CHAPTER 6

Rule models

RULE MODELS ARE the second major type of logical machine learning models. Generally speaking, they offer more flexibility than tree models: for instance, while decision tree branches are mutually exclusive, the potential overlap of rules may give additional information. This flexibility comes at a price, however: while it is very tempting to view a rule as a single, independent piece of information, this is often not adequate because of the way the rules are learned. Particularly in supervised learning, a rule model is more than just a set of rules: the specification of how the rules are to be combined to form predictions is a crucial part of the model.

There are essentially two approaches to supervised rule learning. One is inspired by decision tree learning: find a combination of literals – the *body* of the rule, which is what we previously called a concept – that covers a sufficiently homogeneous set of examples, and find a label to put in the *head* of the rule. The second approach goes in the opposite direction: first select a class you want to learn, and then find rule bodies that cover (large subsets of) the examples of that class. The first approach naturally leads to a model consisting of an ordered sequence of rules – a *rule list* – as will be discussed in Section 6.1. The second approach treats collections of rules as unordered *rule sets* and is the topic of Section 6.2. We shall see how these models differ in the way they handle rule overlap. The third section of the chapter covers discovery of subgroups and association rules.

https://doi.org/10.1017/CBO9780511973000.008 Published online by Cambridge University Press

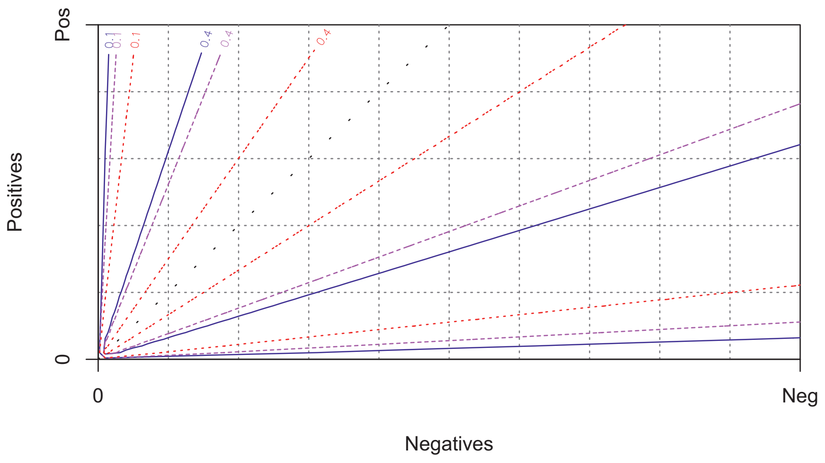


Figure 6.1. ROC isometrics for entropy (rescaled to have a maximum value of 1/2), Gini index and minority class. The grey dotted symmetry line is defined by $\dot{p} = 1/2$: each isometric has two parts, one above the symmetry line (where impurity decreases with increasing empirical probability $\dot{p}$) and its mirror image below the symmetry line (where impurity is proportional to $\dot{p}$). If these impurity measures are used as search heuristic, as they are in rule learning, only the shape of the isometrics matters but not the associated impurity values, and hence all three impurity measures are equivalent.

6.1 Learning ordered rule lists

The key idea of this kind of rule learning algorithm is to keep growing a conjunctive rule body by adding the literal that most improves its homogeneity. That is, we construct a downward path through the hypothesis space, of the kind discussed in Section 4.2, and we stop as soon as some homogeneity criterion is satisfied. It is natural to measure homogeneity in terms of purity, as we did with decision trees. You might think that adding a literal to a rule body is much the same as adding a binary split to a decision tree, as the added literal splits the instances covered by the original rule body in two groups: those instances for which the new literal is true, and those for which the new literal is false. However, one key difference is that in decision tree learning we are interested in the purity of *both* children, which is why we use the weighted average impurity as our search heuristic when constructing the tree. In rule learning, on the other hand, we are only interested in the purity of one of the children: the one in which the added literal is true. It follows that we can directly use any of the impurity measures we considered in the previous chapter (see Figure 5.2 on p.134 if you want to remind yourself which they are), without the need for averaging.

In fact, it doesn't even matter which of those impurity measures we use to guide the search, since they will all give the same result. To see this, notice that the impurity of a concept decreases with the empirical probability $\dot{p}$ (the relative frequency of covered positives) if $\dot{p} > 1/2$ and increases with $\dot{p}$ if $\dot{p} < 1/2$; see Figure 6.1. Whether this increase or decrease is linear or not matters if we are averaging the impurities of several concepts, as in decision tree learning, but not if we are evaluating single concepts. In other words, the difference between these impurity measures vanishes in rule learning, and we might as well take the proportion of the minority class $\min(\dot{p}, 1-\dot{p})$ (or, if you prefer, $1/2 - |\dot{p} - 1/2|$), which is arguably the simplest, as our impurity measure of choice in this section. Just keep in mind that if other authors use entropy or Gini index to compare the impurity of literals or rule bodies this will give the same results (not in terms of impurity values but in terms of which one is best).

We introduce the main algorithm for learning rule lists by means of an example.

Example 6.1 (Learning a rule list). Consider again our small dolphins data set with positive examples

p1: Length = 3 ∧ Gills = no ∧ Beak = yes ∧ Teeth = many
p2: Length = 4 ∧ Gills = no ∧ Beak = yes ∧ Teeth = many
p3: Length = 3 ∧ Gills = no ∧ Beak = yes ∧ Teeth = few
p4: Length = 5 ∧ Gills = no ∧ Beak = yes ∧ Teeth = many
p5: Length = 5 ∧ Gills = no ∧ Beak = yes ∧ Teeth = few

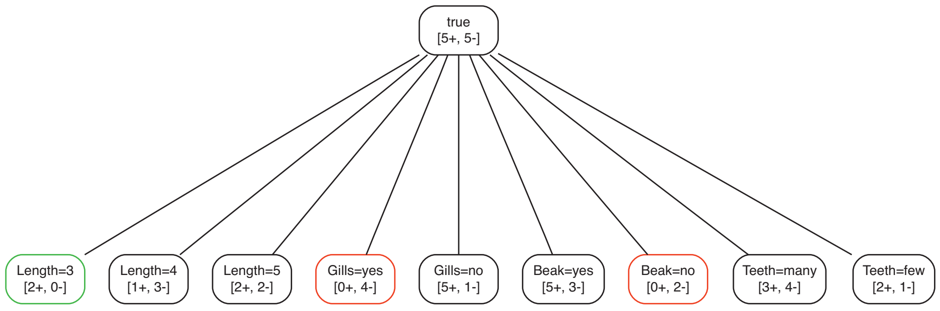
and negatives

n1: Length = 5 ∧ Gills = yes ∧ Beak = yes ∧ Teeth = many
n2: Length = 4 ∧ Gills = yes ∧ Beak = yes ∧ Teeth = many
n3: Length = 5 ∧ Gills = yes ∧ Beak = no ∧ Teeth = many
n4: Length = 4 ∧ Gills = yes ∧ Beak = no ∧ Teeth = many
n5: Length = 4 ∧ Gills = no ∧ Beak = yes ∧ Teeth = few

The nine possible literals are shown with their coverage counts in Figure 6.2 (top). Three of these are pure; in the impurity isometrics plot in Figure 6.2 (bottom) they end up on the x-axis and y-axis. One of the literals covers two positives and two negatives, and therefore has the same impurity as the overall data set; this literal ends up on the ascending diagonal in the coverage plot.

Although impurity in itself does not distinguish between pure literals (we will return to this point later), one could argue that Gills = yes is the best of the three as it covers more examples, so let's formulate our first rule as:

·**if** Gills = yes **then** Class = ⊖·



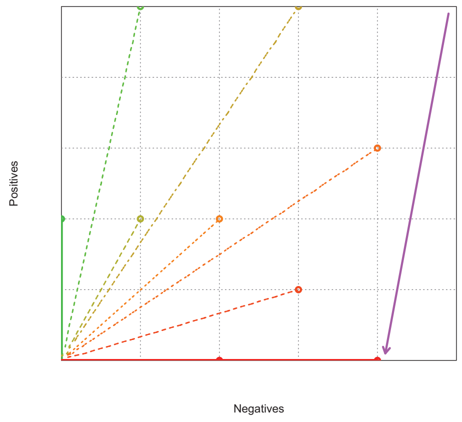


Figure 6.2. **(top)** All literals with their coverage counts on the data in Example 6.1. The ones in green (red) are pure for the positive (negative) class. **(bottom)** The nine literals plotted as points in coverage space, with their impurity values indicated by impurity isometrics (away from the ascending diagonal is better). Impurity values are colour-coded: towards green if $\dot{p} > 1/2$, towards red if $\dot{p} < 1/2$, and orange if $\dot{p} = 1/2$ (on a 45 degree isometric). The violet arrow indicates the selected literal, which excludes all five positives and one negative.

The corresponding coverage point is indicated by the arrow in Figure 6.2 (bottom). You can think of this arrow as the right-most bit of the coverage curve that results if we keep on following a downward path through the hypothesis space by adding literals. In this case we are not interested in following the path further because the concept we found is already pure (we shall see examples later where we have to add several literals before we hit one of the axes). One new thing that we haven't seen before is that this coverage curve lies below the diagonal – this is a consequence of the fact that we haven't fixed the class in advance, and therefore we are just as happy diving deep beneath the ascending

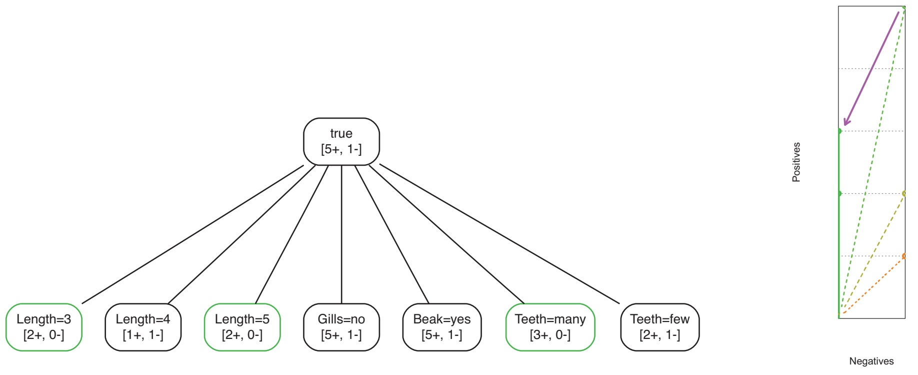


Figure 6.3. (left) Revised coverage counts after removing the four negative examples covered by the first rule found (literals not covering any examples are omitted). **(right)** We are now operating in the right-most 'slice' of Figure 6.2 on p.160.

diagonal as we would be flying high above it. Another way of thinking about this is that if we swap the labels this affects the heads but not the bodies of the learned rules.

Most rule learning algorithms now proceed as follows: they remove the examples covered by the rule just learned from consideration, and proceed with the remaining examples. This strategy is called *separate-and-conquer*, in analogy with the divide-and-conquer strategy of decision trees (the difference is that in separate-and-conquer we end up with one remaining subproblem rather than several as in divide-and-conquer). So we are left with five positive examples and one negative, and we again search for literals with minimum impurity. As is shown in Figure 6.3, we can understand this as working in a smaller coverage space. After going through the numbers, we find the next rule learned is

$$\cdot\textbf{if}\ \mathsf{Teeth} = \mathsf{many}\ \textbf{then}\ \mathsf{Class} = \oplus\cdot$$

As I mentioned earlier, we should be cautious when interpreting this rule on its own, as against the original data set it actually covers more negatives than positives! In other words, the rule implicitly assumes that the previous rule doesn't 'fire'; in the final rule model we will precede it with 'else'.

