Chapter 15
Classifiers in the Form of Rulesets

Some classifiers take the form of so-called *if-then* rules: if the conditions from the *if*-part are satisfied, the example is labeled with the class specified in the *then*-part. Typically, the classifier is represented not by a single rule, but by a set of rules, a *ruleset*. The paradigm has certain advantages. For one thing, the rules capture the underlying logic, and therefore facilitate explanations of why an example has to be labeled with the given class; for another, induction of rulesets is capable of discovering recursive definitions, something that is difficult to accomplish within other machine-learning paradigms.

In our search for techniques that induce rules or rulesets from data, we will rely on ideas borrowed from *Inductive Logic Programming*, a discipline that studies methods for automated creation and improvement of *Prolog* programs. Here, however, we are interested only in classifier induction.

15.1 A Class Described By Rules

To prepare the ground for simple rule-induction algorithms to be presented later, let us take a look at the nature of the rules we will want to use. After this, we will introduce some relevant terminology and define the specific machine-learning task.

The Essence of Rules Table 15.1 contains the training set of the "pies" domain we have encountered earlier. In Chap. 1, the following expression was given as one possible description of the positive class:

```
[ (shape=circle) AND (filling-shade=dark) ] OR
[ NOT(shape=circle) AND (crust-shade=dark) ]
```

When classifying example **x**, the classifier compares the example's attribute values with those in the expression. Thus if **x** is `circular` and its `filling-shade` happens to be `dark`, the expression is *true*, and the classifier therefore labels **x** with

M. Kubat, *An Introduction to Machine Learning*,
DOI 10.1007/978-3-319-63913-0_15

Table 15.1 Twelve training examples expressed in a matrix form

Example	Shape	Crust		Filling		Class
		Size	Shade	Size	Shade	
ex1	Circle	Thick	Gray	Thick	Dark	pos
ex2	Circle	Thick	White	Thick	Dark	pos
ex3	Triangle	Thick	Dark	Thick	Gray	pos
ex4	Circle	Thin	White	Thin	Dark	pos
ex5	Square	Thick	Dark	Thin	White	pos
ex6	Circle	Thick	White	Thin	Dark	pos
ex7	Circle	Thick	Gray	Thick	White	neg
ex8	Square	Thick	White	Thick	Gray	neg
ex9	Triangle	Thin	Gray	Thin	Dark	neg
ex10	Circle	Thick	Dark	Thick	White	neg
ex11	Square	Thick	White	Thick	Dark	neg
ex12	Triangle	Thick	White	Thick	Gray	neg

the positive class. If the expression is *false*, the classifier labels the example with the negative class. Importantly, the expression can be converted into the following two *rules*:

R1: *if* `[ (shape=circle) AND (filling-shade=dark) ]` *then* **pos**.
R2: *if* `[ NOT(shape=circle) AND (crust-shade=dark) ]` *then* **pos**.
else **neg**.

In the terminology of machine learning, each rule consists of an *antecedent* (the *if*-part), which in this context is a conjunction of attribute values, and a *consequent* (the *then*-part) which points to a concrete class label.

Note that the consequents of both rules indicate the positive class. For an example to be labeled as positive, it is necessary that the conditions in the antecedent of at least one rule be satisfied. Otherwise the classifier will label the example with the default class which, in this case, is **neg**. We will remember that when working with rulesets in domains of this kind, one must not forget to specify the default class.

Simplifying Assumptions Throughout this chapter, we will rely on the following simplifying assumptions:

1. All training examples are described by discrete-valued attributes.
2. The training set is noise-free.
3. The training set is consistent: examples described by the same attribute vectors must belong to the same class.

The Machine-Learning Task Our goal is an algorithm for the induction of rulesets from data that satisfy the simplifying assumptions from the previous paragraph. We will limit ourselves to rules whose consequents point to the positive class, the default always being the negative class.

Since the training set is supposed to be consistent and noise-free, we will be interested in classifiers that correctly classify all training examples. This means that

for each positive example, the antecedent of at least one rule will be *true*. For any negative example, no rule's antecedent is *true*, and the example is labeled with the default (negative) class.

A Rule "Covers" An Example Let us introduce one useful term: an example either is or is not *covered* by a rule. A simple illustration will clarify the notion. Consider the following rule:

R: *if* `(shape=circle)` *then* **pos**.

If we apply this rule to the examples from Table 15.1, we will observe that the antecedent's condition, `shape=circle`, is satisfied by the following set of examples: $\{\mathbf{ex}_1, \mathbf{ex}_2, \mathbf{ex}_4, \mathbf{ex}_6, \mathbf{ex}_7, \mathbf{ex}_0\}$. We will say that **R** *covers* these six examples. Generally speaking, a rule covers an example if the expression in the rule's antecedent is *true* for this example. Note that four of the examples covered by this particular rule are positive and two are negative.

