
Tree models

TREE MODELS ARE among the most popular models in machine learning. For example, the pose recognition algorithm in the Kinect motion sensing device for the Xbox game console has decision tree classifiers at its heart (in fact, an ensemble of decision trees called a random forest about which you will learn more in [Chapter 11](#)). Trees are expressive and easy to understand, and of particular appeal to computer scientists due to their recursive ‘divide-and-conquer’ nature.

In fact, the paths through the logical hypothesis space discussed in the previous chapter already constitute a very simple kind of tree. For instance, the feature tree in [Figure 5.1 \(left\)](#) is equivalent to the path in [Figure 4.6 \(left\)](#) on p.117. This equivalence is best seen by tracing the path and the tree from the bottom upward.

1. The left-most leaf of the feature tree represents the concept at the bottom of the path, covering a single positive example.
2. The next concept up in the path generalises the literal `Length = 3` into `Length = [3, 5]` by means of internal disjunction; the added coverage (one positive example) is represented by the second leaf from the left in the feature tree.
3. By dropping the condition `Teeth = few` we add another two covered positives.
4. Dropping the ‘Length’ condition altogether (or extending the internal disjunction with the one remaining value ‘4’) adds the last positive, and also a negative.
5. Dropping `Beak = yes` covers no additional examples (remember the discussion

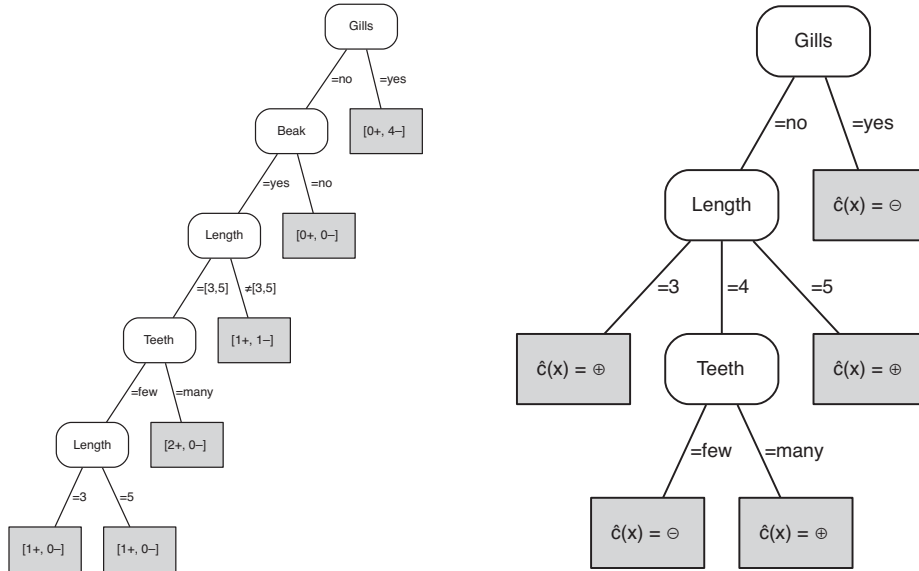


Figure 5.1. (left) The path from Figure 4.6 on p.117, redrawn in the form of a tree. The coverage numbers in the leaves are obtained from the data in Example 4.4. **(right)** A decision tree learned on the same data. This tree separates the positives and negatives perfectly.

about closed concepts in the previous chapter).

6. Finally, dropping *Gills* = no covers the four remaining negatives.

We see that a path through the hypothesis space can be turned into an equivalent feature tree. To obtain a tree that is equivalent to the i -th concept from the bottom in the path, we can either truncate the tree by combining the left-most i leaves into a single leaf representing the concept; or we can label the left-most i leaves positive and the remaining leaves negative, turning the feature tree into a decision tree.

Decision trees do not employ internal disjunction for features with more than two values, but instead allow branching on each separate value. They also allow leaf labellings that do not follow the left-to-right order of the leaves. Such a tree is shown in Figure 5.1 (right). This tree can be turned into a logical expression in many different ways, including:

$$\begin{aligned}
 & (\text{Gills} = \text{no} \wedge \text{Length} = 3) \vee (\text{Gills} = \text{no} \wedge \text{Length} = 4 \wedge \text{Teeth} = \text{many}) \\
 & \quad \vee (\text{Gills} = \text{no} \wedge \text{Length} = 5) \\
 & \text{Gills} = \text{no} \wedge [\text{Length} = 3 \vee (\text{Length} = 4 \wedge \text{Teeth} = \text{many}) \vee \text{Length} = 5] \\
 & \quad \neg [(\text{Gills} = \text{no} \wedge \text{Length} = 4 \wedge \text{Teeth} = \text{few}) \vee \text{Gills} = \text{yes}] \\
 & (\text{Gills} = \text{yes} \vee \text{Length} = [3, 5] \vee \text{Teeth} = \text{many}) \wedge \text{Gills} = \text{no}
 \end{aligned}$$

The first expression is in disjunctive normal form (DNF, see Background 4.1 on p.105)

and is obtained by forming a disjunction of all paths from the root of the tree to leaves labelled positive, where each path gives a conjunction of literals. The second expression is a simplification of the first using the distributive equivalence $(A \wedge B) \vee (A \wedge C) \equiv A \wedge (B \vee C)$. The third expression is obtained by first forming a DNF expression representing the negative class, and then negating it. The fourth expression turns this into CNF by using the De Morgan laws $\neg(A \wedge B) \equiv \neg A \vee \neg B$ and $\neg(A \vee B) \equiv \neg A \wedge \neg B$.


