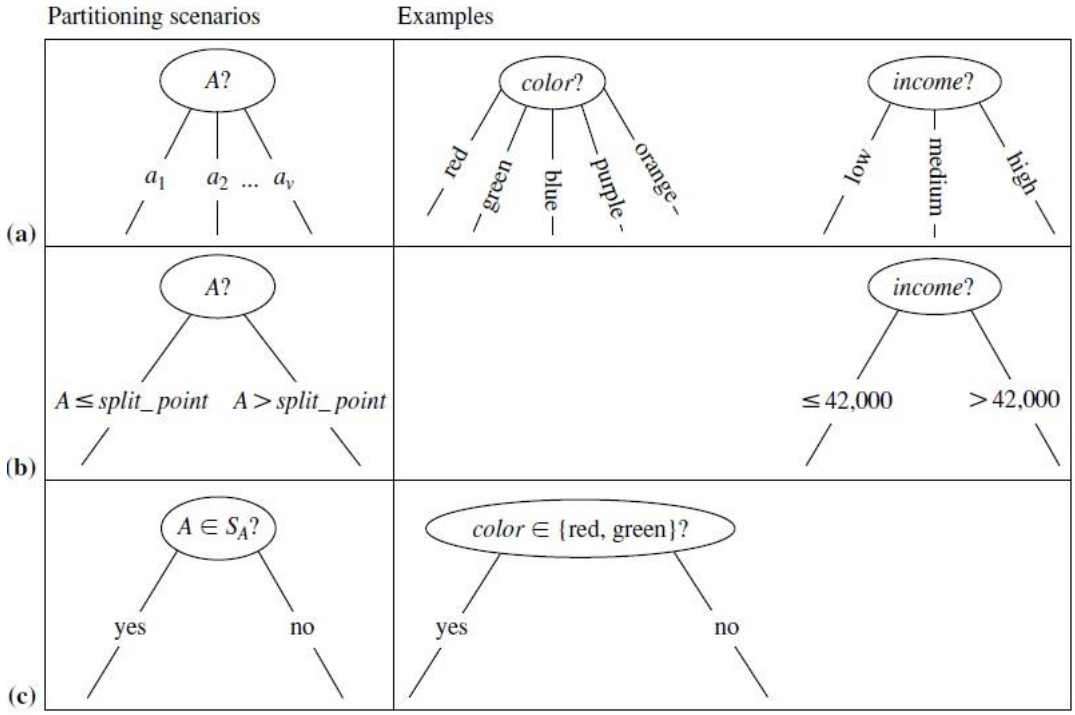


Nominal Attributes: These can have many values. These can be represented in two ways.



Ordinal attributes: These can produce binary or multiway splits. The values can be grouped as long as the grouping does not violate the order property of attribute values.





2. Attribute Selection Measures

- An **attribute selection measure** is a heuristic for selecting the splitting criterion that "best" separates a given data partition, *D*, of class-labeled training tuples into individual classes.
- If we were to split *D* into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (i.e., all the tuples that fall into a given partition would belong to the same class).
- Conceptually, the "best" splitting criterion is the one that most closely results in such a scenario. Attribute selection measures are also known as **splitting rules** because they determine how the tuples at a given node are to be split.

- The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the measure4 is chosen as the splitting attribute for the given tuples.
- If the splitting attribute is continuous-valued or if we are restricted to binary trees, then, respectively, either a split point or a splitting subset must also be determined as part of the splitting criterion.
- The tree node created for partition D is labeled with the splitting criterion, branches are grown for each outcome of the criterion, and the tuples are partitioned accordingly.
- There are three popular attribute selection measures—*information gain, gain ratio*, and *Gini index*.

INFORMATION GAIN

This measure is based on pioneering work by Claude Shannon on information theory, which studied the value or "information content" of message. Let node N represents or hold the tuple of partition D. The attribute with the highest information gain is chosen as the splitting attribute for the node N. The expected information needed to classify a tuple in D is given by,

$$\text{Info(D)} = -\sum_{i=1}^{m} pi \log_2 pi$$

i.e in our example Entropy(D)=info(D)=$-p_y \log_2(p_y) - p_n \log_2(p_n)$

Where Pi is the probability that an arbitrary tuple in D belongs to class Ci and is estimated by $|C_{i,D}|/|D|$. Info(D) is the average amount of information needed to identify the class label of a tuple in D.Info(D) is also known as the entropy of D.The expected information required to classify a tuple from D, based on the partitioning by attribute A is calculated by,

$$\text{Info}_A(D) = \sum_{j=1}^{V} \frac{|Dj|}{|D|} \quad \text{X Info}(D_j)$$

Information gain is defined as the difference between the original information requirement (i.e. based on the classes) and the new requirement (i.e. obtained after partitioning on A)

$$\text{Gain(A)} = \text{Info(D)} - \text{Info}_A(D)$$

The following table presents training set, D, of class labeled tuples randomly selected from the AllElectronics Customer database.

