Decision Tree Classifier

This section introduces a simple classification technique known as the **decision tree** classifier. To illustrate how a decision tree works, consider the classification problem of distinguishing mammals from non-mammals using

the vertebrate data set shown in Table 3.2. Suppose a new species is discovered by scientists. How can we tell whether it is a mammal or a non-mammal? One approach is to pose a series of questions about the characteristics of the species. The first question we may ask is whether the species is cold- or warm-blooded. If it is cold-blooded, then it is definitely not a mammal. Otherwise, it is either a bird or a mammal. In the latter case, we need to ask a follow-up question: Do the females of the species give birth to their young? Those that do give birth are definitely mammals, while those that do not are likely to be non- mammals (with the exception of egg-laying mammals such as the platypusand spiny anteater).

The previous example illustrates how we can solve a classification problem by asking a series of carefully crafted questions about the attributes of the test instance. Each time we receive an answer, we could ask a follow-up question until we can conclusively decide on its class label. The series of questions and their possible answers can be organized into a hierarchical structure called a decision tree. Figure 3.4 shows an example of the decision tree for the mammal classification problem. The tree has three types of nodes:

- A **root node**, with no incoming links and zero or more outgoing links.
- **Internal nodes**, each of which has exactly one incoming link and two or more outgoing links.
- **Leaf** or **terminal** nodes, each of which has exactly one incoming link and no outgoing links.

Every leaf node in the decision tree is associated with a class label. The **non-terminal** nodes, which include the root and internal nodes, contain **attribute test conditions** that are typically defined using a single attribute. Each possible outcome of the attribute test condition is associated with exactly one child of this node. For example, the root node of the tree shown in Figure 3.4 uses the attribute Body Temperature to define an attribute test condition that has two outcomes, warm and cold, resulting in two child nodes.

Given a decision tree, classifying a test instance is straightforward. Starting from the root node, we apply its attribute test condition and follow the appropriate branch based on the outcome of the test. This will lead us either to another internal node, for which a new attribute test condition is applied, or to a leaf node. Once a leaf node is reached, we assign the class label associated with the node to the test instance. As an illustration, Figure 3.5 traces the path used to predict the class label of a flamingo. The path terminates at a leaf node labeled as Non-mammals.

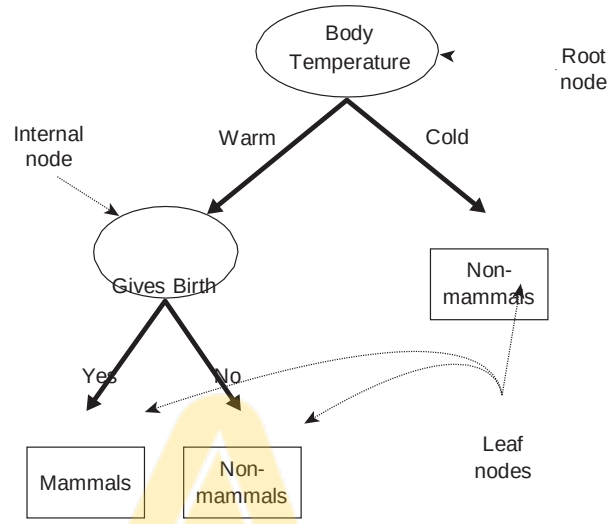


Figure 3.4. A decision tree for the mammal classification problem.

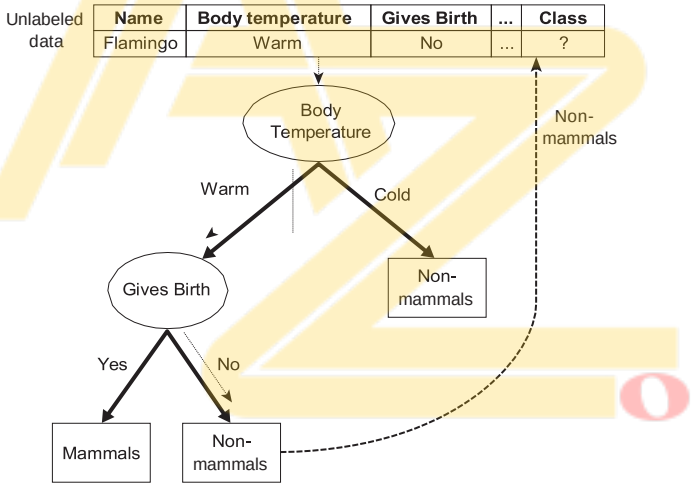


Figure 3.5. Classifying an unlabeled vertebrate. The dashed lines represent the outcomes of applying various attribute test conditions on the unlabeled vertebrate. The vertebrate is eventually assigned to the Non-mammals class.

3.3.1 A Basic Algorithm to Build a Decision Tree

Many possible decision trees that can be constructed from a particular data set. While some trees are better than others, finding an optimal one is computationally expensive due to the exponential size of the search space. Efficient algorithms have been developed to induce a reasonably accurate, albeit

suboptimal, decision tree in a reasonable amount of time. These algorithms usually employ a greedy strategy to grow the decision tree in a top-down fashion by making a series of locally optimal decisions about which attribute to use when partitioning the training data. One of the earliest method is **Hunt's algorithm**, which is the basis for many current implementations of decision tree classifiers, including ID3, C4.5, and CART. This subsection presents Hunt's algorithm and describes some of the design issues that must be considered when building a decision tree.

Hunt's Algorithm

In Hunt's algorithm, a decision tree is grown in a recursive fashion. The tree initially contains a single root node that is associated with all the training instances. If a node is associated with instances from more than one class, it is expanded using an attribute test condition that is determined using a **splitting criterion**. A child leaf node is created for each outcome of the attribute test condition and the instances associated with the parent node aredistributed to the children based on the test outcomes. This node expansion step can then be recursively applied to each child node, as long as it haslabels of more than one class. If all the instances associated with a leaf node have identical class labels, then the node is not expanded any further. Each leaf node is assigned a class label that occurs most frequently in the training instances associated with the node.

To illustrate how the algorithm works, consider the training set shown in Table 3.3 for the loan borrower classification problem. Suppose we apply Hunt's algorithm to fit the training data. The tree initially contains only a single leaf node as shown in Figure 3.6(a). This node is labeled as Defaulted = No, since the majority of the borrowers did not default on their loan payments. The training error of this tree is 30% as three out of the ten training instances have the class label Defaulted = Yes. The leaf node can therefore be further expanded because it contains training instances from more than one class.

Let Home Owner be the attribute chosen to split the training instances. The justification for choosing this attribute as the attribute test condition willbe discussed later. The resulting binary split on the Home Owner attributeis shown in Figure 3.6(b). All the training instances for which Home Owner = Yes are propagated to the left child of the root node and the rest are propagated to the right child. Hunt's algorithm is then recursively applied to each child. The left child becomes a leaf node labeled Defaulted = No, since all instances associated with this node have identical class label Defaulted = No. The right child has instances from each class label. Hence, we split it

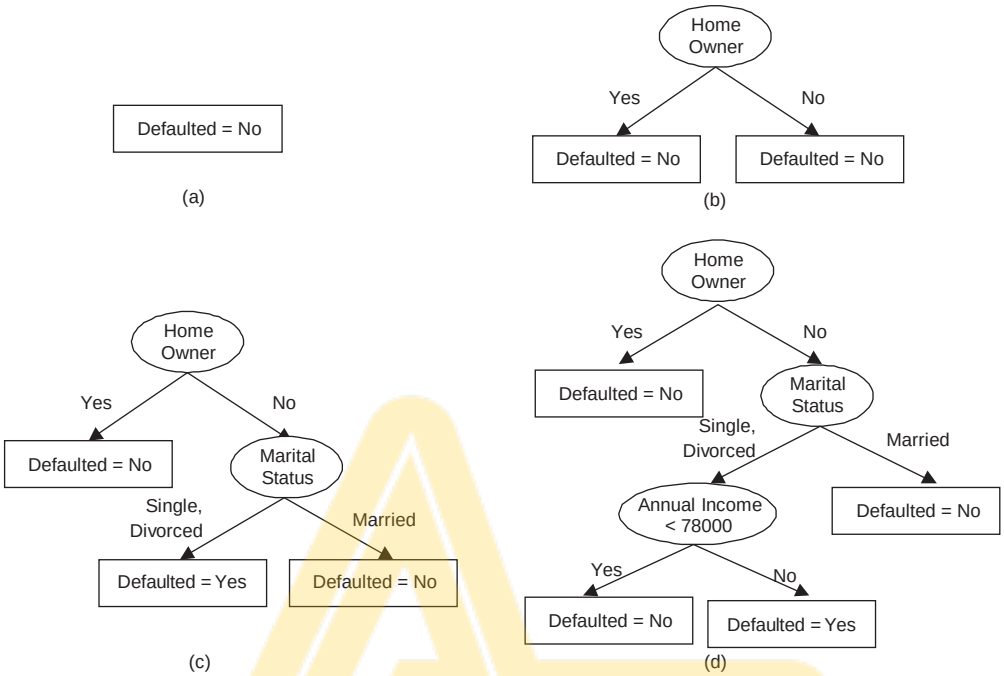


Figure 3.6. Hunt's algorithm for building decision trees.

further. The resulting subtrees after recursively expanding the right child are shown in Figures 3.6(c) and (d).

Hunt's algorithm, as described above, makes some simplifying assumptions that are often not true in practice. In the following, we describe these assumptions and briefly discuss some of the possible ways for handling them.

1. Some of the child nodes created in Hunt's algorithm can be empty if none of the training instances have the particular attribute values. One way to handle this is by declaring each of them as a leaf node witha class label that occurs most frequently among the training instances associated with their parent nodes.

2. If all training instances associated with a node have identical attribute values but different class labels, it is not possible to expand this node any further. One way to handle this case is to declare it a leaf nodeand assign it the class label that occurs most frequently in the training instances associated with this node.

Design Issues of Decision Tree Induction

Hunt's algorithm is a generic procedure for growing decision trees in a greedy fashion. To implement the algorithm, there are two key design issues that must be addressed.

1. **What is the splitting criterion?** At each recursive step, an attribute must be selected to partition the training instances associated with a node into smaller subsets associated with its child nodes. The splitting criterion determines which attribute is chosen as the test condition and how the training instances should be distributed to the child nodes. Thiswill be discussed in Sections 3.3.2 and 3.3.3.

2. **What is the stopping criterion?** The basic algorithm stops expand- ing a node only when all the training instances associated with the node have the same class labels or have identical attribute values. Although these conditions are sufficient, there are reasons to stop expanding a node much earlier even if the leaf node contains training instances from morethan one class. This process is called early termination and the conditionused to determine when a node should be stopped from expanding is called a topping criterion. The advantages of early termination arediscussed in Section 3.4.

3.3.2 Methods for Expressing Attribute 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, as shown in Figure 3.7.

Nominal Attributes Since a nominal attribute can have many values, its attribute test condition can be expressed in two ways, as a multiway split or a binary split as shown in Figure 3.8. For a multiway split (Figure 3.8(a)), the number of outcomes depends on the number of distinct values for the corresponding attribute. For example, if an attribute such as marital status has three distinct values—single, married, or divorced—its test condition will produce a three-way split. It is also possible to create a binary split by partitioning all values taken by the nominal attribute into two groups. For example, some decision tree algorithms, such as CART, produce only binary

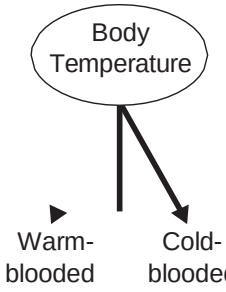


Figure 3.7. Attribute test condition for a binary attribute.