Rule Specialization Suppose we modify the above rule by adding to its antecedent another condition, `filling-shade=dark`, obtaining the following:

R1: *if* `(shape=circle) AND (filling-shade=dark)` *then* **pos**

Checking **R1** against the training set, we realize that it covers the following examples: $\{\mathbf{ex}_1, \mathbf{ex}_2, \mathbf{ex}_4, \mathbf{ex}_6\}$. We observe that this is a subset of the six examples originally covered by **R**. Conveniently, only positive (and no negative) examples are now covered.

This leads us to the definition of another useful term. If a modification of a rule's antecedent reduces the set of covered examples to a subset, we say that the modification has *specialized* the rule. In other words, specialization narrows the set of covered examples to a proper subset. A typical way of specializing a rule is to add a new condition to the rule's antecedent.

Rule Generalization Conversely, a rule is *generalized* if its modification enlarges the set of covered examples to a superset—if the new version covers all examples that were covered by the previous version, plus some additional ones. The easiest way to generalize a rule is by removing a condition from its antecedent. For instance, this happens when we drop from rule **R1** the condition `(filling-shade=dark)`.

Specialization and Generalization of Rulesets We have said we are interested in induction of rulesets that label an example with the positive class if the antecedent of at least one rule is *true* for the example. For instance, this is the case of the ruleset consisting of the rules **R1** and **R2** above.

If we remove one rule from a ruleset, the ruleset may no longer cover some of the previously covered examples. This, we already know, is called specialization. Conversely, adding a new rule to the ruleset will generalize the ruleset because the new rule will add to the set of covered examples.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Explain the nature of rule-based classifiers. What do we mean when we say that a rule *covers* an example? Using this term (*cover*), specify how the induced classifier should behave on a consistent and noise-free training set.
- Define the terms *generalization* and *specialization*. How will you specialize or generalize a rule? How will you specialize or generalize a ruleset?
- List the simplifying assumptions to be used throughout this chapter.

15.2 Inducing Rulesets by Sequential Covering

Let us now introduce a simple technique that induces rulesets from training data satisfying the simplifying assumptions from the previous section.

The Principle The goal is to find a ruleset such that each of its rules covers some positive examples, but no negative examples. Together, the rules should cover all positive examples and no negative ones. The procedure we will use creates one rule at a time, always starting with a very general initial version (covering also negative examples) that is then gradually specialized until all negative examples are excluded from coverage. The circumstance that the rules are created sequentially, and that each is supposed to cover those positive examples that were missed by previous rules, gives the technique its name: *sequential covering*.

Baseline Version of Sequential Covering Table 15.2 provides the pseudocode of a simple method for induction of rulesets. The main body contains the sequential covering algorithm. The idea is to find a rule that covers some positive examples, but no negative examples. Once the rule has been created, the examples it covers are removed from the training set. If no positive examples remain, the algorithm stops; otherwise, the algorithm is applied to the reduced training set.

The lower part describes induction of a single rule. The algorithm starts with the most general version of the antecedent that says, "all examples are positive." Assuming that the training set contains at least one negative example, this statement is obviously incorrect. The algorithm therefore seeks to rectify the situation by specialization, trying to exclude from coverage some negative examples, hopefully without losing the coverage of the positive examples. The specialization operator adds to the rule another conjunct in the form, $a_i = v_j$ (read: the value of attribute a_i is v_j).

A Concrete Example Let us "hand-simulate" the sequential-covering algorithm using the data from Table 15.1. The first rule, with the empty antecedent, covers all training examples. Adding to the empty antecedent the condition `shape=circle`

Table 15.2 The sequential covering algorithm

Input: training set T.

Sequential covering.

Create an empty ruleset.

While at least one positive example remains in T:

1. Create a rule using the algorithm below.
2. Remove from T all examples that satisfy the rule's antecedent.
3. Add the rule to the ruleset.

Create a single rule

Create an initial version of the rule, **R**: *if* () *then* **pos**

1. If **R** does not cover any negative example, stop.
2. Add to **R**'s antecedent a condition, $a_i = v_j$, and return to the previous step.

results in a rule that covers four positive and two negative examples. Adding one more condition, `filling-shade=dark`, specializes the rule so that, while still covering the four positive examples, it now no longer covers any negative example. We have obtained a rule that covers examples $\{\mathbf{ex}_1, \mathbf{ex}_2, \mathbf{ex}_4, \mathbf{ex}_6\}$. Note that is the rule **R1** from the previous section.