Class-Labeled Training Tuples from the *AllElectronics* Customer Database

RID	age	income	student	credit_rating	Class: buys_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

Example

- Each attribute is discrete value.
- Continues valued attribute have been generated
- The class label attribute, buys_computer have 2 distinct values {yes, no}; therefore, there are 2 distinct class (i.e. m=2)
- Let class C1 corresponds to YES and class C2 corresponds to NO
- There are nine tuples of class *yes* and five tuples of class *no.*
- A (root) node *N* is created for the tuples in *D*.
- To find the splitting criterion for these tuples, we must compute the information gain of each attribute.
- Let us consider classes: buys_computer as decision criteria D

$$Info(D) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) = 0.940 \text{ bits.}$$

- Now Calculate Entropy of age.
- Age Can be
  - Youth
  - Middle_aged

- Senior

- Youth

Youth	Class: buys computer
Yes	2
No	3

- Middle aged

middle	Class: buys computer
Yes	4
No	0

- Senior

Senior	Class: buys computer
Yes	3
No	2

$$Info_{age}(D) = \frac{5}{14} \times \left(-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}\right)$$

$$+ \frac{4}{14} \times \left(-\frac{4}{4}\log_2\frac{4}{4}\right)$$

$$+ \frac{5}{14} \times \left(-\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5}\right)$$

$$= 0.694 \text{ bits.}$$

Hence, the gain in information from such a partitioning would be

$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246 \text{ bits.}$$

Similarly, we can compute *Gain*(*income*)= 0.029 bits, *Gain*(*student*)= 0.151 bits, and *Gain*(*credit rating*)= 0.048 bits. Because *age* has the highest information gain among the attributes, it is selected as the splitting attribute.

Gain Ratio:

- The information gain measure is biased toward tests with many outcomes.
- It prefers to select attributes having a large number of values.
- For example, consider an attribute that acts as a unique identifier such as *product ID*.
- A split on *product ID* would result in a large number of partitions (as many as there are values), each one containing just one tuple.
- Because each partition is pure, the information required to classify data set *D* based on this partitioning would be *Infoproduct ID*(*D*)= D_0.

- Therefore, the information gained by partitioning on this attribute is maximal. Clearly, such a partitioning is useless for classification.
- C4.5, a successor of ID3, uses an extension to information gain known as *gain ratio*, which attempts to overcome this bias.

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right).$$

This value represents the potential information generated by splitting the training data set, *D*, into *v* partitions, corresponding to the *v* outcomes of a test on attribute *A*.
The gain ratio is defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}.$$

The attribute with the maximum gain ratio is selected as the splitting attribute.
Computation of gain ratio for the attribute *income*. A test on *income* splits the data into three partitions, namely *low*, *medium*, and *high*, containing four, six, and four tuples, respectively. To compute the gain ratio of *income*.

$$SplitInfo_{income}(D) = -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right)$$

$$= 1.557.$$

Gini Index:

- Gini index measures the impurity of *D*, a data partition or set of training tuples, as

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2,$$

i.e. Gini(D)=1-p_y^2-p_n^2

- where *pi* is the probability that a tuple in *D* belongs to class *Ci* and is estimated by |*Ci*,*D*|/|*D*|.
- The sum is computed over *m* classes.
- The Gini index considers a binary split for each attribute.
- When considering a binary split, we compute a weighted sum of the impurity of each resulting partition.

$$Gini_A(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2).$$

- For each attribute, each of the possible binary splits is considered.

