


4.1 Types of Classifiers

Before presenting specific techniques, we first categorize the different types of classifiers available. One way to distinguish classifiers is by considering the characteristics of their output.

Binary versus Multiclass

Binary classifiers assign each data instance to one of two possible labels, typically denoted as +1 and -1. The positive class usually refers to the category we are more interested in predicting correctly compared to the negative class (e.g., the `spam` category in email classification problems). If there are more than two possible labels available, then the technique is known as a multiclass classifier. As some classifiers were designed for binary classes only, they must be adapted to deal with multiclass problems. Techniques for transforming binary classifiers into multiclass classifiers are described in [Section 4.12](#) .

Deterministic versus Probabilistic

A deterministic classifier produces a discrete-valued label to each data instance it classifies whereas a probabilistic classifier assigns a continuous score between 0 and 1 to indicate how likely it is that an instance belongs to a particular class, where the probability scores for all the classes sum up to 1. Some examples of probabilistic classifiers include the naïve Bayes classifier, Bayesian networks, and logistic regression. Probabilistic classifiers provide additional information about the confidence in assigning an instance to a class than deterministic classifiers. A data instance is typically assigned to the class

with the highest probability score, except when the cost of misclassifying the class with lower probability is significantly higher. We will discuss the topic of cost-sensitive classification with probabilistic outputs in [Section 4.11.2](#).

Another way to distinguish the different types of classifiers is based on their technique for discriminating instances from different classes.

Linear versus Nonlinear

A linear classifier uses a linear separating hyperplane to discriminate instances from different classes whereas a nonlinear classifier enables the construction of more complex, nonlinear decision surfaces. We illustrate an example of a linear classifier (perceptron) and its nonlinear counterpart (multi-layer neural network) in [Section 4.7](#). Although the linearity assumption makes the model less flexible in terms of fitting complex data, linear classifiers are thus less susceptible to model overfitting compared to nonlinear classifiers. Furthermore, one can transform the original set of attributes, $x = (x_1, x_2, \dots, x_d)$, into a more complex feature set, e.g., $\Phi(x) = (x_1, x_2, x_1x_2, x_{12}, x_{22}, \dots)$, before applying the linear classifier. Such feature transformation allows the linear classifier to fit data sets with nonlinear decision surfaces (see [Section 4.9.4](#)).

Global versus Local

A global classifier fits a single model to the entire data set. Unless the model is highly nonlinear, this one-size-fits-all strategy may not be effective when the relationship between the attributes and the class labels varies over the input space. In contrast, a local classifier partitions the input space into smaller regions and fits a distinct model to training instances in each region. The k -nearest neighbor classifier (see [Section 4.3](#)) is a classic example of local classifiers. While local classifiers are more flexible in terms of fitting complex

decision boundaries, they are also more susceptible to the model overfitting problem, especially when the local regions contain few training examples.

Generative versus Discriminative

Given a data instance x , the primary objective of any classifier is to predict the class label, y , of the data instance. However, apart from predicting the class label, we may also be interested in describing the underlying mechanism that *generates* the instances belonging to every class label. For example, in the process of classifying spam email messages, it may be useful to understand the typical characteristics of email messages that are labeled as spam, e.g., specific usage of keywords in the subject or the body of the email. Classifiers that learn a generative model of every class in the process of predicting class labels are known as generative classifiers. Some examples of generative classifiers include the naïve Bayes classifier and Bayesian networks. In contrast, discriminative classifiers directly predict the class labels without explicitly describing the distribution of every class label. They solve a simpler problem than generative models since they do not have the onus of deriving insights about the generative mechanism of data instances. They are thus sometimes preferred over generative models, especially when it is not crucial to obtain information about the properties of every class. Some examples of discriminative classifiers include decision trees, rule-based classifier, nearest neighbor classifier, artificial neural networks, and support vector machines.

4.2 Rule-Based Classifier

A rule-based classifier uses a collection of “if ...then...” rules (also known as a **rule set**) to classify data instances. [Table 4.1](#) shows an example of a rule set generated for the vertebrate classification problem described in the previous chapter. Each classification rule in the rule set can be expressed in the following way:

$$r_i: (\text{Condition}_i) \rightarrow y_i. \quad (4.1)$$

The left-hand side of the rule is called the **rule antecedent** or **precondition**. It contains a conjunction of attribute test conditions:

$$\text{Condition}_i = (A_1 \text{ op } v_1) \wedge (A_2 \text{ op } v_2) \dots (A_k \text{ op } v_k), \quad (4.2)$$

where (A_j, v_j) is an attribute-value pair and op is a comparison operator chosen from the set $\{=, \neq, <, >, \leq, \geq\}$. Each attribute test $(A_j \text{ op } v_j)$ is also known as a conjunct. The right-hand side of the rule is called the **rule consequent**, which contains the predicted class y_i .

A rule r covers a data instance x if the precondition of r matches the attributes of x . r is also said to be fired or triggered whenever it covers a given instance. For an illustration, consider the rule r_1 given in [Table 4.1](#) and the following attributes for two vertebrates: hawk and grizzly bear.