If we remove these four examples from the training set, we are left with only two positive examples, $\mathbf{ex}_3$ and $\mathbf{ex}_5$. The development of another rule again starts from the most general version (empty antecedent). Suppose that we then choose `shape=triangle` as the initial condition. This covers one positive and two negative examples. Adding to the antecedent the term `filling-shade=dark`, we succeed in excluding the negative examples while retaining the coverage of the positive example $\mathtt{ex}_3$, which can now be removed from the training set. After the creation of this second rule, we are left with one positive example $\mathtt{ex}_5$.

We therefore have to create yet another rule whose task will be to cover $\mathtt{ex}_5$ without covering any negative example. Once we find such rule, $\mathtt{ex}_5$ is removed from the training set. After this, we observe that there are no positive examples left, and the procedure can stop. We have created a ruleset consisting of three rules that cover all positive examples and no negative examples.

How to Identify the Best Attribute-Value Pair In the previous example, we always chose the condition to be added to the rule's antecedent more or less at random. But seeing that we could have selected it from quite a few alternatives, we realize that we need a mechanism capable of informing us about the quality of each choice. Perhaps the most natural criterion to be used here is based on information theory, a principle we have encountered in Chap. 6 where we used it in the course of induction of decision trees.

Let N^+_{old} be the number of positive examples covered by the original version of the rule, and let N^-_{old} be number of negative examples covered by the original version of the rule. Likewise, the numbers of positive and negative examples covered by the new version of the rule will be denoted by N^+_{new} and N^-_{new}, respectively.

Since the rule covers only positive examples, the information content of the message that a randomly picked example is labeled by it as positive is calculated as follows (for the old version and for the new version):

$$I_{old} = -\log\left(\frac{N^+_{old}}{N^+_{old} + N^-_{old}}\right)$$

$$I_{new} = -\log\left(\frac{N^+_{new}}{N^+_{new} + N^-_{new}}\right)$$

The difference between these two is the amount of information that has been gained by modifying the rule. Usually, machine-learning professionals normalize the information gain by the number, N_C, of covered examples so as to give preference rule modifications that optimize the number of covered examples. The quality of the rule-improvement is then calculated as follows:

$$Q = N_C \times |I_{new} - I_{old}| \tag{15.1}$$

When comparing alternative ways of modifying a rule, we choose the one with the highest value of Q.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Summarize the principle of the *sequential covering* algorithm.
- Explain the mechanism of gradual specialization of a rule. What do we want to accomplish by this specialization?
- How will you use information gain when looking for the most promising way of specializing a rule?

15.3 Predicates and Recursion

The *sequential covering* algorithm has a much broader scope of applications than the previous section seems to have indicated. Perhaps most importantly, the technique can be employed for induction of concepts expressed in predicate calculus.

Predicates: Greater Expressive Power Than Attributes A serious limitation of attribute-value logic is that it is not sufficiently flexible to capture certain relations among data. For instance, the fact that **y** is located between **x** and **z** can be stated using the predicate `between(x,y,z)`—more accurately, the predicate is the term "`between`," whereas "`(x,y,z)`" is a list of the predicate's arguments.

The reader will agree that trying to express the same relation by means of attributes and their values would be difficult to say the least. An attribute can be seen as a special case of a one-argument predicate. For instance, the fact that, for a given example, `x`, the shape is `circular` can be written as `circular(x)`. But the analogy is no longer as obvious in the case of predicates with more arguments.

Induction of Rules in Predicate Calculus Here is an example of a rule that says that if `x` is a parent of `y`, and at the same time `x` is a woman, then this parent is actually `y`'s mother:

if `parent(x,y)` AND `female(x)` *then* `mother(x,y)`

We can see that this rule has the same structure as the rules **R1** and **R2** we have seen above: a list of conditions in the antecedent followed by a consequent. And indeed, the same sequential covering algorithm can be used here. There is one difference, though. When choosing among candidate predicates to be added to antecedent, we must not forget that the meaning of the predicate changes if we change the arguments. For instance, the previous rule's meaning will change if we replace `parent(x,y)` with `parent(x,z)` because, in this case, the fact that `x` is a parent of `z` surely does not guarantee that `x` is mother of some other subject, `y`.

Rulesets Allow Recursive Definitions The rules can be more interesting than the toy domain from Table 15.1 might lead us to believe. For one thing, they can be recursive—which is the case of the following two rules defining the term `ancestor`.

if `parent(x,y)` *then* `ancestor(x,y)`.
if `parent(x,z)` AND `ancestor(z,y)` *then* `ancestor(x,y)`.