- For a discrete-valued attribute, the subset that gives the minimum Gini index for that attribute is selected as its splitting subset.
- For continuous-valued attributes, each possible split-point must be considered.
- The strategy is similar to that described earlier for information gain, where the midpoint between each pair of (sorted) adjacent values is taken as a possible split-point.
- For a possible split-point of *A*, *D*1 is the set of tuples in *D* satisfying *A* <= *split point*, and *D*2 is the set of tuples in *D* satisfying *A* > *split point*.
- The reduction in impurity that would be incurred by a binary split on a discrete- or continuous-valued attribute *A* is

$$\Delta Gini(A) = Gini(D) - Gini_A(D).$$

The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini index) is selected as the splitting attribute. This attribute and either its splitting subset (for a discrete-valued splitting attribute) or split-point (for a continuous-valued splitting attribute) together form the splitting criterion.

Induction of a decision tree using the Gini index. Let *D* be the training data shown earlier in Table, where there are nine tuples belonging to the class *buys computer* D *yes* and the remaining five tuples belong to the class *buys computer* D *no*. A (root) node *N* is created for the tuples in *D*. The Gini index to compute the
impurity of *D*:

$$Gini(D) = 1 - \left(\frac{9}{14}\right)^2 - \left(\frac{5}{14}\right)^2 = 0.459.$$

Let A=income: {low, medium, high}
Let D1 satisfies the condition income ε {low, medium} and D2 satisfies income ε {high}
Therefore, no of tuples in D1=10 and in D2=4.

$$\begin{aligned} Gini_{income \in \{low,medium\}}(D) &= \frac{10}{14} Gini(D_1) + \frac{4}{14} Gini(D_2) \\ &= \frac{10}{14}\left(1 - \left(\frac{7}{10}\right)^2 - \left(\frac{3}{10}\right)^2\right) + \frac{4}{14}\left(1 - \left(\frac{2}{4}\right)^2 - \left(\frac{2}{4}\right)^2\right) \\ &= 0.443 \end{aligned}$$

Similarly, the Gini index values for {*low, high*} and {*medium*} = 0.458
and for {*medium, high*} and {*low*}=0.450.
Evaluating *age*, we obtain {*youth, senior*} and {*middle_aged*} as the best split for *age* with a Gini index = 0.375; the attributes *student* and *credit rating* are both binary, with Gini index values of 0.367 and 0.429, respectively.
Gini index overall, with a reduction in impurity of 0.459-0.357=0.102.

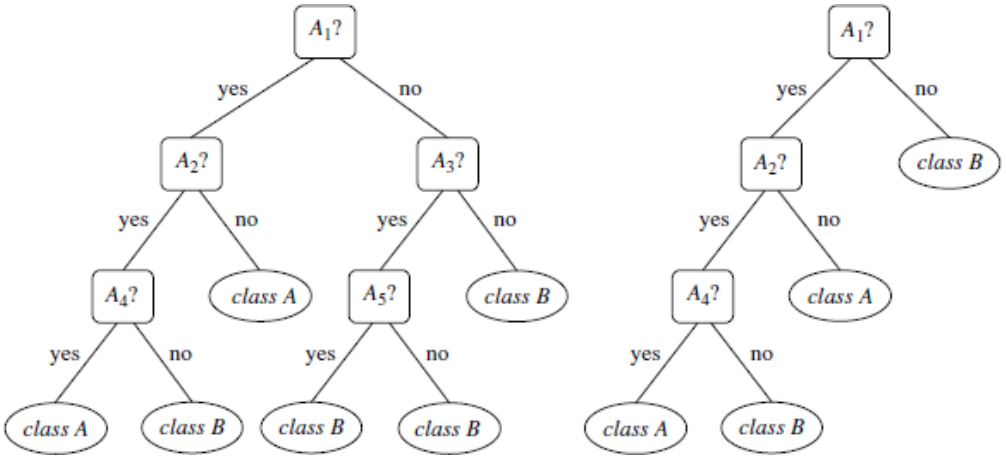
Tree Pruning:

- When a decision tree is built, many of the branches will reflect anomalies in the training data due to noise or outliers.
- Tree pruning methods address this problem of *overfitting* the data. Such methods typically use statistical measures to remove the least-reliable branches.
- Pruned trees tend to be smaller and less complex and, thus, easier to comprehend.
- They are usually faster and better at correctly classifying independent test data (i.e., of previously unseen tuples) than unpruned trees.

"How does tree pruning work?" There are two common approaches to tree pruning: *prepruning* and *postpruning*.