Table 4.1. Example of a rule set for the vertebrate classification problem.

$r_1: (\text{Gives Birth=no}) \wedge (\text{Aerial Creature=yes}) \rightarrow \text{Birds}$ $r_2: (\text{Gives Birth=no}) \wedge (\text{Aquatic Creature=yes}) \rightarrow \text{Fishes}$ $r_3: (\text{Gives Birth=yes}) \wedge (\text{Body Temperature=warm-blooded}) \rightarrow \text{Mammals}$ $r_4: (\text{Gives Birth=no}) \wedge (\text{Aerial Creature=no}) \rightarrow \text{Reptiles}$ $r_5: (\text{Aquatic Creature=semi}) \rightarrow \text{Amphibians}$
--

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hibernates
hawk	warm-blooded	feather	no	no	yes	yes	no
grizzly bear	warm-blooded	fur	yes	no	no	yes	yes

r1 covers the first vertebrate because its precondition is satisfied by the hawk's attributes. The rule does not cover the second vertebrate because grizzly bears give birth to their young and cannot fly, thus violating the precondition of r1.

The quality of a classification rule can be evaluated using measures such as coverage and accuracy. Given a data set D and a classification rule $r : A \rightarrow y$, the coverage of the rule is the fraction of instances in D that trigger the rule r . On the other hand, its accuracy or confidence factor is the fraction of instances triggered by r whose class labels are equal to y . The formal definitions of these measures are

$$\text{Coverage}(r) = \frac{|A|}{|D|} \quad \text{Coverage}(r) = \frac{|A \cap y|}{|A|}, \quad (4.3)$$

where $|A|$ is the number of instances that satisfy the rule antecedent, $|A \cap y|$ is the number of instances that satisfy both the antecedent and consequent, and $|D|$ is the total number of instances.

Example 4.1.

Consider the data set shown in [Table 4.2](#). The rule

$(\text{Gives Birth}=\text{yes}) \wedge (\text{Body Temperature}=\text{warm-blooded}) \rightarrow \text{Mammals}$

eel	cold-blooded	scales	no	yes	no	no	no	Fishes
salamander	cold-blooded	none	no	semi	no	yes	yes	Amphibians

4.2.1 How a Rule-Based Classifier Works

A rule-based classifier classifies a test instance based on the rule triggered by the instance. To illustrate how a rule-based classifier works, consider the rule set shown in [Table 4.1](#) and the following vertebrates:

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hibernates
lemur	warm-blooded	fur	yes	no	no	yes	yes
turtle	cold-blooded	scales	no	semi	no	yes	no
dogfish shark	cold-blooded	scales	yes	yes	no	no	no

- The first vertebrate, which is a lemur, is warm-blooded and gives birth to its young. It triggers the rule r3, and thus, is classified as a mammal.
- The second vertebrate, which is a turtle, triggers the rules r4 and r5. Since the classes predicted by the rules are contradictory (reptiles versus amphibians), their conflicting classes must be resolved.
- None of the rules are applicable to a dogfish shark. In this case, we need to determine what class to assign to such a test instance.

4.2.2 Properties of a Rule Set

The rule set generated by a rule-based classifier can be characterized by the following two properties.

Definition 4.1 (Mutually Exclusive Rule Set).

The rules in a rule set R are mutually exclusive if no two rules in R are triggered by the same instance. This property ensures that every instance is covered by at most one rule in R .

Definition 4.2 (Exhaustive Rule Set).

A rule set R has exhaustive coverage if there is a rule for each combination of attribute values. This property ensures that every instance is covered by at least one rule in R .

Table 4.3. Example of a mutually exclusive and exhaustive rule set.

$r_1: (\text{Body Temperature}=\text{cold-blooded}) \rightarrow \text{Non-mammals}$
 $r_2: (\text{Body Temperature}=\text{warm-blooded}) \wedge (\text{Gives Birth}=\text{yes}) \rightarrow \text{Mammals}$
 $r_3: (\text{Body Temperature}=\text{warm-$

blooded) \wedge (Gives Birth=no) \rightarrow Non-mammals

Together, these two properties ensure that every instance is covered by exactly one rule. An example of a mutually exclusive and exhaustive rule set is shown in [Table 4.3](#). Unfortunately, many rule-based classifiers, including the one shown in [Table 4.1](#), do not have such properties. If the rule set is not exhaustive, then a default rule, $rd: () \rightarrow y_d$, must be added to cover the remaining cases. A default rule has an empty antecedent and is triggered when all other rules have failed. y_d is known as the default class and is typically assigned to the majority class of training instances not covered by the existing rules. If the rule set is not mutually exclusive, then an instance can be covered by more than one rule, some of which may predict conflicting classes.

Definition 4.3 (Ordered Rule Set).

The rules in an ordered rule set R are ranked in decreasing order of their priority. An ordered rule set is also known as a **decision list**.