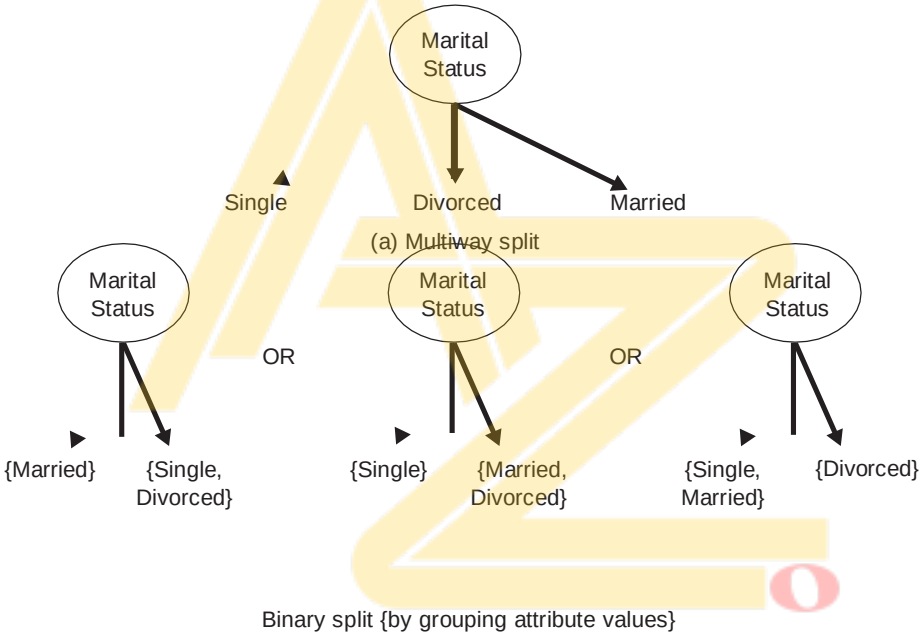


Figure 3.8. Attribute test conditions for nominal attributes.

splits by considering all $2^{k-1} - 1$ ways of creating a binary partition of k attribute values. Figure 3.8(b) illustrates three different ways of grouping the attribute values for marital status into two subsets.

Ordinal Attributes Ordinal attributes can also produce binary or multi- way splits. Ordinal attribute values can be grouped as long as the grouping

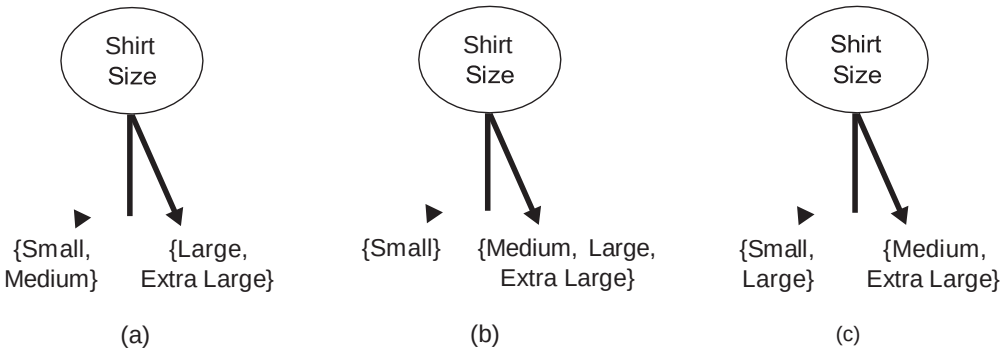


Figure 3.9. Different ways of grouping ordinal attribute values.

does not violate the order property of the attribute values. Figure 3.9 illustrates various ways of splitting training records based on the Shirt Size attribute. The groupings shown in Figures 3.9(a) and (b) preserve the order among the attribute values, whereas the grouping shown in Figure 3.9(c) violates this property because it combines the attribute values Small andLarge into the same partition while Medium and Extra Large are combined into another partition.

Continuous Attributes For continuous attributes, the attribute test condition can be expressed as a comparison test (e.g., $A < v$) producing a binary split, or as a range query of the form $v_i \leq A < v_{i+1}$, for $i = 1, \ldots, k$, producing a multiway split. The difference between these approaches is shownin Figure 3.10. For the binary split, any possible value v between the minimum and maximum attribute values in the training data can be used for construct- ing the comparison test $A < v$. However, it is sufficient to only consider distinct attribute values in the training set as candidate split positions. For the multiway split, any possible collection of attribute value ranges can be used,as long as they are mutually exclusive and cover the entire range of attribute values between the minimum and maximum values observed in the training set. One approach for constructing multiway splits is to apply the discretization strategies described in Section 2.3.6 on page 63. After discretization, a new ordinal value is assigned to each discretized interval, and the attribute test condition is then defined using this newly constructed ordinal attribute.

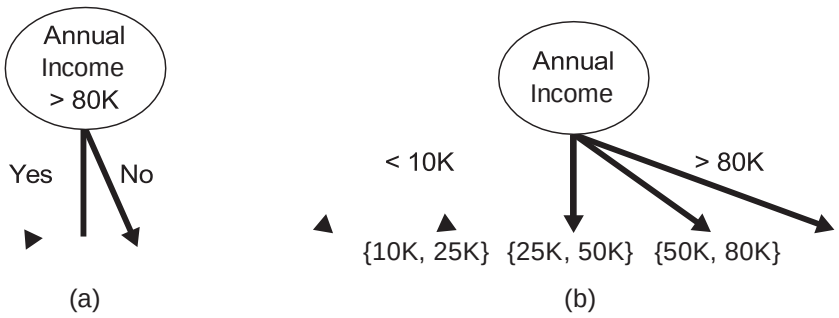


Figure 3.10. Test condition for continuous attributes.

3.3.3 Measures for Selecting an Attribute Test Condition

There are many measures that can be used to determine the goodness of an attribute test condition. These measures try to give preference to attribute test conditions that partition the training instances into *purer* subsets in the child nodes, which mostly have the same class labels. Having purer nodes is useful since a node that has all of its training instances from the same class does not need to be expanded further. In contrast, an impure node containingtraining instances from multiple classes is likely to require several levels of node expansions, thereby increasing the depth of the tree considerably. Larger trees are less desirable as they are more susceptible to model overfitting, a condition that may degrade the classification performance on unseen instances, as will be discussed in Section 3.4. They are also difficult to interpret and incur more training and test time as compared to smaller trees.

In the following, we present different ways of measuring the impurity of a node and the collective impurity of its child nodes, both of which will be used to identify the best attribute test condition for a node.

Impurity Measure for a Single Node

The impurity of a node measures how dissimilar the class labels are for the data instances belonging to a common node. Following are examples of measures

that can be used to evaluate the impurity of a node t:

$$\text{Entropy} = -\sum_{i=0}^{c-1} p_i(t) \log_2 p_i(t), \tag{3.4}$$

$$\text{Gini index} = 1 - \sum_{i=0}^{c-1} p_i(t)^2, \tag{3.5}$$

$$\text{Classification error} = 1 - \max_i [p_i(t)], \tag{3.6}$$

where $p_i(t)$ is the relative frequency of training instances that belong to class i at node t, c is the total number of classes, and $0 \log_2 0 = 0$ in entropy calculations. All three measures give a zero impurity value if a node contains instances from a single class and maximum impurity if the node has equal proportion of instances from multiple classes.

Figure 3.11 compares the relative magnitude of the impurity measures when applied to binary classification problems. Since there are only two classes, $p_0(t) + p_1(t) = 1$. The horizontal axis p refers to the fraction of instances that belong to one of the two classes. Observe that all three measures attain their maximum value when the class distribution is uniform (i.e., $p_0(t) = p_1(t) = 0.5$) and minimum value when all the instances belong to a single class (i.e., either $p_0(t)$ or $p_1(t)$ equals to 1). The following examples illustrate how the values of the impurity measures vary as we alter the class distribution.

Node N_1	Count
Class=0	0
Class=1	6

Gini = $1 - (0/6)^2 - (6/6)^2 = 0$
Entropy = $-(0/6) \log_2(0/6) - (6/6) \log_2(6/6) = 0$
Error = $1 - \max[0/6, 6/6] = 0$

Node N_2	Count
Class=0	1
Class=1	5

Gini = $1 - (1/6)^2 - (5/6)^2 = 0.278$
Entropy = $-(1/6) \log_2(1/6) - (5/6) \log_2(5/6) = 0.650$
Error = $1 - \max[1/6, 5/6] = 0.167$

Node N_3	Count
Class=0	3
Class=1	3

Gini = $1 - (3/6)^2 - (3/6)^2 = 0.5$
Entropy = $-(3/6) \log_2(3/6) - (3/6) \log_2(3/6) = 1$
Error = $1 - \max[3/6, 3/6] = 0.5$

Based on these calculations, node N_1 has the lowest impurity value, followed by N_2 and N_3. This example, along with Figure 3.11, shows the consistency among the impurity measures, i.e., if a node N_1 has lower entropy than node N_2, then the Gini index and error rate of N_1 will also be lower than that

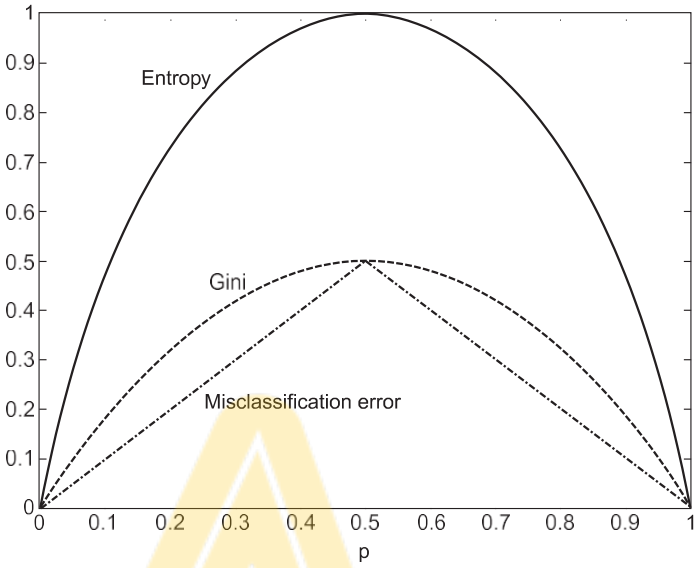


Figure 3.11. Comparison among the impurity measures for binary classification problems.

f N_2. Despite their agreement, the attribute chosen as splitting criterion by the impurity measures can still be different (see Exercise 6 on page 187).

Collective Impurity of Child Nodes

Consider an attribute test condition that splits a node containing N training instances into k children, $\{v_1, v_2, \cdots, v_k\}$, where every child node representsa partition of the data resulting from one of the k outcomes of the attribute test condition. Let $N(v_j)$ be the number of training instances associated witha child node v_j, whose impurity value is $I(v_j)$. Since a training instance in the parent node reaches node v_j for a fraction of $N(v_j)/N$ times, the collective impurity of the child nodes can be computed by taking a weighted sum of the impurities of the child nodes, as follows:

$$I(\text{children}) = \sum_{j=1}^{k} \frac{N(v_j)}{N} I(v_j), \tag{3.7}$$

Example 3.3. [Weighted Entropy] Consider the candidate attribute test condition shown in Figures 3.12(a) and (b) for the loan borrower classification problem. Splitting on the Home Owner attribute will generate two child nodes

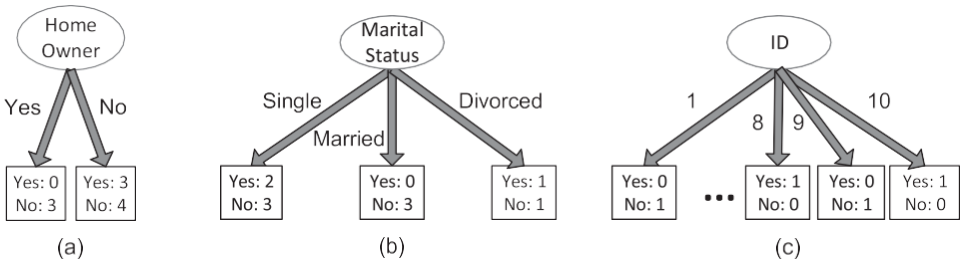


Figure 3.12. Examples of candidate attribute test conditions.

whose weighted entropy can be calculated as follows:

$$I(\text{Home Owner} = \text{yes}) = -\frac{0}{3}\log_2\frac{0}{3} - \frac{3}{3}\log_2\frac{3}{3} = 0$$

$$I(\text{Home Owner} = \text{no}) = -\frac{3}{7}\log_2\frac{3}{7} - \frac{4}{7}\log_2\frac{4}{7} = 0.985$$

$$I(\text{Home Owner}) = \frac{3}{10} \times 0 + \frac{7}{10} \times 0.985 = 0.690$$

Splitting on Marital Status, on the other hand, leads to three child nodes with a weighted entropy given by

$$I(\text{Marital Status} = \text{Single}) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.971$$

$$I(\text{Marital Status} = \text{Married}) = -\frac{0}{3}\log_2\frac{0}{3} - \frac{3}{3}\log_2\frac{3}{3} = 0$$

$$I(\text{Marital Status} = \text{Divorced}) = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1.000$$

$$I(\text{Marital Status}) = \frac{5}{10} \times 0.971 + \frac{3}{10} \times 0 + \frac{2}{10} \times 1 = 0.686$$

Thus, Marital Status has a lower weighted entropy than Home Owner.