There are many other logical expressions that are equivalent to the concept defined by the decision tree. Perhaps it would be possible to obtain an equivalent conjunctive concept? Interestingly, the answer to this question is no: some decision trees represent a conjunctive concept, but many trees don't and this is one of them.¹ *Decision trees are strictly more expressive than conjunctive concepts.* In fact, since decision trees correspond to DNF expressions, and since every logical expression can be equivalently written in DNF, it follows that decision trees are maximally expressive: the only data that they cannot separate is data that is inconsistently labelled, i.e., the same instance appears twice with different labels. This explains why data that isn't conjunctively separable, as in our example, can be separated by a decision tree.

There is a potential problem with using such an expressive hypothesis language. Let Δ be the disjunction of all positive examples, then Δ is in disjunctive normal form. Δ clearly covers all positives – in fact, Δ 's extension is exactly the set of positive examples. In other words, in the hypothesis space of DNF expressions (or of decision trees), Δ is the LGG of the positive examples, but it doesn't cover any other instances. So Δ does not generalise beyond the positive examples, but merely memorises them – talk about overfitting! Turning this argument around, we see that *one way to avoid overfitting and encourage learning is to deliberately choose a restrictive hypothesis language*, such as conjunctive concepts: in such a language, even the LGG operation typically generalises beyond the positive examples. And if our language is expressive enough to represent any set of positive examples, we must make sure that the learning algorithm employs other mechanisms to force generalisation beyond the examples and avoid overfitting – this is called the *inductive bias* of the learning algorithm. As we will see, most learning algorithms that operate in expressive hypothesis spaces have an inductive bias towards less complex hypotheses, either implicitly through the way the hypothesis space is searched, or explicitly by incorporating a complexity penalty in the objective function.

Tree models are not limited to classification but can be employed to solve almost any machine learning task, including ranking and probability estimation, regression and clustering. The tree structure that is common to all those models can be defined

¹ If we allowed the creation of new conjunctive features, we could actually represent this tree as the conjunctive concept $\text{Gills} = \text{no} \wedge F = \text{false}$, where $F \equiv \text{Length} = 4 \wedge \text{Teeth} = \text{few}$ is a new conjunctive feature. The creation of new features during learning is called *constructive induction*, and as shown here can extend the representational power of a logical language.

as follows.

Definition 5.1 (Feature tree). A **feature tree** is a tree such that each internal node (the nodes that are not leaves) is labelled with a feature, and each edge emanating from an internal node is labelled with a literal. The set of literals at a node is called a **split**. Each leaf of the tree represents a logical expression, which is the conjunction of literals encountered on the path from the root of the tree to the leaf. The extension of that conjunction (the set of instances covered by it) is called the **instance space segment** associated with the leaf. 

Essentially, a feature tree is a compact way of representing a number of conjunctive concepts in the hypothesis space. The learning problem is then to decide which of the possible concepts will be best to solve the given task. While rule learners (discussed in the next chapter) essentially learn these concepts one at a time, tree learners perform a top-down search for all these concepts at once.

Algorithm 5.1 gives the generic learning procedure common to most tree learners. It assumes that the following three functions are defined:

Homogeneous(D) returns true if the instances in D are homogeneous enough to be labelled with a single label, and false otherwise;

Label(D) returns the most appropriate label for a set of instances D ;

BestSplit(D, F) returns the best set of literals to be put at the root of the tree.

These functions depend on the task at hand: for instance, for classification tasks a set of instances is homogeneous if they are (mostly) of a single class, and the most appropriate label would be the majority class. For clustering tasks a set of instances is homogenous if they are close together, and the most appropriate label would be some exemplar such as the mean (more on exemplars in Chapter 8).

Algorithm 5.1: GrowTree(D, F) – grow a feature tree from training data.

Input : data D ; set of features F .

Output : feature tree T with labelled leaves.

```

1 if Homogeneous( $D$ ) then return Label( $D$ ) ;           // Homogeneous, Label: see text
2  $S \leftarrow$  BestSplit( $D, F$ ) ;                         // e.g., BestSplit-Class (Algorithm 5.2)
3 split  $D$  into subsets  $D_i$  according to the literals in  $S$ ;
4 for each  $i$  do
5   | if  $D_i \neq \emptyset$  then  $T_i \leftarrow$  GrowTree( $D_i, F$ ) else  $T_i$  is a leaf labelled with Label( $D$ );
6 end
7 return a tree whose root is labelled with  $S$  and whose children are  $T_i$ 
```

Algorithm 5.1 is a *divide-and-conquer* algorithm: it divides the data into subsets, builds a tree for each of those and then combines those subtrees into a single tree. Divide-and-conquer algorithms are a tried-and-tested technique in computer science. They are usually implemented recursively, because each subproblem (to build a tree for a subset of the data) is of the same form as the original problem. This works as long as there is a way to stop the recursion, which is what the first line of the algorithm does. However, it should be noted that such algorithms are *greedy*: whenever there is a choice (such as choosing the best split), the best alternative is selected on the basis of the information then available, and this choice is never reconsidered. This may lead to sub-optimal choices. An alternative would be to use a *backtracking search* algorithm, which can return an optimal solution, at the expense of increased computation time and memory requirements, but we will not explore that further in this book.

In the remainder of this chapter we will instantiate the generic Algorithm 5.1 to classification, ranking and probability estimation, clustering and regression tasks.

5.1 Decision trees

As already indicated, for a classification task we can simply define a set of instances D to be homogenous if they are all from the same class, and the function $\text{Label}(D)$ will then obviously return that class. Notice that in line 5 of Algorithm 5.1 we may be calling $\text{Label}(D)$ with a non-homogeneous set of instances in case one of the D_i is empty, so the general definition of $\text{Label}(D)$ is that it returns the majority class of the instances in D .² This leaves us to decide how to define the function $\text{BestSplit}(D, F)$.