We are now left with two positives and one negative (Figure 6.4). This time it makes sense to choose the rule that covers the single remaining negative, which is

$$\cdot\textbf{if}\ \mathsf{Length} = 4\ \textbf{then}\ \mathsf{Class} = \ominus\cdot$$

Since the remaining examples are all positive, we can invoke a *default rule* to cover those examples for which all other rules fail. Put together, the learned rule model is

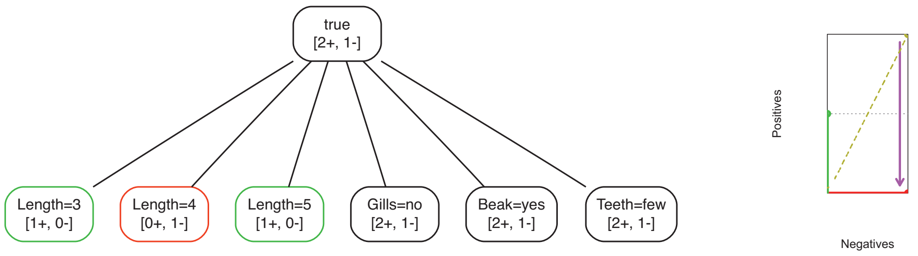


Figure 6.4. (left) The third rule covers the one remaining negative example, so that the remaining positives can be swept up by a default rule. **(right)** This will collapse the coverage space.

then as follows:

$$\cdot\textbf{if } \mathsf{Gills} = \mathsf{yes} \textbf{ then } \mathsf{Class} = \ominus\cdot$$
$$\cdot\textbf{else if } \mathsf{Teeth} = \mathsf{many} \textbf{ then } \mathsf{Class} = \oplus\cdot$$
$$\cdot\textbf{else if } \mathsf{Length} = 4 \textbf{ then } \mathsf{Class} = \ominus\cdot$$
$$\cdot\textbf{else } \mathsf{Class} = \oplus\cdot$$

Organising rules in a list is one way of dealing with overlaps among rules. For example, we know from the data that there are several examples with both Gills = yes and Teeth = many, but the rule list above tells us that the first rule takes precedence in such cases. Alternatively, we could rewrite the rule list such that the rules are mutually exclusive. This is useful because it means that we can use each rule without reference to the other rules, and also ignore their ordering. The only slight complication is that we need negated literals (or internal disjunction) for those features that have more than two values, such as ‘Length’:

$$\cdot\textbf{if } \mathsf{Gills} = \mathsf{yes} \textbf{ then } \mathsf{Class} = \ominus\cdot$$
$$\cdot\textbf{if } \mathsf{Gills} = \mathsf{no} \wedge \mathsf{Teeth} = \mathsf{many} \textbf{ then } \mathsf{Class} = \oplus\cdot$$
$$\cdot\textbf{if } \mathsf{Gills} = \mathsf{no} \wedge \mathsf{Teeth} = \mathsf{few} \wedge \mathsf{Length} = 4 \textbf{ then } \mathsf{Class} = \ominus\cdot$$
$$\cdot\textbf{if } \mathsf{Gills} = \mathsf{no} \wedge \mathsf{Teeth} = \mathsf{few} \wedge \mathsf{Length} \neq 4 \textbf{ then } \mathsf{Class} = \oplus\cdot$$

In this example we rely on the fact that this particular set of rules has a single literal in each rule – in the general case we would need non-conjunctive rule bodies. For example, consider the following rule list:

$$\cdot\textbf{if } P \wedge Q \textbf{ then } \mathsf{Class} = \oplus\cdot$$
$$\cdot\textbf{else if } R \textbf{ then } \mathsf{Class} = \ominus\cdot$$

If we wanted to make these mutually exclusive the second rule would become

$$\cdot\textbf{if } \neg(P \wedge Q) \wedge R \textbf{ then } \mathsf{Class} = \ominus\cdot$$

or equivalently,

$$\cdot\textbf{if } (\neg P \vee \neg Q) \wedge R \textbf{ then } \mathsf{Class} = \ominus\cdot$$

Clearly, making rules mutually exclusive leads to less compact rules, which explains why rule lists are a powerful and popular format.

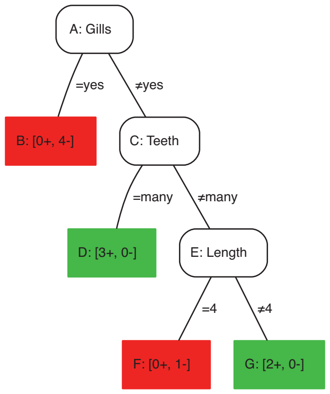
Algorithm 6.1 specifies the separate-and-conquer rule learning strategy in more detail. While there are still training examples left, the algorithm learns another rule and removes all examples covered by the rule from the data set. This algorithm, which is the basis for the majority of rule learning systems, is also called the *covering algorithm*. The algorithm for learning a single rule is given in Algorithm 6.2. Similar to decision trees, it uses the functions Homogeneous(D) and Label(D) to decide whether further specialisation is needed and what class to put in the head of the rule, respectively. It also employs a function BestLiteral(D, L) that selects the best literal to add to the rule from the candidates in L given data D; in our example above, this literal would be selected on purity.

Many variations on these algorithms exist in the literature. The conditions in the while-loops are often relaxed to other *stopping criteria* in order to deal with noisy data. For example, in Algorithm 6.1 we may want to stop when no class has more than a certain number of examples left, and include a default rule for the remaining examples. Likewise, in Algorithm 6.2 we may want to stop if D drops below a certain size.

Rule lists have much in common with decision trees. We can therefore analyse the construction of a rule list in the same way as we did in Figure 5.3 on p.137. This is shown for the running example in Figure 6.5. For example, adding the first rule is depicted in coverage space by splitting the ascending diagonal A into a horizontal segment B representing the new rule and another diagonal segment C representing the new coverage space. Adding the second rule causes segment C to split into vertical segment D (the second rule) and diagonal segment E (the third coverage space). Finally, E is split into a horizontal and a vertical segment (the third rule and the default rule, respectively). The remaining segments B, D, F and G are now all horizontal or vertical, signalling that the rules we learned are pure.

Algorithm 6.1: LearnRuleList(D) – learn an ordered list of rules.

```
Input   : labelled training data $D$.
Output  : rule list $R$.
1 $R \leftarrow \emptyset$;
2 while $D \neq \emptyset$ do
3     $r \leftarrow$ LearnRule($D$) ;                 // LearnRule: see Algorithm 6.2
4     append $r$ to the end of $R$;
5     $D \leftarrow D \setminus \{x \in D | x$ is covered by $r\}$;
6 end
7 return $R$
```



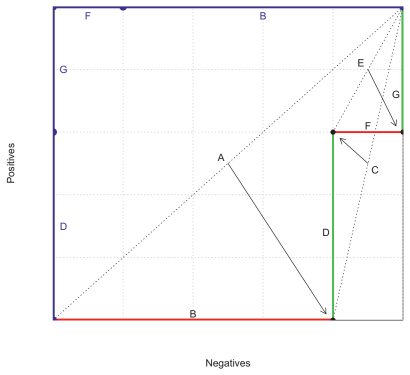


Figure 6.5. **(left)** A right-branching feature tree corresponding to a list of single-literal rules. **(right)** The construction of this feature tree depicted in coverage space. The leaves of the tree are either purely positive (in green) or purely negative (in red). Reordering these leaves on their empirical probability results in the blue coverage curve. As the rule list separates the classes this is a perfect coverage curve.

Rule lists for ranking and probability estimation

Turning a rule list into a ranker or probability estimator is as easy as it was for decision trees. Due to the covering algorithm we have access to the local class distributions

Algorithm 6.2: LearnRule(D) – learn a single rule.

```
Input  : labelled training data $D$.
Output : rule $r$.
$b$ ← true;
$L$ ← set of available literals;
while not Homogeneous($D$) do
    $l$ ← BestLiteral($D, L$) ;                    // e.g., highest purity; see text
    $b \leftarrow b \wedge l$;
    $D \leftarrow \{x \in D | x$ is covered by $b\}$;
    $L \leftarrow L \setminus \{l' \in L | l'$ uses same feature as $l\}$;
end
$C$ ← Label($D$) ;                                  // e.g., majority class
$r$ ← ·if $b$ then Class $= C$·;
return $r$
```

associated with each rule. We can therefore base our scores on the empirical probabilities. In the case of two classes we can rank the instances on decreasing empirical probability of the positive class, giving rise to a coverage curve with one segment for each rule. It is important to note that the ranking order of the rules is different from their order in the rule list, just as the ranking order of the leaves of a tree is different from their left-to-right order.

Example 6.2 (Rule lists as rankers). Consider the following two concepts:

(A)	Length = 4	p2	n2, n4–5
(B)	Beak = yes	p1–5	n1–2, n5

Indicated on the right is each concept's coverage over the whole training set. Using these concepts as rule bodies, we can construct the rule list AB:

·**if** Length = 4 **then** Class = ⊖·	[1+, 3−]
·**else if** Beak = yes **then** Class = ⊕·	[4+, 1−]
·**else** Class = ⊖·	[0+, 1−]

The coverage curve of this rule list is given in Figure 6.6. The first segment of the curve corresponds to all instances which are covered by B but not by A, which is why we use the set-theoretical notation B \ A. Notice that while this segment corresponds to the second rule in the rule list, it comes first in the coverage curve because it has the highest proportion of positives. The second coverage segment corresponds to rule A, and the third coverage segment denoted '-' corresponds to the default rule. This segment comes last, not because it represents the last rule, but because it happens to cover no positives.

We can also construct a rule list in the opposite order, BA:

·**if** Beak = yes **then** Class = ⊕·	[5+, 3−]
·**else if** Length = 4 **then** Class = ⊖·	[0+, 1−]
·**else** Class = ⊖·	[0+, 1−]

The coverage curve of this rule list is also depicted in Figure 6.6. This time, the first segment corresponds to the first segment in the rule list (B), and the second and third segment are tied between rule A (after the instances covered by B are taken away: A \ B) and the default rule.

Which of these rule lists is a better ranker? We can see that AB makes fewer ranking errors than BA (4.5 vs. 7.5), and thus has better AUC (0.82 vs. 0.70). We also see that,

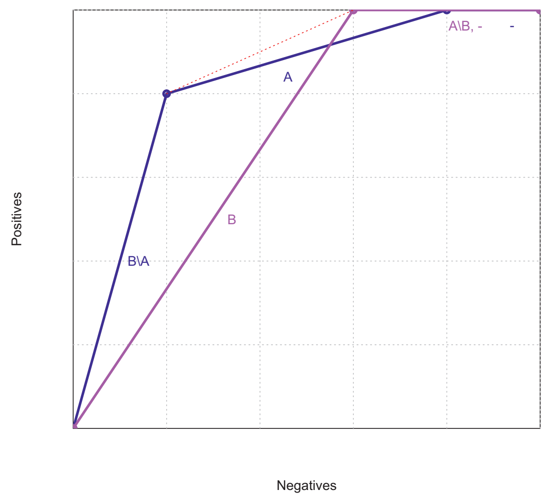


Figure 6.6. Coverage curves of two rule lists consisting of the rules from Example 6.2, in different order (AB in blue and BA in violet). B \ A corresponds to the coverage of rule B once the coverage of rule A is taken away, and '-' denotes the default rule. Neither curve dominates the other, and thus each has operating conditions under which it is superior. The dotted segment in red connecting the two curves corresponds to the overlap of the two rules A ∧ B, which is not accessible by either rule list.

if accuracy is our performance criterion, AB would be optimal, achieving 0.80 accuracy ($tpr = 0.80$ and $tnr = 0.80$) where BA only manages 0.70 ($tpr = 1$ and $tnr = 0.40$). However, if performance on the positives is 3 times as important as performance on the negatives, then BA's optimal operating point outperforms AB's. Hence, each rule list contains information not present in the other, and so neither is uniformly better.