Identifying the best attribute test condition

To determine the goodness of an attribute test condition, we need to compare the degree of impurity of the parent node (before splitting) with the weighted degree of impurity of the child nodes (after splitting). The larger their difference, the better the test condition. This difference, Δ, also termed as the **gain** in purity of an attribute test condition, can be defined as follows:

$$\Delta = I(\text{parent}) - I(\text{children}), \tag{3.8}$$

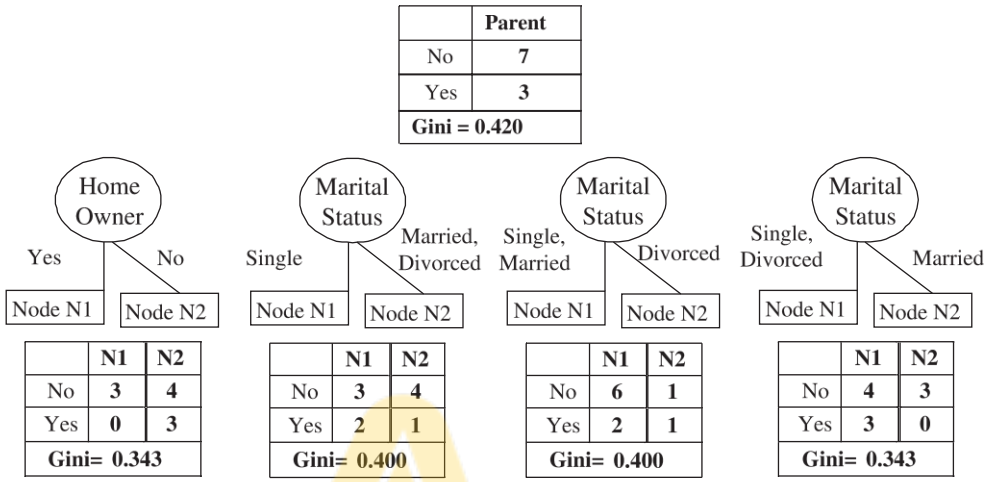


Figure 3.13. Splitting criteria for the loan borrower classification problem using Gini index.

where I(parent) is the impurity of a node before splitting and I(children) is the weighted impurity measure after splitting. It can be shown that the gain is non-negative since I(parent) $\geq$ I(children) for any reasonable measure such as those presented above. The higher the gain, the purer are the classes in the child nodes relative to the parent node. The splitting criterion in the decision tree learning algorithm selects the attribute test condition that shows the maximum gain. Note that maximizing the gain at a given node is equivalent to minimizing the weighted impurity measure of its children since I(parent) is the same for all candidate attribute test conditions. Finally, when entropy is used as the impurity measure, the difference in entropy is commonly known as **information gain**, Δ_{info}.

In the following, we present illustrative approaches for identifying the best attribute test condition given qualitative or quantitative attributes.

Splitting of Qualitative Attributes

Consider the first two candidate splits shown in Figure 3.12 involving qualitative attributes Home Owner and Marital Status. The initial class distribution at the parent node is (0.3, 0.7), since there are 3 instances of class Yes and 7 instances of class No in the training data. Thus,

$$I(\text{parent}) = -\frac{3}{10}\log_2\frac{3}{10} - \frac{7}{10}\log_2\frac{7}{10} = 0.881$$

The information gains for Home Owner and Marital Status are each given by

$$\Delta_{info}(\text{Home Owner}) = 0.881 - 0.690 = 0.191$$
$$\Delta_{info}(\text{Marital Status}) = 0.881 - 0.686 = 0.195$$

The information gain for Marital Status is thus higher due to its lower weighted entropy, which will thus be considered for splitting.

Binary Splitting of Qualitative Attributes

Consider building a decision tree using only binary splits and the Gini index as the impurity measure. Figure 3.13 shows examples of four candidate splitting criteria for the Home Owner and Marital Status attributes. Since there are 3 borrowers in the training set who defaulted and 7 others who repaid their loan (see Table in Figure 3.13), the Gini index of the parent node before splitting is

$$1 - \left(\frac{3}{10}\right)^2 - \left(\frac{7}{10}\right)^2 = 0.420.$$

If Home Owner is chosen as the splitting attribute, the Gini index for the child nodes N_1 and N_2 are 0 and 0.490, respectively. The weighted average Gini index for the children is

$$(3/10) \times 0 + (7/10) \times 0.490 = 0.343,$$

where the weights represent the proportion of training instances assigned to each child. The gain using Home Owner as splitting attribute is 0.420 - 0.343 = 0.077. Similarly, we can apply a binary split on the Marital Status attribute. However, since Marital Status is a nominal attribute with three outcomes, there are three possible ways to group the attribute values into a binary split. The weighted average Gini index of the children for each candidate binarysplit is shown in Figure 3.13. Based on these results, Home Owner and the lastbinary split using Marital Status are clearly the best candidates, since they both produce the lowest weighted average Gini index. Binary splits can also be used for ordinal attributes, if the binary partitioning of the attribute values does not violate the ordering property of the values.

Binary Splitting of Quantitative Attributes

Consider the problem of identifying the best binary split Annual Income $\leq \tau$ for the preceding loan approval classification problem. As discussed previously,

Class	No	No	No	Yes	Yes	Yes	No	No	No	No
	Annual Income (in '000s)									
→	60	70	75	85	90	95	100	120	125	220
Sorted Values →	55	65	72.5	80	87.5	92.5	97.5	110	122.5	172.5	230
Split Positions	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >
Yes	0 3	0 3	0 3	0 3	1 2	2 1	3 0	3 0	3 0	3 0	3 0
No	0 7	1 6	2 5	3 4	3 4	3 4	3 4	4 3	5 2	6 1	7 0
Gini	0.420	0.400	0.375	0.343	0.417	0.400	***0.300***	0.343	0.375	0.400	0.420

Figure 3.14. Splitting continuous attributes.

even though τ can take any value between the minimum and maximum values of annual income in the training set, it is sufficient to only consider the annual income values observed in the training set as candidate split positions. For each candidate τ , the training set is scanned once to count the number of borrowers with annual income less than or greater than τ along with their class proportions. We can then compute the Gini index at each candidatesplit position and choose the τ that produces the lowest value. Computing theGini index at each candidate split position requires $O(N)$ operations, where$N$ is the number of training instances. Since there are at most N possible candidates, the overall complexity of this brute-force method is $O(N^2)$. It is possible to reduce the complexity of this problem to $O(N \log N)$ by using a method described as follows (see illustration in Figure 3.14). In this method, we first sort the training instances based on their annual income, a one-time cost that requires $O(N \log N)$ operations. The candidate split positions are given by the midpoints between every two adjacent sorted values: $55,000, $65,000, $72,500, and so on. For the first candidate, since none of the instances has an annual income less than or equal to $55,000, the Gini index for the child node with Annual Income < $55,000 is equal to zero. In contrast, there are 3 training instances of class Yes and 7 instances of class No with annual income greater than $55,000. The Gini index for this node is 0.420. The weighted average Gini index for the first candidate split position, τ = $55, 000, is equal to $0 \times 0 + 1 \times 0.420 = 0.420$.

For the next candidate, τ = $65, 000, the class distribution of its child nodes can be obtained with a simple update of the distribution for the previous candidate. This is because, as τ increases from $55,000 to $65,000, there is only one training instance affected by the change. By examining the class labelof the affected training instance, the new class distribution is obtained. For example, as τ increases to $65,000, there is only one borrower in the training

set, with an annual income of $60,000, affected by this change. Since the class label for the borrower is No, the count for class No increases from 0 to 1 (for Annual Income $\leq$ $65,000) and decreases from 7 to 6 (for Annual Income $>$ $65,000), as shown in Figure 3.14. The distribution for the Yes class remains unaffected. The updated Gini index for this candidate split position is 0.400.

This procedure is repeated until the Gini index for all candidates are found. The best split position corresponds to the one that produces the lowest Gini index, which occurs at $\tau = \$97,500$. Since the Gini index at each candidate split position can be computed in $O(1)$ time, the complexity of finding the best split position is $O(N)$ once all the values are kept sorted, a one-time operation that takes $O(N \log N)$ time. The overall complexity of this method is thus $O(N \log N)$, which is much smaller than the $O(N^2)$ time taken by the brute-force method. The amount of computation can be further reduced by considering only candidate split positions located between two adjacent sorted instances with different class labels. For example, we do not need to consider candidate split positions located between $60,000 and $75,000 because all three instances with annual income in this range ($60,000, $70,000, and $75,000) have the same class labels. Choosing a split position within this range only increases the degree of impurity, compared to a split position located outside this range. Therefore, the candidate split positions at $\tau = \$65,000$ and $\tau = \$72,500$ can be ignored. Similarly, we do not need to consider the candidate split positions at $87,500, $92,500, $110,000, $122,500, and $172,500 because they are located between two adjacent instances with the same labels. This strategy reduces the number of candidate split positions to consider from 9 to 2 (excluding the two boundary cases $\tau = \$55,000$ and $\tau = \$230,000$).

Gain Ratio

One potential limitation of impurity measures such as entropy and Gini index is that they tend to favor qualitative attributes with large number of distinct values. Figure 3.12 shows three candidate attributes for partitioning the data set given in Table 3.3. As previously mentioned, the attribute Marital Status is a better choice than the attribute Home Owner, because it provides a larger information gain. However, if we compare them against Customer ID, the latter produces the purest partitions with the maximum information gain, since the weighted entropy and Gini index is equal to zero for its children. Yet, Customer ID is not a good attribute for splitting because it has a unique value for each instance. Even though a test condition involving Customer ID will accurately classify every instance in the training data, we cannot use such a test condition on new test instances with Customer ID values that haven't

been seen before during training. This example suggests having a low impurity value alone is insufficient to find a good attribute test condition for a node. As we will see later in Section 3.4, having more number of child nodes can make a decision tree more complex and consequently more susceptible to overfitting. Hence, the number of children produced by the splitting attribute should also be taken into consideration while deciding the best attribute test condition.

There are two ways to overcome this problem. One way is to generate only binary decision trees, thus avoiding the difficulty of handling attributes with varying number of partitions. This strategy is employed by decision tree classifiers such as CART. Another way is to modify the splitting criterion to take into account the number of partitions produced by the attribute. For example, in the C4.5 decision tree algorithm, a measure known as **gain ratio** is used to compensate for attributes that produce a large number of child nodes. This measure is computed as follows:

$$\text{Gain ratio} = \frac{\Delta_{\text{info}}}{\text{Split Info}} = \frac{\text{Entropy(Parent)} - \sum_{i=1}^{k} \frac{N(v_i)}{N} \text{Entropy}(v_i)}{-\sum_{i=1}^{k} \frac{N(v_i)}{N} \log_2 \frac{N(v_i)}{N}} \tag{3.9}$$

where $N(v_i)$ is the number of instances assigned to node v_i and k is the total number of splits. The split information measures the entropy of splitting a node into its child nodes and evaluates if the split results in a larger numberof equally-sized child nodes or not. For example, if every partition has the same number of instances, then $\forall i : N(v_i)/N = 1/k$ and the split information would be equal to $\log_2 k$. Thus, if an attribute produces a large number of splits, its split information is also large, which in turn, reduces the gain ratio.

Example 3.4. [Gain Ratio] Consider the data set given in Exercise 2 on page 185. We want to select the best attribute test condition among the following three attributes: Gender, Car Type, and Customer ID. The entropy before splitting is

$$\text{Entropy(parent)} = -\frac{10}{20}\log_2 \frac{10}{20} - \frac{10}{20}\log_2 \frac{10}{20} = 1.$$

If Gender is used as attribute test condition:

$$\text{Entropy(children)} = \frac{10}{20}\left(-\frac{6}{10}\log_2 \frac{6}{10} - \frac{4}{10}\log_2 \frac{4}{10}\right) \times 2 = 0.971$$

$$\text{Gain Ratio} = \frac{1 - 0.971}{-\frac{10}{20}\log_2 \frac{10}{20} - \frac{10}{20}\log_2 \frac{10}{20}} = \frac{0.029}{1} = 0.029$$

If Car Type is used as attribute test condition:

$$\text{Entropy(children)} = \frac{4}{20}\left(-\frac{1}{4}\log_2\frac{1}{4}-\frac{3}{4}\log_2\frac{3}{4}\right)+\frac{8}{20}\times 0$$

$$+\frac{8}{20}\left(-\frac{1}{8}\log_2\frac{1}{8}-\frac{7}{8}\log_2\frac{7}{8}\right)=0.380$$

$$\text{Gain Ratio} = \frac{1-0.380}{-\frac{4}{20}\log_2\frac{4}{20}-\frac{8}{20}\log_2\frac{8}{20}-\frac{8}{20}\log_2\frac{8}{20}} = \frac{0.620}{1.52} = 0.41$$

Finally, if Customer ID is used as attribute test condition:

$$\text{Entropy(children)} = \frac{1}{20}\left(-\frac{1}{1}\log_2\frac{1}{1}-\frac{0}{1}\log_2\frac{0}{1}\right)\times 20 = 0$$

$$\text{Gain Ratio} = \frac{1-0}{-\frac{1}{20}\log_2\frac{1}{20}\times 20} = \frac{1}{4.32} = 0.23$$

Thus, even though Customer ID has the highest information gain, its gain ratio is lower than Car Type since it produces a larger number of splits.