The rank of a rule can be defined in many ways, e.g., based on its accuracy or total description length. When a test instance is presented, it will be classified by the highest-ranked rule that covers the instance. This avoids the problem of having conflicting classes predicted by multiple classification rules if the rule set is not mutually exclusive.

An alternative way to handle a non-mutually exclusive rule set without ordering the rules is to consider the consequent of each rule triggered by a test instance as a vote for a particular class. The votes are then tallied to determine the class label of the test instance. The instance is usually assigned to the class that receives the highest number of votes. The vote may also be weighted by the rule's accuracy. Using unordered rules to build a rule-based classifier has both advantages and disadvantages. Unordered rules are less susceptible to errors caused by the wrong rule being selected to classify a test instance unlike classifiers based on ordered rules, which are sensitive to the choice of rule-ordering criteria. Model building is also less expensive because the rules do not need to be kept in sorted order. Nevertheless, classifying a test instance can be quite expensive because the attributes of the test instance must be compared against the precondition of every rule in the rule set.

In the next two sections, we present techniques for extracting an ordered rule set from data. A rule-based classifier can be constructed using (1) direct methods, which extract classification rules directly from data, and (2) indirect methods, which extract classification rules from more complex classification models, such as decision trees and neural networks. Detailed discussions of these methods are presented in [Sections 4.2.3](#) and [4.2.4](#), respectively.

4.2.3 Direct Methods for Rule Extraction

To illustrate the direct method, we consider a widely-used rule induction algorithm called RIPPER. This algorithm scales almost linearly with the number of training instances and is particularly suited for building models from

data sets with imbalanced class distributions. RIPPER also works well with noisy data because it uses a validation set to prevent model overfitting.

RIPPER uses the **sequential covering** algorithm to extract rules directly from data. Rules are grown in a greedy fashion one class at a time. For binary class problems, RIPPER chooses the majority class as its default class and learns the rules to detect instances from the minority class. For multiclass problems, the classes are ordered according to their prevalence in the training set. Let (y_1, y_2, \dots, y_c) be the ordered list of classes, where y_1 is the least prevalent class and y_c is the most prevalent class. All training instances that belong to y_1 are initially labeled as positive examples, while those that belong to other classes are labeled as negative examples. The sequential covering algorithm learns a set of rules to discriminate the positive from negative examples. Next, all training instances from y_2 are labeled as positive, while those from classes y_3, y_4, \dots, y_c are labeled as negative. The sequential covering algorithm would learn the next set of rules to distinguish y_2 from other remaining classes. This process is repeated until we are left with only one class, y_c , which is designated as the default class.

Example 4.1. Sequential covering algorithm.

```
1: Let  $E$  be the training instances and  $A$  be the set of attribute-  
value pairs,  $\{(A_j, v_j)\}$ .  
2: Let  $Y_o$  be an ordered set of classes  $\{y_1, y_2, \dots, y_k\}$ .  
3: Let  $R = \{ \}$  be the initial rule list.  
4: for each class  $y \in Y_o - \{y_k\}$  do  
5:   while stopping condition is not met do  
6:      $r \leftarrow \text{Learn-One-Rule}(E, A, y)$ .  
7:     Remove training instances from  $E$  that are covered by  $r$ .  
8:     Add  $r$  to the bottom of the rule list:  $R \leftarrow R \vee r$ .
```

```
9: end while
10: end for
11: Insert the default rule,  $\{\} \rightarrow y_k$ , to the bottom of the rule list
R.
```

A summary of the sequential covering algorithm is shown in [Algorithm 4.1](#). The algorithm starts with an empty decision list, R , and extracts rules for each class based on the ordering specified by the class prevalence. It iteratively extracts the rules for a given class y using the Learn-One-Rule function. Once such a rule is found, all the training instances covered by the rule are eliminated. The new rule is added to the bottom of the decision list R . This procedure is repeated until the stopping criterion is met. The algorithm then proceeds to generate rules for the next class.

[Figure 4.1](#) demonstrates how the sequential covering algorithm works for a data set that contains a collection of positive and negative examples. The rule R_1 , whose coverage is shown in [Figure 4.1\(b\)](#), is extracted first because it covers the largest fraction of positive examples. All the training instances covered by R_1 are subsequently removed and the algorithm proceeds to look for the next best rule, which is R_2 .

Learn-One-Rule Function

Finding an optimal rule is computationally expensive due to the exponential search space to explore. The Learn-One-Rule function addresses this problem by growing the rules in a greedy fashion. It generates an initial rule $r: \{\} \rightarrow +$, where the left-hand side is an empty set and the right-hand side corresponds to the positive class. It then refines the rule until a certain stopping criterion is met. The accuracy of the initial rule may be poor because some of the training instances covered by the rule belong to the negative

class. A new conjunct must be added to the rule antecedent to improve its accuracy.

