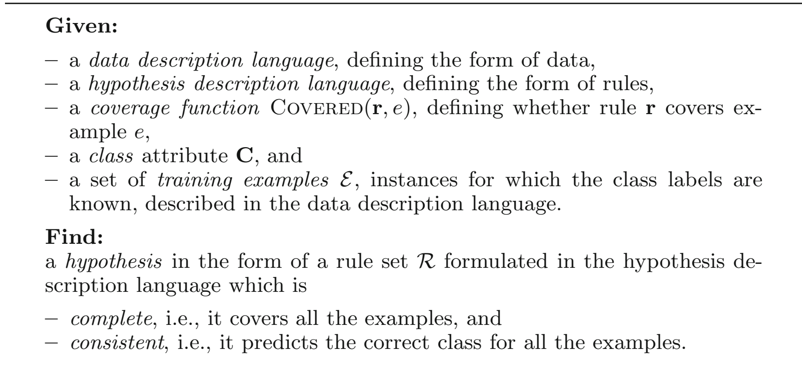
Chapter 2
Rule Learning in a Nutshell

This chapter gives a brief overview of inductive rule learning and may therefore serve as a guide through the rest of the book. Later chapters will expand upon the material presented here and discuss advanced approaches, whereas this chapter only presents the core concepts. The chapter describes search heuristics and rule quality criteria, the basic covering algorithm, illustrates classification rule learning on simple propositional learning problems, shows how to use the learned rules for classifying new instances, and introduces the basic evaluation criteria and methodology for rule-set evaluation.

After defining the learning task in Sect. 2.1, we start with discussing data (Sect. 2.2) and rule representation (Sect. 2.3) for the standard propositional rule learning framework, in which training examples are represented in a single table, and the outputs are if–then rules. Section 2.4 outlines the rule construction process, followed by a more detailed description of its parts: the induction of individual rules is presented as a search problem in Sect. 2.5, and the learning of rule sets in Sect. 2.6. One of the classical rule learning algorithms, CN2, is described in more detail in Sect. 2.7. Section 2.8 shows how to use the induced rule sets for the classification of new instances, and the subsequent Sect. 2.9 discusses evaluation of the classification quality of the induced rule sets and presents cross-validation as a means for evaluating the predictive accuracy of rules. Finally, Sect. 2.10 gives a brief historical account of some influential rule learning systems.

[†]This chapter is partly based on (Flach & Lavrač, 2003).

J. Fürnkranz et al., *Foundations of Rule Learning*, Cognitive Technologies,
DOI 10.1007/978-3-540-75197-7_2, © Springer-Verlag Berlin Heidelberg 2012



Given:

– a *data description language*, defining the form of data,
– a *hypothesis description language*, defining the form of rules,
– a *coverage function* COVERED($\mathbf{r}, e$), defining whether rule $\mathbf{r}$ covers example e,
– a *class* attribute $\mathbf{C}$, and
– a set of *training examples* $\mathcal{E}$, instances for which the class labels are known, described in the data description language.

Find:

a *hypothesis* in the form of a rule set $\mathcal{R}$ formulated in the hypothesis description language which is

– *complete*, i.e., it covers all the examples, and
– *consistent*, i.e., it predicts the correct class for all the examples.

Fig. 2.1 Definition of the classification rule learning task

2.1 Problem Definition

Informally, we can define the problem of learning classification rules as follows:

> Given a set of training examples, find a set of classification rules that can be used for prediction or classification of new instances.

Note that we distinguish between the terms *examples* and *instances*. Both are usually described by *attribute values*. Examples refer to instances labeled by a *class label*, whereas instances themselves bear no class label. An instance is *covered* by a rule if its description satisfies the rule conditions, and it is *not covered* if its description does not satisfy the rule conditions. An example is *correctly covered* by the rule if it is covered and the class of the rule equals the class label of the example, or *incorrectly covered* if its description satisfies the rule conditions, but the class label of the rule is not equal to the class label of the example.

The above informal definition leaves out several details. A more formal definition is shown in Fig. 2.1. It includes important additional preliminaries for the learning task, such as the representation formalism used for describing the data (*data description language*) and for describing the induced set of rules (*hypothesis description language*). We use the term *hypothesis* to denote the output of learning because of the hypothetical nature of induction, which can never guarantee that the output of inductive learning will not be falsified by new evidence presented to the learner. However, we will also often use the terms *model* or *theory* as synonyms for hypothesis. Finally, we also need a *coverage function*, which connects the hypothesis description with the data description. The restrictions imposed by the languages defining the format and scope of data and knowledge representation are also referred to as the *language bias* of the learning problem.

Note that the definition of the classification rule learning task of Fig. 2.1 describes an idealistic scenario with no errors in the data where a complete and consistent

---

Given:

- a *data description language*, imposing a bias on the form of data,
- a *target concept*, typically denoted with $\oplus$,
- a *hypothesis description language*, imposing a bias on the form of rules,
- a *coverage function* COVERED($\mathbf{r}, e$) defining whether rule $\mathbf{r}$ covers example e,
- a set of *positive examples* $\mathcal{P}$, instances for which it is known that they belong to the target concept
- a set of *negative examples* $\mathcal{N}$, instances for which it is known that they do not belong to the target concept

Find:

a *hypothesis* as a set of rules $\mathcal{R}$ described in the hypothesis description language, providing the definition of the target concept which is

- *complete*, i.e., it covers all examples that belong to the concept, and
- *consistent*, i.e., it does not cover any example that does not belong to the concept.

Fig. 2.2 Definition of the concept learning task

hypothesis can be induced. However, in realistic situations, completeness and consistency have to be replaced with less strict criteria for measuring the quality of the induced rule set.

Propositional rules. This chapter focuses on *propositional rule induction* or *attribute-value rule learning*. Representatives of this class of learners are CN2 (Clark & Boswell, 1991; Clark & Niblett, 1989) and RIPPER (Cohen, 1995). An example of rule learning from the statistics literature is PRIM (Friedman & Fisher, 1999). In this language, a classification rule is an expression of the form:

IF *Conditions* THEN *c*

where *c* is the *class label*, and the *Conditions* are a conjunction of simple logical tests describing the properties of instances that have to be satisfied for the rule to 'fire'. Thus, a rule essentially corresponds to an implication $Conditions \rightarrow c$ in propositional logic, which we will typically write in the opposite direction of the implication sign ($c \leftarrow Conditions$).

Concept learning. Most rule learning algorithms assume a *concept learning* task, a special case of the classification learning problem, shown in Fig. 2.2. Here the task is to learn a set of rules that describe a single *target class c* (often denoted as $\oplus$), also called the *target concept*. As training information, we are given a set of *positive* examples, for which we know that they belong to the target concept, and a set of *negative* examples, for which we know that they do *not* belong to the concept. In this case, it is typically sufficient to learn a theory for the target class only. All instances that are not covered by any of the learned rules will be classified as negative. Thus, a *complete* hypothesis is one that covers all positive examples, and

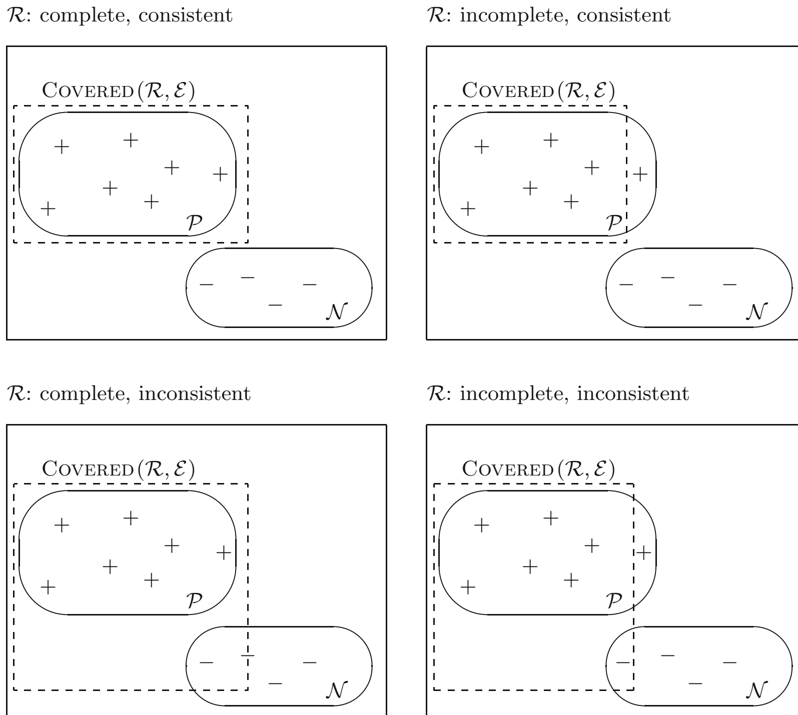


Fig. 2.3 Completeness and consistency of a hypothesis (rule set $\mathcal{R}$)

a *consistent* hypothesis is one that covers no negative examples. Figure 2.3 shows a schematic depiction of (in-)complete and (in-)consistent hypotheses.

Given this concept learning perspective, iterative application of single concept learning tasks allows us to deal with general *multiclass classification* problems. Suppose that training instances are labeled with three class labels: c_1, c_2, and c_3. The above definition of the learning task can be applied if we form three different learning tasks. In the first task, instances labeled with class c_1 are treated as the positive examples, and instances labeled c_2 and c_3 are the negative examples. In the next run, class c_2 will be considered as the positive class, and finally, in the third run, rules for class c_3 will be learned. Due to this simple transformation of a multiclass learning problem into a number of concept learning tasks, concept learning is a central topic of inductive rule learning. This type of transformation of multiclass problems to two-class concept learning problems is also known as *one-against-all* class binarization. Alternative ways for handling multiple classes are discussed in Chap. 10.

Overfitting. Generally speaking, consistency and completeness—as required in the task definition of Fig. 2.1—are very strict conditions. They are unrealistic in learning from large, *noisy* datasets, which contain random errors in the data, either due to

incorrect class labels or errors in instance descriptions. Learning a complete and consistent hypothesis is undesirable in the presence of noise, because the hypothesis will try to explain the errors as well. This is known as *overfitting* the data.

It is also possible that the data description language or the hypothesis description language are not expressive enough to allow a complete and consistent hypothesis, in which case the target class needs to be approximated. Another complication is caused by target classes that are not strictly disjoint. To deal with these cases, the consistency and completeness requirements need to be relaxed and replaced with some other evaluation criteria, such as sufficient *coverage* of positive examples, high *predictive accuracy* of the hypothesis or its *significance* above the requested, predefined threshold. These measures can be used both as heuristics to guide rule construction and as measures to evaluate the quality of induced hypotheses. Some of these measures and related issues will be discussed in more detail in Sect. 2.7 and, subsequently, in Chaps. 7 and 9.