3.3.4 Algorithm for Decision Tree Induction

Algorithm 3.1 presents a pseudocode for decision tree induction algorithm. The input to this algorithm is a set of training instances E along with the attribute set F . The algorithm works by recursively selecting the best attribute to split the data (Step 7) and expanding the nodes of the tree (Steps 11 and 12) until the stopping criterion is met (Step 1). The details of this algorithm are explained below.

1. The createNode() function extends the decision tree by creating a new node. A node in the decision tree either has a test condition, denoted as *node.test_cond*, or a class label, denoted as *node.label*.

2. The find best split() function determines the attribute test condi- tion for partitioning the training instances associated with a node. The splitting attribute chosen depends on the impurity measure used. The popular measures include entropy and the Gini index.

3. The Classify() function determines the class label to be assigned to a leaf node. For each leaf node t, let $p(i|t)$ denote the fraction of training instances from class i associated with the node t. The label assigned

Algorithm 3.1 A skeleton decision tree induction algorithm.

```
TreeGrowth (E, F)
 1: if stopping_cond(E,F) = true then
 2:    leaf = createNode().
 3:    leaf.label = Classify(E).
 4:    return leaf.
 5: else
 6:    root = createNode().
 7:    root.test_cond = find best split(E, F).
 8:    let V = {v|v is a possible outcome of root.test_cond }.
 9:    for each v ∈ V do
10:       E_v = {e | root.test_cond(e) = v and e ∈ E}.
11:       child = TreeGrowth(E_v, F).
12:       add child as descendent of root and label the edge (root → child) as v.
13:    end for
14: end if
15: return root.
```

to the leaf node is typically the one that occurs most frequently in the training instances that are associated with this node.

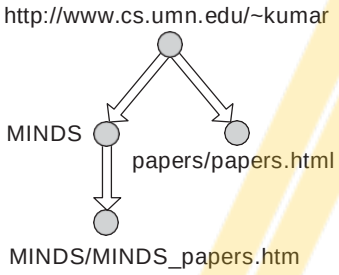
$$leaf.label = \underset{i}{\text{argmax}}\, p(i|t), \tag{3.10}$$

where the argmax operator returns the class i that maximizes $p(i|t)$. Besides providing the information needed to determine the class label of a leaf node, $p(i|t)$ can also be used as a rough estimate of the probability that an instance assigned to the leaf node t belongs to class i. Sections 4.11.2 and 4.11.4 in the next chapter describe how such probability estimates can be used to determine the performance of a decision tree under different cost functions.

4. The stopping cond() function is used to terminate the tree-growing process by checking whether all the instances have identical class label or attribute values. Since decision tree classifiers employ a top-down, recursive partitioning approach for building a model, the number of training instances associated with a node decreases as the depth of the tree increases. As a result, a leaf node may contain too few training instances to make a statistically significant decision about its class label. This is known as the **data fragmentation** problem. One way to avoidthis problem is to disallow splitting of a node when the number of instances associated with the node fall below a certain threshold. A

Session	IP Address	Timestamp	Request Method	Requested Web Page	Protocol	Status	Number of Bytes	Referrer	User Agent
1	160.11.11.11	08/Aug/2004 10:15:21	GET	http://www.cs.umn.edu/~kumar	HTTP/1.1	200	6424		Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:15:34	GET	http://www.cs.umn.edu/~kumar/MINDS	HTTP/1.1	200	41378	http://www.cs.umn.edu/~kumar	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:15:41	GET	http://www.cs.umn.edu/~kumar/MINDS/MINDS_papers.htm	HTTP/1.1	200	1018516	http://www.cs.umn.edu/~kumar/MINDS	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:16:11	GET	http://www.cs.umn.edu/~kumar/papers/papers.html	HTTP/1.1	200	7463	http://www.cs.umn.edu/~kumar	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
2	35.9.2.2	08/Aug/2004 10:16:15	GET	http://www.cs.umn.edu/~steinbac	HTTP/1.0	200	3149		Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7) Gecko/20040616

(a) Example of a Web server log.



(b) Graph of a Web session.

Attribute Name	Description
totalPages	Total number of pages retrieved in a Web session
ImagePages	Total number of image pages retrieved in a Web session
TotalTime	Total amount of time spent by Web site visitor
RepeatedAccess	The same page requested more than once in a Web session
ErrorRequest	Errors in requesting for Web pages
GET	Percentage of requests made using GET method
POST	Percentage of requests made using POST method
HEAD	Percentage of requests made using HEAD method
Breadth	Breadth of Web traversal
Depth	Depth of Web traversal
MultiIP	Session with multiple IP addresses
MultiAgent	Session with multiple user agents

(c) Derived attributes for Web robot detection.

Figure 3.15. Input data for web robot detection.

more systematic way to control the size of a decision tree (number of leaf nodes) will be discussed in Section 3.5.4.

3.3.5 Example Application: Web Robot Detection

Consider the task of distinguishing the access patterns of web robots from those generated by human users. A web robot (also known as a web crawler) is a software program that automatically retrieves files from one or more websites by following the hyperlinks extracted from an initial set of seed URLs. These programs have been deployed for various purposes, from gathering web pages on behalf of search engines to more malicious activities such as spamming and committing click frauds in online advertisements.

The web robot detection problem can be cast as a binary classification task. The input data for the classification task is a web server log, a sampleof which is shown in Figure 3.15(a). Each line in the log file corresponds to a

request made by a client (i.e., a human user or a web robot) to the web server. The fields recorded in the web log include the client's IP address, timestamp of the request, URL of the requested file, size of the file, and **user agent**, which is a field that contains identifying information about the client. For human users, the user agent field specifies the type of web browser or mobile device used to fetch the files, whereas for web robots, it should technically contain the name of the crawler program. However, web robots may conceal their true identities by declaring their user agent fields to be identical to known browsers. Therefore, user agent is not a reliable field to detect web robots.

The first step toward building a classification model is to precisely define a data instance and associated attributes. A simple approach is to consider each log entry as a data instance and use the appropriate fields in the log file as its attribute set. This approach, however, is inadequate for several reasons. First, many of the attributes are nominal-valued and have a wide range of domain values. For example, the number of unique client IP addresses, URLs, and referrers in a log file can be very large. These attributes are undesirable for building a decision tree because their split information is extremely high (see Equation (3.9)). In addition, it might not be possible to classify test instances containing IP addresses, URLs, or referrers that are not present in the training data. Finally, by considering each log entry as a separate data instance, we disregard the sequence of web pages retrieved by the client—a critical piece of information that can help distinguish web robot accesses from those of a human user.

A better alternative is to consider each web session as a data instance. A web session is a sequence of requests made by a client during a given visit to the website. Each web session can be modeled as a directed graph, in which the nodes correspond to web pages and the edges correspond to hyperlinks connecting one web page to another. Figure 3.15(b) shows a graphical representation of the first web session given in the log file. Every web session can be characterized using some meaningful attributes about the graph that contain discriminatory information. Figure 3.15(c) shows some of the attributes extracted from the graph, including the depth and breadth of its corresponding tree rooted at the entry point to the website. For example, the depth and breadth of the tree shown in Figure 3.15(b) are both equal to two.

The derived attributes shown in Figure 3.15(c) are more informative than the original attributes given in the log file because they characterize the behavior of the client at the website. Using this approach, a data set containing 2916 instances was created, with equal numbers of sessions due to web robots (class 1) and human users (class 0). 10% of the data were reserved for training while the remaining 90% were used for testing. The induced decision tree is

shown in Figure 3.16, which has an error rate equal to 3.8% on the training set and 5.3% on the test set. In addition to its low error rate, the tree also reveals some interesting properties that can help discriminate web robots fromhuman users:

1. Accesses by web robots tend to be broad but shallow, whereas accesses by human users tend to be more focused (narrow but deep).

2. Web robots seldom retrieve the image pages associated with a web page.

3. Sessions due to web robots tend to be long and contain a large number of requested pages.

4. Web robots are more likely to make repeated requests for the same web page than human users since the web pages retrieved by human users are often cached by the browser.

3.3.6 Characteristics of Decision Tree Classifiers

The following is a summary of the important characteristics of decision tree induction algorithms.

1. **Applicability:** Decision trees are a nonparametric approach for building classification models. This approach does not require any prior assumption about the probability distribution governing the class and attributes of the data, and thus, is applicable to a wide variety of data sets. It is also applicable to both categorical and continuous data without requiring the attributes to be transformed into a common representation via binarization, normalization, or standardization. Unlike some binary classifiers described in Chapter 4, it can also deal with multiclass problems without the need to decompose them into multiple binary classification tasks. Another appealing feature of decision tree classifiers is that the induced trees, especially the shorter ones, are relatively easy to interpret. The accuracies of the trees are also quite comparable to other classification techniques for many simple data sets.

2. **Expressiveness:** A decision tree provides a universal representation for discrete-valued functions. In other words, it can encode any func- tion of discrete-valued attributes. This is because every discrete-valued function can be represented as an assignment table, where every unique combination of discrete attributes is assigned a class label. Since every

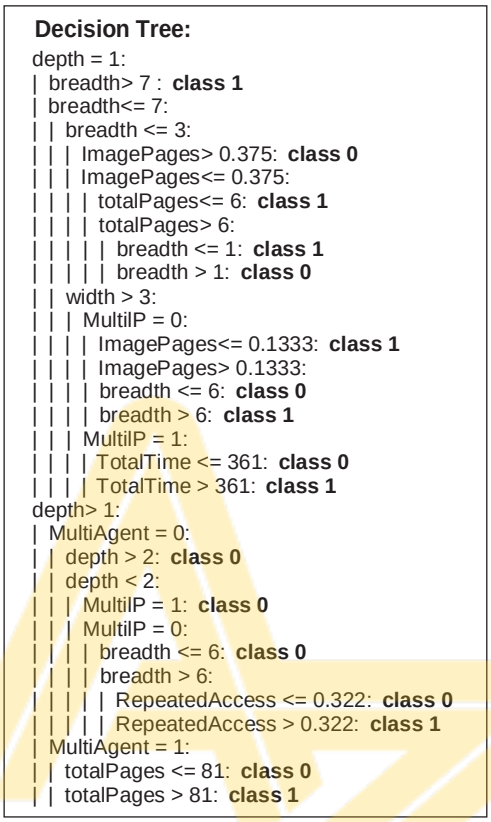


Figure 3.16. Decision tree model for web robot detection.

combination of attributes can be represented as a leaf in the decision tree, we can always find a decision tree whose label assignments at the leaf nodes matches with the assignment table of the original function. Decision trees can also help in providing compact representations of functions when some of the unique combinations of attributes can be represented by the same leaf node. For example, Figure 3.17 shows the assignment table of the Boolean function $(A \wedge B) \vee (C \wedge D)$ involving four binary attributes, resulting in a total of $2^4 = 16$ possible assignments. The tree shown in Figure 3.17 shows a compressed encoding of this assignment table. Instead of requiring a fully-grown tree with 16 leaf nodes, it is possible to encode the function using a simpler tree with only 7 leaf nodes. Nevertheless, not all decision trees for discrete-valued attributes can be simplified. One notable example is the parity function,

A	B	C	D	class
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

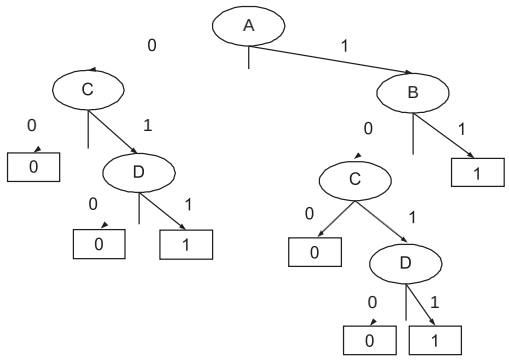


Figure 3.17. Decision tree for the Boolean function $(A \wedge B) \vee (C \wedge D)$.

whose value is 1 when there is an even number of true values among its Boolean attributes, and 0 otherwise. Accurate modeling of such a function requires a full decision tree with 2^d nodes, where d is the number of Boolean attributes (see Exercise 1 on page 185).