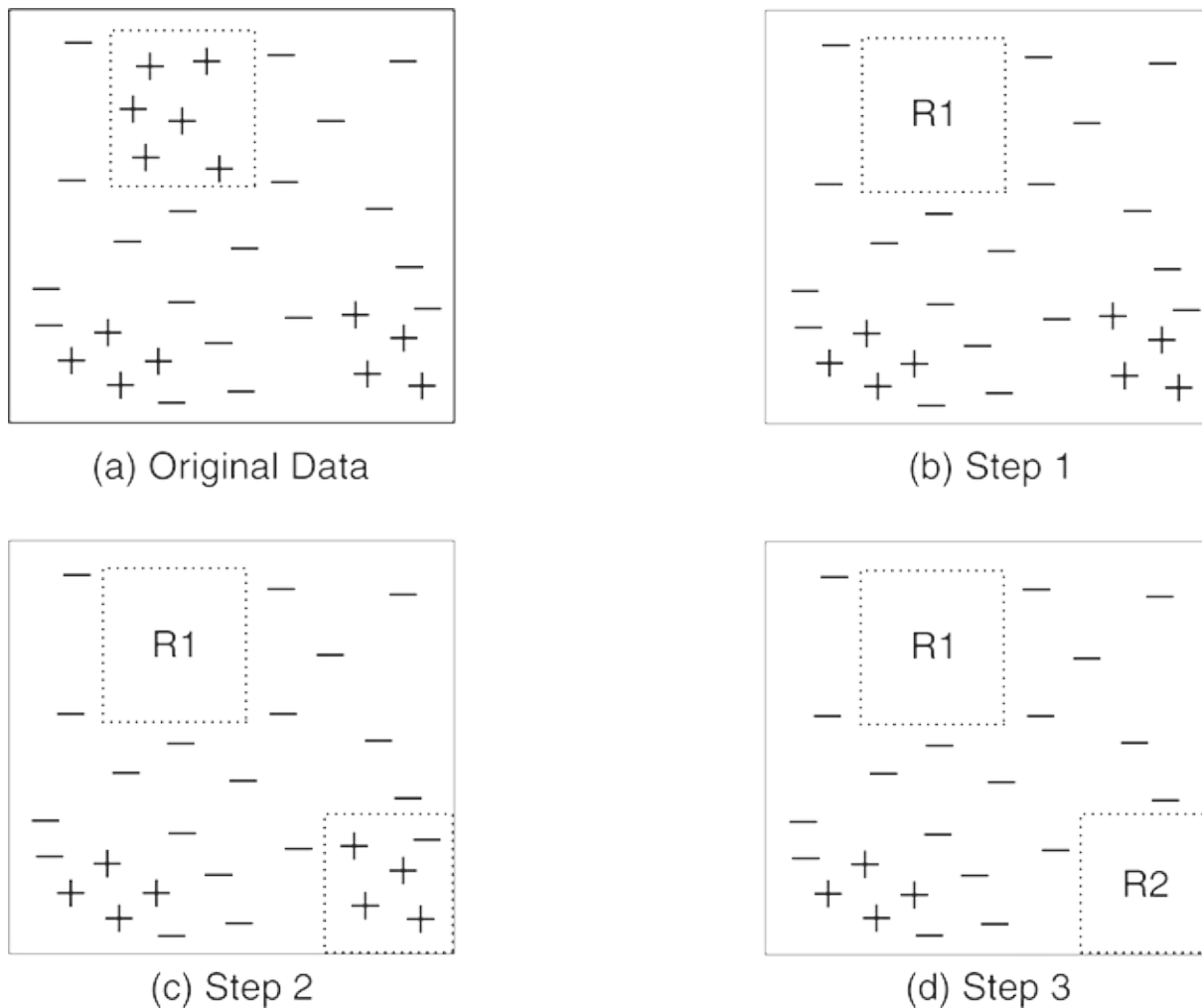


Figure 4.1.

An example of the sequential covering algorithm.

RIPPER uses the FOIL's information gain measure to choose the best conjunct to be added into the rule antecedent. The measure takes into consideration both the gain in accuracy and support of a candidate rule, where support is defined as the number of positive examples covered by the rule. For example, suppose the rule $r: A \rightarrow +$ initially covers p_0 positive examples and n_0 negative examples. After adding a new conjunct B , the extended rule $r': A \wedge B \rightarrow +$ covers p_1 positive examples and n_1 negative

examples. The FOIL's information gain of the extended rule is computed as follows:

$$\text{FOIL's information gain} = p_1 \times (\log_2 p_1 p_1 + n_1 - \log_2 p_0 p_0 + n_0). \quad (4.4)$$

RIPPER chooses the conjunct with highest FOIL's information gain to extend the rule, as illustrated in the next example.

Example 4.2. [Foil's Information Gain]

Consider the training set for the vertebrate classification problem shown in [Table 4.2](#). Suppose the target class for the Learn-One-Rule function is mammals. Initially, the antecedent of the rule $\{\} \rightarrow \text{Mammals}$ covers 5 positive and 10 negative examples. Thus, the accuracy of the rule is only 0.333. Next, consider the following three candidate conjuncts to be added to the left-hand side of the rule: Skin cover=hair, Body temperature=warm, and Has legs=No. The number of positive and negative examples covered by the rule after adding each conjunct along with their respective accuracy and FOIL's information gain are shown in the following table.