The main reason for this is that the segment A ∧ B – the overlap of the two rules – is not accessible by either rule list. In Figure 6.6 this is indicated by the dotted segment connecting the segment B from rule list BA and the segment B \ A from rule list AB. It follows that this segment contains exactly those examples that are in B but not in B \ A, hence in A ∧ B. In order to access the rule overlap, we need to either combine the two rule lists or go beyond the power of rule lists. This will be investigated further at the end of the next section.

There are thus several connections between rule lists and decision trees. Furthermore, *rule lists are similar to decision trees in that the empirical probabilities associated with each rule yield convex ROC and coverage curves on the training data*. We have access to those empirical probabilities because of the coverage algorithm, which removes all training instances covered by one rule before learning the next (Algorithm 6.1). As a result, rule lists produce probabilities that are well-calibrated on the training set. Some rule learning algorithms in the literature reorder the rule list after all rules have been constructed. In this case, convexity cannot be guaranteed unless we re-evaluate the coverage of each rule in the reordered rule list.

6.2 Learning unordered rule sets

We next consider the alternative approach to rule learning, where rules are learned for one class at a time. This means we can further simplify our search heuristic: rather than minimising $\min(\dot{p}, 1-\dot{p})$, we can maximise $\dot{p}$, the empirical probability of the class we are learning. This search heuristic is conventionally referred to by its 'evaluation measure name' *precision* (see Table 2.3 on p.57).

Example 6.3 (Learning a rule set for one class). We continue the dolphin example. Figure 6.7 shows that the first rule learned for the positive class is

$$\cdot\textbf{if } \mathsf{Length} = 3 \textbf{ then } \mathsf{Class} = \oplus\cdot$$

The two examples covered by this rule are removed, and a new rule is learned. We now encounter a new situation, as none of the candidates is pure (Figure 6.8). We thus start a second-level search, from which the following pure rule emerges:

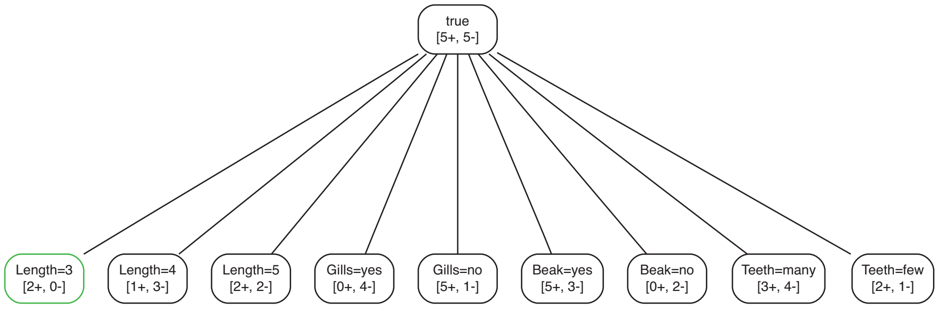
$$\cdot\textbf{if } \mathsf{Gills} = \mathsf{no} \wedge \mathsf{Length} = 5 \textbf{ then } \mathsf{Class} = \oplus\cdot$$

To cover the remaining positive, we again need a rule with two conditions (Figure 6.9):

$$\cdot\textbf{if } \mathsf{Gills} = \mathsf{no} \wedge \mathsf{Teeth} = \mathsf{many} \textbf{ then } \mathsf{Class} = \oplus\cdot$$

Notice that, even though these rules are overlapping, their overlap only covers positive examples (since each of them is pure) and so there is no need to organise them in an if-then-else list.

We now have a rule set for the positive class. With two classes this might be considered sufficient, as we can classify everything that isn't covered by the positive rules as negative. However, this might introduce a bias towards the negative class as all difficult cases we're unsure about get automatically classified as negative. So let's learn some rules for the negative class. By the same procedure as in Example 6.3 we find the following rules (you may want to check this): $\cdot\textbf{if } \mathsf{Gills} = \mathsf{yes} \textbf{ then } \mathsf{Class} = \ominus\cdot$ first, followed by $\cdot\textbf{if } \mathsf{Length} = 4 \wedge \mathsf{Teeth} = \mathsf{few} \textbf{ then } \mathsf{Class} = \ominus\cdot$. The final rule set with rules for



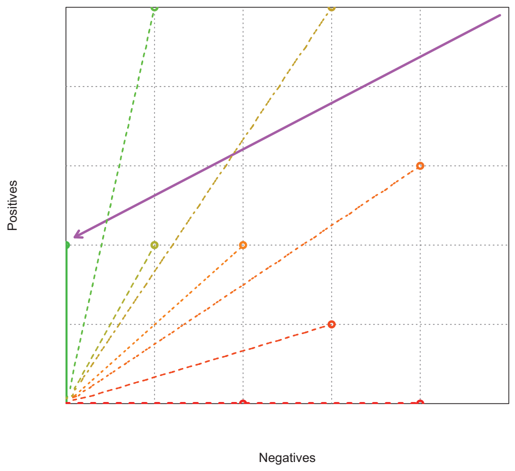
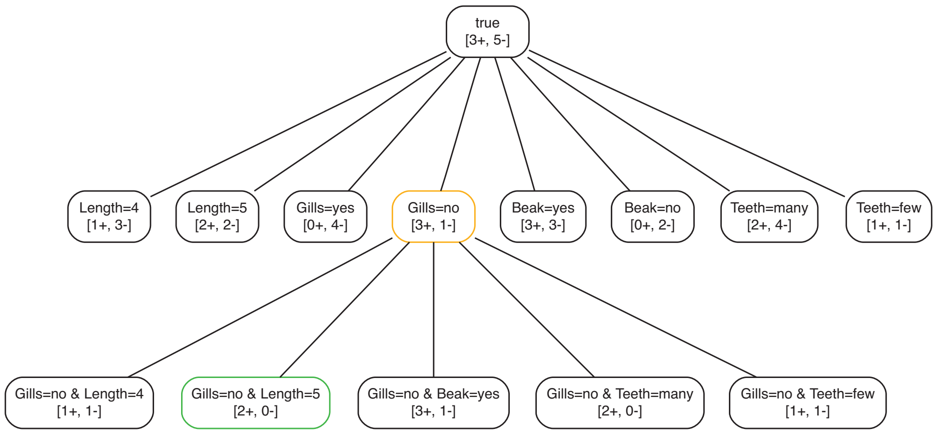


Figure 6.7. (top) The first rule is learned for the positive class. **(bottom)** Precision isometrics look identical to impurity isometrics (Figure 6.2); however, the difference is that precision is lowest on the x-axis and highest on the y-axis, while purity is lowest on the ascending diagonal and highest on both the x-axis and the y-axis.

both classes is therefore

(R1) ·**if** Length = 3 **then** Class = ⊕·
(R2) ·**if** Gills = no ∧ Length = 5 **then** Class = ⊕·
(R3) ·**if** Gills = no ∧ Teeth = many **then** Class = ⊕·
(R4) ·**if** Gills = yes **then** Class = ⊖·
(R5) ·**if** Length = 4 ∧ Teeth = few **then** Class = ⊖·

The algorithm for learning a rule set is given in Algorithm 6.3. The main differences



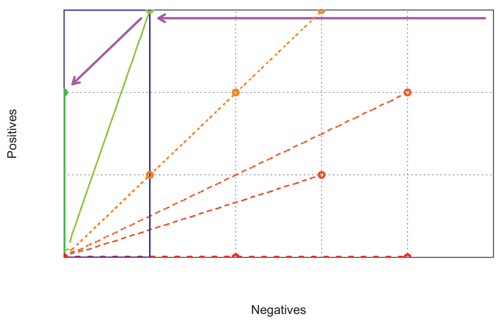
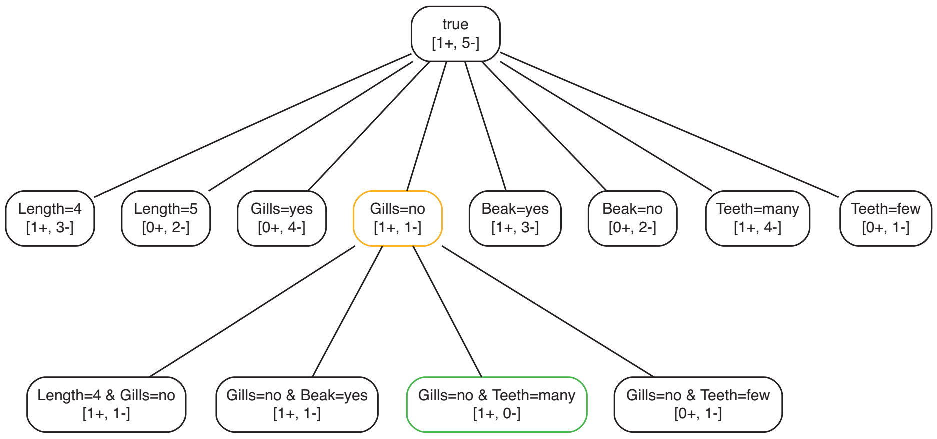


Figure 6.8. **(top)** The second rule needs two literals: we use maximum precision to select both. **(bottom)** The coverage space is smaller because the two positives covered by the first rule are removed. The blue box on the left indicates an even smaller coverage space in which the search for the second literal is carried out, after the condition Gills = no filters out four negatives. Inside the blue box precision isometrics overlap with those in the outer box (this is not necessarily the case with search heuristics other than precision).

with ☞*LearnRuleList* (Algorithm 6.1 on p.163) is that we now iterate over each class in turn, and furthermore that only covered examples for the class that we are currently learning are removed after a rule is found. The reason for this second change is that rule sets are not executed in any particular order, and so covered negatives are not filtered out by other rules. Algorithm 6.4 gives the algorithm for learning a single rule for a particular class, which is very similar to ☞*LearnRule* (Algorithm 6.2 on p.164) except (*i*) the best literal is now chosen with regard to the class to be learned, C_i; and(*ii*) the head of the rule is always labelled with C_i. An interesting variation that is sometimes



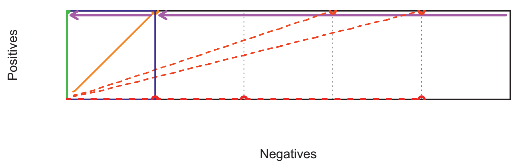


Figure 6.9. (top) The third and final rule again needs two literals. **(bottom)** The first literal excludes four negatives, the second excludes the one remaining negative.

encountered in the literature is to initialise the set of available literals *L* to those occurring in a given *seed example* belonging to the class to be learned: the advantage is that this cuts back the search space, but a possible disadvantage is that the choice of seed example may be sub-optimal.