Let's assume for the moment that we are dealing with Boolean features, so D is split into D_1 and D_2 . Let's also assume we have two classes, and denote by D^{\oplus} and D^{\ominus} the positives and negatives in D (and likewise for D_1^{\oplus} etc.). The question is how to assess the utility of a feature in terms of splitting the examples into positives and negatives. Clearly, the best situation is where $D_1^{\oplus} = D^{\oplus}$ and $D_1^{\ominus} = \emptyset$, or where $D_1^{\oplus} = \emptyset$ and $D_1^{\ominus} = D^{\ominus}$. In that case, the two children of the split are said to be *pure*. So we need to measure the impurity of a set of n^{\oplus} positives and n^{\ominus} negatives. One important principle that we will adhere to is that the impurity should only depend on the relative magnitude of n^{\oplus} and n^{\ominus} , and should not change if we multiply both with the same amount. This in turn means that impurity can be defined in terms of the proportion $\hat{p} = n^{\oplus} / (n^{\oplus} + n^{\ominus})$, which we remember from Section 2.2 as the *empirical probability* of the positive class. Furthermore, impurity should not change if we swap the positive and negative class, which means that it should stay the same if we replace \hat{p} with $1 - \hat{p}$. We also want a function that is 0 whenever $\hat{p} = 0$ or $\hat{p} = 1$ and that reaches its maximum

²If there is more than one largest class we will make an arbitrary choice between them, usually uniformly random.

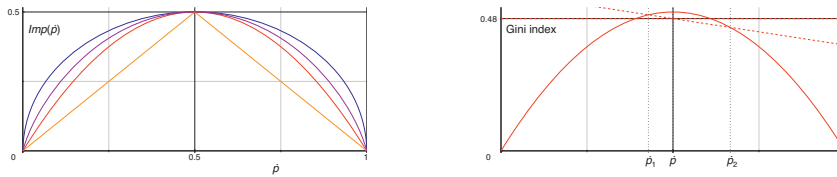


Figure 5.2. (left) Impurity functions plotted against the empirical probability of the positive class. From the bottom: the relative size of the minority class, $\min(\dot{p}, 1 - \dot{p})$; the Gini index, $2\dot{p}(1 - \dot{p})$; entropy, $-\dot{p}\log_2 \dot{p} - (1 - \dot{p})\log_2(1 - \dot{p})$ (divided by 2 so that it reaches its maximum in the same point as the others); and the (rescaled) square root of the Gini index, $\sqrt{\dot{p}(1 - \dot{p})}$ – notice that this last function describes a semi-circle. **(right)** Geometric construction to determine the impurity of a split (**Teeth** = [many, few] from [Example 5.1](#)): \dot{p} is the empirical probability of the parent, and \dot{p}_1 and \dot{p}_2 are the empirical probabilities of the children.

for $\dot{p} = 1/2$. The following functions all fit the bill.

Minority class $\min(\dot{p}, 1 - \dot{p})$ – this is sometimes referred to as the error rate, as it measures the proportion of misclassified examples if the leaf was labelled with the majority class; the purer the set of examples, the fewer errors this will make. This impurity measure can equivalently be written as $1/2 - |\dot{p} - 1/2|$.

Gini index $2\dot{p}(1 - \dot{p})$ – this is the expected error if we label examples in the leaf randomly: positive with probability \dot{p} and negative with probability $1 - \dot{p}$. The probability of a false positive is then $\dot{p}(1 - \dot{p})$ and the probability of a false negative $(1 - \dot{p})\dot{p}$.³

entropy $-\dot{p}\log_2 \dot{p} - (1 - \dot{p})\log_2(1 - \dot{p})$ – this is the expected information, in bits, conveyed by somebody telling you the class of a randomly drawn example; the purer the set of examples, the more predictable this message becomes and the smaller the expected information.

A plot of these three impurity measures can be seen in [Figure 5.2 \(left\)](#), some of them rescaled so that they all reach their maximum at (0.5, 0.5). I have added a fourth one: the square root of the Gini index, which I will indicate as $\sqrt{\text{Gini}}$, and which has an advantage over the others, as we will see later. Indicating the impurity of a single leaf D_j as $\text{Imp}(D_j)$, the impurity of a set of mutually exclusive leaves $\{D_1, \dots, D_l\}$ is then

³When I looked up ‘Gini index’ on Wikipedia I was referred to a page describing the *Gini coefficient*, which – in a machine learning context – is a linear rescaling of the *AUC* to the interval $[-1, 1]$. This is quite a different concept, and the only thing that the Gini index and the Gini coefficient have in common is that they were both proposed by the Italian statistician Corrado Gini, so it is good to be aware of potential confusion.

defined as a weighted average

$$\text{Imp}(\{D_1, \dots, D_l\}) = \sum_{j=1}^l \frac{|D_j|}{|D|} \text{Imp}(D_j) \quad (5.1)$$

where $D = D_1 \cup \dots \cup D_l$. For a binary split there is a nice geometric construction to find $\text{Imp}(\{D_1, D_2\})$ given the empirical probabilities of the parent and the children, which is illustrated in Figure 5.2 (right):

1. We first find the impurity values $\text{Imp}(D_1)$ and $\text{Imp}(D_2)$ of the two children on the impurity curve (here the Gini index).
2. We then connect these two values by a straight line, as any weighted average of the two must be on that line.
3. Since the empirical probability of the parent is also a weighted average of the empirical probabilities of the children, with the same weights (i.e., $\hat{p} = \frac{|D_1|}{|D|} \hat{p}_1 + \frac{|D_2|}{|D|} \hat{p}_2$ – the derivation is given in Equation 5.2 on p.139), \hat{p} gives us the correct interpolation point.

This construction will work with any of the impurity measures plotted in Figure 5.2 (left). Note that, if the class distribution in the parent is very skewed, the empirical probability of both children may end up to the left or to the right of the $\hat{p} = 0.5$ vertical. This isn't a problem – except for the minority class impurity measure, as the geometric construction makes it clear that all such splits will be evaluated as having the same weighted average impurity. For this reason its use as an impurity measure is often discouraged.