Background knowledge. The above definition of the learning task assumes that the learner has no prior knowledge about the problem and that it learns exclusively from training examples. However, difficult learning problems typically require a substantial body of prior knowledge. We refer to declarative prior knowledge as *background knowledge*. Using background knowledge, the learner may express the induced hypotheses in a more natural and concise manner. In this chapter we mostly disregard background knowledge, except in the process of constructing features (attribute values) used as ingredients in forming rule conditions. However, background knowledge plays a crucial role in relational rule learning, addressed in Chap. 5.

2.2 Data Representation

In classification tasks as defined in Fig. 2.1, the input to a classification rule learner consists of a set of training examples, i.e., instances with known class labels. Typically, these instances are described in a so-called attribute-value representation: An *instance* description has the form $(v_{1,j}, \ldots, v_{n,j})$, where each $v_{i,j}$ is the value of *attribute* $\mathbf{A}_i$, $i \in \{1, \ldots, A\}$. An attribute can either have a finite set of values (*discrete*) or take real numbers as values (*continuous* or *numerical*). An *example* e_j is a vector of attribute values labeled by a class label $e_j = (v_{1,j}, \ldots, v_{n,j}, c_j)$, where each $v_{i,j}$ is a value of attribute $\mathbf{A}_i$, and $c_j \in \{c_1, \ldots, c_C\}$ is one of the C possible values of class attribute $\mathbf{C}$. The class attribute is also often called the *target attribute*. A *dataset* is a set of examples. We will normally organize a dataset in tabular form, with columns for the attributes and rows or *tuples* for the examples.

As an example, consider the dataset in Table 2.1.[1] Like the dataset of Table 1.1, it characterizes a number of individuals by four attributes: EducationMaritalStatus,

[1]The dataset is adapted from the well-known contact lenses dataset (Cendrowska, 1987; Witten & Frank, 2005).

Table 2.1 A sample three-class dataset

No.	Education	Marital status	Sex	Has children	Car
1	Primary	Married	Female	No	Mini
2	Primary	Married	Male	No	Sports
3	Primary	Married	Female	Yes	Mini
4	Primary	Married	Male	Yes	Family
5	Primary	Single	Female	No	Mini
6	Primary	Single	Male	No	Sports
7	Secondary	Married	Female	No	Mini
8	Secondary	Married	Male	No	Sports
9	Secondary	Married	Male	Yes	Family
10	Secondary	Single	Female	No	Mini
11	Secondary	Single	Female	Yes	Mini
12	Secondary	Single	Male	Yes	Mini
13	University	Married	Male	No	Mini
14	University	Married	Female	Yes	Mini
15	University	Single	Female	No	Mini
16	University	Single	Male	No	Sports
17	University	Single	Female	Yes	Mini
18	University	Single	Male	Yes	Mini

Sex, and HasChildren. However, the target value is now not a binary decision (whether a certain issue is approved or not), but a three-valued attribute, which encodes what car the person is driving. For ease of reference, we have numbered the examples from 1 to 18.

The reader may notice that the set of examples is *incomplete* in the sense that not all possible combinations of attribute values are present. This situation is typical for real-world applications where the training set consists only of a small fraction of all possible examples. The task of a rule learner is to learn a rule set that serves a twofold purpose:

1. The learned rule set should help to uncover the hidden relationship between the input attributes and the class value, and
2. it should generalize this relationship to new, previously unseen examples.

Table 2.2 shows the remaining six examples in this domain, for which we do not know their classification during training, indicated by question marks in the last column. However, the class labels can, in principle, be determined, and their values are shown in square brackets. If these classifications are known, such a dataset is also known as a *test set*, if its purpose is to evaluate the predictive quality of the learned theory, or a *validation set*, if its purpose is to provide an internal evaluation that the learning algorithm may use to improve its performance.

In the following, we will use the examples from Table 2.1 as the training set, and the examples of Table 2.2 as the test set of a rule learning algorithm.

Table 2.2 A test set for the database of Table 2.1

No.	Education	Marital status	Sex	Has children	Car
19	Primary	Single	Female	Yes	? [mini]
20	Primary	Single	Male	Yes	? [family]
21	Secondary	Married	Female	Yes	? [mini]
22	Secondary	Single	Male	No	? [sports]
23	University	Married	Male	Yes	? [family]
24	University	Married	Female	No	? [mini]

2.3 Rule Representation

Given a set of preclassified objects (called *examples*), usually described by attribute values, a rule learning system constructs one or more rules of the form:

$$\texttt{IF } \mathbf{f}_1 \texttt{ AND } \mathbf{f}_2 \texttt{ AND} \ldots \texttt{AND } \mathbf{f}_L \texttt{ THEN } \mathit{Class} = c_i$$

The condition part of the rule is a logical conjunction of features (also called *conditions*), where a *feature* $\mathbf{f}_k$ is a test that checks whether the example to classify has the specified property or not. The number L of such features (or conditions) is called the *rule length*.

In the attribute-value framework that we sketched in the previous section, features $\mathbf{f}_k$ typically have the form $\mathbf{A}_i = v_{i,j}$ for discrete attributes, and $\mathbf{A}_i < v$ or $\mathbf{A}_i \geq v$ for continuous attributes (here, v is a threshold value that does not need to correspond to a value of the attribute observed in examples). The conclusion of the rule contains a class value c_i. In essence, this means that for all examples that satisfy the body of the rule, the rule predicts the class value c_i.

The condition part of a rule $\mathbf{r}$ is also known as the *antecedent* or the *body* (B) of the rule, and the conclusion is also known as the *consequent* or the *head* (H) of the rule. The terms 'head' and 'body' have their origins in common notation in clausal logic, where an implication is denoted as B $\rightarrow$ H, or equivalently, H $\leftarrow$ B, of the form

$$c_i \leftarrow \mathbf{f}_1 \wedge \mathbf{f}_2 \wedge \ldots \wedge \mathbf{f}_L$$

We will also frequently use this formal syntax, as well as the equivalent Prolog-like syntax

```
ci :- f1, f2, ..., fL.
```

In logical terminology, the body consists of a conjunction of *literals*, and the head is a single literal. Such rules are also known as *determinate clauses*. General clause may have a disjunction of literals in the head. More on the logical foundations can be found in Chap. 5.

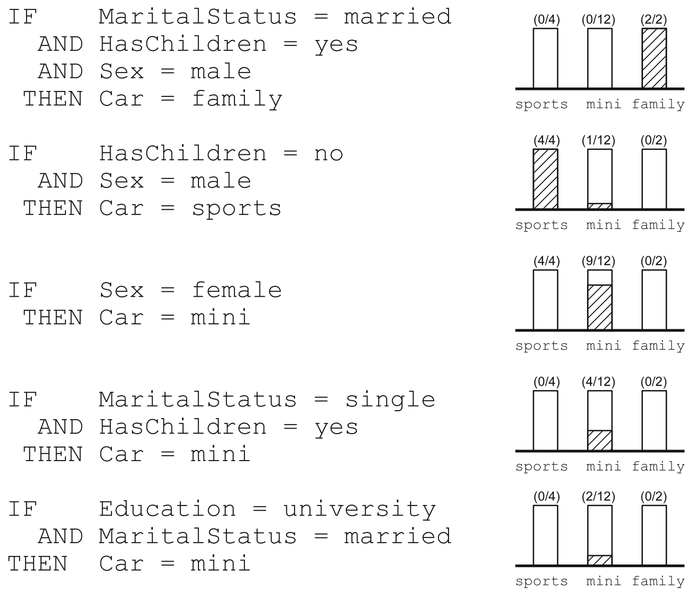
An example set of rules that could have been induced in our sample domain is shown in Fig. 2.4a. The numbers between square brackets indicate the number of covered examples from each class. All the rules, except for the second, cover only examples from a single class, i.e., these rules are *consistent*. On the other hand, the second rule is inconsistent because it misclassifies one training example (#13). Note that the fourth and fifth rule would each misclassify one example from the test set (#20 and #23), but this is not known to the learner. The first rule is *complete* with regard to the class family (covers all the examples of this class), the second is complete with regard to the class sports. Again, this only refers to the training examples that are known to the learner, the first rule would not be complete for class family with respect to the entire domain because it does not cover example #20 of the test set.

Collectively, the rules classify all the training examples, i.e., the learned theory is complete for the given training set (and, in fact, for the entire domain). The theory is not consistent, because it misclassifies one training example. However, we will see later that this is not necessarily bad due to a phenomenon called *overfitting* (cf. Sect. 2.7).

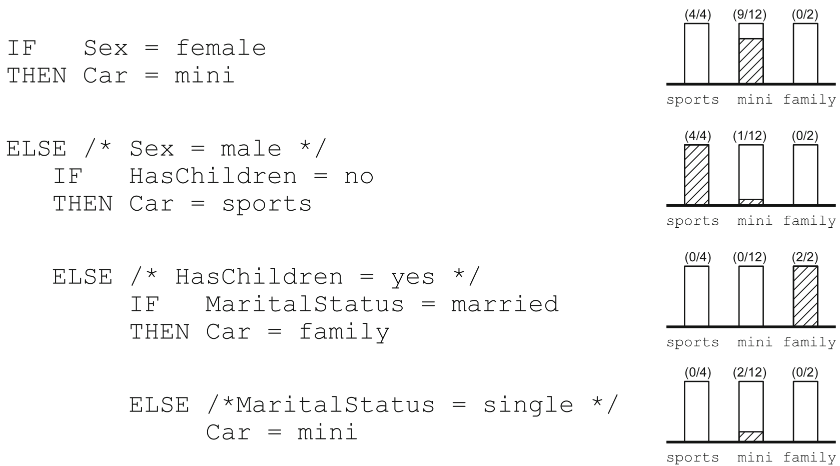
Also note that the counts for the class mini add up to 16 examples, while there are only 12 examples from this class. Thus, some examples must be covered by more than one rule. This is possible, because the rules are *overlapping*. For example, example 13 is covered by the second and by the fifth rule. As both rules make contradicting predictions, there must be a procedure for determining the final prediction (cf. Sect. 2.8).

This is not the case for the *decision list*, shown in Fig. 2.4b. Here the rules are tried from top to bottom, and the first rule that fires is used to assign the class label to the instance to be classified. Thus, the class counts of each rule only show the examples that are not covered by previous rules. Moreover, the rule set ends in a *default rule* that will be used for class assignment when none of the previous rules fire.

The numbers that show the class distribution of the examples covered by a rule are not necessary. If desired, we can simply ignore them and interpret the rule categorically. However, the rules also give an indication about the reliability of a rule. Generally speaking, the more biased the distribution is towards a single class, and the more examples are covered by the rule, the more reliable is the rule. For example, intuitively the third rule in Fig. 2.4a is more reliable than the second rule, because it covers more examples, and it also covers only examples of a single class. Rules one, four, and five are also consistent, but they cover fewer examples. Indeed, it turns out that rules four and five misclassify examples in the test set. This intuitive understanding of rule reliability will be formalized in Sect. 2.5.3, where it is used for choosing among a set of candidate rules.