One issue with using precision as search heuristic is that it tends to focus a bit too much on finding pure rules, thereby occasionally missing near-pure rules that can be specialised into a more general pure rule. Consider Figure 6.10 (top): precision favours the rule ·**if** Length = 3 **then** Class = ⊕·, even though the near-pure literal Gills = no leads to the pure rule ·**if** Gills = no ∧ Teeth = many **then** Class = ⊕·. A convenient way to deal with this 'myopia' of precision is the Laplace correction, which ensures that [5+, 1−] is 'corrected' to [6+, 2−] and thus considered to be of the same quality as [2+, 0−] aka [3+, 1−] (Figure 6.10 (bottom)). Another way to reduce myopia further and break such ties is to employ a *beam search*: rather than greedily going for the best candidate, we maintain a fixed number of alternate candidates. In the example, a small beam size would already allow us to find the more general rule:

- ☞ the first beam would include the candidate bodies Length = 3 and Gills = no;
- ☞ we then add all possible specialisations of non-pure elements of the beam;
- ☞ of the remaining set – i.e., elements of the original beam plus all added specialisations – we keep only the best few, preferring the ones that were already on the beam in case of ties, as they are shorter;
- ☞ we stop when all beam elements are pure, and we select the best one.

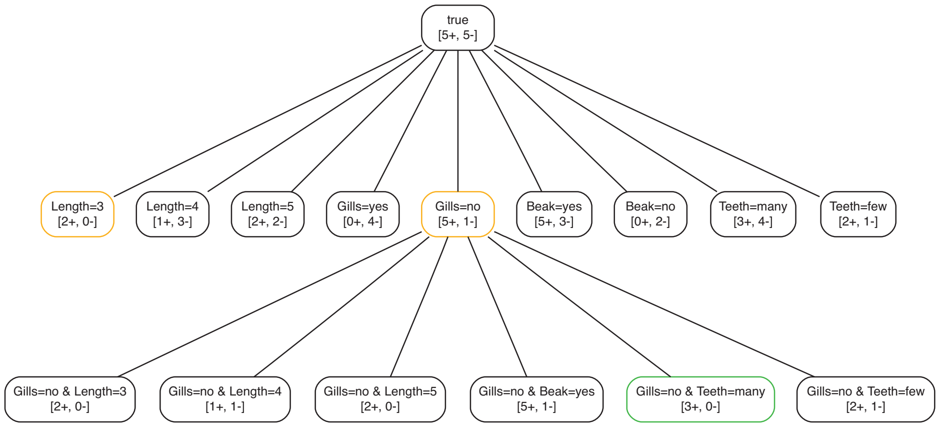
Now that we have seen how to learn a rule set, we turn to the question of how to employ a rule set model as a classifier. Suppose we encounter a new instance, say Length = 3 ∧ Gills = yes ∧ Beak = yes ∧ Teeth = many. With the rule list on p.162 the

Algorithm 6.3: LearnRuleSet(D) – learn an unordered set of rules.

```
Input  : labelled training data $D$.
Output : rule set $R$.
1  $R \leftarrow \emptyset$;
2  for every class $C_i$ do
3      $D_i \leftarrow D$;
4      while $D_i$ contains examples of class $C_i$ do
5          $r \leftarrow$ LearnRuleForClass($D_i, C_i$) ;      // LearnRuleForClass: see Algorithm 6.4
6          $R \leftarrow R \cup \{r\}$;
7          $D_i \leftarrow D_i \setminus \{x \in C_i | x$ is covered by $r\}$ ;      // remove only positives
8      end
9  end
10 return $R$
```

Algorithm 6.4: LearnRuleForClass(D, C_i) – learn a single rule for a given class.

```
Input  : labelled training data $D$; class $C_i$.
Output : rule $r$.
1  $b \leftarrow$ true;
2  $L \leftarrow$ set of available literals ;      // can be initialised by seed example
3  while not Homogeneous($D$) do
4      $l \leftarrow$ BestLiteral($D, L, C_i$) ;      // e.g. maximising precision on class $C_i$
5      $b \leftarrow b \wedge l$;
6      $D \leftarrow \{x \in D | x$ is covered by $b\}$;
7      $L \leftarrow L \setminus \{l' \in L | l'$ uses same feature as $l\}$;
8  end
9  $r \leftarrow$ ·if $b$ then Class = $C_i$·;
10 return $r$
```



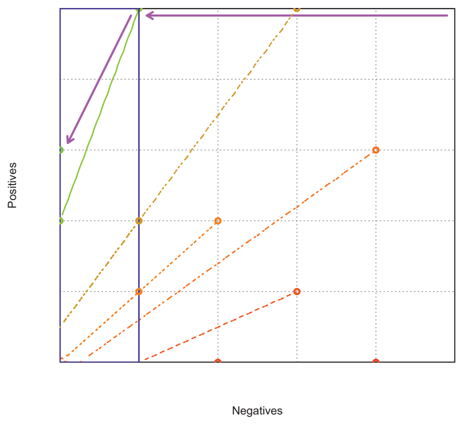


Figure 6.10. **(top)** Using Laplace-corrected precision allows learning a better rule in the first iteration. **(bottom)** Laplace correction adds one positive and one negative pseudo-count, which means that the isometrics now rotate around $(-1,-1)$ in coverage space, resulting in a preference for more general rules.

first rule would fire and hence the instance is classified as negative. With the rule set on p.168 we have that both R1 and R4 fire and make contradictory predictions. How can we resolve this? In order to answer that question, it is easier to consider a more general question first: how do we use a rule set for ranking and probability estimation?

Rule sets for ranking and probability estimation

In the general case, for a rule set consisting of r rules there are up to 2^r different ways in which rules can overlap, and hence 2^r instance space segments. Even though many of these segments will be empty because rules are mutually exclusive, in general we will have more instance space segments than rules. As a consequence, we have to estimate the coverage of some of these segments.

Example 6.4 (Rule sets as rankers). Consider the following rule set (the first two rules were also used in Example 6.2):

(A)	·**if** Length = 4 **then** Class = ⊖·	[1+,3−]
(B)	·**if** Beak = yes **then** Class = ⊕·	[5+,3−]
(C)	·**if** Length = 5 **then** Class = ⊖·	[2+,2−]

The figures on the right indicate coverage of each rule over the whole training set. For instances covered by single rules we can use these coverage counts to calculate probability estimates: e.g., an instance covered only by rule A would receive probability $\hat{p}(\mathsf{A}) = 1/4 = 0.25$, and similarly $\hat{p}(\mathsf{B}) = 5/8 = 0.63$ and $\hat{p}(\mathsf{C}) = 2/4 = 0.50$.

Clearly A and C are mutually exclusive, so the only overlaps we need to take into account are AB and BC. A simple trick that is often applied is to average the coverage of the rules involved: for example, the coverage of AB is estimated as [3+,3−] yielding $\hat{p}(\mathsf{AB}) = 3/6 = 0.50$. Similarly, $\hat{p}(\mathsf{BC}) = 3.5/6 = 0.58$. The corresponding ranking is thus B – BC – [AB, C] – A, resulting in the orange training set coverage curve in Figure 6.11.

Let us now compare this rule set with the following rule list ABC:

·**if** Length = 4 **then** Class = ⊖·	[1+,3−]
·**else if** Beak = yes **then** Class = ⊕·	[4+,1−]
·**else if** Length = 5 **then** Class = ⊖·	[0+,1−]

The coverage curve of this rule list is indicated in Figure 6.11 as the blue line. We see that the rule set outperforms the rule list, by virtue of being able to distinguish between examples covered by B only and those covered by both B and C.

While in this example the rule set outperformed the rule list, this cannot be guaranteed in general. Due to the fact that the coverage counts of some segments have to be estimated, a rule set coverage curve is not guaranteed to be convex even on the

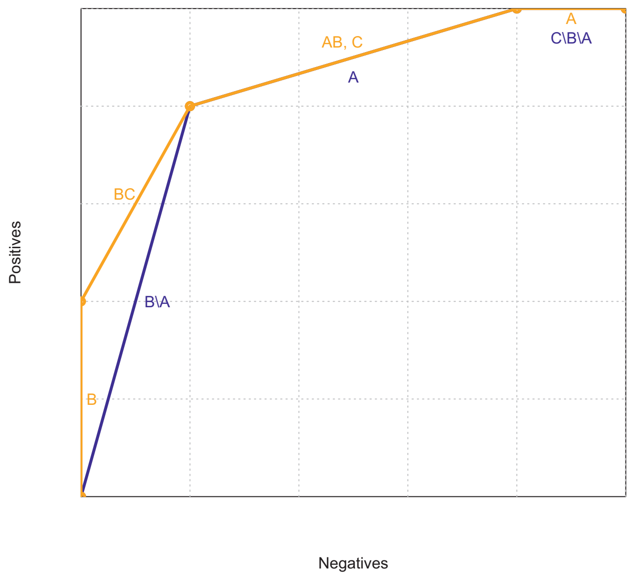


Figure 6.11. Coverage curves of the rule set in Example 6.4 (in orange) and the rule list ABC (in blue). The rule set partitions the instance space in smaller segments, which in this case lead to better ranking performance.

training set. For example, suppose that rule C also covers p1, then this won't affect the performance of the rule list (since p1 is already covered by B), but it would break the tie between AB and C in favour of the latter and thus introduce a concavity.

If we want to turn such a ranker into a classifier, we have to find the best operating point on the coverage curve. Assuming accuracy as our performance criterion, the point ($fpr = 0.2, tpr = 0.8$) is optimal, which can be achieved by classifying instances with $\hat{p} > 0.5$ as positive and the rest as negative. If such calibration of the decision threshold is problematic (for example, in the case of more than two classes), we can simply assign the class with the highest average coverage, making a random choice in case of a tie.

A closer look at rule overlap

We have seen that rule lists always give convex training set coverage curves, but that there is no globally optimal ordering of a given set of rules. The main reason is that rule lists don't give us access to the overlap of two rules $A \wedge B$: we either have access to $A = (A \wedge B) \vee (A \wedge \neg B)$ if the rule order is AB, or $B = (A \wedge B) \vee (\neg A \wedge B)$ if it is BA. More generally, a rule list of r rules results in only r instance space segments (or $r+1$

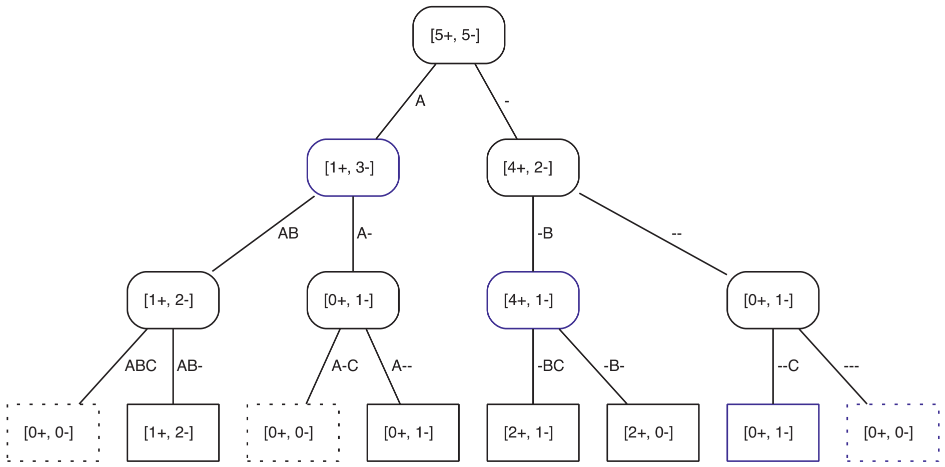


Figure 6.12. A rule tree constructed from the rules in Example 6.5. Nodes are labelled with their coverage (dotted leaves have empty coverage), and branch labels indicate particular areas in the instance space (e.g., A-C denotes $A \wedge \neg B \wedge C$). The blue nodes are the instance space segments corresponding to the rule list ABC: the rule tree has better performance because it is able to split them further.

in case we add a default rule). This means that we cannot take advantage of most of the 2^r ways in which rules can overlap. Rule sets, on the other hand, can potentially give access to such overlaps, but the need for the coverage counts of overlapping segments to be estimated means that we have to sacrifice convexity. In order to understand this further, we introduce in this section the concept of a *rule tree*: a complete feature tree using the rules as features.