Candidate rule	p1	n1	Accuracy	Info Gain
{Skin Cover=hair} → mammals	3	0	1.000	4.755
{Body temperature=wam} → mammals	5	1	0.714	5.498
{Has legs=No} → mammals	2	4	0.200	-0.737

Although Skin cover=hair has the highest accuracy among the three candidates, the conjunct Body temperature=warm has the highest FOIL's information gain. Thus, it is chosen to extend the rule (see [Figure 4.2](#)).

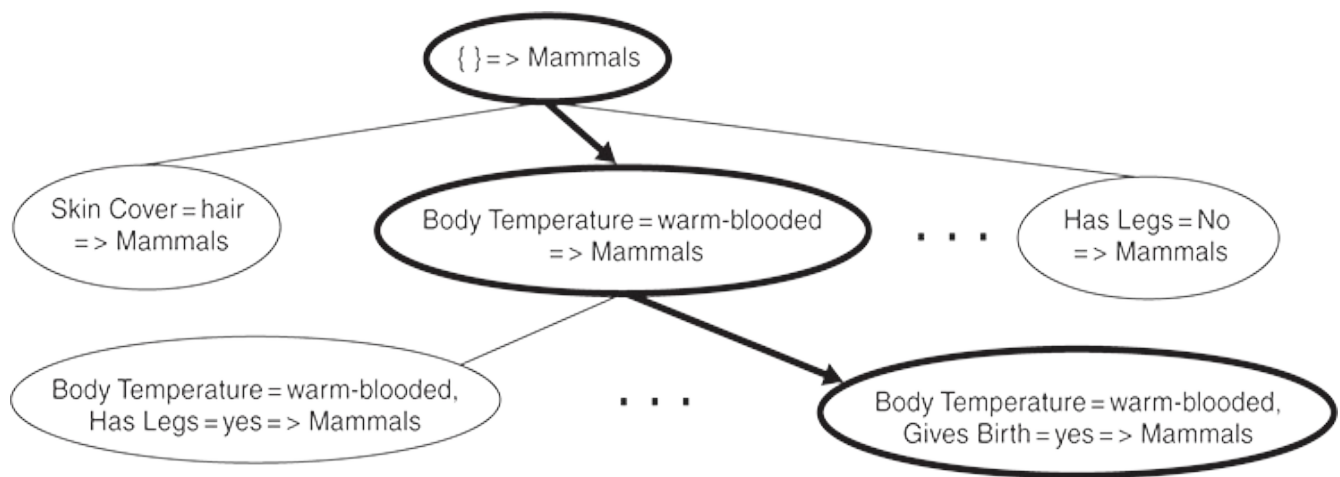
This process continues until adding new conjuncts no longer improves the information gain measure.

Rule Pruning

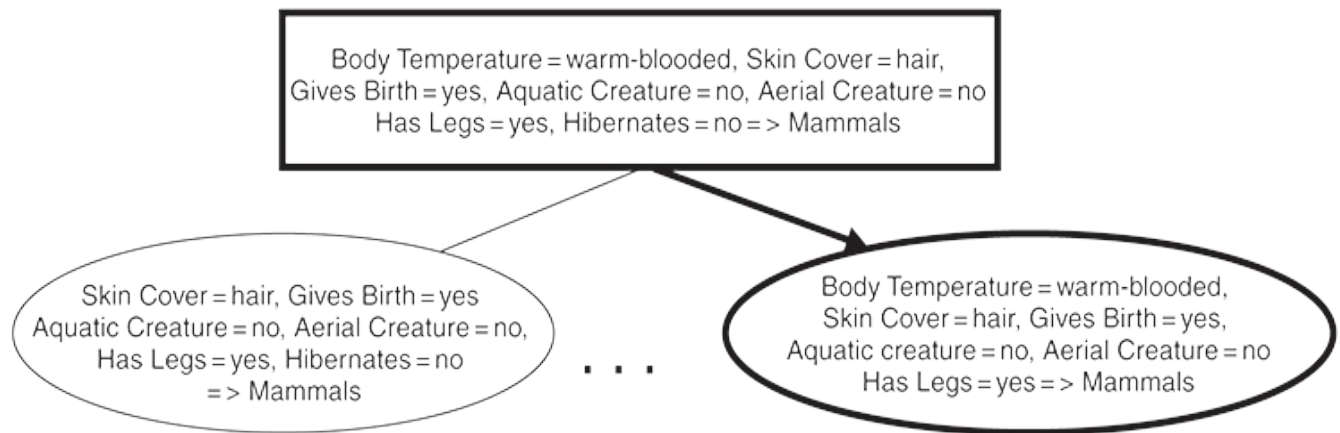
The rules generated by the Learn-One-Rule function can be pruned to improve their generalization errors. RIPPER prunes the rules based on their performance on the validation set. The following metric is computed to determine whether pruning is needed: $(p-n)/(p+n)$, where $p(n)$ is the number of positive (negative) examples in the validation set covered by the rule. This metric is monotonically related to the rule's accuracy on the validation set. If the metric improves after pruning, then the conjunct is removed. Pruning is done starting from the last conjunct added to the rule. For example, given a rule $ABCD \rightarrow y$, RIPPER checks whether D should be pruned first, followed by CD , BCD , etc. While the original rule covers only positive examples, the pruned rule may cover some of the negative examples in the training set.