- In the **prepruning** approach, a tree is "pruned" by halting its construction early. Upon halting, the node becomes a leaf. The leaf may hold the most frequent class among the subset tuples or the probability distribution of those tuples.
- If partitioning the tuples at a node would result in a split that falls below a prespecified threshold, then further partitioning of the given subset is halted. There are difficulties, however, in choosing an appropriate threshold.
- In the post pruning, which removes subtrees from a "fully grown" tree. A subtree at a given node is pruned by removing its branches and replacing it with a leaf. The leaf is labeled with the most frequent class among the subtree being replaced.

Fig:



Unpruned and Pruned Trees

- The **cost complexity** pruning algorithm used in CART is an example of the pos tpruning approach.
- This approach considers the cost complexity of a tree to be a function of the number of leaves in the tree and the error rate of the tree (where the **error rate** is the percentage of tuples misclassified by the tree). It starts from the bottom of the tree.
- For each internal node, *N*, it computes the cost complexity of the subtree at *N*, and the cost complexity of the subtree at *N* if it were to be pruned (i.e., replaced by a leaf node).
- The two values are compared. If pruning the subtree at node *N* would result in a smaller cost complexity, then the subtree is pruned. Otherwise, it is kept.
- A **pruning set** of class-labeled tuples is used to estimate cost complexity.

- This set is independent of the training set used to build the unpruned tree and of any test set used for accuracy estimation.
- The algorithm generates a set of progressively pruned trees. In general, the smallest decision tree that minimizes the cost complexity is preferred.
- C4.5 uses a method called **pessimistic pruning**, which is similar to the cost complexity method in that it also uses error rate estimates to make decisions regarding subtree pruning.

3. Bayesian Classification:

- Bayesian classifiers are statistical classifiers.
- They can predict class membership probabilities, such as the probability that a given tuple belongs to a particular class.

- Bayesian classification is based on Bayes' theorem.

Bayes' Theorem:

- Let X be a data tuple. In Bayesian terms, X is considered — "evidence" and it is described by measurements made on a set of n attributes.
- Let H be some hypothesis, such as that the data tuple X belongs to a specified class C.
- For classification problems, we want to determine P(H|X), the probability that the hypothesis H holds given the —evidence‖ or observed data tuple X.
- P(H|X) is the posterior probability, or a posteriori probability, of H conditioned on X.
- Bayes' theorem is useful in that it provides a way of calculating the posterior probability, P(H|X), from P(H), P(X|H), and P(X).

$$P(H|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|H)P(H)}{P(\boldsymbol{X})}.$$

Naïve Bayesian Classification:

The naïve Bayesian classifier, or simple Bayesian classifier, works as follows:

1. Let=be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, X = (x1, x2, …,xn), depicting n measurements made on the tuple from n attributes, respectively, A1, A2, …, An.
2. Suppose that there are m classes, C1, C2, …, Cm. Given a tuple, X, the classifier will predict that X belongs to the class having the highest posterior probability, conditioned on X. That is, the naïve Bayesian classifier predicts that tuple X belongs to the class Ci if and only if

$$P(C_i|\boldsymbol{X}) > P(C_j|\boldsymbol{X}) \quad \text{for } 1 \le j \le m, j \ne i.$$

Thus we maximize(C_j |X). The class Ci for which (C_j |X). is maximized is called the maximum posteriori hypothesis. By Bayes' theorem

$$P(C_i|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|C_i)P(C_i)}{P(\boldsymbol{X})}.$$

3. As P(X) is constant for all classes, only P(X|Ci)P(Ci) need be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, P(C1) = P(C2) = …= P(Cm), and we would therefore maximize P(X|Ci). Otherwise, we maximize P(X|Ci)P(Ci).

4. Given data sets with many attributes, it would be extremely computationally expensive to compute P(X|Ci). In order to reduce computation in evaluating P(X|Ci), the naive assumption of class conditional independence is made. This presumes that the values of the attributes are conditionally independent of one another, given the class label of the tuple. Thus,

$$P(\mathbf{X}|C_i) = \prod_{k=1}^{n} P(x_k|C_i)$$
$$= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i).$$

5. We can easily estimate the probabilities P(x1|Ci), P(x2|Ci), : : : , P(xn|Ci) from the training tuples.

6. For each attribute, we look at whether the attribute is categorical or continuous- valued. For instance, to compute P(X|Ci), we consider the following:

- If A_k is categorical, then P(x_k|Ci) is the number of tuples of class Ci in=having the value x_k for A_k, divided by |Ci ,D| the number of tuples of class Ci in D.
- If A_k is continuous-valued, then we need to do a bit more work, but the calculation is pretty straightforward.