Example 6.5 (Rule tree). From the rules in Example 6.4 we can construct the rule tree in Figure 6.12. The use of a tree rather than a list allows further splitting of the segments of the rule list. For example, the node labelled A is further split into AB ($A \wedge B$) and A- ($A \wedge \neg B$). As the latter is pure, we obtain a better coverage curve (the red line in Figure 6.13).

As we see in this example, the rule tree coverage curve dominates the rule list coverage curve. This is true in general: there is no other information regarding rule overlap than that contained in a rule tree, and any given rule list will usually convey only part of that information. Conversely, we may wonder whether any operating point on the rule tree curve is reachable by a particular rule list. The answer to this is negative, as a

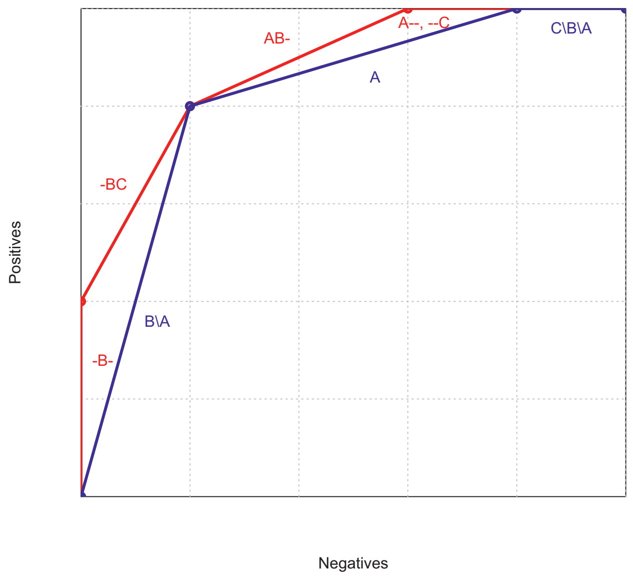


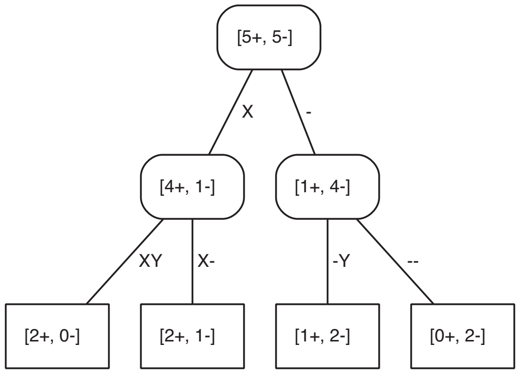
Figure 6.13. The blue line is the coverage curve of the rule list ABC in Example 6.4. This curve is dominated by the red coverage curve, corresponding to the rule tree in Figure 6.12. The rule tree also improves upon the rule set (orange curve in Figure 6.11), as it has access to exact coverage counts in all segments and thus recognises that AB- goes before - -C.

simple counter-example shows (Figure 6.14).

In summary, of the three rule models considered, only rule trees can unlock the full potential of rule overlap as they have the capacity to represent all 2^r overlap areas of r rules and give access to exact coverage counts for each area. Rule lists also convey exact coverage counts but for fewer segments; rule sets distinguish the same segments as rule trees but have to estimate coverage counts for the overlap areas. On the other hand, rule trees are expensive as their size is exponential in the number of rules. Another disadvantage is that the coverage counts have to be obtained in a separate step, after the rules have been learned. I have included rule trees here mainly for conceptual reasons: to gain a better understanding of the more common rule list and rule set models.

6.3 Descriptive rule learning

As we have seen, the rule format lends itself naturally to predictive models, built from rules with the target variable in the head. It is not hard to come up with ways to extend



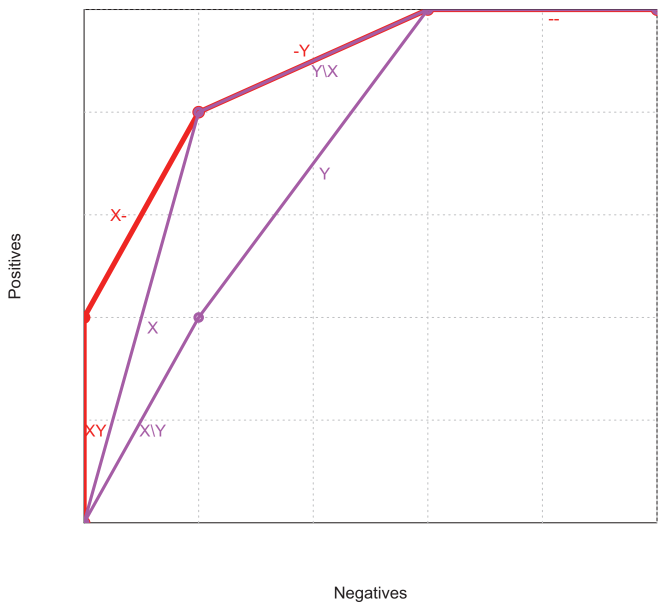


Figure 6.14. (top) A rule tree built on two rules X and Y. **(bottom)** The rule tree coverage curve strictly dominates the convex hull of the two rule list curves. This means that there is an operating point $[2+,0-]$ that cannot be achieved by either rule list.

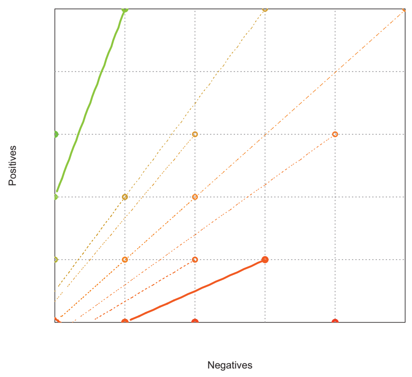
rule models to regression and clustering tasks, in a similar way to what we did for tree models at the end of Chapter 5, but I will not elaborate on that here. Instead I will show how the rule format can equally easily be used to build descriptive models. As explained in Section 1.1, descriptive models can be learned in either a supervised or an unsupervised way. As an example of the supervised setting we will discuss how to adapt the given rule learning algorithms to subgroup discovery. For unsupervised learning of descriptive rule models we will take a look at frequent item sets and association rule discovery.

Rule learning for subgroup discovery

When learning classification models it is natural to look for rules that identify pure subsets of the training examples: i.e., sets of examples that are all of the same class and that all satisfy the same conjunctive concept. However, as we have seen in Section 3.3, sometimes we are less interested in predicting a class and more interested in finding interesting patterns. We defined subgroups as mappings $\hat{g} : \mathscr{X} \to \{\mathsf{true}, \mathsf{false}\}$ – or alternatively, subsets of the instance space – that are learned from a set of labelled examples $(x_i, l(x_i))$, where $l : \mathscr{X} \to \mathscr{C}$ is the true labelling function. A good subgroup is one whose class distribution is significantly different from the overall population. This is by definition true for pure subgroups, but these are not the only interesting ones. For instance, one could argue that the complement of a subgroup is as interesting as the subgroup itself: in our dolphin example, the concept Gills = yes, which covers four negatives and no positives, could be considered as interesting as its complement Gills = no, which covers one negative and all positives. This means that we need to move away from impurity-based evaluation measures.

Like concepts, subgroups can be plotted as points in coverage space, with the positives in the subgroup on the y-axis and the negatives on the x-axis. Any subgroup plotted on the ascending diagonal has the same proportion of positives as the overall population; these are the least interesting subgroups as they have the same statistics as random samples. Subgroups above (below) the diagonal have a larger (smaller) proportion of positives than the population. So one way to measure the quality of subgroups is to take one of the heuristics used for rule learning and measure the absolute deviation from the default value on the diagonal. For example, the precision of any subgroup on the diagonal is equal to the proportion of positives, so this leads to $|prec - pos|$ as one possible quality measure. For reasons already discussed it is often better to use Laplace-corrected precision $prec^{\mathrm{L}}$, leading to the alternative measure $|prec^{\mathrm{L}} - pos|$. As can be seen in Figure 6.15 (left), the introduction of pseudo-counts means that $[5+, 1-]$ is evaluated as $[6+, 2-]$ and is thus as interesting as the pure concept $[2+, 0-]$ which is evaluated as $[3+, 1-]$.

However, this doesn't quite put complementary subgroups on an equal footing, as $[5+, 1-]$ is still considered to be of lower quality than $[0+, 4-]$. In order to achieve this complementarity we need an evaluation measure whose isometrics all run parallel to the ascending diagonal. As it turns out, we have already seen such an evaluation measure in Section 2.1, where we called it *average recall* (see, e.g., Figure 2.4 on p.61). Notice that subgroups on the diagonal always have average recall 0.5, regardless of the class distribution. So, a good subgroup evaluation measure is $|avg\text{-}rec - 0.5|$. Average recall can be written as $(1 + tpr - fpr)/2$, and thus $|avg\text{-}rec - 0.5| = |tpr - fpr|/2$. It is sometimes desirable not to take the absolute value, so that the sign of the difference tells us whether we are above or below the diagonal. A related subgroup evaluation measure is



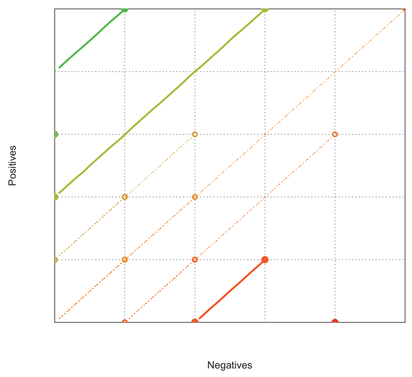


Figure 6.15. (left) Subgroups and their isometrics according to Laplace-corrected precision. The solid, outermost isometrics indicate the best subgroups. **(right)** The ranking changes if we order the subgroups on average recall. For example, [5+, 1−] is now better than [3+, 0−] and as good as [0+, 4−].

Subgroup	*Coverage*	$prec^{L}$	*Rank*	*avg-rec*	*Rank*
Gills = yes	[0+, 4−]	0.17	1	0.10	1–2
Gills = no ∧ Teeth = many	[3+, 0−]	0.80	2	0.80	3
Gills = no	[5+, 1−]	0.75	3–9	0.90	1–2
Beak = no	[0+, 2−]	0.25	3–9	0.30	4–11
Gills = yes ∧ Beak = yes	[0+, 2−]	0.25	3–9	0.30	4–11
Length = 3	[2+, 0−]	0.75	3–9	0.70	4–11
Length = 4 ∧ Gills = yes	[0+, 2−]	0.25	3–9	0.30	4–11
Length = 5 ∧ Gills = no	[2+, 0−]	0.75	3–9	0.70	4–11
Length = 5 ∧ Gills = yes	[0+, 2−]	0.25	3–9	0.30	4–11
Length = 4	[1+, 3−]	0.33	10	0.30	4–11
Beak = yes	[5+, 3−]	0.60	11	0.70	4–11

Table 6.1. Detailed evaluation of the top subgroups. Using Laplace-corrected precision we can evaluate the quality of a subgroup as $|prec^{L} - pos|$. Alternatively, we can use average recall to define the quality of a subgroup as $|avg\text{-}rec - 0.5|$. These two quality measures result in slightly different rankings.

weighted relative accuracy, which can be written as $pos \cdot neg(tpr - fpr)$.