Building the Rule Set

After generating a rule, all the positive and negative examples covered by the rule are eliminated. The rule is then added into the rule set as long as it does not violate the stopping condition, which is based on the minimum description length principle. If the new rule increases the total description length of the rule set by at least d bits, then RIPPER stops adding rules into its rule set (by default, d is chosen to be 64 bits). Another stopping condition used by RIPPER is that the error rate of the rule on the validation set must not exceed 50%.



(a) General-to-specific



(b) Specific-to-general


Figure 4.2.

General-to-specific and specific-to-general rule-growing strategies.

RIPPER also performs additional optimization steps to determine whether some of the existing rules in the rule set can be replaced by better alternative rules. Readers who are interested in the details of the optimization method may refer to the reference cited at the end of this chapter.

Instance Elimination

After a rule is extracted, RIPPER eliminates the positive and negative examples covered by the rule. The rationale for doing this is illustrated in the next example.

Figure 4.3  shows three possible rules, R_1 , R_2 , and R_3 , extracted from a training set that contains 29 positive examples and 21 negative examples. The accuracies of R_1 , R_2 , and R_3 are 12/15 (80%), 7/10 (70%), and 8/12 (66.7%), respectively. R_1 is generated first because it has the highest accuracy. After generating R_1 , the algorithm must remove the examples covered by the rule so that the next rule generated by the algorithm is different than R_1 . The question is, should it remove the positive examples only, negative examples only, or both? To answer this, suppose the algorithm must choose between generating R_2 or R_3 after R_1 . Even though R_2 has a higher accuracy than R_3 (70% versus 66.7%), observe that the region covered by R_2 is disjoint from R_1 , while the region covered by R_3 overlaps with R_1 . As a result, R_1 and R_3 together cover 18 positive and 5 negative examples (resulting in an overall accuracy of 78.3%), whereas R_1 and R_2 together cover 19 positive and 6 negative examples (resulting in a lower overall accuracy of 76%). If the positive examples covered by R_1 are not removed, then we may overestimate the effective accuracy of R_3 . If the negative examples covered by R_1 are not removed, then we may underestimate the accuracy of R_3 . In the latter case, we might end up preferring R_2 over R_3 even though half of the false positive errors committed by R_3 have already been accounted for by the preceding rule, R_1 . This example shows that the effective accuracy after adding R_2 or R_3 to the rule set becomes evident only when both positive and negative examples covered by R_1 are removed.

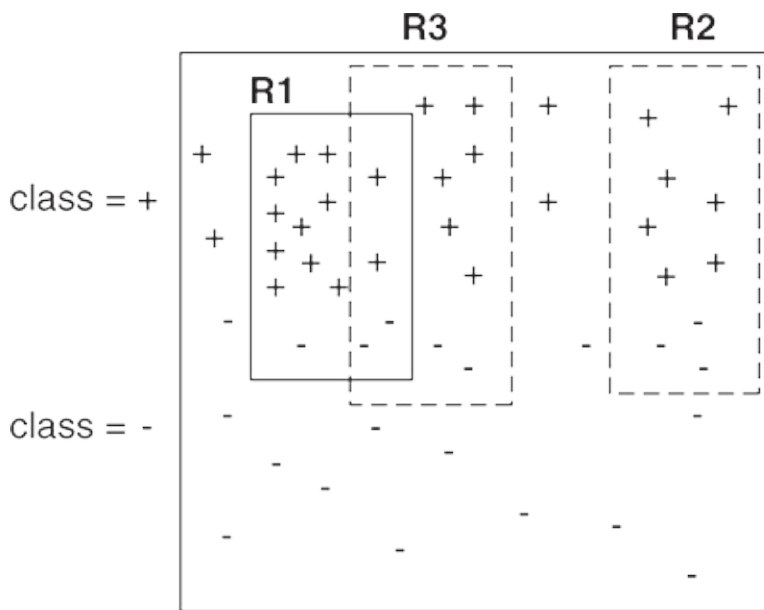



Figure 4.3.

Elimination of training instances by the sequential covering algorithm. *R1*, *R2*, and *R3* represent regions covered by three different rules.

4.2.4 Indirect Methods for Rule Extraction

This section presents a method for generating a rule set from a decision tree. In principle, every path from the root node to the leaf node of a decision tree can be expressed as a classification rule. The test conditions encountered along the path form the conjuncts of the rule antecedent, while the class label at the leaf node is assigned to the rule consequent. [Figure 4.4](#)  shows an example of a rule set generated from a decision tree. Notice that the rule set is exhaustive and contains mutually exclusive rules. However, some of the rules can be simplified as shown in the next example.