Example 5.1 (Calculating impurity). Consider again the data in Example 4.4 on p.115. We want to find the best feature to put at the root of the decision tree. The four features available result in the following splits:

Length = [3,4,5] [2+,0-][1+,3-][2+,2-]
 Gills = [yes,no] [0+,4-][5+,1-]
 Beak = [yes,no] [5+,3-][0+,2-]
 Teeth = [many,few] [3+,4-][2+,1-]

Let's calculate the impurity of the first split. We have three segments: the first one is pure and so has entropy 0; the second one has entropy $-(1/4)\log_2(1/4) - (3/4)\log_2(3/4) = 0.5 + 0.31 = 0.81$; the third one has entropy 1. The total entropy is then the weighted average of these, which is $2/10 \cdot 0 + 4/10 \cdot 0.81 + 4/10 \cdot 1 = 0.72$.

Similar calculations for the other three features give the following entropies:

$$\begin{aligned}
\text{Gills} & 4/10 \cdot 0 + 6/10 \cdot \left(-(5/6) \log_2(5/6) - (1/6) \log_2(1/6) \right) = 0.39; \\
\text{Beak} & 8/10 \cdot \left(-(5/8) \log_2(5/8) - (3/8) \log_2(3/8) \right) + 2/10 \cdot 0 = 0.76; \\
\text{Teeth} & 7/10 \cdot \left(-(3/7) \log_2(3/7) - (4/7) \log_2(4/7) \right) \\
& + 3/10 \cdot \left(-(2/3) \log_2(2/3) - (1/3) \log_2(1/3) \right) = 0.97.
\end{aligned}$$

We thus clearly see that ‘Gills’ is an excellent feature to split on; ‘Teeth’ is poor; and the other two are somewhere in between.

The calculations for the Gini index are as follows (notice that these are on a scale from 0 to 0.5):

$$\begin{aligned}
\text{Length} & 2/10 \cdot 2 \cdot (2/2 \cdot 0/2) + 4/10 \cdot 2 \cdot (1/4 \cdot 3/4) + 4/10 \cdot 2 \cdot (2/4 \cdot 2/4) = 0.35; \\
\text{Gills} & 4/10 \cdot 0 + 6/10 \cdot 2 \cdot (5/6 \cdot 1/6) = 0.17; \\
\text{Beak} & 8/10 \cdot 2 \cdot (5/8 \cdot 3/8) + 2/10 \cdot 0 = 0.38; \\
\text{Teeth} & 7/10 \cdot 2 \cdot (3/7 \cdot 4/7) + 3/10 \cdot 2 \cdot (2/3 \cdot 1/3) = 0.48.
\end{aligned}$$

As expected, the two impurity measures are in close agreement. See [Figure 5.2 \(right\)](#) for a geometric illustration of the last calculation concerning ‘Teeth’.

Adapting these impurity measures to $k > 2$ classes is done by summing the per-class impurities in a one-versus-rest manner. In particular, k -class entropy is defined as $\sum_{i=1}^k -\dot{p}_i \log_2 \dot{p}_i$, and the k -class Gini index as $\sum_{i=1}^k \dot{p}_i(1 - \dot{p}_i)$. In assessing the quality of a feature for splitting a parent node D into leaves D_1, \dots, D_l , it is customary to look at the purity gain $\text{Imp}(D) - \text{Imp}(\{D_1, \dots, D_l\})$. If purity is measured by entropy, this is called the *information gain* splitting criterion, as it measures the increase in information about the class gained by including the feature. However, note that [Algorithm 5.1](#) only compares splits with the same parent, and so we can ignore the impurity of the parent and search for the feature which results in the lowest weighted average impurity of its children ([Algorithm 5.2](#)).

We now have a fully instantiated decision tree learning algorithm, so let’s see what tree it learns on our dolphin data. We have already seen that the best feature to split on at the root of the tree is ‘Gills’: the condition **Gills = yes** leads to a pure leaf $[0+, 4-]$ labelled negative, and a predominantly positive child $[5+, 1-]$. For the next split we have the choice between ‘Length’ and ‘Teeth’, as splitting on ‘Beak’ does not decrease the impurity. ‘Length’ results in a $[2+, 0-][1+, 1-][2+, 0-]$ split and ‘Teeth’ in a $[3+, 0-][2+, 1-]$ split; both entropy and Gini index consider the former purer than the latter. We then use ‘Teeth’ to split the one remaining impure node. The resulting tree is the one shown previously in [Figure 5.1](#) on [p.130](#), and reproduced in [Figure 5.3 \(left\)](#). We have learned

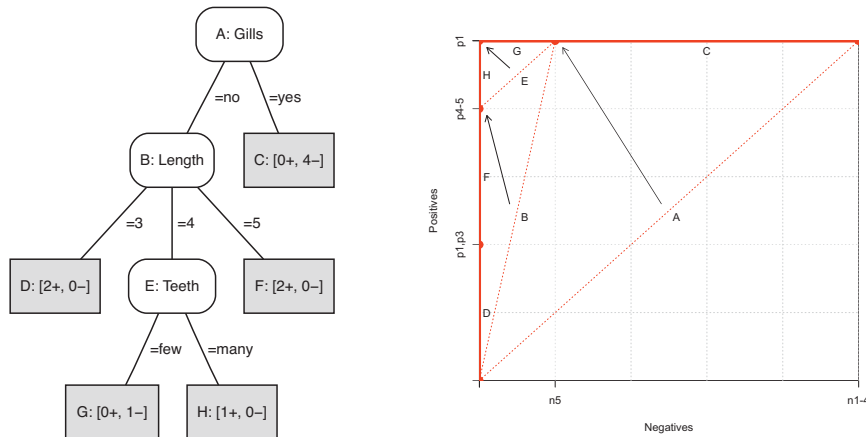


Figure 5.3. (left) Decision tree learned from the data in Example 4.4 on p.115. **(right)** Each internal and leaf node of the tree corresponds to a line segment in coverage space: vertical segments for pure positive nodes, horizontal segments for pure negative nodes, and diagonal segments for impure nodes.

our first decision tree!