As can be seen by comparing the two isometrics plots in Figure 6.15, using average recall rather than Laplace-corrected precision has an effect on the ranking of some of the subgroups. Detailed calculations are given in Table 6.1.

Subgroup	*Coverage*	*avg-rec*	*Wgtd coverage*	*W-avg-rec*	*Rank*
Gills = yes	[0+, 4−]	0.10	[0+, **3**−]	0.07	1–2
Gills = no	[5+, 1−]	0.90	[**4.5**+, **0.5**−]	0.93	1–2
Gills = no ∧ Teeth = many	[3+, 0−]	0.80	[**2.5**+, 0−]	0.78	3
Length = 5 ∧ Gills = yes	[0+, 2−]	0.30	[0+, 2−]	0.21	4
Length = 3	[2+, 0−]	0.70	[2+, 0−]	0.72	5–6
Length = 5 ∧ Gills = no	[2+, 0−]	0.70	[2+, 0−]	0.72	5–6
Beak = no	[0+, 2−]	0.30	[0+, **1.5**−]	0.29	7–9
Gills = yes ∧ Beak = yes	[0+, 2−]	0.30	[0+, **1.5**−]	0.29	7–9
Beak = yes	[5+, 3−]	0.70	[**4.5**+, **2**−]	0.71	7–9
Length = 4	[1+, 3−]	0.30	[**0.5**+, **1.5**−]	0.34	10
Length = 4 ∧ Gills = yes	[0+, 2−]	0.30	[0+, **1**−]	0.36	11

Table 6.2. The 'Wgtd coverage' column shows how the weighted coverage of the subgroups is affected if the weights of the examples covered by Length = 4 are reduced to 1/2. 'W-*avg-rec*' shows how how the *avg-rec* numbers as calculated in Table 6.1 are affected by the weighting, leading to further differentiation between subgroups that were previously considered equivalent.

Example 6.6 (Comparing Laplace-corrected precision and average recall). Table 6.1 ranks ten subgroups in the dolphin example in terms of Laplace-corrected precision and average recall. One difference is that Gills = no ∧ Teeth = many with coverage [3+, 0−] is better than Gills = no with coverage [5+, 1−] in terms of Laplace-corrected precision, but worse in terms of average recall, as the latter ranks it equally with its complement Gills = yes.

The second difference between classification rule learning and subgroup discovery is that in the latter case we are naturally interested in overlapping rules, whereas the standard covering algorithm doesn't encourage this as examples already covered are removed from the training set. One way of dealing with this is by assigning weights to examples that are decreased every time an example is covered by a newly learned rule. A scheme that works well in practice is to initialise the example weights to 1 and halve them every time a new rule covers the example. Search heuristics are then evaluated in terms of the cumulative weight of covered examples, rather than just their number.

Example 6.7 (The effect of weighted covering). Suppose the first subgroup

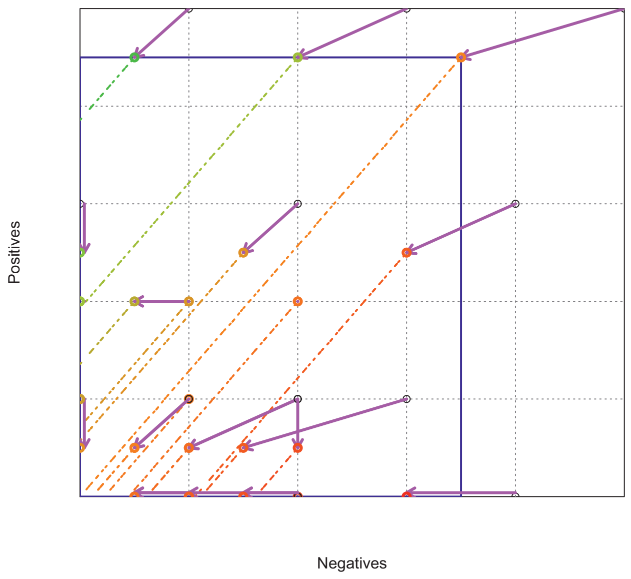


Figure 6.16. Visualisation of the effect of weighted covering. If the first subgroup found is Length = 4, then this halves the weight of one positive and three negatives, shrinking the coverage space to the blue box. The arrows indicate how this affects the weighted coverage of other subgroups, depending on which of the reduced-weight examples they cover.

found is Length = 4, reducing the weight of the one positive and three negatives covered by it to 1/2. Detailed calculations of how this affects the weighted coverage of subgroups are given in Table 6.2. We can see how the coverage space shrinks to the blue box in Figure 6.16. It also affects the weighted coverage of the subgroups overlapping with Length = 4, as indicated by the arrows. Some subgroups end up closer to the diagonal and hence lose importance: for instance, Length = 4 itself, which moves from [3+, 1−] to [1.5+, 0.5−]. Others move away from the diagonal and hence gain importance: for example Length = 5 ∧ Gills = yes at [0+, 2−].

The *weighted covering* algorithm is given in Algorithm 6.5. Notice that this algorithm can be applied to discover subgroups over $k > 2$ classes, as long as the evaluation measure used to learn single rules can handle more than two classes. This is clearly the case for average recall used in our examples. Other possibilities include measures derived from the Chi-squared test and mutual information-based measures.

Association rule mining

I will now introduce a new kind of rule that can be learned in a wholly unsupervised manner and is prominent in data mining applications. Suppose we observed eight customers who each bought one or more of apples, beer, crisps and nappies:

Transaction	*Items*
1	nappies
2	beer, crisps
3	apples, nappies
4	beer, crisps, nappies
5	apples
6	apples, beer, crisps, nappies
7	apples, crisps
8	crisps

Each *transaction* in this table involves a set of *items*; conversely, for each item we can list the transactions in which it was involved: transactions 1, 3, 4 and 6 for nappies, transactions 3, 5, 6 and 7 for apples, and so on. We can also do this for sets of items: e.g., beer and crisps were bought together in transactions 2, 4 and 6; we say that item set $\{\text{beer}, \text{crisps}\}$ *covers* transaction set $\{2,4,6\}$. There are 16 of such item sets (including the empty set, which covers all transactions); using the subset relation between transaction sets as partial order, they form a lattice (Figure 6.17).

Let us call the number of transactions covered by an item set I its *support*, denoted $\text{Supp}(I)$ (sometimes called frequency). We are interested in *frequent item sets*, which exceed a given support threshold f_0. Support is *monotonic*: when moving down a path in the item set lattice it can never increase. This means that the set of frequent item

Algorithm 6.5: WeightedCovering(D) – learn overlapping rules by weighting examples.

```
Input  : labelled training data $D$ with instance weights initialised to 1.
Output : rule list $R$.
1 $R \leftarrow \emptyset$;
2 while some examples in $D$ have weight 1 do
3     $r \leftarrow$ LearnRule($D$) ;                // LearnRule: see Algorithm 6.2
4     append $r$ to the end of $R$;
5     decrease the weights of examples covered by $r$;
6 end
7 return $R$
```

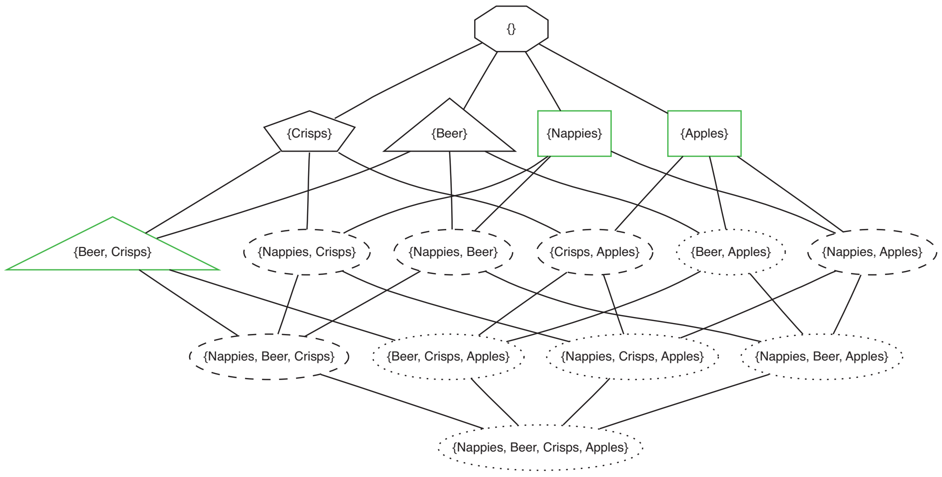


Figure 6.17. An item set lattice. Item sets in dotted ovals cover a single transaction; in dashed ovals, two transactions; in triangles, three transactions; and in polygons with n sides, n transactions. The maximal item sets with support 3 or more are indicated in green.

sets is *convex* and is fully determined by its lower boundary of largest item sets: in the example these maximal[1] frequent item sets are, for $f_0 = 3$: {apples}, {beer, crisps} and {nappies}. So, at least three transactions involved apples; at least three involved nappies; at least three involved both beer and crisps; and any other combination of items was bought less often.

Because of the monotonicity property of item set support, frequent item sets can be found by a simple enumerative breadth-first or level-wise search algorithm (Algorithm 6.6). The algorithm maintains a priority queue, initially holding only the empty item set which covers all transactions. Taking the next candidate item set I off the priority queue, it generates all its possible extensions (supersets containing one more item, the downward neighbours in the item set lattice), and adds them to the priority queue if they exceed the support threshold (at the back, to achieve the desired breadth-first behaviour). If at least one of I's extensions is frequent, I is not maximal and can be discarded; otherwise I is added to the set of maximal frequent item sets found.

We can speed up calculations by restricting attention to *closed item sets*. These are completely analogous to the ☞*closed concepts* discussed at the end of Section 4.2: a closed item set contains all items that are involved in every transaction it covers. For example, {beer, crisps} covers transactions 2, 4 and 6; the only items involved in each of those transactions are beer and crisps, and so the item set is closed. However, {beer} is not closed, as it covers the same transactions, hence its closure is {beer, crisps}. If two item sets that are connected in the lattice have the same coverage, the smaller item set

[1] 'Maximal' here means that no superset is frequent.

cannot be closed. The lattice of closed item sets is shown in Figure 6.18. Notice that maximal frequent item sets are necessarily closed (as extending them will decrease their coverage below the support threshold, otherwise they aren't maximal), and are thus unaffected by this restriction; but it does allow a more efficient search.

So what is the point of these frequent item sets? The answer is that we will use them to build *association rules*, which are rules of the form ·**if** B **then** H· where both body B and head H are item sets that frequently appear in transactions together. Pick any edge in Figure 6.17, say the edge between {beer} and {nappies, beer}. We know that the support of the former is 3 and of the latter, 2: that is, three transactions involve beer and two of those involve nappies as well. We say that the *confidence* of the association rule ·**if** beer **then** nappies· is 2/3. Likewise, the edge between {nappies} and {nappies, beer} demonstrates that the confidence of the rule ·**if** nappies **then** beer· is 2/4. There are also rules with confidence 1, such as ·**if** beer **then** crisps·; and rules with empty bodies, such as ·**if** true **then** crisps·, which has confidence 5/8 (i.e., five out of eight transactions involve crisps).