The meaning of two rules is easy to see. Ancestor is a parent, or at least the parent's ancestor. For instance, a grandparent is the parent of a parent—and therefore an ancestor.

A Concrete Example of Induction Let us illustrate induction of rulesets using the problem from Table 15.3. Here, two concepts (classes), `parent` and `ancestor`, are characterized by a list of positive examples under the assumption that any example that is not in this list should be regarded as a negative example. Our goal is to induce the definition of `ancestor`, using the predicate `parent`.

We begin with the most-general rule, *if* () *then* `ancestor(x,y)`. In the next step, we want to add a condition to the antecedent. To this end, we may consider various possibilities, but the simplest appears to be `parent(x,y)`—which will also be supported by the information-gain criterion. We have obtained the following rule:

Table 15.3 Illustration of induction from examples described using predicate logic

Consider the knowledge base consisting of the following positive examples of classes `parent` and `ancestor`, defined using prolog-like facts (any other example will be regarded as negative).

```
parent(eve,ted)      ancestor(eve,ted)ancestor(eve,ivy)
parent(tom,ted)      ancestor(tom,ted)ancestor(eve,ann)
parent(tom,liz)      ancestor(tom,ted)ancestor(eve,jim)
parent(ted,ivy)      ancestor(tom,ted)ancestor(tim,ivy)
parent(ted,ann)      ancestor(tom,ted)ancestor(eve,ann)
parent(ann,jim)      ancestor(tom,ted)ancestor(eve,jim)
                                      ancestor(ted,jim)
```

From the above examples, the algorithm creates the following first version of the rule. Note that this rule does not cover any negative examples.

R3: *if* `parent(x,y)` *then* `ancestor(x,y)`

Removing all positive examples of this rule, the following set of positive examples of `ancestor(x,y)` remains:

```
ancestor(eve,ivy)
ancestor(eve,ann)
ancestor(eve,jim)
ancestor(tim,ivy)
ancestor(eve,ann)
ancestor(eve,jim)
```

To cover these, another rule is created:

if `parent(x,z)` *then* `ancestor(x,y)`

After specialization, the second rule is turned into following:

R4: *if* `parent(x,z)` AND `ancestor(z,y)` *then* `ancestor(x,y)`

These two rules **R3** and **R4** now cover all positive examples and no negative examples.

R3: *if* `parent(x,y)` *then* `ancestor(x,y)`

Observing that the rule covers only positive examples and no negative examples, we realize there is no need to specialize it.

However, the rule covers only the `ancestor` examples from the middle column, and no examples from the rightmost column. Obviously, we need at least one more rule. When considering the conditions to be added to the empty antecedent of the

next rule, we may consider the following (note that this is always the same predicate, but each time with a different set of arguments):

```
parent(x,z)
parent(z,y)
```

Suppose that the first leads to higher information gain. Seeing that the rule still covers some negative examples, we want to specialize it by adding another condition to its antecedent. Seeing that the `parent` predicate does not lead us anywhere, we try the predicate `ancestor`, again with various lists of arguments. Evaluating the information gain of all alternatives, we realize that the best option is `ancestor(z,y)`. This is how we obtain the second rule:

R4: *if* `parent(x,z)` AND `ancestor(z,y)` *then* `ancestor(x,y)`.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- How can a class be expressed using predicates? In what sense is the language of predicates richer than the language of attributes?
- Give an example of a recursively defined class. Can you think of a different example than `ancestor`?

15.4 More Advanced Search Operators

The technique described in the previous sections followed a simple strategy: in its attempts to find a good ruleset, the algorithm always sought to modify the rule(s) by specialization and generalization, evaluating alternative options by the information-gain criterion.

Operators for Ruleset Modification In reality, the search for rules can be more flexible than that. Other ruleset-modifying operators have been suggested. These, as we will see, do not necessarily represent specialization or generalization, but if we take a look at them, we realize they make sense. Let us mention in passing that these operators have been derived with the help of a well-known principle from logic, so-called *inverse resolution*. For our specific needs, however, the method of their derivation is unimportant.

In the following, we will simplify the formalism by writing a comma instead of AND, and using an arrow instead of the *if-then* construct. In all of the four cases, the operator converts the ruleset on the left into the ruleset on the right. The leftmost column gives the traditional names of these operators.