3. **Computational Efficiency:** Since the number of possible decision trees can be very large, many decision tree algorithms employ a heuristic-based approach to guide their search in the vast hypothesis space. For example, the algorithm presented in Section 3.3.4 uses a greedy, top-down, recursive partitioning strategy for growing a decision tree. For many data sets, such techniques quickly construct a reasonably good decision tree even when the training set size is very large. Furthermore, once a decision tree has been built, classifying a test record is extremely fast, with a worst-case complexity of $O(w)$, where w is the maximum depth of the tree.

4. **Handling Missing Values:** A decision tree classifier can handle missing attribute values in a number of ways, both in the training and the test sets. When there are missing values in the test set, the classifier must decide which branch to follow if the value of a splitting node attribute is missing for a given test instance. One approach, known as the **probabilistic split method**, which is employed by the C4.5 decision tree classifier, distributes the data instance to every child of the splitting node according to the probability that the missing attribute has a particular value. In contrast, the CART algorithm uses the **surrogate split method**, where the instance whose splitting attribute value is

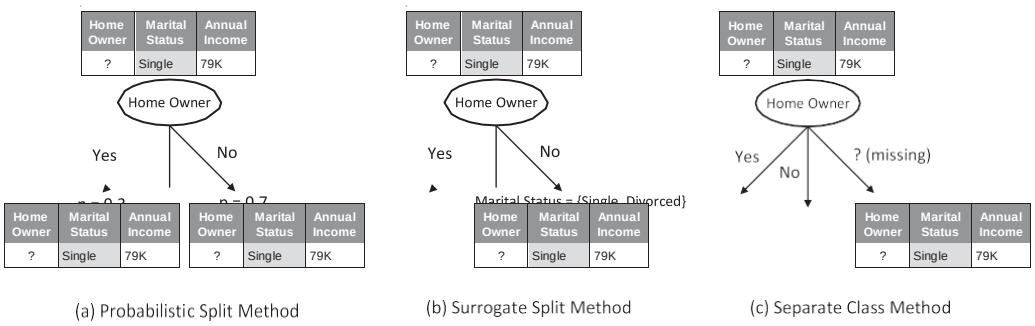


Figure 3.18. Methods for handling missing attribute values in decision tree classifier.

missing is assigned to one of the child nodes based on the value of another non-missing surrogate attribute whose splits most resemble the partitions made by the missing attribute. Another approach, known as the **separate class method** is used by the CHAID algorithm, where the missing value is treated as a separate categorical value distinct from other values of the splitting attribute. Figure 3.18 shows an example of the three different ways for handling missing values in a decision tree classifier. Other strategies for dealing with missing values are based on data preprocessing, where the instance with missing value is either imputed with the mode (for categorical attribute) or mean (for continuous attribute) value or discarded before the classifier is trained.

During training, if an attribute *v* has missing values in some of thetraining instances associated with a node, we need a way to measurethe gain in purity if *v* is used for splitting. One simple way is to exclude instances with missing values of *v* in the counting of instances associatedwith every child node, generated for every possible outcome of *v*. Further, if *v* is chosen as the attribute test condition at a node, training instances with missing values of *v* can be propagated to the child nodes usingany of the methods described above for handling missing values in test instances.

5. **Handling Interactions among Attributes:** Attributes are consid- ered interacting if they are able to distinguish between classes when used together, but individually they provide little or no information.Due to the greedy nature of the splitting criteria in decision trees, such attributes could be passed over in favor of other attributes that are not asuseful. This could result in more complex decision trees than necessary.

Hence, decision trees can perform poorly when there are interactions among attributes.

To illustrate this point, consider the three-dimensional data shown in Figure 3.19(a), which contains 2000 data points from one of two classes, denoted as + and ◦ in the diagram. Figure 3.19(b) shows the distribution of the two classes in the two-dimensional space involving attributes X and Y, which is a noisy version of the XOR Boolean function. Wecan see that even though the two classes are well-separated in this two-dimensional space, neither of the two attributes contain sufficient information to distinguish between the two classes when used alone. For example, the entropies of the following attribute test conditions:$X \leq 10$ and $Y \leq 10$, are close to 1, indicating that neither X nor Y provide any reduction in the impurity measure when used individually.X and Y thus represent a case of interaction among attributes. The data set also contains a third attribute, Z, in which both classes are distributed uniformly, as shown in Figures 3.19(c) and 3.19(d), and hence, the entropy of any split involving Z is close to 1. As a result,Z is as likely to be chosen for splitting as the interacting but useful attributes, X and Y. For further illustration of this issue, readers are referred to Example 3.7 in Section 3.4.1 and Exercise 7 at the end of this chapter.

6. **Handling Irrelevant Attributes:** An attribute is irrelevant if it is not useful for the classification task. Since irrelevant attributes are poorly associated with the target class labels, they will provide little or no gain in purity and thus will be passed over by other more relevant features. Hence, the presence of a small number of irrelevant attributes will not impact the decision tree construction process. However, not all attributes that provide little to no gain are irrelevant (see Figure 3.19). Hence, if the classification problem is complex (e.g., involving interactions among attributes) and there are a large number of irrelevant attributes, then some of these attributes may be accidentally chosen during the tree-growing process, since they may provide a better gain than a relevant attribute just by random chance. Feature selection techniques can help to improve the accuracy of decision trees by eliminating the irrelevant attributes during preprocessing. We will investigate the issue of too many irrelevant attributes in Section 3.4.1.

7. **Handling Redundant Attributes:** An attribute is redundant if it is strongly correlated with another attribute in the data. Since redundant

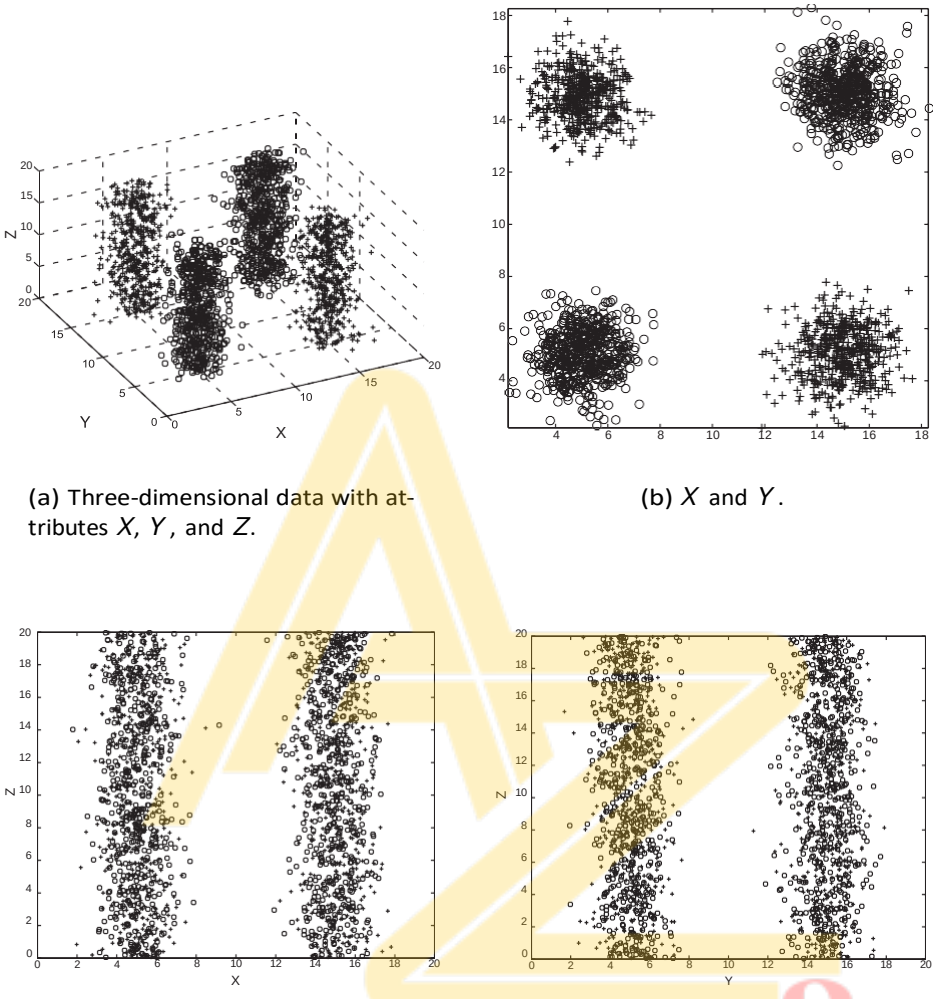


Figure 3.19. Example of a XOR data involving X and Y, along with an irrelevant attribute Z.

attributes show similar gains in purity if they are selected for splitting, only one of them will be selected as an attribute test condition in the decision tree algorithm. Decision trees can thus handle the presence of redundant attributes.

8. **Using Rectilinear Splits:** The test conditions described so far in this chapter involve using only a single attribute at a time. As a consequence,

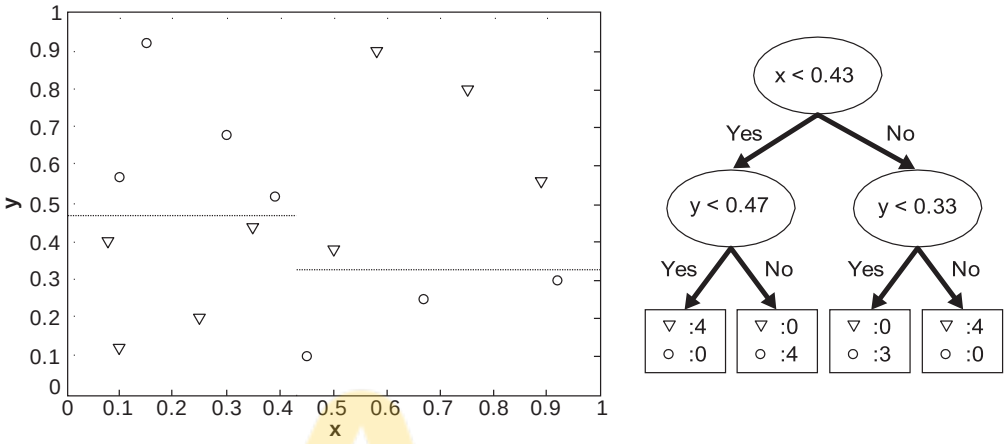


Figure 3.20. Example of a decision tree and its decision boundaries for a two-dimensional data set.

the tree-growing procedure can be viewed as the process of partitioning the attribute space into disjoint regions until each region contains records of the same class. The border between two neighboring regions of different classes is known as a **decision boundary**. Figure 3.20 shows the decision tree as well as the decision boundary for a binary classification problem. Since the test condition involves only a single attribute, the decision boundaries are rectilinear; i.e., parallel to the coordinate axes. This limits the expressiveness of decision trees in representing decision boundaries of data sets with continuous attributes. Figure 3.21 shows a two-dimensional data set involving binary classes that cannot be perfectly classified by a decision tree whose attribute test conditions are defined based on single attributes. The binary classes in the data set are generated from two skewed Gaussian distributions, centered at (8,8) and (12,12), respectively. The true decision boundary is represented by the diagonal dashed line, whereas the rectilinear decision boundary produced by the decision tree classifier is shown by the thick solid line.