The tree represents a partition of the instance space, and therefore also assigns a class to the 14 instances that were not part of the training set – which is why we can say that the tree generalises the training data. Leaf C leaves three feature values unspecified, with a total of $3 \cdot 2 \cdot 2 = 12$ possible combinations of values; four of these were supplied as training examples, so leaf C covers eight unlabelled instances and classifies them as negative. Similarly, two unlabelled instances are classified as positive by leaf

Algorithm 5.2: **BestSplit-Class**(D, F) – find the best split for a decision tree.

Input : data D ; set of features F .

Output : feature f to split on.

```

1  $I_{\min} \leftarrow 1$ ;
2 for each  $f \in F$  do
3   split  $D$  into subsets  $D_1, \dots, D_l$  according to the values  $v_j$  of  $f$ ;
4   if  $\text{Imp}(\{D_1, \dots, D_l\}) < I_{\min}$  then
5      $I_{\min} \leftarrow \text{Imp}(\{D_1, \dots, D_l\})$ ;
6      $f_{\text{best}} \leftarrow f$ ;
7   end
8 end
9 return  $f_{\text{best}}$ 

```

D, and a further two by leaf F; one is classified as negative by leaf G, and the remaining one as positive by leaf H. The fact that more unlabelled instances are classified as negative (9) than as positive (5) is thus mostly due to leaf C: because it is a leaf high up in the tree, it covers many instances. One could argue that the fact that four out of five negatives have gills is the strongest regularity found in the data.


It is also worth tracing the construction of this tree in coverage space (Figure 5.3 (right)). Every node of the tree, internal or leaf, covers a certain number of positives and negatives and hence can be plotted as a line segment in coverage space. For instance, the root of the tree covers all positives and all negatives, and hence is represented by the ascending diagonal A. Once we add our first split, segment A is replaced by segment B (an impure node and hence diagonal) and segment C, which is pure and not split any further. Segment B is further split into D (pure and positive), E (impure) and F (pure and positive). Finally, E is split into two pure nodes.

This idea of a decision tree coverage curve ‘pulling itself up’ from the ascending diagonal in a divide-and-conquer fashion is appealing – but unfortunately it is not true in general. The ordering of coverage curve segments is purely based on the class distributions in the leaves and does not bear any direct relationship to the tree structure. To understand this better, we will now look at how tree models can be turned into rankers and probability estimators.

5.2 Ranking and probability estimation trees

Grouping classifiers such as decision trees divide the instance space into segments, and so can be turned into rankers by learning an ordering on those segments. Unlike some other grouping models, decision trees have access to the local class distributions in the segments or leaves, which can directly be used to construct a leaf ordering that is optimal for the training data. So, for instance, in Figure 5.3 this ordering is [D – F] – H – G – C, resulting in a perfect ranking ($AUC = 1$). The ordering can simply be obtained from the empirical probabilities \hat{p} , breaking ties as much as possible by giving precedence to leaves covering a larger number of positives.⁴ Why is this ordering optimal? Well, the slope of a coverage curve segment with empirical probability \hat{p} is $\hat{p}/(1 - \hat{p})$; since $\hat{p} \mapsto \frac{\hat{p}}{1 - \hat{p}}$ is a monotonic transformation (if $\hat{p} > \hat{p}'$ then $\frac{\hat{p}}{1 - \hat{p}} > \frac{\hat{p}'}{1 - \hat{p}'}$) sorting the segments on non-increasing empirical probabilities ensures that they are also sorted on non-increasing slope, and so the curve is convex. This is an important point, so I’ll say it again: *the ranking obtained from the empirical probabilities in the leaves of a decision tree yields a convex ROC curve on the training data*. As we shall see later in

⁴Tie breaking – although it does not alter the shape of the coverage curve and isn’t essential in that sense – can also be achieved by subtracting $\epsilon \ll 1$ from the number of positives covered. The Laplace correction also breaks ties in favour of larger leaves but isn’t a monotonic transformation and so might change the shape of the coverage curve.

the book, some other grouping models including  *rule lists* (Section 6.1) share this property, but no grading model does.



As already noted, the segment ordering cannot be deduced from the tree structure. The reason is essentially that, even if we know the empirical probability associated with the parent of a split, this doesn't constrain the empirical probabilities of its children. For instance, let $[n^{\oplus}, n^{\ominus}]$ be the class distribution in the parent with $n = n^{\oplus} + n^{\ominus}$, and let $[n_1^{\oplus}, n_1^{\ominus}]$ and $[n_2^{\oplus}, n_2^{\ominus}]$ be the class distributions in the children, with $n_1 = n_1^{\oplus} + n_1^{\ominus}$ and $n_2 = n_2^{\oplus} + n_2^{\ominus}$. We then have

$$\dot{p} = \frac{n^{\oplus}}{n} = \frac{n_1}{n} \frac{n_1^{\oplus}}{n_1} + \frac{n_2}{n} \frac{n_2^{\oplus}}{n_2} = \frac{n_1}{n} \dot{p}_1 + \frac{n_2}{n} \dot{p}_2 \quad (5.2)$$

In other words, the empirical probability of the parent is a weighted average of the empirical probabilities of its children; but this only tells us that $\dot{p}_1 \leq \dot{p} \leq \dot{p}_2$ or $\dot{p}_2 \leq \dot{p} \leq \dot{p}_1$. Even if the place of the parent segment in the coverage curve is known, its children may come much earlier or later in the ordering.