Example:

age	**income**	**student**	**credit_rating**	**buys_computer**
youth	high	no	fair	no
youth	high	no	excellent	no
middle_aged	high	no	fair	yes
senior	medium	no	fair	yes
senior	low	yes	fair	yes
senior	low	yes	excellent	no
middle_aged	low	yes	excellent	yes
youth	medium	no	fair	no
youth	low	yes	fair	yes
senior	medium	yes	fair	yes
youth	medium	yes	excellent	yes
middle_aged	medium	no	excellent	yes
middle_aged	high	yes	fair	yes
senior	medium	no	excellent	no

We wish to predict the class label of a tuple using naïve Bayesian classification, given the same training data above. The training data were shown above in Table. The data tuples are described by the attributes *age*, *income*, *student*, and *credit rating*. The class label attribute, *buys computer*, has two distinct values (namely, {yes, no}). Let *C*1 correspond to the class *buys computer=yes* and *C*2 correspond to b*uys computer=no.* The tuple we wish to classify is

X={age= "youth", income= "medium", student= "yes", credit_rating= "fair"}

We need to maximize P(X|Ci)P(Ci), for i=1,2. P(Ci), the prior probability of each class, can be computed based on the training tuples:

P(*buys computer* = *yes*) = 9/14 = 0.643
P(*buys computer* = *no*) = 5/14 = 0.357

To compute P(X|Ci), for *i* = 1, 2, we compute the following conditional probabilities:

P(*age* = *youth* | *buys computer* = *yes*) = 2/9 = 0.222
P(*income=medium* | *buys computer=yes*) = 4/9 = 0.444
P(*student=yes* | *buys computer=yes*) = 6/9 = 0.667
P(*credit rating=fair* | *buys computer=yes*) = 6/9 = 0.667

P(*age=youth* | *buys computer=no*) = 3/5 = 0.600
P(*income=medium* | *buys computer=no*) = 2/5 = 0.400
P(*student=yes* | *buys computer=no*) = 1/5 = 0.200
P(*credit rating=fair* | *buys computer=no*) = 2/5 = 0.400

Using these probabilities, we obtain
P(***X*** | *buys computer=yes*) = *P*(*age=youth* | *buys computer=yes*)
× *P*(*income=medium* | *buys computer=yes*)
× *P*(*student=yes* | *buys computer=yes*)

× *P*(*credit rating=fair* | *buys computer=yes*)

= 0.222 × 0.444 × 0.667 × 0.667 = 0.044.

Similarly,

P(***X*** | *buys computer=no*) = 0.600 × 0.400 × 0.200 × 0.400 = 0.019.

To find the class, C_i, that P(X|Ci)P(Ci), we compute

P(***X*** | *buys computer=yes*) *P*(*buys computer=yes*) = 0.044 × 0.643 = 0.028

P(***X*** | *buys computer=no*) *P*(*buys computer=no*) = 0.019 × 0.357 = 0.007

Therefore, the naïve Bayesian classifier predicts *buys computer = yes* for tuple ***X***.

4. Rule-Based Classification

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,

*R*1: IF *age = youth* AND *student = yes* THEN *buys computer = yes*.

- The "IF" part (or left side) of a rule is known as the **rule antecedent** or **precondition**.
- The "THEN" part (or right side) is the **rule consequent**.

*R*1 can also be written as

*R*1: (*age = youth*) ^ (*student = yes*)=>(*buys_computer = yes*).

A rule *R* can be assessed by its coverage and accuracy. Given a tuple, ***X***, from a class labelled data set, *D*, let n_{covers} be the number of tuples covered by *R*; $n_{corrects}$ 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

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

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

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).

Example:

*R*1: IF *age* = *youth*	AND *student* = *no*	THEN *buys_computer* = *no*
*R*2: IF *age* = *youth*	AND *student* = *yes*	THEN *buys_computer* = *yes*
*R*3: IF *age* = *middle_aged*		THEN *buys_computer* = *yes*
*R*4: IF *age* = *senior*	AND *credit_rating* = *excellent*	THEN *buys_computer* = *yes*
*R*5: IF *age* = *senior*	AND *credit_rating* = *fair*	THEN *buys_computer* = *no*

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**.

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

Input:

- *D*, a data set of 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)          Rule_set = Rule_set + Rule; // add new rule to rule set
(7)      until terminating condition;
(8)  endfor
(9)  return Rule_Set;
```

Basic sequential covering algorithm.