In contrast, an **oblique decision tree** may overcome this limitation by allowing the test condition to be specified using more than one attribute. For example, the binary classification data shown in Figure 3.21 can be easily represented by an oblique decision tree with a single root node with test condition

$$x + y < 20.$$

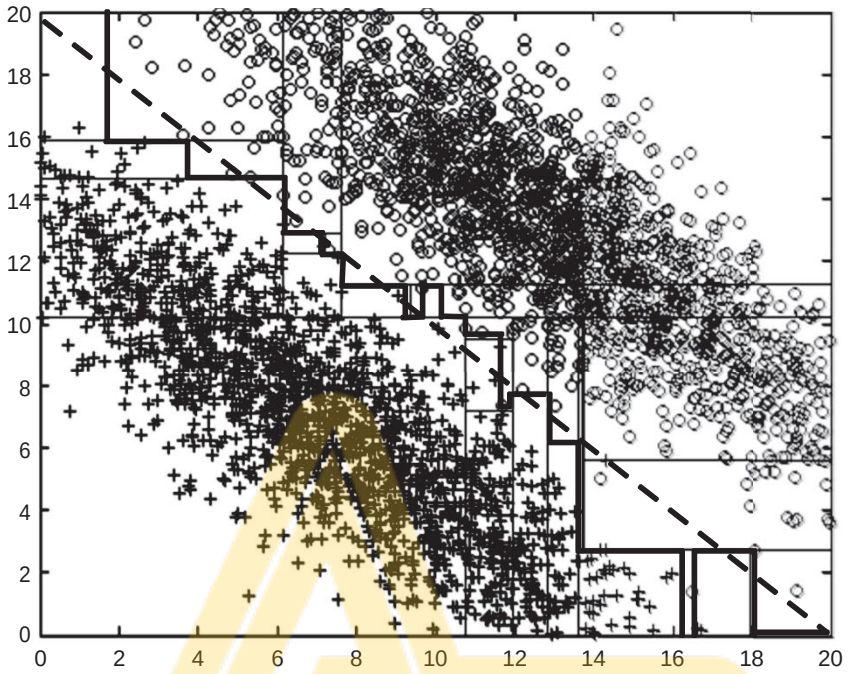


Figure 3.21. Example of data set that cannot be partitioned optimally using a decision tree with single attribute test conditions. The true decision boundary is shown by the dashed line.

Although an oblique decision tree is more expressive and can produce more compact trees, finding the optimal test condition is computationally more expensive.

9. **Choice of Impurity Measure:** It should be noted that the choice of impurity measure often has little effect on the performance of decision tree classifiers since many of the impurity measures are quite consistent with each other, as shown in Figure 3.11 on page 129. Instead, the strategy used to prune the tree has a greater impact on the final tree than the choice of impurity measure.

3.5 Model Evaluation

The previous section discussed several approaches for model selection that can be used to learn a classification model from a training set *D.train*. Herewe discuss methods for estimating its generalization performance, i.e. its performance on unseen instances outside of *D.train*. This process is known as **model evaluation**.

Note that model selection approaches discussed in Section 3.5 also compute an estimate of the generalization performance using the training set *D.train*.

However, these estimates are *biased* indicators of the performance on unseen instances, since they were used to guide the selection of classification model. For example, if we use the validation error rate for model selection (as described in Section 3.5.1), the resulting model would be deliberately chosen to minimize the errors on the validation set. The validation error rate may thus under-estimate the true generalization error rate, and hence cannot be reliably used for model evaluation.

A correct approach for model evaluation would be to assess the performance of a learned model on a labeled test set has not been used at any stage of model selection. This can be achieved by partitioning the entire set of labeled instances D, into two disjoint subsets, *D.train*, which is used for model selection and *D.test*, which is used for computing the test error rate, err_{test}. In the following, we present two different approaches for partitioning D into *D.train* and *D.test*, and computing the test error rate, err_{test}.

3.6.1 Holdout Method

The most basic technique for partitioning a labeled data set is the holdout method, where the labeled set D is randomly partitioned into two disjoint sets, called the training set *D.train* and the test set *D.test*. A classification model is then induced from *D.train* using the model selection approaches presented in Section 3.5, and its error rate on *D.test*, err_{test}, is used as an estimate of the generalization error rate. The proportion of data reserved for training and for testing is typically at the discretion of the analysts, e.g.,two-thirds for training and one-third for testing.

Similar to the trade-off faced while partitioning *D.train* into *D.tr* and *D.val* in Section 3.5.1, choosing the right fraction of labeled data to be used for training and testing is non-trivial. If the size of *D.train* is small, the learned classification model may be improperly learned using an insufficient number of training instances, resulting in a biased estimation of generalization performance. On the other hand, if the size of *D.test* is small, err_{test} may be less reliable as it would be computed over a small number of test instances. Moreover, err_{test} can have a high variance as we change the random partitioning of D into *D.train* and *D.test*.

The holdout method can be repeated several times to obtain a distribution of the test error rates, an approach known as **random subsampling** or repeated holdout method. This method produces a distribution of the error rates that can be used to understand the variance of err_{test}.

3.6.2 Cross-Validation

Cross-validation is a widely-used model evaluation method that aims to make effective use of all labeled instances in D for both training and testing. To illustrate this method, suppose that we are given a labeled set that we have randomly partitioned into three equal-sized subsets, S_1, S_2, and S_3, as shown in Figure 3.33. For the first run, we train a model using subsets S_2 and S_3 (shown as empty blocks) and test the model on subset S_1. The test error rate

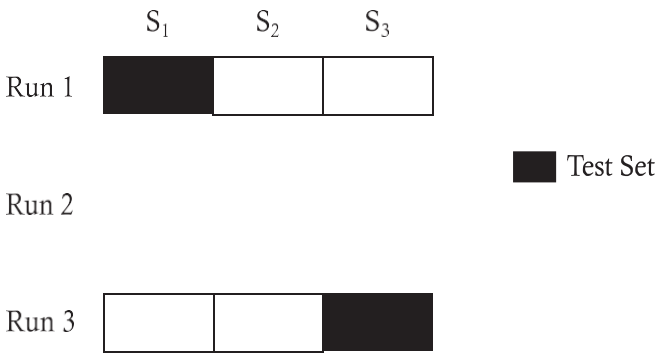


Figure 3.33. Example demonstrating the technique of 3-fold cross-validation.

on S_1, denoted as $err(S_1)$, is thus computed in the first run. Similarly, for the second run, we use S_1 and S_3 as the training set and S_2 as the test set, to compute the test error rate, $err(S_2)$, on S_2. Finally, we use S_1 and S_3 for training in the third run, while S_3 is used for testing, thus resulting in the test error rate $err(S_3)$ for S_3. The overall test error rate is obtained by summing up the number of errors committed in each test subset across all runs and dividing it by the total number of instances. This approach is called three-fold cross-validation.

The k-fold cross-validation method generalizes this approach by segmenting the labeled data D (of size N) into k equal-sized partitions (or folds). During the i^{th} run, one of the partitions of D is chosen as $D.test(i)$ for testing, while the rest of the partitions are used as $D.train(i)$ for training. A model $m(i)$ is learned using $D.train(i)$ and applied on $D.test(i)$ to obtain the sum of test errors, $err_{sum}(i)$. This procedure is repeated k times. The total test error rate, err_{test}, is then computed as

$$err_{test} = \frac{\sum_{i=1}^{k} err_{sum}(i)}{N}. \tag{3.14}$$

Every instance in the data is thus used for testing exactly once and for training exactly $(k-1)$ times. Also, every run uses $(k-1)/k$ fraction of the data for training and $1/k$ fraction for testing.

The right choice of k in k-fold cross-validation depends on a number of characteristics of the problem. A small value of k will result in a smallertraining set at every run, which will result in a larger estimate of generalizationerror rate than what is expected of a model trained over the entire labeledset. On the other hand, a high value of k results in a larger training set at

every run, which reduces the bias in the estimate of generalization error rate. In the extreme case, when $k = N$, every run uses exactly one data instance for testing and the remainder of the data for testing. This special case of *k*-fold cross-validation is called the **leave-one-out** approach. This approach has the advantage of utilizing as much data as possible for training. However, leave-one-out can produce quite misleading results in some special scenarios, as illustrated in Exercise 11. Furthermore, leave-one-out can be computationally expensive for large data sets as the cross-validation procedure needs to be repeated *N* times. For most practical applications, the choice of *k* between 5 and 10 provides a reasonable approach for estimating the generalization error rate, because each fold is able to make use of 80% to 90% of the labeled data for training.

The *k*-fold cross-validation method, as described above, produces a single estimate of the generalization error rate, without providing any information about the variance of the estimate. To obtain this information, we can run *k*-fold cross-validation for every possible partitioning of the data into *k* partitions, and obtain a distribution of test error rates computed for every such partitioning. The average test error rate across all possible partitionings serves as a more robust estimate of generalization error rate. This approach of estimating the generalization error rate and its variance is known as the **complete cross-validation** approach. Even though such an estimate is quite robust, it is usually too expensive to consider all possible partitionings of a large data set into *k* partitions. A more practical solution is to repeat the cross-validation approach multiple times, using a different random partitioning of the data into *k* partitions at every time, and use the average test error rateas the estimate of generalization error rate. Note that since there is onlyone possible partitioning for the leave-one-out approach, it is not possible to estimate the variance of generalization error rate, which is another limitation of this method.

The *k*-fold cross-validation does not guarantee that the fraction of positive and negative instances in every partition of the data is equal to the fraction observed in the overall data. A simple solution to this problem is to perform a stratified sampling of the positive and negative instances into *k* partitions, an approach called **stratified cross-validation**.

In *k*-fold cross-validation, a different model is learned at every run and the performance of every one of the *k* models on their respective test folds is then aggregated to compute the overall test error rate, err_{test}. Hence, err_{test} does not reflect the generalization error rate of any of the *k* models. Instead, it reflects the *expected* generalization error rate of the *model selectionapproach*, when applied on a training set of the same size as one of the training

folds (N (k − 1)/k). This is different than the err_{test} computed in the holdout method, which exactly corresponds to the specific model learned over *D.train*. Hence, although effectively utilizing every data instance in D for training and testing, the err_{test} computed in the cross-validation method does not represent the performance of a single model learned over a specific *D.train*.

Nonetheless, in practice, err_{test} is typically used as an estimate of the generalization error of a model built on D. One motivation for this is that when the size of the training folds is closer to the size of the overall data (when k is large), then err_{test} resembles the expected performance of a model learned over a data set of the same size as D. For example, when k is 10, everytraining fold is 90% of the overall data. The err_{test} then should approach the expected performance of a model learned over 90% of the overall data, whichwill be close to the expected performance of a model learned over D.

Rule-Based Classifier

- Classify records by using a collection of "if...then..." rules
- Rule: (*Condition*) → *y*
  - where
    - *Condition* is a conjunction of tests on attributes
    - *y* is the class label
  - Examples of classification rules:
    - (Blood Type=Warm) ∧ (Lay Eggs=Yes) → Birds
    - (Taxable Income < 50K) ∧ (Refund=Yes) → Evade=No

Rule-based Classifier (Example)

Name	Blood Type	Give Birth	Can Fly	Live in Water	Class
human	warm	yes	no	no	mammals
python	cold	no	no	no	reptiles
salmon	cold	no	no	yes	fishes
whale	warm	yes	no	yes	mammals
frog	cold	no	no	sometimes	amphibians
komodo	cold	no	no	no	reptiles
bat	warm	yes	yes	no	mammals
pigeon	warm	no	yes	no	birds
cat	warm	yes	no	no	mammals
leopard shark	cold	yes	no	yes	fishes
turtle	cold	no	no	sometimes	reptiles
penguin	warm	no	no	sometimes	birds
porcupine	warm	yes	no	no	mammals
eel	cold	no	no	yes	fishes
salamander	cold	no	no	sometimes	amphibians
gila monster	cold	no	no	no	reptiles
platypus	warm	no	no	no	mammals
owl	warm	no	yes	no	birds
dolphin	warm	yes	no	yes	mammals
eagle	warm	no	yes	no	birds

R1: (Give Birth = no) ∧ (Can Fly = yes) → Birds
R2: (Give Birth = no) ∧ (Live in Water = yes) → Fishes
R3: (Give Birth = yes) ∧ (Blood Type = warm) → Mammals
R4: (Give Birth = no) ∧ (Can Fly = no) → Reptiles
R5: (Live in Water = sometimes) → Amphibians

Application of Rule-Based Classifier

- A rule *r* covers an instance **x** if the attributes of the instance satisfy the condition of the rule

R1: (Give Birth = no) ∧ (Can Fly = yes) → Birds
R2: (Give Birth = no) ∧ (Live in Water = yes) → Fishes
R3: (Give Birth = yes) ∧ (Blood Type = warm) → Mammals
R4: (Give Birth = no) ∧ (Can Fly = no) → Reptiles
R5: (Live in Water = sometimes) → Amphibians

Name	Blood Type	Give Birth	Can Fly	Live in Water	Class
hawk	warm	no	yes	no	?
grizzly bear	warm	yes	no	no	?

The rule R1 covers a hawk => Bird
The rule R3 covers the grizzly bear => Mammal

Rule Coverage and Accuracy

- Coverage of a rule:
  - Fraction of records that satisfy the antecedent of a rule
- Accuracy of a rule:
  - Fraction of records that satisfy the antecedent that also satisfy the consequent of a rule

Tid	Refund	Marital Status	Taxable Income	Class
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

(Status=Single) → No

Coverage = 40%, Accuracy = 50%

How does Rule-based Classifier Work?