(a)



(b)

Fig. 2.4 Different types of rule-based theories induced from the car dataset. (**a**) Rule set. (**b**) Decision list

2.4 Rule Learning Process

Using a training set like the one of Table 2.1, the rule learning process is performed on three levels:

Feature construction. In this phase the object descriptions in the training data are turned into sets of binary features. For attribute-value data, we have already seen that features typically have the form $\mathbf{A}_i = v_{i,j}$ for a discrete attribute $\mathbf{A}_i$, or $\mathbf{A}_i < v$ or $\mathbf{A}_i \geq v$ if $\mathbf{A}_i$ is a numerical attribute. For different types of object representations (e.g., multirelational data, textual data, multimedia data, etc.), more sophisticated feature construction techniques can be used. Features and feature construction are the topic of Chap. 4.

Rule construction. Once the feature set is fixed, individual rules can be constructed, each covering a part of the example space. Typically, this is done by fixing the head of the rule to a single class value $\mathbf{C} = c_j$, and heuristically searching for the conjunction of features that is most predictive for this class. In this way the classification task is converted into a concept learning task in which examples of class c_i are positive and other examples are negative.

Hypothesis construction. A hypothesis consists of a set of rules. In propositional rule learning, hypothesis construction can be simplified by learning individual rules sequentially, for instance, by employing the covering algorithm, which will be described in Sect. 2.6. Using this algorithm, we can form either *unordered rule sets* or *ordered rule sets* (also known as *decision lists*). In first-order rule learning, the situation is more complex if recursion is employed, in which case rules cannot be learned independently. We will discuss this in Chap. 5.

Figure 2.5 illustrates a typical rule learning process, using several subroutines that we will detail further below. At the upper level, we have a multiclass classification problem which is transformed into a series of concept learning tasks. For each concept learning task there is a training set consisting of positive and negative examples of the target concept. For example, for learning the concept family, the dataset of Table 2.1 will be transformed into a set consisting of two positive examples (#4 and #9) and 16 negative examples (all others). Similar transformations are then made for the concepts sports (4 positive and 12 negative examples) and mini (12 positive and 6 negative examples).

The set of relevant features for each concept learning task can be constructed with the FEATURECONSTRUCTION algorithm, which will be discussed in more detail in Chap. 4. The LEARNONERULE algorithm uses these features to construct a rule body for the given target class. By iterative application of this algorithm the complete rule set can be obtained. In each iteration of the LEARNSETOFRULES algorithm, the set of examples is reduced by eliminating the examples covered in the previous iteration. When all positive examples have been covered, or some other stopping criterion is satisfied, the concept learning task is completed. The set of rules describing the target class is returned to the LEARNRULEBASE algorithm and included into the set of rules for classification.

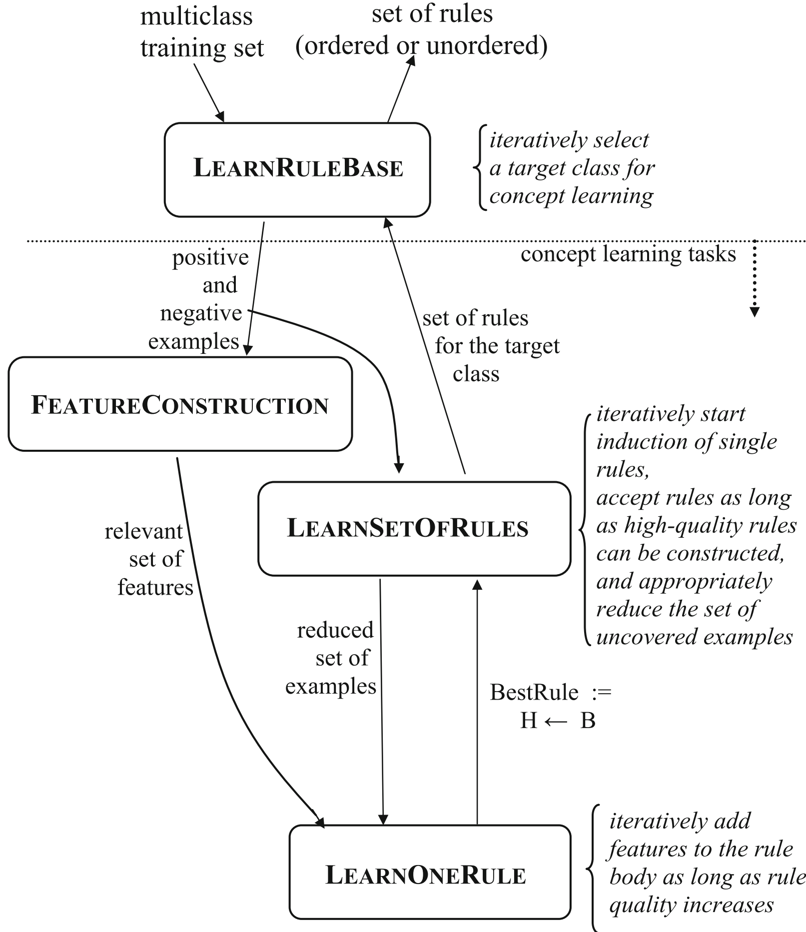


Fig. 2.5 Rule learning process

In the following sections, we will take a closer look at the key subroutines of this process, learning a single rule from data, and assembling multiple rules to a hypothesis in the form of a rule-based theory.

2.5 Learning a Single Rule

Learning of individual rules can be regarded as a search problem (Mitchell, 1982). To formulate the problem in this way, we have to define

– An appropriate *search space*
– A *search strategy* for searching through this space

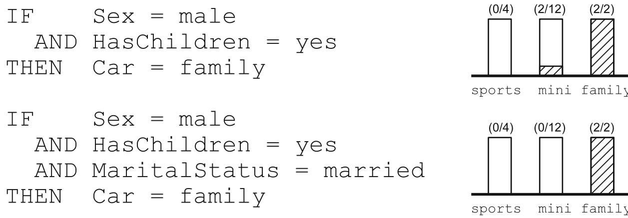


Fig. 2.6 The upper rule is more general than the lower rule

– A *quality function* that evaluates the rules in order to determine whether a candidate rule is a solution or how close it is to a solution.

We will briefly address these elements in the following sections.

2.5.1 Search Space

The search space of possible solutions is determined by the hypothesis language. In propositional rule learning, this is the space of all rules of the form $c \leftarrow \mathrm{B}$, with c being one of the classes, and B being a conjunction of features as described above (Sect. 2.3).

Generality relation. Enumerating the whole space of possible rules is often infeasible, even in the simple case of propositional rules over attribute-value data. It is therefore a good idea to structure the search space in order to search the space systematically, and to enable pruning of some parts of the search space. Nearly all symbolic inductive learning techniques structure the search by means of the dual notions of *generalization* and *specialization* (Mitchell, 1997).

Generality is most easily defined in terms of coverage. Let $\textsc{Covered}(\mathbf{r}, \mathcal{E})$ stand for the subset of examples in $\mathcal{E}$ which are covered by rule $\mathbf{r}$.

Definition 2.5.1 (Generality). A rule $\mathbf{r}$ is said to be *more general than* rule $\mathbf{r}'$, denoted as $\mathbf{r}' \subseteq \mathbf{r}$, iff

– Both $\mathbf{r}$ and $\mathbf{r}'$ have the same consequent, and
– $\textsc{Covered}(\mathbf{r}', \mathcal{E}) \subseteq \textsc{Covered}(\mathbf{r}, \mathcal{E})$.

We also say that $\mathbf{r}'$ is *more specific than* $\mathbf{r}$.

As an illustration, consider the two rules shown in Fig. 2.6.
The second rule has more features in its body and thus imposes more constraints on the examples it covers than the first. Thus, it will cover fewer examples and is therefore more specific than the first. In terms of coverage, the first rule covers four instances of Table 2.1 (examples 4, 9, 12, and 18), whereas the second rule covers

only two of them (4 and 9). Consequently, the first rule is more general than the second rule.

In case of continuous attributes, conditions involving inequalities are compared in the obvious way: e.g., a condition like `Age < 25` is more general than `Age < 20`. On the other hand, condition `Age = 22` would be less general than the first, but is incomparable to the second because it is neither a subset nor a superset of this rule.

The above definition of generality is sometimes called *semantic* generality because it is concerned with the semantics of the rules reflected in the examples they cover. However, computing this generality relation requires us to evaluate rules against a given dataset, which is costly. For learning conjunctive rules, a simple *syntactic* criterion can be used instead: given the same rule consequent, rule $\mathbf{r}$ is more general than rule $\mathbf{r}'$ if the antecedent of $\mathbf{r}'$ imposes at least the same constraints as the antecedent of $\mathbf{r}$, i.e., when $\text{CONDITIONS}(\mathbf{r}) \subseteq \text{CONDITIONS}(\mathbf{r}')$. For example, in Fig. 2.6, the lower rule is also a syntactic specialization of the upper rule, because the latter can be transformed into the former by deleting the condition `MaritalStatus = married`.

It is easy to see that syntactic generality defines a sufficient, but not necessary condition for semantic generality. For example, specialization could also operate over different attribute values (e.g., `Vienna` ⊆ `Austria` ⊆ `Europe`) or over different attributes (e.g., `Pregnancy = yes` ⊆ `Sex = female`).

Structuring the search space. The generality relation can be used to structure the hypothesis space by ordering rules according to this relation. It is easily seen that the relation of generality between rules is reflexive, antisymmetric, and transitive, hence a partial order.

The search space has a unique most general rule, the *universal rule* $\mathbf{r}^{\top}$, which has the body true and thus covers all examples, and a unique most specific rule, the *empty rule* $\mathbf{r}_{\perp}$, which has the body false and thus covers no examples. All other rules are more specific than the universal rule and more general than the empty rule. Thus, the universal rule is also called the *top rule*, and the empty rule is also called the *bottom rule* of the hypothesis space, which is indicated by the symbols $\top$ and $\perp$. However, the term bottom rule is also often used to refer to the most specific rule $\mathbf{r}_e$ that covers a given example e. Such a bottom rule typically consists of a conjunction of all features that are true for this particular example. We will use the terms universal rule and empty rule for the unique most general and most specific rules in the hypothesis space, and reserve the term bottom rule for the most specific rule relative to a given example.