But we only want to construct association rules that involve frequent items. The rule ·**if** beer ∧ apples **then** crisps· has confidence 1, but there is only one transaction involving all three and so this rule is not strongly supported by the data. So we first use Algorithm 6.6 to mine for frequent item sets; we then select bodies B and heads H from

Algorithm 6.6: FrequentItems(D, f_0) – find all maximal item sets exceeding a given support threshold.

```
Input  : data $D \subseteq \mathscr{X}$; support threshold $f_0$.
Output : set of maximal frequent item sets $M$.
$M \leftarrow \emptyset$;
initialise priority queue $Q$ to contain the empty item set;
while $Q$ is not empty do
    $I \leftarrow$ next item set deleted from front of $Q$;
    $max \leftarrow$ true ;                      // flag to indicate whether $I$ is maximal
    for each possible extension $I'$ of $I$ do
        if Supp$(I') \geq f_0$ then
            $max \leftarrow$ false ;        // frequent extension found, so $I$ is not maximal
            add $I'$ to back of $Q$;
        end
    end
    if $max =$ true then $M \leftarrow M \cup \{I\}$;
end
return $M$
```

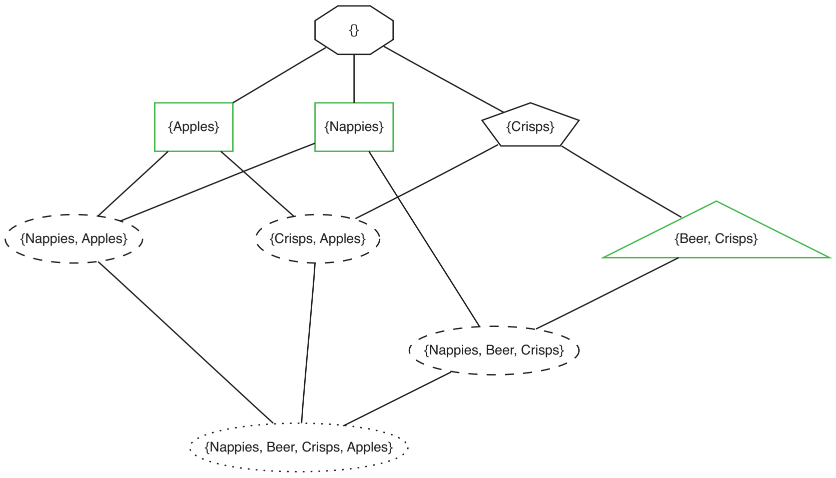


Figure 6.18. Closed item set lattice corresponding to the item sets in Figure 6.17. This lattice has the property that no two adjacent item sets have the same coverage.

the frequent sets m, discarding rules whose confidence is below a given confidence threshold. Algorithm 6.7 gives the basic algorithm. Notice that we are free to discard some of the items in the maximal frequent sets (i.e., $H \cup B$ may be smaller than m), because any subset of a frequent item set is frequent as well.

A run of the algorithm with support threshold 3 and confidence threshold 0.6 gives the following association rules:

·**if** beer **then** crisps·	support 3, confidence 3/3
·**if** crisps **then** beer·	support 3, confidence 3/5

Algorithm 6.7: AssociationRules(D, f_0, c_0) – find all association rules exceeding given support and confidence thresholds.

```
Input  : data $D \subseteq \mathscr{X}$; support threshold $f_0$; confidence threshold $c_0$.
Output : set of association rules $R$.
1 $R \leftarrow \emptyset$;
2 $M \leftarrow$ FrequentItems$(D, f_0)$ ;                // FrequentItems: see Algorithm 6.6
3 for each $m \in M$ do
4     for each $H \subseteq m$ and $B \subseteq m$ such that $H \cap B = \emptyset$ do
5         if Supp$(B \cup H)$/Supp$(B) \geq c_0$ then $R \leftarrow R \cup \{$·if $B$ then $H$·$\}$
6     end
7 end
8 return $R$
```

·**if** true **then** crisps· support 5, confidence 5/8

Association rule mining often includes a *post-processing* stage in which superfluous rules are filtered out, e.g., special cases which don't have higher confidence than the general case. One quantity that is often used in post-processing is *lift*, defined as

$$\mathrm{Lift}(\cdot\textbf{if } B \textbf{ then } H\cdot) = \frac{n \cdot \mathrm{Supp}(B \cup H)}{\mathrm{Supp}(B) \cdot \mathrm{Supp}(H)}$$

where n is the number of transactions. For example, for the the first two association rules above we would have lifts of $\frac{8\cdot 3}{3\cdot 5} = 1.6$, as $\mathrm{Lift}(\cdot\textbf{if } B \textbf{ then } H\cdot) = \mathrm{Lift}(\cdot\textbf{if } H \textbf{ then } B\cdot)$. For the third rule we have $\mathrm{Lift}(\cdot\textbf{if } \text{true} \textbf{ then } \text{crisps}\cdot) = \frac{8\cdot 5}{8\cdot 5} = 1$. This holds for any rule with $B = \emptyset$, as

$$\mathrm{Lift}(\cdot\textbf{if } \emptyset \textbf{ then } H\cdot) = \frac{n \cdot \mathrm{Supp}(\emptyset \cup H)}{\mathrm{Supp}(\emptyset) \cdot \mathrm{Supp}(H)} = \frac{n \cdot \mathrm{Supp}(H)}{n \cdot \mathrm{Supp}(H)} = 1$$

More generally, a lift of 1 means that $\mathrm{Supp}(B \cup H)$ is entirely determined by the *marginal* frequencies $\mathrm{Supp}(B)$ and $\mathrm{Supp}(H)$ and is not the result of any meaningful interaction between B and H. Only association rules with lift larger than 1 are of interest.

Quantities like confidence and lift can also be understood from a probabilistic context. Let $\mathrm{Supp}(I)/n$ be an estimate of the probability $p(I)$ that a transaction involves all items in I, then confidence estimates the conditional probability $p(H|B)$. In a classification context, where H denotes the actual class and B the predicted class, this would be called precision (see Table 2.3 on p.57), and in this chapter we have already used it as a search heuristic in rule learning. Lift then measures whether the events 'a random transaction involves all items in B' and 'a random transaction involves all items in H' are statistically independent.

It is worth noting that the heads of association rules can contain multiple items. For instance, suppose we are interested in the rule ·**if** nappies **then** beer·, which has support 2 and confidence 2/4. However, {nappies, beer} is not a closed item set: its closure is {nappies, beer, crisps}. So ·**if** nappies **then** beer· is actually a special case of ·**if** nappies **then** beer ∧ crisps·, which has the same support and confidence but involves only closed item sets.

We can also apply frequent item set analysis to our dolphin data set, if we treat each literal Feature = Value as an item, keeping in mind that different values of the same feature are mutually exclusive. Item sets then correspond to concepts, transactions to instances, and the extension of a concept is exactly the set of transactions covered by an item set. The item set lattice is therefore the same as what we previously called the hypothesis space, with the proviso that we are not considering negative examples in this scenario (Figure 6.19). The reduction to closed concepts/item sets is shown in Figure 6.20. We can see that, for instance, the rule

·**if** Gills = no ∧ Beak = yes **then** Teeth = many·

has support 3 and confidence 3/5 (but you may want to check whether it has any lift!).

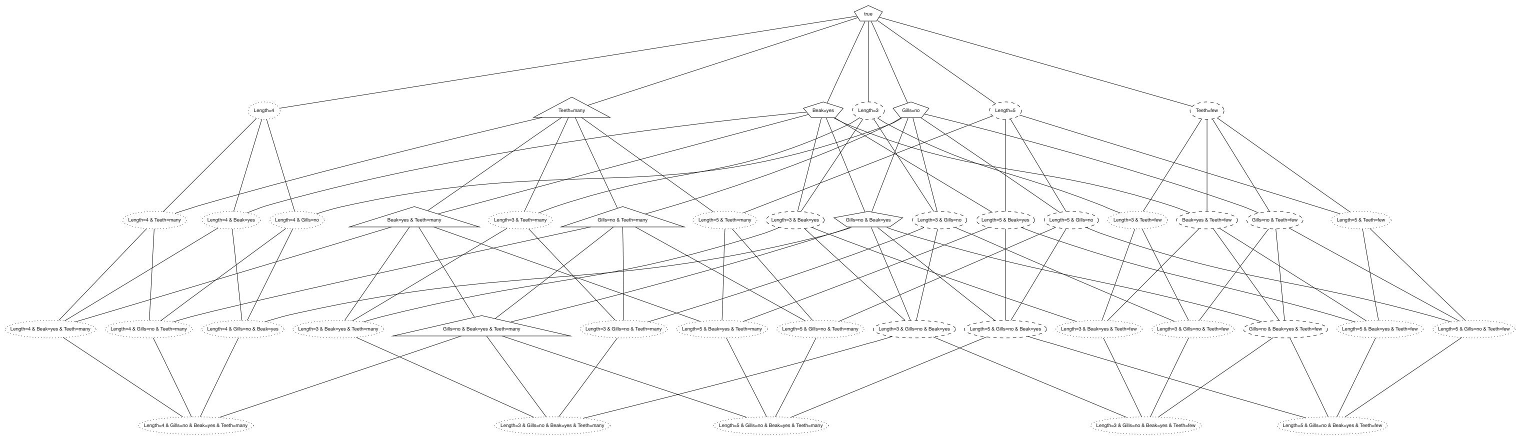

true
Length=4
Teeth=many
Beak=yes
Length=3
Gills=no
Length=5
Teeth=few
Length=4 & Teeth=many
Length=4 & Beak=yes
Length=4 & Gills=no
Beak=yes & Teeth=many
Length=3 & Teeth=many
Gills=no & Teeth=many
Length=5 & Teeth=many
Length=3 & Beak=yes
Gills=no & Beak=yes
Length=3 & Gills=no
Length=5 & Beak=yes
Length=5 & Gills=no
Length=3 & Teeth=few
Beak=yes & Teeth=few
Gills=no & Teeth=few
Length=5 & Teeth=few
Length=4 & Beak=yes & Teeth=many
Length=4 & Gills=no & Teeth=many
Length=4 & Gills=no & Beak=yes
Length=3 & Beak=yes & Teeth=many
Gills=no & Beak=yes & Teeth=many
Length=3 & Gills=no & Teeth=many
Length=5 & Beak=yes & Teeth=many
Length=5 & Gills=no & Teeth=many
Length=3 & Gills=no & Beak=yes
Length=5 & Gills=no & Beak=yes
Length=3 & Beak=yes & Teeth=few
Length=3 & Gills=no & Teeth=few
Gills=no & Beak=yes & Teeth=few
Length=5 & Beak=yes & Teeth=few
Length=5 & Gills=no & Teeth=few
Length=4 & Gills=no & Beak=yes & Teeth=many
Length=3 & Gills=no & Beak=yes & Teeth=many
Length=5 & Gills=no & Beak=yes & Teeth=many
Length=3 & Gills=no & Beak=yes & Teeth=few
Length=5 & Gills=no & Beak=yes & Teeth=few

Figure 6.19. The item set lattice corresponding to the positive examples of the dolphin example in Example 4.4 on p.115. Each 'item' is a literal Feature = Value; each feature can occur at most once in an item set. The resulting structure is exactly the same as what was called the hypothesis space in Chapter 4.