R1: (Give Birth = no) ∧ (Can Fly = yes) → Birds
R2: (Give Birth = no) ∧ (Live in Water = yes) → Fishes
R3: (Give Birth = yes) ∧ (Blood Type = warm) → Mammals
R4: (Give Birth = no) ∧ (Can Fly = no) → Reptiles
R5: (Live in Water = sometimes) → Amphibians

Name	Blood Type	Give Birth	Can Fly	Live in Water	Class
lemur	warm	yes	no	no	?
turtle	cold	no	no	sometimes	?
dogfish shark	cold	yes	no	yes	?

A lemur triggers rule R3, so it is classified as a mammal
A turtle triggers both R4 and R5
A dogfish shark triggers none of the rules

Characteristics of Rule Sets: Strategy 1

- Mutually exclusive rules
  - Classifier contains mutually exclusive rules if the rules are independent of each other
  - Every record is covered by at most one rule
- Exhaustive rules
  - Classifier has exhaustive coverage if it accounts for every possible combination of attribute values
  - Each record is covered by at least one rule

Characteristics of Rule Sets: Strategy 2

- Rules are not mutually exclusive
  - A record may trigger more than one rule
  - Solution?
    - Ordered rule set
    - Unordered rule set – use voting schemes
- Rules are not exhaustive
  - A record may not trigger any rules
  - Solution?
    - Use a default class

Ordered Rule Set

- Rules are rank ordered according to their priority
  - An ordered rule set is known as a decision list
- When a test record is presented to the classifier
  - It is assigned to the class label of the highest ranked rule it has triggered
  - If none of the rules fired, it is assigned to the default class

R1: (Give Birth = no) ∧ (Can Fly = yes) → Birds
R2: (Give Birth = no) ∧ (Live in Water = yes) → Fishes
R3: (Give Birth = yes) ∧ (Blood Type = warm) → Mammals
R4: (Give Birth = no) ∧ (Can Fly = no) → Reptiles
R5: (Live in Water = sometimes) → Amphibians

Name	Blood Type	Give Birth	Can Fly	Live in Water	Class
turtle	cold	no	no	sometimes	?

Rule Ordering Schemes

- Rule-based ordering
  - Individual rules are ranked based on their quality
- Class-based ordering
  - Rules that belong to the same class appear together

Rule-based Ordering

(Refund=Yes) ==> No

(Refund=No, Marital Status={Single,Divorced}, Taxable Income<80K) ==> No

(Refund=No, Marital Status={Single,Divorced}, Taxable Income>80K) ==> Yes

(Refund=No, Marital Status={Married}) ==> No

Class-based Ordering

(Refund=Yes) ==> No

(Refund=No, Marital Status={Single,Divorced}, Taxable Income<80K) ==> No

(Refund=No, Marital Status={Married}) ==> No

(Refund=No, Marital Status={Single,Divorced}, Taxable Income>80K) ==> Yes

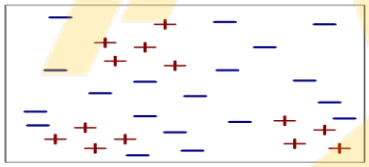
Building Classification Rules

- Direct Method:
  - Extract rules directly from data
  - Examples: RIPPER, CN2, Holte's 1R
- Indirect Method:
  - Extract rules from other classification models (e.g. decision trees, neural networks, etc).
  - Examples: C4.5rules

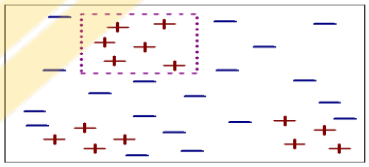
Direct Method: Sequential Covering

1. Start from an empty rule
2. Grow a rule using the Learn-One-Rule function
3. Remove training records covered by the rule
4. Repeat Step (2) and (3) until stopping criterion is met

Example of Sequential Covering

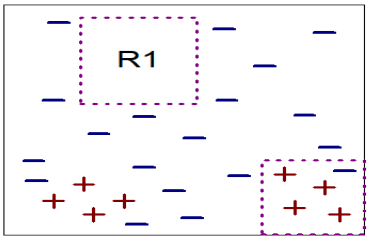


(i) Original Data

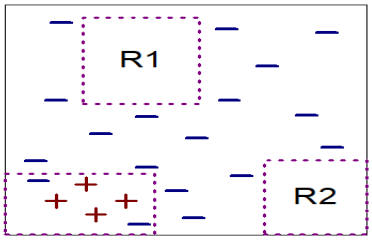


(ii) Step 1

Example of Sequential Covering...



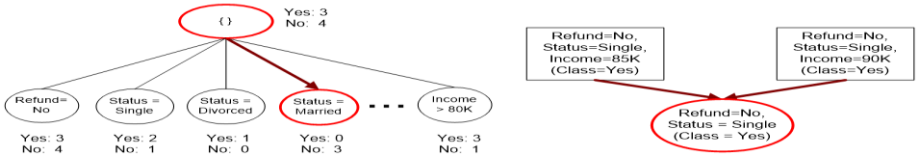
(iii) Step 2



(iv) Step 3

Rule Growing

- Two common strategies



Rule Evaluation

- Foil's Information Gain

FOIL: First Order Inductive Learner – an early rule-based learning algorithm

  - R0: {} => class (initial rule)
  - R1: {A} => class (rule after adding conjunct)
  - $Gain(R_0, R_1) = p_1 \times [\, log_2 \left(\frac{p_1}{p_1 + n_1}\right) - log_2 \left(\frac{p_0}{p_0 + n_0}\right)]$
  - p_0: number of positive instances covered by R0
    n_0: number of negative instances covered by R0
    p_1: number of positive instances covered by R1
    n_1: number of negative instances covered by R1

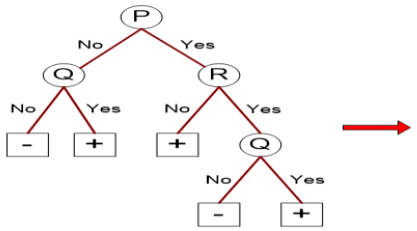
Direct Method: RIPPER

- For 2-class problem, choose one of the classes as positive class, and the other as negative class
  - Learn rules for positive class
  - Negative class will be default class
- For multi-class problem
  - Order the classes according to increasing class prevalence (fraction of instances that belong to a particular class)
  - Learn the rule set for smallest class first, treat the rest as negative class
  - Repeat with next smallest class as positive class

Direct Method: RIPPER

- Growing a rule:
  - Start from empty rule
  - Add conjuncts as long as they improve FOIL's information gain
  - Stop when rule no longer covers negative examples
  - Prune the rule immediately using incremental reduced error pruning
  - Measure for pruning: v = (p-n)/(p+n)
    - p: number of positive examples covered by the rule in the validation set
    - n: number of negative examples covered by the rule in the validation set
  - Pruning method: delete any final sequence of conditions that maximizes v
- Optimize the rule set:
  - For each rule *r* in the rule set ***R***
    - Consider 2 alternative rules:
      - Replacement rule (r*): grow new rule from scratch
      - Revised rule(r'): add conjuncts to extend the rule *r*
    - Compare the rule set for *r* against the rule set for r* and r'
    - Choose rule set that minimizes MDL principle
  - Repeat rule generation and rule optimization for the remaining positive examples

Indirect Methods



Rule Set

r1: (P=No,Q=No) ==> -
r2: (P=No,Q=Yes) ==> +
r3: (P=Yes,R=No) ==> +
r4: (P=Yes,R=Yes,Q=No) ==> -
r5: (P=Yes,R=Yes,Q=Yes) ==> +

Indirect Method: C4.5rules

- Extract rules from an unpruned decision tree
- For each rule, r: A → y,
  - consider an alternative rule r': A' → y where A' is obtained by removing one of the conjuncts in A
  - Compare the pessimistic error rate for r against all r's
  - Prune if one of the alternative rules has lower pessimistic error rate
  - Repeat until we can no longer improve generalization error

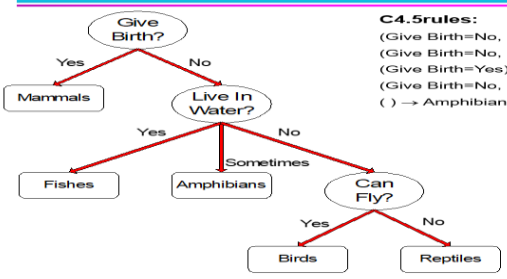
Indirect Method: C4.5rules

- Instead of ordering the rules, order subsets of rules (class ordering)
  - Each subset is a collection of rules with the same rule consequent (class)
  - Compute description length of each subset
    - Description length = L(error) + g L(model)
    - g is a parameter that takes into account the presence of redundant attributes in a rule set (default value = 0.5)

Example

Name	Give Birth	Lay Eggs	Can Fly	Live in Water	Have Legs	Class
human	yes	no	no	no	yes	mammals
python	no	yes	no	no	no	reptiles
salmon	no	yes	no	yes	no	fishes
whale	yes	no	no	yes	no	mammals
frog	no	yes	no	sometimes	yes	amphibians
komodo	no	yes	no	no	yes	reptiles
bat	yes	no	yes	no	yes	mammals
pigeon	no	yes	yes	no	yes	birds
cat	yes	no	no	no	yes	mammals
leopard shark	yes	no	no	yes	no	fishes
turtle	no	yes	no	sometimes	yes	reptiles
penguin	no	yes	no	sometimes	yes	birds
porcupine	yes	no	no	no	yes	mammals
eel	no	yes	no	yes	no	fishes
salamander	no	yes	no	sometimes	yes	amphibians
gila monster	no	yes	no	no	yes	reptiles
platypus	no	yes	no	no	yes	mammals
owl	no	yes	yes	no	yes	birds
dolphin	yes	no	no	yes	no	mammals
eagle	no	yes	yes	no	yes	birds

C4.5 versus C4.5rules versus RIPPER



C4.5rules:
(Give Birth=No, Can Fly=Yes) → Birds
(Give Birth=No, Live in Water=Yes) → Fishes
(Give Birth=Yes) → Mammals
(Give Birth=No, Can Fly=No, Live in Water=No) → Reptiles
( ) → Amphibians

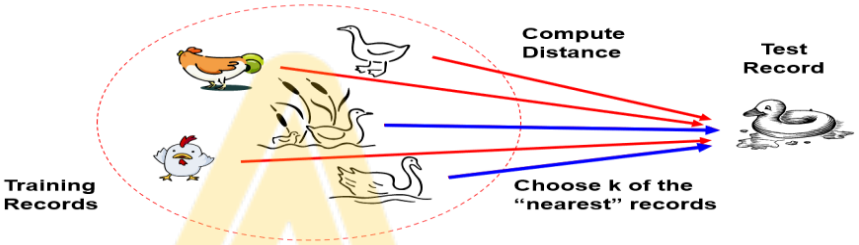
RIPPER:
(Live in Water=Yes) → Fishes
(Have Legs=No) → Reptiles
(Give Birth=No, Can Fly=No, Live In Water=No) → Reptiles
(Can Fly=Yes,Give Birth=No) → Birds
() → Mammals

Advantages of Rule-Based Classifiers

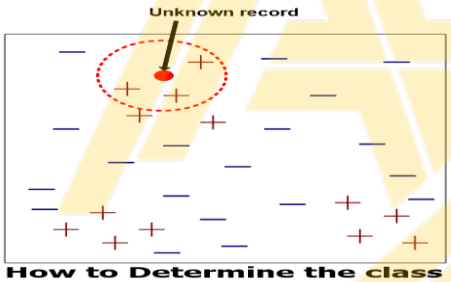
- Has characteristics quite similar to decision trees
  - As highly expressive as decision trees
  - Easy to interpret (if rules are ordered by class)
  - Performance comparable to decision trees
    - Can handle redundant and irrelevant attributes
    - Variable interaction can cause issues (e.g., X-OR problem)
- Better suited for handling imbalanced classes
- Harder to handle missing values in the test set

Nearest Neighbor Classifiers

- Basic idea:
  - If it walks like a duck, quacks like a duck, then it's probably a duck



Nearest-Neighbor Classifiers



- Requires the following:
  - A set of labeled records
  - Proximity metric to compute distance/similarity between a pair of records
    - e.g., Euclidean distance
  - The value of *k*, the number of nearest neighbors to retrieve
  - A method for using class labels of K nearest neighbors to determine the class label of unknown record (e.g., by taking majority vote)

How to Determine the class label of a Test Sample?