The syntactic generality relation can be used to define a so-called *refinement operator* that allows navigation in this ordered space. A rule can be *specialized* by conjunctively adding a condition to the rule, or it can be *generalized* by deleting one of its conditions. Figure 2.7 shows the space of all generalizations of the conjunction `MaritalStatus = married, HasChildren=yes, Sex=male`. This rule could be reached by six different paths that start from the universal rule at the top. Each step on this path consists of refining the rule in the

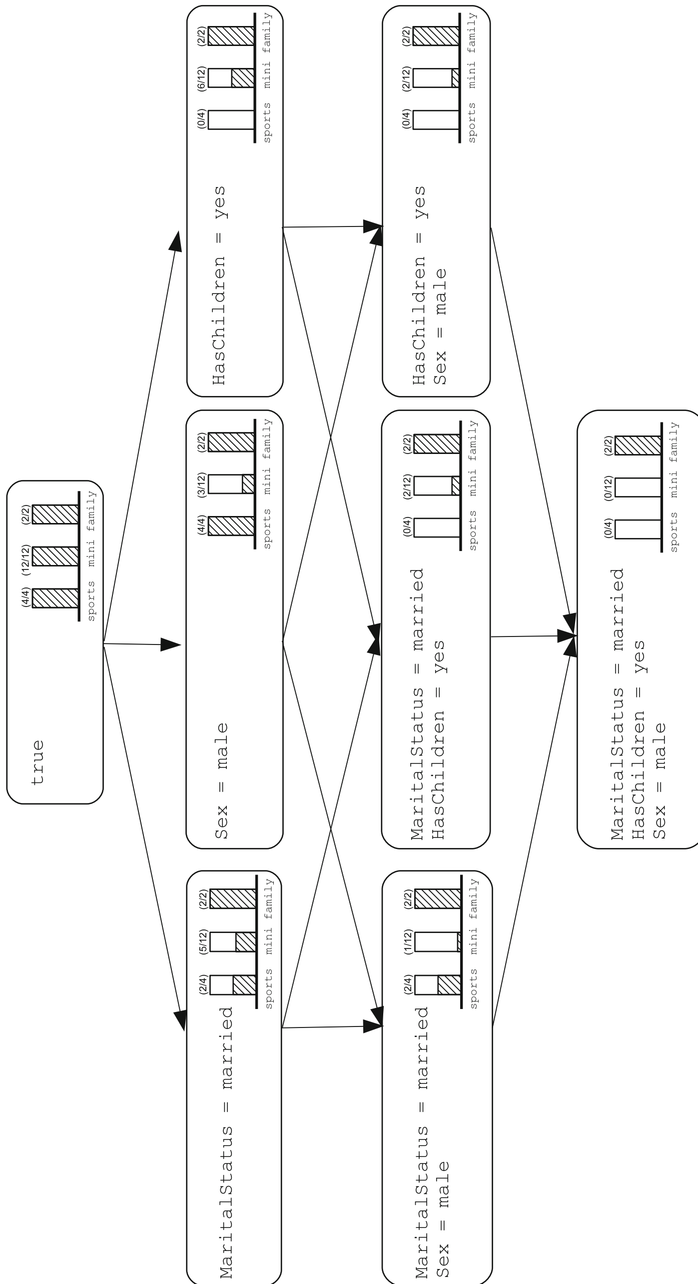


Fig. 2.7 All generalizations of `MaritalStatus = married`, `HasChildren = yes`, `Sex = male`, shown as a generalization hierarchy

current node by adding a condition, resulting in a more specific rule that covers fewer examples. Thus, since a more specific rule will cover (the same or) a subset of the already covered examples, making a rule more specific (or *specializing it*) is a way to obtain consistent (pure) rules which cover only examples of the target class. In this case, each path successively removes examples of all classes other than family, eventually resulting in a rule that covers all examples of this class and no examples from other classes.

Note, however, that Fig. 2.7 only shows a small snapshot of the actual search space. In principle, the universal rule could be refined into nine rules with a single condition (one for each possible value of each of the four attributes), which in turn can be refined into 30 rules with 2 conditions, 44 rules with 3 conditions, and 24 rules with 4 conditions before we arrive at the empty rule. Thus, the total search space has $1 + 9 + 30 + 44 + 24 + 1 = 109$ rules. The number of paths through this graph is $24 \times 4! = 576$.

Thus it is important to avoid searching unpromising branches and to avoid searching parts of the graph multiple times. By exploiting the monotonicity of the generality relation, the partially ordered search space can be searched efficiently because

- When generalizing rule $\mathbf{r}'$ to $\mathbf{r}$ all training examples covered by $\mathbf{r}'$ will also be covered by $\mathbf{r}$,
- When specializing rule $\mathbf{r}$ to $\mathbf{r}'$ all training examples not covered by $\mathbf{r}$ will also not be covered by $\mathbf{r}'$.

Both properties can be used to prune large parts of the search space of rules. The second property is often used in conjunction with positive examples. If a rule does not cover a positive example, all specializations of that rule can be pruned, as they also cannot cover the example. Similarly, the first property is often used with negative examples: if a rule covers a negative example, all its generalizations can be pruned since they will cover that negative example as well.

Searching through such a *refinement graph*, i.e., a graph which has rules as its nodes and applications of a refinement operator as edges, can be seen as a balancing act between rule coverage (the proportion of examples covered by a rule) and rule precision (the proportion of examples correctly classified by a rule). We will address the issue of rule quality evaluation in Sect. 2.5.3.

2.5.2 Search Strategy

For learning a single rule, most learners use one of the following search strategies.

- *General-to-specific* or top-down learners start from the most general rule and repeatedly specialize it as long as the found rules still cover negative examples. Specialization stops when a rule is consistent. During the search, general-to-specific learners ensure that the rules considered cover at least one positive example.

---

function LEARNONERULE(c_i,$\mathcal{P}_i$,$\mathcal{N}_i$)

Input:
c_i: a class value
$\mathcal{P}_i$: a set of positive examples for class c_i
$\mathcal{N}_i$: a set of negative examples for class c_i
$\mathcal{F}$: a set of features

Algorithm:
$\mathbf{r} := (c_i \leftarrow \mathrm{B})$, where $\mathrm{B} \leftarrow \emptyset$
repeat
build refinements $\rho(\mathbf{r}) \leftarrow \{\mathbf{r}' \mid \mathbf{r}' = (c_i \leftarrow \mathrm{B} \wedge \mathbf{f})\}$ for all $\mathbf{f} \in \mathcal{F}$
evaluate all $\mathbf{r}' \in \rho(\mathbf{r})$ according to a quality criterion
$\mathbf{r}$:= the best refinement $\mathbf{r}'$ in $\rho(\mathbf{r})$
until $\mathbf{r}$ satisfies a quality threshold
or covers no examples from $\mathcal{N}_i$

Output:
learned rule $\mathbf{r}$

Fig. 2.8 A general-to-specific hill-climbing algorithm for single rule learning

– *Specific-to-general* or bottom-up learners start from a most specific rule (either the empty rule or a bottom rule for a given example), and then generalize the rule until it cannot further be generalized without covering negative examples.

The first approach generates rules from the top of the generality ordering downwards, whereas the second proceeds from the bottom of the generality ordering upwards. Typically, top-down search will find more general rules than bottom-up search, and is thus less cautious and makes larger inductive leaps. General-to-specific search is very well suited for learning in the presence of noise because it can easily be guided by heuristics.

Specific-to-general search strategies, on the other hand, seem better suited for situations where fewer examples are available and for interactive and incremental processing. These learners are, however, quite susceptible to noise in the data, and cannot be used for hill-climbing searches, such as a bottom-up version of the LEARNONERULE algorithm introduced below. Bottom-up algorithms must therefore be combined with more elaborate refinement operators. Even though bottom-up learners enjoyed some popularity in inductive logic programming, most practical systems nowadays use a top-down strategy.

Using a refinement operator, it is easy to define a simple general-to-specific search algorithm for learning individual rules. A possible implementation of this algorithm, called LEARNONERULE, is sketched in Fig. 2.8. The algorithm repeatedly refines the current best rule, and selects the best of all computed refinements according to some quality criterion. This amounts to a *top-down hill-climbing*[2]

[2]If the term 'top-down hill-climbing' sounds contradictory: hill-climbing refers to the process of greedily moving towards a (local) optimum of the evaluation function, whereas top-down refers to the fact that the search space is searched by successively specializing the candidate rules, thereby moving downwards in the generalization hierarchy induced by the rules.

search strategy. LEARNONERULE is, essentially, equivalent to the algorithm used in the PRISM learning system (Cendrowska, 1987). It is straightforward to modify the algorithm to return not only one but a beam of the b best rules, using the so-called *beam search* strategy.[3] This strategy is, for example, used in the CN2 learning algorithm.

The LEARNONERULE algorithm contains several heuristic choices. For example, it uses a heuristic quality function for selecting the best refinement, and it stops rule refinement either when a stopping criterion is satisfied or when no further refinement is possible. We will briefly discuss these options in the next section, but refer to Chaps. 7 and 9 for more details.

2.5.3 Evaluating the Quality of Rules

A key issue in the LEARNONERULE algorithm of Fig. 2.8 is how to evaluate and compare different rules, so that the search can be focused on finding the best possible rule refinement. Numerous measures are used for rule evaluation in machine learning and data mining. In classification rule induction, frequently used measures include *precision*, *information gain*, *correlation*, the *m-estimate*, the *Laplace estimate*, and others. In this section, we focus on the basic principle underlying these measures, namely a simultaneous optimization of consistency and coverage, and present a few simple measures. Two more measures will be presented in Sect. 2.7, but a detailed discussion of rule learning heuristics will follow in Chap. 7.

Terminological and notational conventions. In concept learning, examples are either positive or negative examples of a given target class $\oplus$, and they are covered (predicted positive) or not covered (predicted negative) by a rule **r** or set of rules $\mathcal{R}$. Positive examples correctly predicted to be positive are called *true positives*, correctly predicted negative examples are called *true negatives*, positives incorrectly predicted as negative are called *false negatives*, and negatives predicted as positive are called *false positives*. This situation can be plotted in the form of a 2×2 table, as shown in Table 2.3.

In the following, we will briefly introduce some of our notational conventions. A summary can be found in Table I in a separate section in the frontmatter (pp. xi–xiii). We will use the letters $\mathcal{E}$, $\mathcal{P}$, and $\mathcal{N}$ to refer to all examples, the positive examples, and the negative examples, respectively. Calligraphic font is used for denoting sets, and the corresponding uppercase letters E, P, and N are used for denoting the sizes of these sets. Table 2.3 thus shows the four possible subsets into which the example set $\mathcal{E}$ can be divided, depending on whether the example is positive or negative, and

---

[3]Beam search is a heuristic search algorithm that explores a graph by expanding just a limited set of the most promising nodes (cf. also Sect. 6.3.1).

Table 2.3 Confusion matrix depicting the notation for sets of covered and uncovered positive and negative examples (in *calligraphic font*) and their respective absolute numbers (in *parantheses*)

Examples	Covered	Not Covered	
Positive	$\hat{\mathcal{P}}$ $(\hat{P})$ (true positives)	$\bar{\mathcal{P}}$ $(\bar{P})$ (false negatives)	$\mathcal{P}$ (P)
Negative	$\hat{\mathcal{N}}$ $(\hat{N})$ (false positives)	$\bar{\mathcal{N}}$ $(\bar{N})$ (true negatives)	$\mathcal{N}$ (N)
Total	$\hat{\mathcal{E}}$ $(\hat{E})$	$\bar{\mathcal{E}}$ $(\bar{E})$	$\mathcal{E}$ (E)

whether it is covered or not covered by rule **r**. Coverage is denoted by adding a hat (ˆ) on top of a letter; noncoverage is denoted by a bar (¯).