Example 5.2 (Growing a tree). Consider the tree in Figure 5.4 (top). Each node is labelled with the numbers of positive and negative examples covered by it: so, for instance, the root of the tree is labelled with the overall class distribution (50 positives and 100 negatives), resulting in the trivial ranking [50+, 100−]. The corresponding one-segment coverage curve is the ascending diagonal (Figure 5.4 (bottom)). Adding split (1) refines this ranking into [30+, 35−][20+, 65−], resulting in a two-segment curve. Adding splits (2) and (3) again breaks up the segment corresponding to the parent into two segments corresponding to the children. However, the ranking produced by the full tree – [15+, 3−][29+, 10−][5+, 62−][1+, 25−] – is different from the left-to-right ordering of its leaves, hence we need to reorder the segments of the coverage curve, leading to the top-most, solid curve.

So, adding a split to a decision tree can be interpreted in terms of coverage curves as the following two-step process:

-  split the corresponding curve segment into two or more segments;
-  reorder the segments on decreasing slope.

The whole process of growing a decision tree can be understood as an iteration of these two steps; or alternatively as a sequence of splitting steps followed by one overall re-ordering step. It is this last step that guarantees that the coverage curve is convex (on the training data).

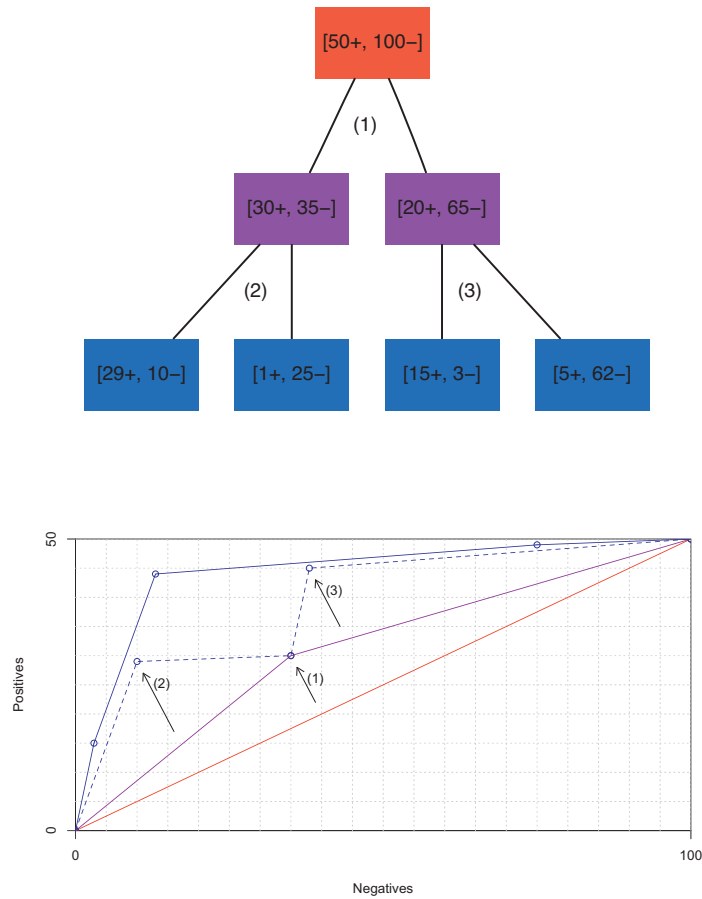


Figure 5.4. (top) Abstract representation of a tree with numbers of positive and negative examples covered in each node. Binary splits are added to the tree in the order indicated. **(bottom)** Adding a split to the tree will add new segments to the coverage curve as indicated by the arrows. After a split is added the segments may need reordering, and so only the solid lines represent actual coverage curves.

It is instructive to take this analysis a step further by considering all possible rankings that can be constructed with the given tree. One way to do that is to consider the tree as a feature tree, without any class labels, and ask ourselves in how many ways we can label the tree, and what performance that would yield, given that we know the numbers of positives and negatives covered in each leaf. In general, if a feature tree has l leaves and we have c classes, then the number of possible labellings of leaves with classes is c^l ; in the example of Figure 5.4 this is $2^4 = 16$. Figure 5.5 depicts these 16 labellings in coverage space. As you might expect, there is a lot of symmetry in this

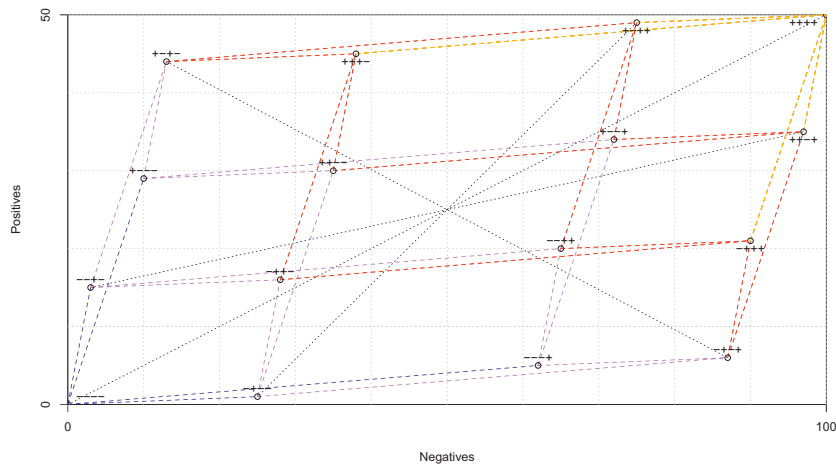


Figure 5.5. Graphical depiction of all possible labellings and all possible rankings that can be obtained with the four-leaf decision tree in Figure 5.4. There are $2^4 = 16$ possible leaf labellings; e.g., ‘+--+’ denotes labelling the first and third leaf from the left as + and the second and fourth leaf as -. Also indicated are some pairwise symmetries (dotted lines): e.g., +--+ and -+-+ are each other’s inverse and end up at opposite ends of the plot. There are $4! = 24$ possible blue-violet-red-orange paths through these points which start in ---- and switch each leaf to + in some order; these represent all possible four-segment coverage curves or rankings.

plot. For instance, labellings occur in pairs (say +--+ and -+-+) that occur in opposite locations in the plot (see if you can figure out what is meant by ‘opposite’ here). We obtain a ranking by starting in ---- in the lower left-hand corner, and switching each leaf to + in some order. For instance, the optimal coverage curve follows the order ----, --+-, +-+-, -+++, +++++. For a tree with l leaves there are $l!$ permutations of its leaves and thus $l!$ possible coverage curves (24 in our example).