– identification: $\left\{\begin{array}{l} b, x \to a \\ b, c, d \to a \end{array}\right\} \Rightarrow \left\{\begin{array}{l} b, x \to a \\ c, d \to x \end{array}\right\}$

– absorption: $\left\{\begin{array}{l} c, d \to x \\ b, c, d \to a \end{array}\right\} \Rightarrow \left\{\begin{array}{l} c, d \to x \\ b, x \to a \end{array}\right\}$

– inter-construction: $\left\{\begin{array}{l} v, b, c \to a \\ w, b, c \to a \end{array}\right\} \Rightarrow \left\{\begin{array}{l} u, b, c \to a \\ v \to u \\ w \to u \end{array}\right\}$

– intra-construction: $\left\{\begin{array}{l} v, b, c \to a \\ w, b, c \to a \end{array}\right\} \Rightarrow \left\{\begin{array}{l} v, u \to a \\ w, u \to a \\ b, c \to u \end{array}\right\}$

Note that these replacements are not deductive: the rules on the right are never perfectly equivalent to those on the left. And yet, they do appear to make sense intuitively.

How to Improve Existing Rulesets? The operators from the previous paragraph can be used to improve rulesets that have been induced by the sequential covering algorithm. We can even consider a situation where not one, but several different classes were induced, which gave rise to several rulesets.

These rulesets can then be improved applying the hill-climbing search technique. The search operators are those listed in the previous paragraph. The evaluation function may give preference to more compact rules that classify correctly some auxiliary set of training examples meant to represent a concrete application domain.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- List the ruleset-modifying operators listing in this section. Which field of logic has helped derive them?
- Suggest how you might use these rules in an attempt to improve a given ruleset?

15.5 Summary and Historical Remarks

- Some classifiers have the form of rules. A rule consists of an antecedent (a list of conjuncted conditions) and a consequent (a class label). If the rule's antecedent is *true*, for the given example, then the example is labeled with the label pointed to by the consequent.

- If a rule's antecedent is *true*, for an example, we say that the rule *covers* the example.
- In the course of rule induction, we often rely on *specialization*. This reduces the set of covered examples to its subset. A rule is specialized if we add a condition to its antecedent. Conversely, *generalization* enlarges the set of covered examples to its superset.
- Usually, we induce a set of rules, a *ruleset*. The classifier then labels an example as positive if the antecedent of at least one of the rules is *true*. Adding a rule to a ruleset represents generalization. Removing a rule would represent specialization.
- The chapter introduced a simple algorithm for induction of rulesets from noise-free and consistent training data described by discrete attributes. The algorithm can so to some degree be optimized with the help of a criterion derived from information theory.
- The same algorithm can be used for induction of rules in domains where the examples are described using *predicate* calculus. Even recursive rules can thus be discovered.
- Some other "search operators" have been developed by the field of *inverse resolution*. They do not necessarily represent specialization or deduction.

Historical Remarks Induction of rules belongs to the oldest tasks of machine learning since the days when this discipline was seen as a means of inducing knowledge artificial-intelligence systems. The sequential-covering algorithm is a simplified version of an algorithm by Clark and Niblett [16]. Its use for induction of predicate-based rule was inspired by the FOIL algorithm developed by Quinlan [77]. The additional operators from Sect. 15.4 are based on the operators introduced by Muggleton and Buntine [70] in the framework of their work on *inverse resolution*.

15.6 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

Exercises

1. Hand-simulate the algorithm of *sequential covering* for the data from Table 15.1. Ignoring information gain, indicate how the first rule is created if we start from `crust-shade=gray`.

2. Show that, when we choose different ways of specializing a rule (adding different attribute-value pairs), we in the end obtain a different ruleset, often of a different size.

Give It Some Thought

1. Think of some other examples of classes (different from those discussed in this chapter) that are best defined recursively.
2. Think about how classes that are by nature recursive would be difficult to address in the framework of attribute-value logic. Demonstrate the superior power of the predicate calculus.
3. Suggest a learning procedure for "knowledge refinement." In this task, we assume that certain classes have already been defined in predicate calculus. When presented with another set of examples, the knowledge-refinement technique seeks to optimize the existing rules, either my making them more compact, or by making them more accurate in the presence of noise.

Computer Assignments

1. Write a computer program that implements the *sequential covering* algorithm. Use some simple criterion (not necessarily information gain) to choose which condition to add to a rule's antecedent.
2. In the UCI repository, find a domain satisfying the criteria specified in Sect. 15.1. Apply to it the program developed in the previous step.
3. How would you represent two-argument or three-argument predicates if you wanted to implement your machine-learning program in *C++*, *Java*? or some other programming language of a similar nature?
4. Write a program that applies the *sequential covering* algorithm to examples described in predicate calculus.