Goals of rule learning heuristics. The goal of a rule learning algorithm is to find a simple set of rules that explains the training data and generalizes well to unseen data. This means that individual rules have to simultaneously optimize two criteria:

- *Coverage:* the number of positive examples that are covered by the rule ($\hat{P}$) should be maximized, and
- *Consistency:* the number of negative examples that are covered by the rule ($\hat{N}$) should be minimized.

Thus, we have a multi-objective optimization problem, namely to simultaneously maximize $\hat{P}$ and minimize $\hat{N}$. Equivalently, one can minimize $\bar{P} = P - \hat{P}$ and maximize $\bar{N} = N - \hat{N}$. Thus, the quality of a rule can be characterized by four of the entries in the confusion matrix. As P and N are constant for a given dataset, the heuristics effectively only differ in the way they trade off completeness (maximizing $\hat{P}$) and consistency (minimizing $\hat{N}$). Thus they may be viewed as functions $H(\hat{P}, \hat{N})$.

What follows is a very short selection of rule quality measures. All of them are applicable for a single rule **r** but, in principle, they can also be used for evaluating a set of rules constructed for the positive class (an example is covered by a rule set if it is covered by at least one rule from the set). The presented selection does not aim for completeness or quality, but is meant to illustrate the main problems and principles. An exhaustive survey and analysis of rule evaluation measures is presented in Chap. 7.

Selected rule learning heuristics. As discussed above, the two key values that characterize the quality of a rule are $\hat{P}$, the number of covered positive examples, and $\hat{N}$, the number of covered negative examples. Optimizing each one individually is insufficient, as it will either neglect consistency or completeness.

A simple way to trade these values off is to form a linear combination, in the simplest case

$$\boxed{CovDiff(\mathbf{r}) = \hat{P} - \hat{N}}$$

which gives equal weight to both components. One can also normalize the two components and use the difference between the *true positive rate* ($\hat{\pi}$) and the *false positive rate* ($\hat{\nu}$).

$$\boxed{RateDiff(\mathbf{r}) = \frac{\hat{P}}{P} - \frac{\hat{N}}{N} = \hat{\pi} - \hat{\nu}}$$

Instead of taking the difference, one can also compute the relative frequency of positive examples in all the covered examples:

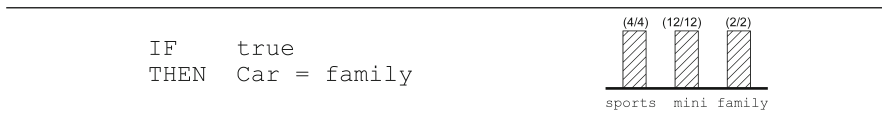
$$\boxed{Precision(\mathbf{r}) = \frac{\hat{P}}{\hat{P} + \hat{N}} = \frac{\hat{P}}{\hat{E}}}$$

Essentially, this measure estimates the probability $\Pr(\oplus \mid \mathrm{B})$ that an example that is covered by (the body of) a rule **r** is positive. This measure is known under several names, including *precision*, *confidence*, and *rule accuracy*. We will stick with the first term.

These are only three simple examples that are meant to illustrate how a trade-off between consistency and coverage is achieved. They are not among the best-performing heuristics. Later in this chapter (in Sect. 2.7.2), we will introduce two more heuristics that are commonly used to fight overfitting.

2.5.4 *Example*

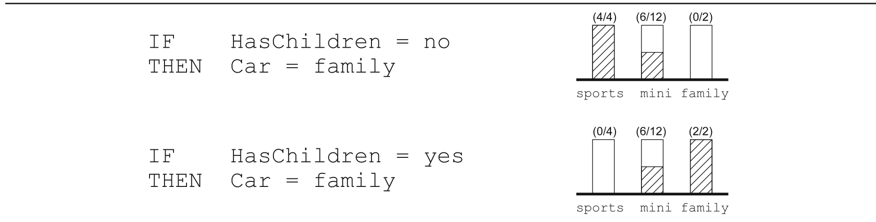
We will now look at a concrete example of a rule learning algorithm at work. We again use the car database from Table 2.1, and, for the moment, rule precision as a measure of rule quality. Consider calling LEARNONERULE to learn the first rule for the class `Car = family`. The rule is initialized with an empty body, so that it classifies all examples into class `family`.



This rule covers all four examples of class `sports`, all two examples of class `family`, and all 12 examples of class `mini`. Given 2 true positives and 16 false positives, it has precision $\frac{2}{18} = 0.11$.

In the next run of the `repeat` loop, the algorithm of Fig. 2.8 will need to select the most promising refinement by conjunctively adding the best feature to

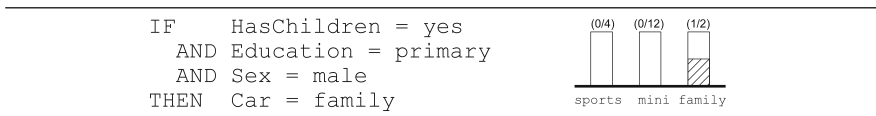
the currently empty rule body. In this case there are as many refinements as there are values for all attributes; there are $3 + 2 + 2 + 2 = 9$ possible refinements in the car domain. Shown below are the two possible refinements that concern the attribute `HasChildren`:



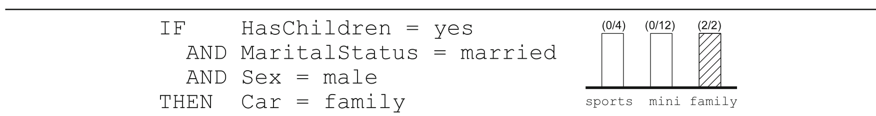
Clearly the second refinement is better than the first for predicting the class family. Its precision is estimated at $\frac{2}{8} = 0.25$. As it turns out, this rule is the best one in this iteration, and we proceed to refine it further.

Table 2.4 presents all seven possible refinements in the second iteration. Next to *Precision*, heuristic values for *CovDiff*, *RateDiff*, and *Laplace* are presented.[4] In bold are the best refinements for each evaluation measure. It can be noticed that for *CovDiff*, *Precision*, and *Laplace* there are three best solutions, while for *RateDiff* there are only two. Selecting at random among optimal solutions and using, for example, *Precision*, it can happen that we select the first refinement `HasChildren = yes AND Education = primary`, which is not an ideal solution according to *RateDiff*. The example demonstrates a common fact that different heuristics may result in different refinement selections and consequently also in different final solutions.

This is confirmed by the third iteration. If refinement `HasChildren = yes AND Education = primary` is used, then the final solution is:



This rule covers one example of class `family` and no examples of other classes. In contrast to that, if we start with `HasChildren = yes AND MaritalStatus = married` then all heuristics will successfully find the optimal solution:



[4] *Laplace* will be defined in Sect. 2.7.

Table 2.4 All possible refinements of the rule `IF HasChildren = yes THEN Car = family` in the second iteration step of LEARNONERULE. Shown is the feature that is added to the rule, the number of covered examples of each of the three classes, and the evaluation of four different heuristics

	Covered examples of class			Heuristic evaluation			
Added feature	Sports	Mini	Family	*CovDiff*	*RateDiff*	*Precision*	*Laplace*
`Education = primary`	0	1	1	**0**	0.437	**0.5**	**0.5**
`Education = secondary`	0	2	1	−1	0.375	0.333	0.4
`Education = university`	0	3	0	−3	−0.187	0.0	0.2
`MaritalStatus = married`	0	2	2	**0**	**0.875**	**0.5**	**0.5**
`MaritalStatus = single`	0	4	0	−4	−0.25	0.0	0.167
`Sex = male`	0	2	2	**0**	**0.875**	**0.5**	**0.5**
`Sex = female`	0	4	0	−4	−0.25	0.0	0.167

2.6 Learning a Rule-Based Model

Real-world hypotheses can only rarely be formulated with a single rule. Thus, both general-to-specific learners and specific-to-general learners repeat the procedure of single rule learning on a reduced example set, if the constructed rule by itself does not cover all positive examples. They use thus an iterative process to compute disjunctive hypotheses consisting of more than one rule.

In this section, we briefly discuss methods that repeatedly call the LEARNONE RULE algorithm to learn multiple rules and combine them into a rule set. We will first discuss the covering algorithm, which forms the basis of most rule learning algorithms, and then discuss how we can deal with multiclass problems.

2.6.1 The Covering Algorithm

The *covering* or *separate-and-conquer* strategy has its origins in the AQ family of algorithms (Michalski, 1969). The term *separate-and-conquer* has been coined by Pagallo and Haussler (1990) because of the way of developing a theory that characterizes this learning strategy: learn a rule that covers a part of the given training examples, remove the covered examples from the training set (the *separate* part), and recursively learn another rule that covers some of the remaining examples (the *conquer* part) until no examples remain. The terminological choice is a matter of personal taste; both terms can be found in the literature.

The basic covering algorithm shown in Fig. 2.9 learns a set of rules $\mathcal{R}_i$ for a given class c_i. It starts to learn a rule by calling the LEARNONERULE algorithm. After the found rule is added to the hypothesis, examples covered by that rule are deleted from the current set of examples, so that they will not influence the generation of subsequent rules. This is done via calls to COVERED($\mathbf{r}$, $\mathcal{E}$), which returns the subset of examples in $\mathcal{E}$ that are covered by rule $\mathbf{r}$. This cycle of adding rules and removing covered examples is repeated until no more examples of the given class remain. In this case, all examples of this class are covered by at least one rule. We will see later (Sect. 2.7) that sometimes it may be advisable to leave some examples uncovered, i.e., no more rules will be added as soon as some external *stopping criterion* is satisfied.

2.6.2 Learning a Rule Base for Classification Problems

The basic LEARNSETOFRULES algorithm can only learn a rule set for a single class. In a concept learning setting, this rule set can be used to predict whether an example is a member of the class c_i or not. However, many real-world problems are multiclass, i.e., it is necessary to learn rules for more than one class.

```
function LearnSetOfRules(c_i, P_i, N_i)
  Input:
    c_i: a class value
    P_i: a set of positive examples for class c_i
    N_i: a set of negative examples for class c_i, where N_i = E \ P_i

  Algorithm:
    P_i^cur := P_i, N_i^cur := N_i
    R_i := ∅
    repeat
        r := LearnOneRule(c_i, P_i^cur, N_i^cur)
        R_i := R_i ∪ {r}
        P_i^cur := P_i^cur \ Covered(r, P_i^cur)
        N_i^cur := N_i^cur \ Covered(r, N_i^cur)
    until R_i satisfies a quality threshold or P_i^cur is empty

  Output:
    R_i the rule set learned for class c_i
```

Fig. 2.9 The covering algorithm for rule sets

```
function LearnRuleBase(E)
  Input:
    E set of training examples

  Algorithm:
    R := ∅
    for each class c_i, i = 1 to C do
        P_i := {subset of examples in E with class label c_i}
        N_i := {subset of examples in E with other class labels}
        R_i := LearnSetOfRules(c_i, P_i, N_i)
        R := R ∪ R_i
    endfor
    R := R ∪ {default rule (c_max ← true)}
        where c_max is the majority class in E.

  Output:
    R the learned rule set
```

Fig. 2.10 Constructing a set of rules in a multiclass learning setting

A straightforward way to tackle such problems is to learn a *rule base* $\mathcal{R} = \bigcup_i \mathcal{R}_i$ that consists of a rule set $\mathcal{R}_i$ for each class. This can be learned with the algorithm LearnRuleBase, shown in Fig. 2.10, which simply iterates calls to LearnSetOfRules over all the C classes c_i. In each iteration the current positive class will be learned against the negatives provided by all other classes.