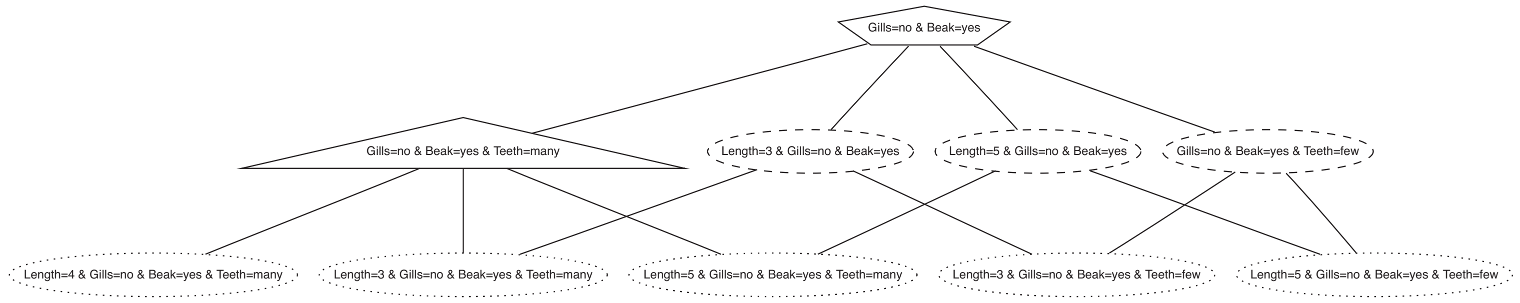

Gills=no & Beak=yes
Gills=no & Beak=yes & Teeth=many
Length=3 & Gills=no & Beak=yes
Length=5 & Gills=no & Beak=yes
Gills=no & Beak=yes & Teeth=few
Length=4 & Gills=no & Beak=yes & Teeth=many
Length=3 & Gills=no & Beak=yes & Teeth=many
Length=5 & Gills=no & Beak=yes & Teeth=many
Length=3 & Gills=no & Beak=yes & Teeth=few
Length=5 & Gills=no & Beak=yes & Teeth=few

Figure 6.20. Closed item set lattice corresponding to the item sets in Figure 6.19.

6.4 First-order rule learning

In Section 4.3 we briefly touched upon using first-order logic as a concept language. The main difference is that literals are no longer simple feature-value pairs but can have a much richer structure. All rule learning approaches covered in this chapter have been upgraded in the literature to learn rules expressed in first-order logic. In this section we will take a brief look at how this might work.

Many approaches to learning in first-order logic are based on the logic programming language Prolog, and learning first-order rules is often called *inductive logic programming* (*ILP*). Logically speaking, Prolog rules are *Horn clauses* with a single literal in the head – we encountered Horn clauses before in Section 4.3. Prolog notation is slightly different from first-order logic notation. So, instead of

$$\forall x: \mathsf{BodyPart}(x, \mathsf{PairOf}(\mathsf{Gill})) \rightarrow \mathsf{Fish}(x)$$

we write

```
fish(X):-bodyPart(X,pairOf(gills)).
```

The main differences are:

- ☞ rules are written back-to-front in 'head-if-body' fashion;
- ☞ variables start with a capital letter; constants, predicates and function symbols (called *functors* in Prolog) start with lower-case;
- ☞ variables are implicitly universally quantified.

With regard to the third point, it is worth pointing out the difference between variables occurring in both head and body, and variables occurring in the body only. Consider the following Prolog clause:

```
someAnimal(X):-bodyPart(X,pairOf(Y)).
```

There are two equivalent ways of writing this rule in first-order logic:

$$\forall x: \forall y: \mathsf{BodyPart}(x, \mathsf{PairOf}(y)) \rightarrow \mathsf{SomeAnimal}(x)$$
$$\forall x: \big(\exists y: \mathsf{BodyPart}(x, \mathsf{PairOf}(y))\big) \rightarrow \mathsf{SomeAnimal}(x)$$

The first logical statement reads 'for all x and y, if x has a pair of ys as body parts then x is some kind of animal' whereas the second states 'for all x, if there exists a y such that x has a pair of ys as body parts then x is some kind of animal'. Crucially, in the second form the scope of the existential quantifier is the if-part of the rule, whereas universal quantifiers always range over the whole clause. Variables occurring in the body but not in the head of Prolog clauses are called *local variables*; they are the source of much additional complexity in learning first-order rules over propositional rules.

If we want to learn an ordered list of Prolog clauses, we can reuse ☞*LearnRuleList* (Algorithm 6.1 on p.163) in its entirety and most of ☞*LearnRule* (Algorithm 6.2 on p.164). What needs adjusting is the choice of literal to be added to the clause. Possible literals can be enumerated by listing the predicates, functors and constants that can be used to build a new literal. For example, if we have a binary predicate `bodyPart`, a unary functor `pairOf` and constants `gill` and `tail`, then we can build a variety of literals such as

```
bodyPart(X,Y)
bodyPart(X,gill)
bodyPart(X,tail)
bodyPart(X,pairOf(Y))
bodyPart(X,pairOf(gill))
bodyPart(X,pairOf(tail))
bodyPart(X,pairOf(pairOf(Y)))
bodyPart(X,pairOf(pairOf(gill)))
bodyPart(X,pairOf(pairOf(tail)))
```

and so on. Notice that the presence of functors means that our hypothesis language becomes infinite! Also, I have only listed literals that somehow 'made sense': there are many less sensible possibilities, such as `bodyPart(pairOf(gill),tail)` or `bodyPart(X,X)`, to name but a few. Although Prolog is an untyped language, many of these unwanted literals can be excluded by adding type information (in logic programming and ILP often done through 'mode declarations' which also specify particular input–output patterns of a predicate's arguments).

It is clear from these examples that there can be relationships between literals, and therefore between the clauses that contain them. For example, consider the following three clauses:

```
fish(X):-bodyPart(X,Y).
fish(X):-bodyPart(X,pairOf(Z)).
fish(X):-bodyPart(X,pairOf(gill)).
```

The first clause defines everything with some body part to be a fish. The second clause specialises this to everything with a pair of unspecified body parts. The third specialises this to everything with a pair of gills. A reasonable search strategy would be to try hypotheses in this order, and only move on to a specialised version if the more general clause is ruled out by negative examples. This is what in fact happens in top–down ILP systems. A simple trick is to represent substitution of terms for variables explicitly by adding equality literals, so the above sequence of clauses becomes

```
fish(X):-bodyPart(X,Y).
fish(X):-bodyPart(X,Y),Y=pairOf(Z).
fish(X):-bodyPart(X,Y),Y=pairOf(Z),Z=gill.
```

As an alternative for enumerating the literals to be considered for inclusion in a clause body we can derive them from the data in a bottom–up fashion. Suppose we have the following information about a dolphin:

```
bodyPart(dolphin42,tail).
bodyPart(dolphin42,pairOf(gills)).
bodyPart(dolphin42,pairOf(eyes)).
```

and this about a tunafish:

```
bodyPart(tuna123,pairOf(gills)).
```

By forming the LGG of each of the literals in the first example with the literal from the second example we obtain each of the generalised literals considered earlier.

This short discussion of rule learning in first-order logic has left out many important details and may therefore give an overly simplified view of the problem. While the problem of learning Prolog clauses can be stated quite succinctly, naive approaches are computationally intractable and 'the devil is in the detail'. The basic approaches sketched here can be extended to include background knowledge, which then affects the generality ordering of the hypothesis space. For example, if our background knowledge includes the clause

```
bodyPart(X,scales):-bodyPart(X,pairOf(gill)).
```

then the first of the following two hypotheses is more general than the second:

```
fish(X):-bodyPart(X,scales).
fish(X):-bodyPart(X,pairOf(gill)).
```

However, this cannot be determined purely by syntactic means and requires logical inference.

Another intriguing possibility offered by first-order logic is the possibility of learning recursive clauses. For instance, part of our hypothesis could be the following clause:

```
fish(X):-relatedSpecies(X,Y),fish(Y).
```

This blurs the distinction between background predicates that can be used in the body of hypotheses and target predicates that are to be learned, and introduces computational challenges such as non-termination. However, this doesn't mean that it cannot be done. Related techniques can be used to learn multiple, interrelated predicates at once, and to invent new background predicates that are completely unobserved.

6.5 Rule models: Summary and further reading

In a decision tree, a branch from root to a leaf can be interpreted as a conjunctive classification rule. Rule models generalise this by being more flexible about the way in which several rules are combined into a model. The typical rule learning algorithm is the covering algorithm, which iteratively learns one rule and then removes the examples covered by that rule. This approach was pioneered by Michalski (1975) with his AQ system, which became highly developed over three decades (Wojtusiak *et al.*, 2006). General overviews are provided by Fürnkranz (1999, 2010) and Fürnkranz, Gamberger and Lavrač (2012). Coverage plots were first used by Fürnkranz and Flach (2005) to achieve a better understanding of rule learning algorithms and demonstrate the close relationship (and in many cases, equivalence) of commonly used search heuristics.

- ☞ Rules can overlap and thus we need a strategy to resolve potential conflicts between rules. One such strategy is to combine the rules in an ordered rule list, which was the subject of Section 6.1. Rivest (1987) compares this approach with decision trees, calling the rule-based model a decision list (I prefer the term 'rule list' as it doesn't carry a suggestion that the elements of the list are single literals). Well-known rule list learners include CN2 (Clark and Niblett, 1989) and Ripper (Cohen, 1995), the latter being particularly effective at avoiding overfitting through incremental reduced-error pruning (Fürnkranz and Widmer, 1994). Also notable is the Opus system (Webb, 1995), which distinguishes itself by performing a complete search through the space of all possible rules.

- ☞ In Section 6.2 we looked at unordered rule sets as an alternative to ordered rule lists. The covering algorithm is adapted to learn rules for a single class at a time, and to remove only covered examples of the class currently under consideration. CN2 can be run in unordered mode to learn rule sets (Clark and Boswell, 1991). Conceptually, both rule lists and rule sets are special cases of rule trees, which distinguish all possible Boolean combinations of a given set of rules. This allows us to see that rule lists lead to fewer instance space segments than rule sets (over the set of rules); on the other hand, rule list coverage curves can be made convex on the training set, whereas rule sets need to estimate the class distribution in the regions where rules overlap.

- ☞ Rule models can be used for descriptive tasks, and in Section 6.3 we considered rule learning for subgroup discovery. The weighted covering algorithm was introduced as an adaption of CN2 by Lavrač, Kavšek, Flach and Todorovski (2004); Abudawood and Flach (2009) generalise this to more than two classes. Algorithm 6.7 learns association rules and is adapted from the well-known Apriori algorithm due to Agrawal, Mannila, Srikant, Toivonen and Verkamo (1996). There is a very wide choice of alternative algorithms, surveyed by Han *et al.* (2007). Association rules can also be used to build effective classifiers (Liu *et al.*, 1998; Li *et al.*, 2001).

☞ The topic of first-order rule learning briefly considered in Section 6.4 has been studied for the last 40 years and has a very rich history. De Raedt (2008) provides an excellent recent introduction, and an overview of recent advances and open problems is provided by Muggleton *et al.* (2012). Flach (1994) gives an introduction to Prolog and also provides high-level implementations of some of the key techniques in inductive logic programming. The FOIL system by Quinlan (1990) implements a top–down learning algorithm similar to the one discussed here. The bottom–up technique was pioneered in the Golem system (Muggleton and Feng, 1990) and further refined in Progol (Muggleton, 1995) and in Aleph (Srinivasan, 2007), two of the most widely used ILP systems. First-order rules can also be learned in an unsupervised fashion, for example by Tertius which learns first-order clauses (not necessarily Horn) (Flach and Lachiche, 2001) and Warmr which learns first-order association rules (King *et al.*, 2001). Higher-order logic provides more powerful data types that can be highly beneficial in learning (Lloyd, 2003). A more recent development is the combination of probabilistic modelling with first-order logic, leading to the area of statistical relational learning (De Raedt and Kersting, 2010).