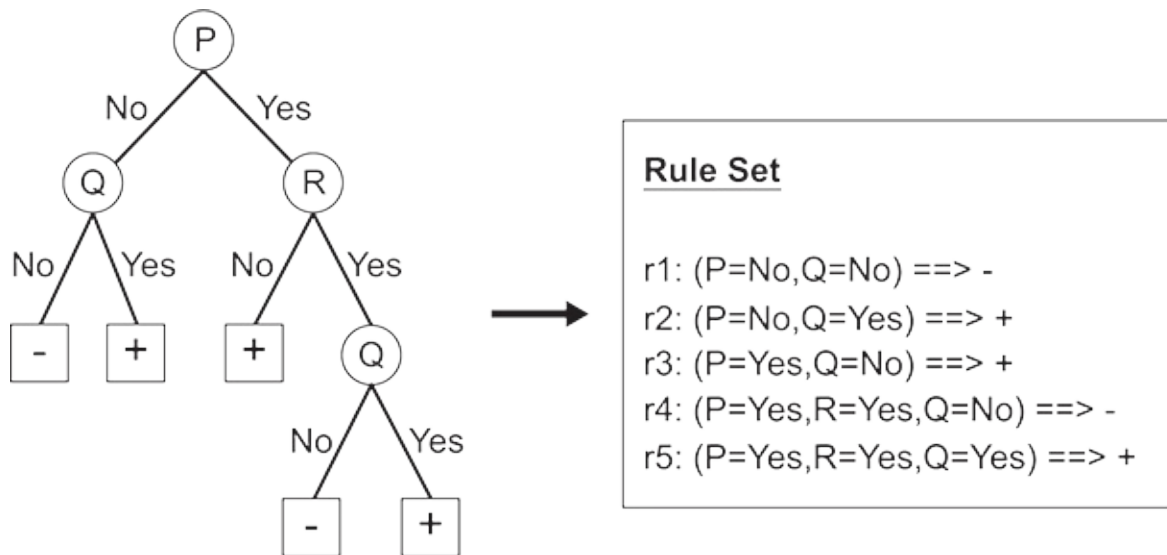


Figure 4.4.
 Converting a decision tree into classification rules.

Example 4.3.

Consider the following three rules from [Figure 4.4](#):

$$r2: (P=No) \wedge (Q=Yes) \rightarrow +$$

$$r3: (P=Yes) \wedge (R=No) \rightarrow +$$

$$r5: (P=Yes) \wedge (R=Yes) \wedge (Q=Yes) \rightarrow +.$$

Observe that the rule set always predicts a positive class when the value of Q is Yes. Therefore, we may simplify the rules as follows:

$$r2': (Q=Yes) \rightarrow +$$

$$r3: (P=Yes) \wedge (R=No) \rightarrow +.$$

r3 is retained to cover the remaining instances of the positive class. Although the rules obtained after simplification are no longer mutually exclusive, they are less complex and are easier to interpret.

In the following, we describe an approach used by the C4.5rules algorithm to generate a rule set from a decision tree. [Figure 4.5](#) shows the decision tree

and resulting classification rules obtained for the data set given in [Table 4.2](#).

Rule Generation

Classification rules are extracted for every path from the root to one of the leaf nodes in the decision tree. Given a classification rule $r:A \rightarrow y$, we consider a simplified rule, $r':A' \rightarrow y$ where A' is obtained by removing one of the conjuncts in A . The simplified rule with the lowest pessimistic error rate is retained provided its error rate is less than that of the original rule. The rule-pruning step is repeated until the pessimistic error of the rule cannot be improved further. Because some of the rules may become identical after pruning, the duplicate rules are discarded.