At the end, we need to learn a *default rule*, which simply predicts the majority class in the data set. This rule is necessary in order to make sure that new examples that may not be covered by any of the learned rules, can nevertheless be assigned a class value.

This strategy of repeatedly learning one rule set for each class is also known as the *one-against-all* learning strategy. We note in passing that other strategies are possible. This, and several other learning strategies (including strategies for learning decision lists) are the subject of Chap. 10.

2.7 Overfitting Avoidance

Most top-down rule learners can be fit into the high-level description provided in the previous sections. For doing so, we need to configure the LEARNONERULE algorithm of Fig. 2.8 with appropriate heuristics for

- Evaluating the quality of a single rule,
- Deciding when to stop refining a rule, and
- Deciding when to stop adding rules to a rule set for a given class.

So far, we have defined very simple rule evaluation criteria, and used consistency and completeness as stopping criteria. However, these choices are appropriate only in idealistic situations. For practical applications, one has to deal with the problem of *overfitting*, which is a common phenomenon in data analysis (cf. also Sect. 2.1). Essentially, the problem is that rule sets that exactly fit the training data often do not generalize well to unseen data. In such cases, heuristics are needed to trade off the quality of a rule or rule set with other factors, such as their complexity.

In the following, we will briefly discuss the choices that are made by the CN2 learning algorithm. More elaborate descriptions of rule evaluation criteria can be found in Chap. 7, and stopping criteria are discussed in more detail in Chap. 9.

2.7.1 Rule Evaluation in CN2

Rules are evaluated on a training set of examples, but we are interested in estimates of their performance on the whole example set. In particular for rules that cover only a few examples, their evaluation values may not be representative for the entire domain. For simplicity, we illustrate this problem by estimating the precision heuristic, but in principle the argument applies to any function where a population probability is to be estimated from sample frequencies.

A key problem with precision is that for very low numbers of $\hat{N}$ and $\hat{P}$, this measure is not very robust. If both $\hat{P}$ and $\hat{N}$ are low, one extra covered positive or negative example may significantly change the evaluation value. Compare, e.g., two rules $\mathbf{r}_1$ and $\mathbf{r}_2$, both covering no negative examples ($\hat{N}_1 = \hat{N}_2 = 0$), but the first one covers 1 positive ($\hat{P}_1 = 1$), and the second one covers 99 positive examples ($\hat{P}_2 = 99$). Both have a precision of 1.0. However, if it turns out that each rule covers one additional negative example ($\hat{N}_1 = \hat{N}_2 = 1$), the evaluation of $\mathbf{r}_1$ drops to $\frac{1}{1+1} = 0.5$, while the evaluation of $\mathbf{r}_2$ is still very high ($\frac{99}{1+99} = 0.99$).

The Laplace estimate addresses this problem by adding two 'virtual' covered examples, one for each class, resulting in the formula

$$\mathit{Laplace}(\mathbf{r}) = \frac{\hat{P}+1}{(\hat{P}+1)+(\hat{N}+1)} = \frac{\hat{P}+1}{\hat{E}+2}$$

In the above example, the estimate for $\mathbf{r}_1$ would be $\frac{2}{3}$, rather than 1, which is much closer to the value $\frac{1}{2}$ that results from covering one additional negative example. The estimate asymptotically approaches 1 if the number of true positives increases, but with finite amounts of data the probability estimate will never be exactly 1 (or 0). In fact, the Laplace correction of the relative frequency of covered positive examples is an unbiased estimate for the probability $\Pr(\oplus \mid \mathrm{B})$ of the positive class given the body of the rule.[5] Example calculations for the Laplace values can be seen in Table 2.4.

2.7.2 *Stopping Criteria in* CN2

CN2 can use a significance measure to enforce the induction of reliable rules. If CN2 is used to induce sets of unordered rules, the rules are usually required to be highly *significant* (at the 99 % level), and thus reliable, representing a regularity unlikely to have occurred by chance. To test significance, CN2 uses the *likelihood ratio statistic* (Clark & Niblett, 1989) that compares the class probability distributions in the set of covered examples with the distribution over the training set, i.e., the distribution of examples covered by the rule compared to the prior distribution of examples in the training set. A rule is deemed reliable if these two distributions are significantly different. For a two-class problem, rule significance is measured as follows:

$$\mathit{LRS}(\mathbf{r}) = 2 \cdot \left(\hat{P} \cdot \log_2 \frac{\frac{\hat{P}}{\hat{P}+\hat{N}}}{\frac{P}{P+N}} + \hat{N} \cdot \log_2 \frac{\frac{\hat{N}}{\hat{P}+\hat{N}}}{\frac{N}{P+N}} \right)$$

[5] If $C > 2$ classes are used, the relative frequencies for each class should be estimated with $\frac{\hat{P}_i+1}{\sum_{j=1}^{C} \hat{P}_j + C}$, where $\hat{P}_i$ is the number of examples of class i covered by the rule and C is the number of classes. However, if we estimate the probability whether an example that is covered by the body of a rule is also covered by its head or not, we have a binary distinction even in multiclass problems.

For a multiclass problem, the value of the likelihood ratio statistic is computed as follows[6]:

$$LRS(\mathbf{r}) = 2 \cdot \sum_{i=1}^{C} \hat{P}_i \log_2 \frac{\hat{\pi}_i}{\gamma_i}$$

where $\hat{\pi}_i = \frac{\hat{P}_i}{\hat{E}}$ is the proportion of covered examples of class c_i, and $\gamma_i = \frac{P_i}{E}$ is the overall proportion of examples of class c_i. This statistic is distributed as χ^2 with $C - 1$ degrees of freedom. The rule is only considered significant if its likelihood ratio is above a specified significance threshold.

2.8 Making a Prediction

After having learned a rule set, how do we use it for classifying new instances? Classification with a decision list is quite straightforward. To classify a new instance, the rules are tried in order. The first rule that covers the instance is used for classification/prediction. If no induced rule fires, a default rule is invoked, which typically predicts the majority class of uncovered training examples.

In the case of unordered rule sets, the situation is more complicated because all the rules are tried and predictions of those that cover the example are collected. Two problems have to be dealt with:

- *Conflict resolution:* multiple overlapping rules may make possibly conflicting predictions for the same example
- *Uncovered examples:* no rule may cover the example.

As we have seen in the previous section, the second case is handled by adding a default rule to the rule set.

The typical solution for the first problem is to use a voting mechanism to obtain the final prediction. Conflicting decisions are resolved by taking into account the number of examples of each class (from the training set) covered by each rule. CN2, for example, sums up the class distributions attached to the rules to determine the most probable class. For example, in the rule set of Fig. 2.4a, example 13 is covered by rules nos. 2 and 5. If we sum up the class counts for each rule, we obtain 4 counts for `sports`, 3 counts for `mini`, and none for `family`. As a result, the person would still be assigned to the `sports` car class. These counts can also be

[6]Clark and Niblett (1989, p. 269) define the likelihood ratio statistic in the form $LRS(\mathbf{r}) = 2 \cdot \sum_{i=1}^{C} \hat{P}_i \cdot \log_2 \frac{\hat{P}_i}{\mathbb{E}\hat{P}_i}$, where $\mathbb{E}\hat{P}_i = \gamma_i \cdot \hat{E}$ is the expected *number* of examples of class c_i that the rule would cover if the covered examples were distributed with the same relative class frequencies as in the original dataset. A simple transformation gives our formulation with ratios of observed relative frequencies and expected relative frequencies.

used for probabilistic estimation (Džeroski, Cestnik, & Petrovski, 1993). In this case, we would predict class `sports` with a probability of $4/7$ and class `mini` with a probability of $3/7$. Class `family` would be ruled out.

As an alternative to standard CN2 classification, where the rule class distribution is computed in terms of the numbers of examples covered, class distribution can be given also in terms of probabilities (computed by the relative frequencies $\hat{\pi}$ of each class). In the above example, rule 2 predicts `sports` with a probability of $4/5$, and class `mini` with a probability of $1/5$, whereas rule 5 predicts `mini` with a probability of 1. Now the probabilities are averaged (instead of summing the numbers of examples), resulting in the final probability distribution $[0.4, 0, 0.6]$, and now the correct class `mini` is predicted. By using this voting scheme the rules covering a small number of examples are not so heavily penalized (as is the case in CN2) when classifying a new example. However, this may also become a problem when some of the rules overfit the data. Thus, often a Laplace-correction is used for estimating these probabilities from the data, as discussed in the previous section.

One may also use different aggregations than averaging the class probability distributions. RIPPER (Cohen, 1995), for example, resolves ties by using the rule that has the largest (Laplace-corrected) probability estimate for its majority class. Note that this approach is equivalent to converting the rule set into a decision list by sorting its rules according to the above-mentioned probability estimates and using the first one that fires.

In the literature on rule learning, we can find a number of more elaborate approaches for handling conflicting predictions and for predictions in the case when no rule covers an example. We will discuss a few of them in Sect. 10.2.

2.9 Estimating the Predictive Accuracy of Rules

Classification quality of the rule set is measured by the classification accuracy, which is defined as the percentage of the total number of correctly classified examples in all classes relative to the total number of tested examples. As a special case, for a binary classification problem, rule set accuracy is computed as follows:

$$\mathit{Accuracy}(\mathcal{R}) = \frac{\hat{P} + \bar{N}}{P + N}$$

Note that accuracy measures the classification accuracy of the whole rule set on both positive and negative examples. It should not be confused with *rule accuracy*, for which we use the term *precision*. The coverage difference measure *CovDiff* (Sect. 2.5.3) is equivalent to evaluating the accuracy of a single rule, when we consider that $\bar{N} = N - \hat{N}$, and that P and N are constant and can be omitted when comparing the accuracy of different rules in the same domain.

Instead of accuracy, results are often presented with *classification error*, which is simply

$$Error(\mathcal{R}) = 1 - Accuracy(\mathcal{R}) = \frac{\bar{P} + \hat{N}}{P + N}$$

Obviously, both measures are equivalent, and the choice is a matter of personal taste.