- Take the majority vote of class labels among the k-nearest neighbors
- Weight the vote according to distance
  - weight factor, $w = 1/d^2$

Choice of proximity measure matters

- For documents, cosine is better than correlation or Euclidean

1 1 1 1 1 1 1 1 1 1 1 0		0 0 0 0 0 0 0 0 0 0 0 1
0 1 1 1 1 1 1 1 1 1 1 1	vs	1 0 0 0 0 0 0 0 0 0 0 0

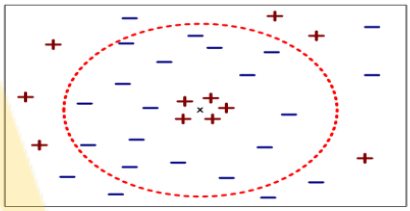
Euclidean distance = 1.4142 for both pairs, but the cosine similarity measure has different values for these pairs.

Nearest Neighbor Classification...

- **Data preprocessing is often required**
  - Attributes may have to be scaled to prevent distance measures from being dominated by one of the attributes
    - Example:
      - height of a person may vary from 1.5m to 1.8m
      - weight of a person may vary from 90lb to 300lb
      - income of a person may vary from $10K to $1M
  - Time series are often standardized to have 0 means a standard deviation of 1

Nearest Neighbor Classification...

- Choosing the value of k:
  - If k is too small, sensitive to noise points
  - If k is too large, neighborhood may include points from other classes



Bayes Classifier

- A probabilistic framework for solving classification problems
- Conditional Probability:

$$P(Y \mid X) = \frac{P(X, Y)}{P(X)}$$

$$P(X \mid Y) = \frac{P(X, Y)}{P(Y)}$$

- Bayes theorem:

$$P(Y \mid X) = \frac{P(X \mid Y)P(Y)}{P(X)}$$

Using Bayes Theorem for Classification

- Consider each attribute and class label as random variables
- Given a record with attributes (X_1, X_2,…, X_d), the goal is to predict class Y
  - Specifically, we want to find the value of Y that maximizes P(Y| X_1, X_2,…, X_d)
- Can we estimate P(Y| X_1, X_2,…, X_d) directly from data?

Tid	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Using Bayes Theorem for Classification

- Approach:
  - compute posterior probability P(Y | X_1, X_2, …, X_d) using the Bayes theorem

$$P(Y \mid X_1 X_2 \ldots X_n) = \frac{P(X_1 X_2 \ldots X_d \mid Y)P(Y)}{P(X_1 X_2 \ldots X_d)}$$

  - *Maximum a-posteriori*: Choose Y that maximizes P(Y | X_1, X_2, …, X_d)
  - Equivalent to choosing value of Y that maximizes P(X_1, X_2, …, X_d|Y) P(Y)
- How to estimate P(X_1, X_2, …, X_d | Y)?

Example Data

Given a Test Record:

$$X = (\text{Refund} = \text{No}, \text{Divorced}, \text{Income} = 120\text{K})$$

Tid	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

- We need to estimate
  P(Evade = Yes | X) and P(Evade = No | X)

  In the following we will replace
  Evade = Yes by Yes, and
  Evade = No by No

Given a Test Record:

$$X = (\text{Refund} = \text{No}, \text{Divorced}, \text{Income} = 120\text{K})$$

Tid	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Using Bayes Theorem:

- $P(\text{Yes} \mid X) = \dfrac{P(X \mid \text{Yes})P(\text{Yes})}{P(X)}$
- $P(\text{No} \mid X) = \dfrac{P(X \mid \text{No})P(\text{No})}{P(X)}$
- How to estimate P(X | Yes) and P(X | No)?

Conditional Independence

- **X** and **Y** are conditionally independent given **Z** if **P(X|YZ) = P(X|Z)**
- Example: Arm length and reading skills
  - Young child has shorter arm length and limited reading skills, compared to adults
  - If age is fixed, no apparent relationship between arm length and reading skills
  - Arm length and reading skills are conditionally independent given age

Naïve Bayes Classifier

- Assume independence among attributes X_i when class is given:
  - $P(X_1, X_2, \ldots, X_d \mid Y_j) = P(X_1 \mid Y_j)\, P(X_2 \mid Y_j) \ldots P(X_d \mid Y_j)$
  - Now we can estimate $P(X_i \mid Y_j)$ for all X_i and Y_j combinations from the training data
  - New point is classified to Y_j if $P(Y_j) \prod P(X_i \mid Y_j)$ is maximal.

Naïve Bayes on Example Data

Given a Test Record:

$$X = (\text{Refund} = \text{No}, \text{Divorced}, \text{Income} = 120\text{K})$$

Tid	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

P(X | Yes) =
P(Refund = No | Yes) x
P(Divorced | Yes) x
P(Income = 120K | Yes)

P(X | No) =
P(Refund = No | No) x
P(Divorced | No) x
P(Income = 120K | No)

Estimate Probabilities from Data

Tid	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

- P(y) = fraction of instances of class y
  - e.g., P(No) = 7/10, P(Yes) = 3/10
- For categorical attributes:
  $P(X_i = c \mid y) = n_c / n$
  - where $|X_i = c|$ is number of instances having attribute value $X_i = c$ and belonging to class y
  - Examples:

    P(Status=Married|No) = 4/7
    P(Refund=Yes|Yes)=0

Estimate Probabilities from Data

- For continuous attributes:
  - Discretization: Partition the range into bins:
    - Replace continuous value with bin value
      - Attribute changed from continuous to ordinal
  - Probability density estimation:
    - Assume attribute follows a normal distribution
    - Use data to estimate parameters of distribution (e.g., mean and standard deviation)
    - Once probability distribution is known, use it to estimate the conditional probability $P(X_i|Y)$

Estimate Probabilities from Data

Tid	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

- Normal distribution:
$$P(X_i \mid Y_j) = \frac{1}{\sqrt{2\pi\sigma_{ij}^2}} e^{\frac{-(X_i-\mu_{ij})^2}{2\sigma_{ij}^2}}$$
  - One for each (X_i, Y_j) pair
- For (Income, Class=No):
  - If Class=No
    - sample mean = 110
    - sample variance = 2975

$$P(Income = 120 \mid No) = \frac{1}{\sqrt{2\pi}(54.54)} e^{\frac{-(120-110)^2}{2(2975)}} = 0.0072$$

Example of Naïve Bayes Classifier

Given a Test Record:

$$X = (\text{Refund} = \text{No}, \text{Divorced}, \text{Income} = 120\text{K})$$

Naïve Bayes Classifier:

P(Refund = Yes | No) = 3/7
P(Refund = No | No) = 4/7
P(Refund = Yes | Yes) = 0
P(Refund = No | Yes) = 1
P(Marital Status = Single | No) = 2/7
P(Marital Status = Divorced | No) = 1/7
P(Marital Status = Married | No) = 4/7
P(Marital Status = Single | Yes) = 2/3
P(Marital Status = Divorced | Yes) = 1/3
P(Marital Status = Married | Yes) = 0

For Taxable Income:
If class = No: sample mean = 110
sample variance = 2975
If class = Yes: sample mean = 90
sample variance = 25

- P(X | No) = P(Refund=No | No)
  × P(Divorced | No)
  × P(Income=120K | No)
  = 4/7 × 1/7 × 0.0072 = 0.0006
- P(X | Yes) = P(Refund=No | Yes)
  × P(Divorced | Yes)
  × P(Income=120K | Yes)
  = 1 × 1/3 × 1.2 × 10^{-9} = 4 × 10^{-10}

Since P(X|No)P(No) > P(X|Yes)P(Yes)
Therefore P(No|X) > P(Yes|X)
=> Class = No

Naïve Bayes Classifier can make decisions with partial information about attributes in the test record

Even in absence of information about any attributes, we can use Apriori Probabilities of Class Variable:

P(Yes) = 3/10
P(No) = 7/10

Naïve Bayes Classifier:

P(Refund = Yes | No) = 3/7
P(Refund = No | No) = 4/7
P(Refund = Yes | Yes) = 0
P(Refund = No | Yes) = 1
P(Marital Status = Single | No) = 2/7
P(Marital Status = Divorced | No) = 1/7
P(Marital Status = Married | No) = 4/7
P(Marital Status = Single | Yes) = 2/3
P(Marital Status = Divorced | Yes) = 1/3
P(Marital Status = Married | Yes) = 0

For Taxable Income:
If class = No: sample mean = 110
sample variance = 2975
If class = Yes: sample mean = 90
sample variance = 25

If we only know that marital status is Divorced, then:

P(Yes | Divorced) = 1/3 x 3/10 / P(Divorced)
P(No | Divorced) = 1/7 x 7/10 / P(Divorced)

If we also know that Refund = No, then

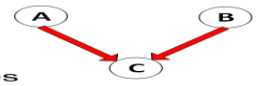
P(Yes | Refund = No, Divorced) = 1 x 1/3 x 3/10 / P(Divorced, Refund = No)
P(No | Refund = No, Divorced) = 4/7 x 1/7 x 7/10 / P(Divorced, Refund = No)

If we also know that Taxable Income = 120, then

P(Yes | Refund = No, Divorced, Income = 120) = 1.2 x10^{-9} x 1 x 1/3 x 3/10 / P(Divorced, Refund = No, Income = 120)
P(No | Refund = No, Divorced Income = 120) = 0.0072 x 4/7 x 1/7 x 7/10 / P(Divorced, Refund = No, Income = 120)

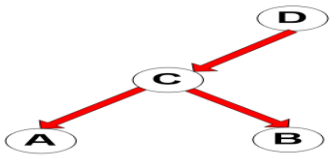
Bayesian Belief Networks

- Provides graphical representation of probabilistic relationships among a set of random variables
- Consists of:
  - A directed acyclic graph (dag)
    - Node corresponds to a variable
    - Arc corresponds to dependence relationship between a pair of variables



  - A probability table associating each node to its immediate parent

Conditional Independence

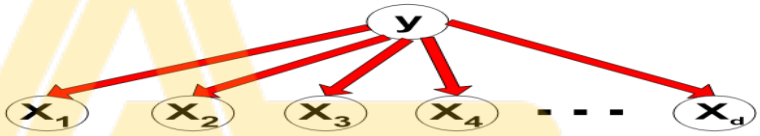


D is parent of C
A is child of C
B is descendant of D
D is ancestor of A

- A node in a Bayesian network is conditionally independent of all of its nondescendants, if its parents are known

Conditional Independence

- Naïve Bayes assumption:



Probability Tables

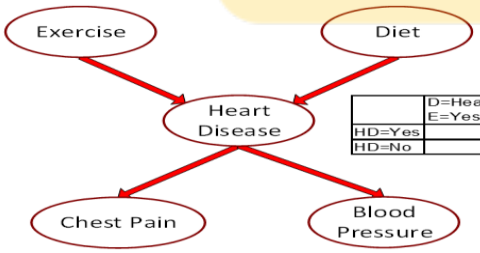
- If X does not have any parents, table contains prior probability P(X)
- If X has only one parent (Y), table contains conditional probability P(X|Y)

- If X has multiple parents ($Y_1, Y_2, \ldots, Y_k$), table contains conditional probability $P(X|Y_1, Y_2, \ldots, Y_k)$

Example of Bayesian Belief Network

Exercise=Yes	0.7
Exercise=No	0.3

Diet=Healthy	0.25
Diet=Unhealthy	0.75



	D=Healthy E=Yes	D=Healthy E=No	D=Unhealthy E=Yes	D=Unhealthy E=No
HD=Yes	0.25	0.45	0.55	0.75
HD=No	0.75	0.55	0.45	0.25

	HD=Yes	HD=No
CP=Yes	0.8	0.01
CP=No	0.2	0.99

	HD=Yes	HD=No
BP=High	0.85	0.2
BP=Low	0.15	0.8

Example of Inferencing using BBN

- Given: X = (E=No, D=Yes, CP=Yes, BP=High)
  - Compute P(HD|E,D,CP,BP)?
- P(HD=Yes| E=No,D=Yes) = 0.55
  P(CP=Yes| HD=Yes) = 0.8
  P(BP=High| HD=Yes) = 0.85
  - P(HD=Yes|E=No,D=Yes,CP=Yes,BP=High) $\propto 0.55 \times 0.8 \times 0.85 = 0.374$
- P(HD=No| E=No,D=Yes) = 0.45
  P(CP=Yes| HD=No) = 0.01
  P(BP=High| HD=No) = 0.2
  - P(HD=No|E=No,D=Yes,CP=Yes,BP=High) $\propto 0.45 \times 0.01 \times 0.2 = 0.0009$

Classify X as Yes

Reference: Introduction to Data Mining, 2nd Edition By Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, Vipin Kumar, ISBN-13: 9780133128901