If I were to choose a single image that would convey the essence of tree models, it would be Figure 5.5. What it visualises is that the class distributions in the leaves of an unlabelled feature tree can be used to turn one and the same tree into a decision tree, a ranking tree, or a probability estimation tree:

- 🔗 to turn a feature tree into a ranker, we order its leaves on non-increasing empirical probabilities, which is provably optimal on the training set;
- 🔗 to turn the tree into a probability estimator, we predict the empirical probabilities in each leaf, applying Laplace or m -estimate smoothing to make these estimates more robust for small leaves;
- 🔗 to turn the tree into a classifier, we choose the operating conditions and find the

operating point that is optimal under those operating conditions.

The last procedure was explained in [Section 2.2](#). We will illustrate it here, assuming the training set class ratio $clr = 50/100$ is representative. We have a choice of five labellings, depending on the expected cost ratio $c = c_{FN}/c_{FP}$ of misclassifying a positive in proportion to the cost of misclassifying a negative:

- $+ - + -$ would be the labelling of choice if $c = 1$, or more generally if $10/29 < c < 62/5$;
- $+ - ++$ would be chosen if $62/5 < c < 25/1$;
- $++++$ would be chosen if $25/1 < c$; i.e., we would always predict positive if false negatives are more than 25 times as costly as false positives, because then even predicting positive in the second leaf would reduce cost;
- $--+-$ would be chosen if $3/15 < c < 10/29$;
- $----$ would be chosen if $c < 3/15$; i.e., we would always predict negative if false positives are more than 5 times as costly as false negatives, because then even predicting negative in the third leaf would reduce cost.

The first of these options corresponds to the majority class labelling, which is what most textbook treatments of decision trees recommend, and also what I suggested when I discussed the function $\text{Label}(D)$ in the context of [Algorithm 5.1](#). In many circumstances this will indeed be the most practical thing to do. However, it is important to be aware of the underlying assumptions of such a labelling: these assumptions are that the training set class distribution is representative and the costs are uniform; or, more generally, that the product of the expected cost and class ratios is equal to the class ratio as observed in the training set. (This actually suggests a useful device for manipulating the training set to reflect an expected class ratio: to mimic an expected class ratio of c , we can oversample the positive training examples with a factor c if $c > 1$, or oversample the negatives with a factor $1/c$ if $c < 1$. We will return to this suggestion below.)

So let's assume that the class distribution is representative and that false negatives (e.g., not diagnosing a disease in a patient) are about 20 times more costly than false positives. As we have just seen, the optimal labelling under these operating conditions is $+ - ++$, which means that we only use the second leaf to filter out negatives. In other words, the right two leaves can be merged into one – their parent. Rather aptly, the operation of merging all leaves in a subtree is called *pruning* the subtree. The process is illustrated in [Figure 5.6](#). The advantage of pruning is that we can simplify the tree without affecting the chosen operating point, which is sometimes useful if we want to communicate the tree model to somebody else. The disadvantage is that we lose ranking performance, as illustrated in [Figure 5.6 \(bottom\)](#). Pruning is therefore not recommended unless [\(i\)](#) you only intend to use the tree for classification, not for

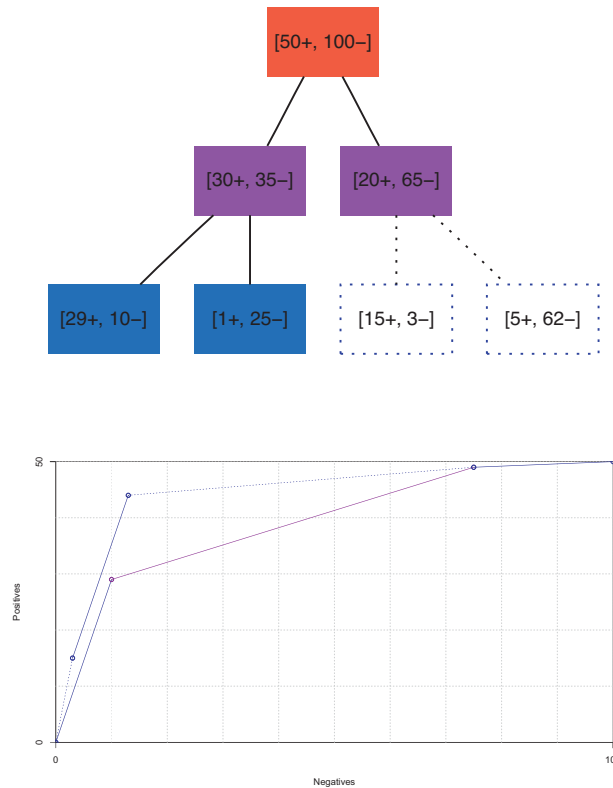


Figure 5.6. (top) To achieve the labelling $+ - ++$ we don't need the right-most split, which can therefore be pruned away. **(bottom)** Pruning doesn't affect the chosen operating point, but it does decrease the ranking performance of the tree.

ranking or probability estimation; and (ii) you can define the expected operating conditions with sufficient precision. One popular algorithm for pruning decision trees is called *reduced-error pruning*, and is given in Algorithm 5.3. The algorithm employs a separate *pruning set* of labelled data not seen during training, as pruning will never improve accuracy over the training data. However, if tree simplicity is not really an issue, I recommend keeping the entire tree intact and choosing the operating point through the leaf labelling only; this can similarly be done using a hold-out data set.