2.9.1 Hold-Out Set Evaluation

It is not valid to estimate the predictive accuracy of a theory on the same dataset that was used for learning. As an illustration, consider a rule learning algorithm that simply constructs one rule for each positive training example by formulating the body of the rule as a conjunction of all attribute values that appear in the definition of the example, and classifying everything else as negative. Such a set of rules would have 100 % accuracy[7] if estimated on the training data, but in reality, this theory would misclassify all examples that belong to the positive class except those that it has already seen during training. Thus, no generalization takes place, and the rule is worthless for practical purposes.

In order to solve this problem, predictive accuracy should be estimated on a separate part of the data, a so-called *test set*, that was removed (held out) from the training phase. Recall that Table 2.2 shows such a test set for the car domain in Table 2.1. If we classify these examples with the decision list of Fig. 2.4a, example 20 will be misclassified as `mini` (by the default rule), while all other examples will be classified correctly. Thus, the estimated predictive accuracy of this rule set would be $5/6 \approx 83.3$ %. On the training data, the decision list made only one mistake in 18 examples, resulting in an accuracy of $17/18 \approx 94.4$ %. Even though these measures are computed from a very small data sample and are thus quite unreliable, the result that the test accuracy is lower than the training accuracy is typical.

2.9.2 Cross-Validation

While hold-out set evaluation produces more accurate estimates for the predictive accuracy, it also results in a waste of training data because only part of the available data can be used for learning. It may also be quite unreliable if only a single, small test set is used, as in the above example.

Cross-validation is a common method for solving this problem: while all of the data are used for learning, the accuracy of the resulting theory is estimated by performing k hold-out experiments as described above. For this purpose, the data is divided into n parts. In each experiment, $n-1$ parts are combined into a training set,

---

[7] We assume that there are no contradictory examples in the training set, an assumption that does not always hold in practice, but the basic argument remains the same.

---

function CROSSVALIDATION($\mathcal{E}$,k)

Input:
$\mathcal{E}$ set of training examples
k the number of folds

Algorithm:
randomly partition $\mathcal{E}$ into f disjoint folds $\mathcal{E}_j$ of approximately the same size
for $j = 1$ to f **do**
$\mathcal{E}_{test} := \mathcal{E}_j$
$\mathcal{E}_{train} := \mathcal{E} \setminus \mathcal{E}_{test}$
$\mathcal{R} :=$ LEARNRULEBASE($\mathcal{E}_{train}$)
$v_j :=$ EVALUATE($\mathcal{R}, \mathcal{E}_{test}$)
endfor
$v := \frac{1}{k} \sum_{j=1}^{k} v_j$

Output:
v the estimated quality

Fig. 2.11 Estimation of the prediction quality of a rule set induced by LEARNRULEBASE via cross-validation

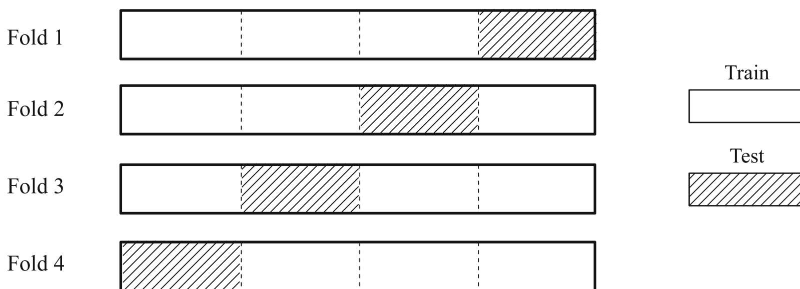


Fig. 2.12 A visual representation of fourfold cross-validation, where predictive accuracy is estimated as the average rule set accuracy of four hold-out experiments

and the remaining part is used for testing. A theory is then learned on the training set and evaluated on the test set. This is repeated until each part (and thus each training example) has been used for testing once. The final accuracy is then estimated as an average of the accuracy estimates computed in each such hold-out experiment. The cross-validation algorithm is shown in Fig. 2.11. Note that the procedure can be used for estimating any aspect of the quality of the learned rules. It is thus shown with a generic function EVALUATE, which can be instantiated with any common evaluation measure, such as accuracy, recall and precision, area under the ROC curve, etc.

Figure 2.12 shows a schematic depiction of a fourfold cross-validation. In practice, tenfold cross-validation is most commonly used. An interesting special case is *leave-one-out cross-validation*, where in each iteration only a single example is held out from training and subsequently used for testing. Because of the large number of theories to be learned (one for each training example), this is only feasible for small datasets.

2.9.3 Benchmark Datasets

Results on a single dataset are typically not very meaningful. Therefore, machine learning techniques are often evaluated on a large set of benchmark datasets. There are several collections of benchmark datasets available; the best-known is the *UCI machine learning repository* (Frand & Asuncion, 2010).[8]

This approach has several advantages, including that tests on a large collection of databases reduce the risk of getting spurious results, and that published results on these datasets are, in principle, reproducible. Some researchers are even working on experiment databases that collect experimental results on these benchmark datasets for various algorithms under various experimental setups in order to increase the repeatability and the reliability of reported results (Blockeel & Vanschoren, 2007).

On the other hand, this approach also has disadvantages. Most notably, it is unclear how representative these datasets are for real-world applications because many datasets have been donated because someone has (typically successfully) presented them in a publication. Thus there is a certain danger that the collection is biased against hard problems, and that algorithms that are *overfitting the UCI repository* may actually not be applicable to real-world problems. Although there is some evidence that these problems may be overestimated (Soares, 2003), the debate is still going on. In any case, using these datasets is still common practice, and we will also occasionally show results on some of these databases in this book.

2.10 A Brief History of Predictive Rule Learning Algorithms

In this section we give a historical account of some of the most influential covering algorithms. Most of them are still in use and are regularly cited in the rule learning literature.

2.10.1 AQ

AQ can be considered as the original covering algorithm. Its original version was conceived by Ryszard Michalski in the 1960s (Michalski, 1969). Over the years, numerous versions and variants of the algorithm appear in the literature (Bergadano, Matwin, Michalski, & Zhang, 1992; Kaufman & Michalski, 2000; Michalski, 1980; Michalski & Larson, 1978; Michalski, Mozetič, Hong, & Lavrač, 1986). A very good summary of the early versions of AQ is given in (Clark & Niblett, 1987, 1989).

[8] At the time of this writing, the collection contains 177 datasets in a great variety of different domains, including bioinformatics, medical applications, financial prognosis, game playing, politics, and more.

The basic algorithm features many typical components of the covering algorithm, but it also has a few interesting particularities. The algorithm uses a top-down beam search for finding the best rule.[9] However, contrary to most successors, AQ does not search all possible refinements of a rule; it only considers refinements that cover a particular example, the so-called *seed example*. In each step, it looks for refinements (*extensions* in AQ) of rules that cover the seed example, but do not cover a randomly chosen negative example. All possible refinements that meet this constraint are evaluated with rule learning heuristics (in the simplest version it was positive coverage p), and the best b rules are maintained in the current beam. Refinements are iterated until one or more rules are found that cover only positive examples, and the best rule among those is added to the current theory.

2.10.2 PRISM

PRISM (Cendrowska, 1987) was the first algorithm that used a conventional top-down search without being constrained by a particular, randomly selected pair of positive and negative examples. While this is less efficient than AQ's method, it results in a stable algorithm that does not depend on random choices. Cendrowska (1987) also realized the main advantage of rule sets over decision trees, namely that decision trees are constrained to find a theory with nonoverlapping rules, which can be more complex than a corresponding theory with overlapping rules that could be discovered by PRISM. Although the system has not received wide recognition in the literature, mostly due to its inability to address the problem of noise handling, it can be considered as an important step. A version of the algorithm, with all its practical limitations, is implemented in the WEKA toolbox.

2.10.3 CN2

CN2 (Clark & Niblett, 1989), named after the initials of its inventors, tried to combine ideas from AQ with the then-popular decision tree learning algorithm ID3 (Quinlan, 1983). The key observation was that learning a single rule corresponds to learning a single branch in a decision tree. Thus, Clark and Niblett (1989) proposed to evaluate each possible condition (the first version of CN2 used ID3's information gain) instead of focusing only on conditions that separate a randomly selected pair of positive and negative examples. This is basically the same idea that was also discovered independently in the PRISM learner. However, CN2's contributions extend much further. Most importantly, CN2 was the first rule learning system that recognized the overfitting problem, and proposed first measures to counter it. First,

[9] The beam is called a *star* in AQ's terminology, and the beam width is called the *star size*.

AQ could select a mislabeled negative example as a positive seed example, and therefore be forced to learn a rule that covers this example. CN2's batch selection of conditions is less susceptible to this problem, as it is independent of random selections of particular examples. Second, CN2 included a prepruning method based on a statistical significance test. In particular, it filtered out all rules that were insignificant according to the likelihood ratio statistics described in Sect. 9.2.4. Finally, in a later version, Clark and Boswell (1991) also proposed the use of the Laplace measure as an evaluation heuristic (cf. Sect. 7.3.6). Finally, following AQ's example, CN2 used a beam search, which is less likely to suffer from search myopia than hill-climbing approaches.

Another innovation introduced by CN2 was the ability to handle multiple classes. In the original version of the algorithm, Clark and Niblett (1989) proposed to handle multiple classes just as in ID3, by using information gain as a search heuristic and by picking the majority class among the examples that are covered by the final rule. In effect, this approach learns a decision list, a concept that was previously introduced by Rivest (1987). Clark and Boswell (1991) later suggested an alternative approach, which treats each class as a separate concept, thereby using the examples of this class as the positive examples and all other examples as the negative examples. The main advantage of this approach is that it allows the search for a rule that targets one particular class, instead of optimizing the distribution over all classes, as decision tree algorithms have to do, and as the first version of CN2 did as well. This approach, which they called *unordered*, as opposed to the *ordered* decision list approach, is nowadays better known as one-against-all, and is still widely used (cf. Chap. 10).

Section 2.7 gave a detailed account of CN2 at work. The system found many successors, including *m*FOIL (Džeroski & Bratko, 1992) and ICL (De Raedt & Van Laer, 1995), which upgraded it to first-order logic, or BEXA (Theron & Cloete, 1996), which used a more expressive hypothesis language by including disjunctions of attribute values in the search space.

2.10.4 FOIL

FOIL (First-Order Inductive Learner; Quinlan, 1990) was the first relational learning algorithm that received attention beyond the field of *relational data mining* and *Inductive Logic Programming* (ILP). It basically works like CN2 and PRISM discussed above; it learns a concept with the covering loop and learns individual concepts with a top-down refinement operator.

A minor difference between these algorithms and FOIL is that FOIL does not evaluate the quality of a rule, but its *information gain* heuristic (cf. Sect. 7.5.3) evaluates the improvement of a rule with respect to its predecessor. Thus, the quality measure of the same rule may be different, depending on the order of the conditions in the rule. For this reason, FOIL can only use hill-climbing search, and not a beam search like CN2 does. Another minor difference is the Minimum Description Length (MDL)-based stopping criterion that is employed by FOIL (cf. Sect. 9.3.4).