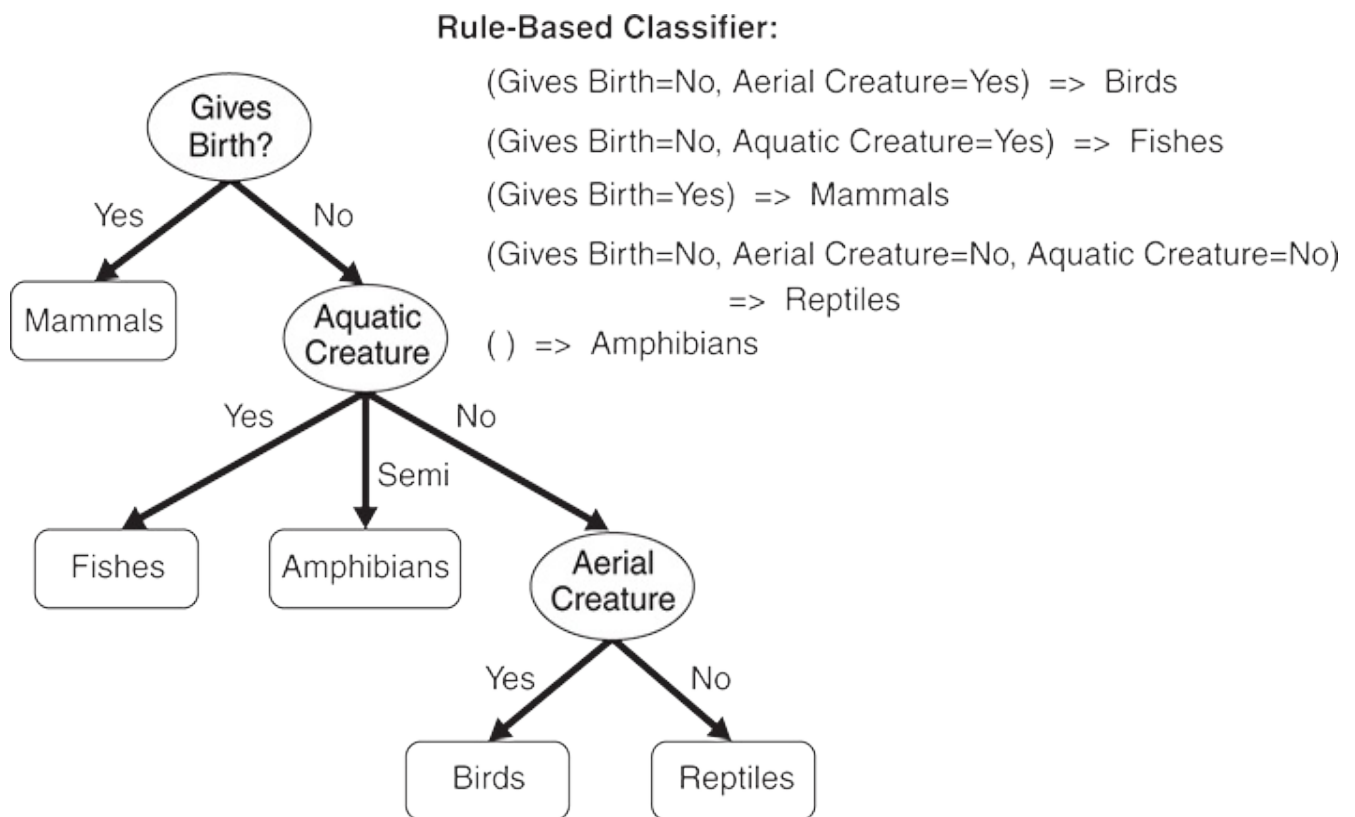


Figure 4.5.

Classification rules extracted from a decision tree for the vertebrate classification problem.

Rule Ordering

After generating the rule set, C4.5rules uses the class-based ordering scheme to order the extracted rules. Rules that predict the same class are grouped together into the same subset. The total description length for each subset is computed, and the classes are arranged in increasing order of their total description length. The class that has the smallest description length is given the highest priority because it is expected to contain the best set of rules. The total description length for a class is given by $L_{\text{exception}} + g \times L_{\text{model}}$, where $L_{\text{exception}}$ is the number of bits needed to encode the misclassified examples, L_{model} is the number of bits needed to encode the model, and g is a tuning parameter whose default value is 0.5. The tuning parameter depends on the number of redundant attributes present in the model. The value of the tuning parameter is small if the model contains many redundant attributes.

4.2.5 Characteristics of Rule-Based Classifiers

1. Rule-based classifiers have very similar characteristics as decision trees. The expressiveness of a rule set is almost equivalent to that of a decision tree because a decision tree can be represented by a set of mutually exclusive and exhaustive rules. Both rule-based and decision tree classifiers create rectilinear partitions of the attribute space and assign a class to each partition. However, a rule-based classifier can

allow multiple rules to be triggered for a given instance, thus enabling the learning of more complex models than decision trees.

2. Like decision trees, rule-based classifiers can handle varying types of categorical and continuous attributes and can easily work in multiclass classification scenarios. Rule-based classifiers are generally used to produce descriptive models that are easier to interpret but give comparable performance to the decision tree classifier.
3. Rule-based classifiers can easily handle the presence of redundant attributes that are highly correlated with one other. This is because once an attribute has been used as a conjunct in a rule antecedent, the remaining redundant attributes would show little to no FOIL's information gain and would thus be ignored.
4. Since irrelevant attributes show poor information gain, rule-based classifiers can avoid selecting irrelevant attributes if there are other relevant attributes that show better information gain. However, if the problem is complex and there are interacting attributes that can collectively distinguish between the classes but individually show poor information gain, it is likely for an irrelevant attribute to be accidentally favored over a relevant attribute just by random chance. Hence, rule-based classifiers can show poor performance in the presence of interacting attributes, when the number of irrelevant attributes is large.
5. The class-based ordering strategy adopted by RIPPER, which emphasizes giving higher priority to rare classes, is well suited for handling training data sets with imbalanced class distributions.
6. Rule-based classifiers are not well-suited for handling missing values in the test set. This is because the position of rules in a rule set follows a certain ordering strategy and even if a test instance is covered by multiple rules, they can assign different class labels depending on their position in the rule set. Hence, if a certain rule involves an attribute that is missing in a test instance, it is difficult to ignore the rule and proceed

to the subsequent rules in the rule set, as such a strategy can result in incorrect class assignments.