The main difference, however, is that FOIL allowed the use of first-order background knowledge. Instead of only being able to use tests on single attributes, FOIL could employ tests that compute relations between multiple attributes, and could also introduce new variables in the body of a rule. Unlike most other ILP algorithms, FOIL was not directly built on top of a Prolog-engine (although it could be), but it implemented its own reasoning module in the form of a simple tuple calculus. A *tuple* is a valid instantiation of all variables that are used in the body of a rule. Adding a literal that introduces a new variable results in extending the set of valid tuples with new variables. As a new variable can often be bound to more than one value, one tuple can be extended to several tuples after a new variable is introduced. Effectively, counting the covered tuples amounts to counting the number of possible proofs for each (positive or negative) covered example. We provide a more detailed description of the algorithm along with an example that illustrates the operation with tuples in Sect. 6.5.1.

New variables may be used to formulate new conditions, i.e., adding a literal that extends the current tuples may be a good idea, even if the literal on its own does not help to discriminate between positive and negative examples, and would therefore not receive a high evaluation by conventional evaluation measures. Quinlan (1991) proposed a simple technique for addressing this problem: whenever the information gain is below a certain percentage of the highest achievable gain, all so-called *determinate literals* are added to the body of a rule. These are literals that add a new variable, but have at most one binding for this variable, so that they will not lead to an increase in the number of tuples (cf. also Sect. 5.5.5).

FOIL was very influential, and numerous successor algorithms built upon its ideas (Quinlan & Cameron-Jones, 1995a). Its influence was not confined to ILP itself, but also propositional variants of the algorithm were developed and used (e.g., Mooney, 1995).

2.10.5 RIPPER

The main problem with all above-mentioned algorithms was overfitting. Even algorithms that employed various techniques for overfitting avoidance turned out to be ineffective. For example, Fürnkranz (1994a) showed that FOIL's MDL-based stopping criterion is not effective: in experiments in the king-rook-king chess endgame domain, with a controlled level of noise in the training data, the size of the theory learned by FOIL grows with the number of training examples. This means that the algorithm overfits the noise in the training data. Similar experiments with CN2 showed the same weakness.

RIPPER (Repeated Incremental Pruning to Produce Error Reduction; Cohen, 1995) was the first rule learning system that effectively countered the overfitting problem. It is based on several previous works, most notably the *incremental reduced error pruning* idea that we describe in detail in Sect. 9.4.1, and added various ideas of its own (cf. also Sect. 9.4.3). Most notably, Cohen (1995) proposed

an interesting post-processing phase for optimizing a rule set in the context of other rules. The key idea is to remove one rule out of a previously learned rule set and try to relearn it not only in the context of previous rules (as would be the case in the regular covering rule), but also in the context of subsequent rules. The resulting algorithm, RIPPER, has been shown to be competitive with C4.5RULES (Quinlan, 1993) without losing I-REP's efficiency (Cohen, 1995).

All in all, RIPPER provided a powerful rule learning system that was used in several practical applications. In fact, RIPPER is still state of the art in inductive rule learning. An efficient C implementation of the algorithm can be obtained from its author, and an alternative implementation named JRIP can be found in the WEKA data mining library (Witten & Frank, 2005). In several independent studies, RIPPER and JRIP have proved to be among the most competitive rule learning algorithms available today. Many authors have also tried to improve RIPPER. A particularly interesting approach is the FURIA algorithm, which incorporates some ideas from fuzzy rule induction into RIPPER (Hühn & Hüllermeier, 2009b).

2.10.6 PROGOL

PROGOL (Muggleton, 1995), another ILP system, was interesting for several reasons. First, it adopted ideas of the AQ algorithm to inductive logic programming. Like AQ, it selects a seed example and computes a minimal generalization of the example with respect to the available background knowledge and a given maximum inference depth. The resulting *bottom rule* is then in a similar way used for constraining the search space for a subsequent top-down search for the best generalization.

Second, unlike AQ, PROGOL does not use a heuristic search for the best generalization of the seed example, but it uses a variant of best-first A^* search in order to find the best possible generalization of the given seed example. This proves to be feasible because of the restrictions of the search space that are imposed by the randomly selected seed example.

PROGOL proved very successful in various applications (Bratko & Muggleton, 1995), some of which were published in scientific journals of their respective application areas (Cootes, Muggleton, & Sternberg, 2003; King et al., 2004; Sternberg & Muggleton, 2003). An efficient implementation of the algorithm (Muggleton & Firth, 2001) is available from http://wwwhomes.doc.ic.ac.uk/~shm/Software/.

2.10.7 ALEPH

To our knowledge, the most widely used ILP system is ALEPH (A Learning Engine for Proposing Hypotheses). Unfortunately, there are no descriptions of this system in the formal literature, and references have been restricted to an extensive user manual

(Srinivasan, 1999). Some of ALEPH's popularity seems to be due to two aspects: that it is in just one physical file,[10] and that it is written in Prolog. Except for that, there is, in fact, one other aspect of the program that makes it different to almost any other ILP system. ALEPH was conceived as a workbench for implementing and testing—under one umbrella—concepts and procedures from a variety of different ILP systems and papers. It is possibly the only ILP system that can construct rules, trees, constraints and features; invent abnormality predicates; perform classification, regression, clustering, and association-rule mining; allow changes in search strategy (general-to-specific (top-down), specific-to-general (bottom-up), bidirectional, and so on); search algorithm (hill-climbing, exhaustive search, stochastic search, etc.); and evaluation functions. It even allows the user to specify their own search procedure, proof strategies, and visualization of hypotheses.

Given this plethora of options, it is not surprising that each user has her own preferred way to use this program, and we will not attempt to prescribe one over the other. It would, however, be amiss if we did not mention two uses of ALEPH that seem to be most popular in the ILP literature.

In the first, ALEPH is configured to identify a set of classification rules, using a randomized version of the covering algorithm. Individual rules in the set are identified using a general-to-specific, compression-based heuristic search guided by most-specific bottom clauses. This is sufficiently similar to the procedure followed by PROGOL (Muggleton, 1995) to be of interest to ILP practioners seeking a Prolog-based approximation to that popular system.

Another interesting use of ALEPH is to identify Boolean features (Joshi, Ramakrishnan, & Srinivasan, 2008; Ramakrishnan, Joshi, Balakrishnan, & Srinivasan, 2008; Specia, Srinivasan, Joshi, Ramakrishnan, & das Graças Volpe Nunes, 2009), that are subsequently used by a propositional learner to construct models (in the cases cited, this learner was a support vector machine). The key idea is to construct bottom clauses that cover individual examples, and to use the resulting rules as features in a subsequent induction phase. The quality of a rule or feature is defined via some syntactic and semantic constraints (such as minimum recall and precision). In this way, ALEPH may also be viewed as a system for first-order subgroup discovery (cf. Chap. 11).

This specific use of rules constructed by an ILP system as Boolean features appears to have been demonstrated first in (Srinivasan & King, 1997). It has since proved to be an extremely useful way to employ an ILP system (Kramer, Lavrač, & Flach, 2001). The idea of constructing propositional representations from first-order ones goes back at least to 1990, with the LINUS system (Lavrač & Džeroski, 1994a; Lavrač, Džeroski, & Grobelnik, 1991), and perhaps even earlier to work in the mid-1980s done by Michalski and coworkers. We will describe such approaches in more detail in Chap. 5.

[10] http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/

2.10.8 Opus

Opus (Optimized Pruning for Unordered Search; Webb, 1995) was the first rule learning system that demonstrated the feasibility of a full exhaustive search through all possible rule bodies for finding a rule that maximizes a given quality criterion (or heuristic function). The key idea is the use of *ordered search* that prevents a rule being generated multiple times. This means that even though there are $l!$ different orders of the conditions of a rule of length l, only one of them can be taken by the learner for finding this rule. In addition, Opus uses several techniques that allow it to prune significant parts of the search space, so that this search method becomes feasible. Follow-up work (Webb, 2000; Webb & Zhang, 2005) has shown that this technique is also an efficient alternative for association rule discovery, provided that the database to mine fits into the memory of the learning system.

2.10.9 CBA

Exhaustive search is also used by association rule discovery algorithms like Apriori or its successors (cf. Sect. 6.3.2). The key difference is the search technique employed. Contrary to Opus, association rule learners typically employ a memory-intensive level-wise breadth-first search. They return not a single best rule, but all rules with a given minimum support and a given minimum confidence.

In general, association rules can have arbitrary combinations of features in the head of a rule. However, the head can also be constrained to hold only features related to the class rule. In this case, an association rule algorithm can be used to discover all classification rules with a given minimum support and confidence. The learner's task is then to combine a subset of these rules into a final theory.

One of the first and best-known algorithms that employed this principle for learning predictive rules is CBA (Liu, Hsu, & Ma, 1998; Liu, Ma, & Wong, 2000). In its simplest version, the algorithm selects the final rule sets by sorting all classification rules according to confidence and incrementally adding rules to the final set until all examples are covered or the quality of the rule set decreases.

There is a variety of variations of this basic algorithm that have been discussed in the literature (e.g., Bayardo Jr., 1997; Jovanoski & Lavrač, 2001; Li, Han, & Pei, 2001; Yin & Han, 2003). Mutter, Hall, and Frank (2004) performed an empirical comparison between some of them. A good survey of pattern-based classification algorithms can be found in (Bringmann, Nijssen, & Zimmermann, 2009).

2.11 Conclusion

This chapter provided a gentle introduction to rule learning, mainly focusing on supervised learning of predictive rules. In this setting, the goal is to learn understandable models for predictive induction. We have seen a broad overview of the main components that are common to all rule learning algorithms, such as the covering loop for learning rule sets or decision lists, the top-down search for single rules, evaluation criteria for rules, and techniques for fighting overfitting. In the following chapters, we will return to these issues and discuss them in more detail.

However, other techniques for generating rule sets are possible. For example, rules can be generated from induced decision trees. As we have seen in Chap. 1, nodes in a decision tree stand for attributes or conditions on attributes, arcs stand for values of attributes or outcomes of those conditions, and leaves assign classes. A decision list like the one in Fig. 2.4b can be seen as a right-branching decision tree. Conversely, each path from the root to a leaf in a decision tree can be seen as a rule.

Standard algorithms for learning decision trees (such as C4.5) are quite similar to the CN2 algorithm for learning decision lists in that the aim of extending a decision tree with another split is to reduce the class impurity in the leaves (usually measured by entropy). However, as discussed in Sect. 1.5, rule sets are often more compact than decision trees. Consequently, a rule set can be considerably simplified during the conversion of a decision tree to a set of rules (Quinlan, 1987a, 1993). For example, Frank and Witten (1998) suggested the PART algorithm, which tries to integrate this simplification into the tree induction process by focusing only on a single branch of a tree.