Sensitivity to skewed class distributions

I just mentioned in passing that one way to make sure the training set reflects the right operating conditions is to duplicate positives or negatives so that the training set class ratio is equal to the product of expected cost and class ratios in deployment of the model. Effectively, this changes the aspect ratio of the rectangle representing the cov-

erage space. The advantage of this method is that it is directly applicable to any model, without need to interfere with search heuristics or evaluation measures. The disadvantage is that it will increase training time – and besides, it may not actually make a difference for the model being learned. I will illustrate this with an example.

Example 5.3 (Cost-sensitivity of splitting criteria). Suppose you have 10 positives and 10 negatives, and you need to choose between the two splits $[8+, 2-][2+, 8-]$ and $[10+, 6-][0+, 4-]$. You duly calculate the weighted average entropy of both splits and conclude that the first split is the better one. Just to be sure, you also calculate the average Gini index, and again the first split wins. You then remember somebody telling you that the square root of the Gini index was a better impurity measure, so you decide to check that one out as well. Lo and behold, it favours the second split...! What to do?

You then remember that mistakes on the positives are about ten times as costly as mistakes on the negatives. You're not quite sure how to work out the maths, and so you decide to simply have ten copies of every positive: the splits are now $[80+, 2-][20+, 8-]$ and $[100+, 6-][0+, 4-]$. You recalculate the three splitting criteria and now all three favour the second split. Even though you're slightly bemused by all this, you settle for the second split since all three splitting criteria are now unanimous in their recommendation.

So what is going on here? Let's first look at the situation with the inflated numbers of positives. Intuitively it is clear that here the second split is preferable, since one of the children is pure and the other one is fairly good as well, though perhaps not as

Algorithm 5.3: $\text{PruneTree}(T, D)$ – reduced-error pruning of a decision tree.

Input : decision tree T ; labelled data D .

Output : pruned tree T' .

```

1 for every internal node  $N$  of  $T$ , starting from the bottom do
2    $T_N \leftarrow$  subtree of  $T$  rooted at  $N$ ;
3    $D_N \leftarrow \{x \in D \mid x \text{ is covered by } N\}$ ;
4   if accuracy of  $T_N$  over  $D_N$  is worse than majority class in  $D_N$  then
5     | replace  $T_N$  in  $T$  by a leaf labelled with the majority class in  $D_N$ ;
6   end
7 end
8 return pruned version of  $T$ 

```

good as [80+, 2-]. But this situation changes if we have only one-tenth of the number of positives, at least according to entropy and Gini index. Using notation introduced earlier, this can be understood as follows. The Gini index of the parent is $2 \frac{n^{\oplus}}{n} \frac{n^{\ominus}}{n}$, and the weighted Gini index of one of the children is $\frac{n_1}{n} 2 \frac{n_1^{\oplus}}{n_1} \frac{n_1^{\ominus}}{n_1}$. So the weighted impurity of the child *in proportion* to the parent's impurity is $\frac{n_1^{\oplus} n_1^{\ominus} / n_1}{n^{\oplus} n^{\ominus} / n}$; let's call this *relative impurity*. The same calculations for $\sqrt{\text{Gini}}$ give

☞ impurity of the parent: $\sqrt{\frac{n^{\oplus}}{n} \frac{n^{\ominus}}{n}};$

☞ weighted impurity of the child: $\frac{n_1}{n} \sqrt{\frac{n_1^{\oplus}}{n_1} \frac{n_1^{\ominus}}{n_1}};$

☞ relative impurity: $\sqrt{\frac{n_1^{\oplus} n_1^{\ominus}}{n^{\oplus} n^{\ominus}}}.$

The important thing to note is that this last ratio doesn't change if we multiply all numbers involving positives with a factor c . That is, $\sqrt{\text{Gini}}$ is designed to minimise relative impurity, and thus is insensitive to changes in class distribution. In contrast, relative impurity for the Gini index includes the ratio n_1/n , which changes if we inflate the number of positives. Something similar happens with entropy. As a result, these two splitting criteria emphasise children covering more examples.

A picture will help to explain this further. Just as accuracy and average recall have isometrics in coverage and ROC space, so do splitting criteria. Owing to their non-linear nature, these isometrics are curved rather than straight. They also occur on either side of the diagonal, as we can swap the left and right child without changing the quality of the split. One might imagine the impurity landscape as a mountain looked down on from above – the summit is a ridge along the ascending diagonal, representing the splits where the children have the same impurity as the parent. This mountain slopes down on either side and reaches ground level in ROC heaven as well as its opposite number ('ROC hell'), as this is where impurity is zero. The isometrics are the contour lines of this mountain – walks around it at constant elevation.

Consider [Figure 5.7 \(top\)](#). The two splits among which you needed to choose in [Example 5.3](#) (before inflating the positives) are indicated as points in this plot. I have drawn six isometrics in the top-left of the plot: two splits times three splitting criteria. A particular splitting criterion prefers the split whose isometric is the *highest* (closest to ROC heaven) of the two: you can see that only one of the three ($\sqrt{\text{Gini}}$) prefers the split on the top-right. [Figure 5.7 \(bottom\)](#) demonstrates how this changes when inflating the positives with a factor 10 (a coverage plot would run off the page here, so I have plotted this in ROC space with the grid indicating how the class distribution has changed). Now all three splitting criteria prefer the top-right split, because the entropy

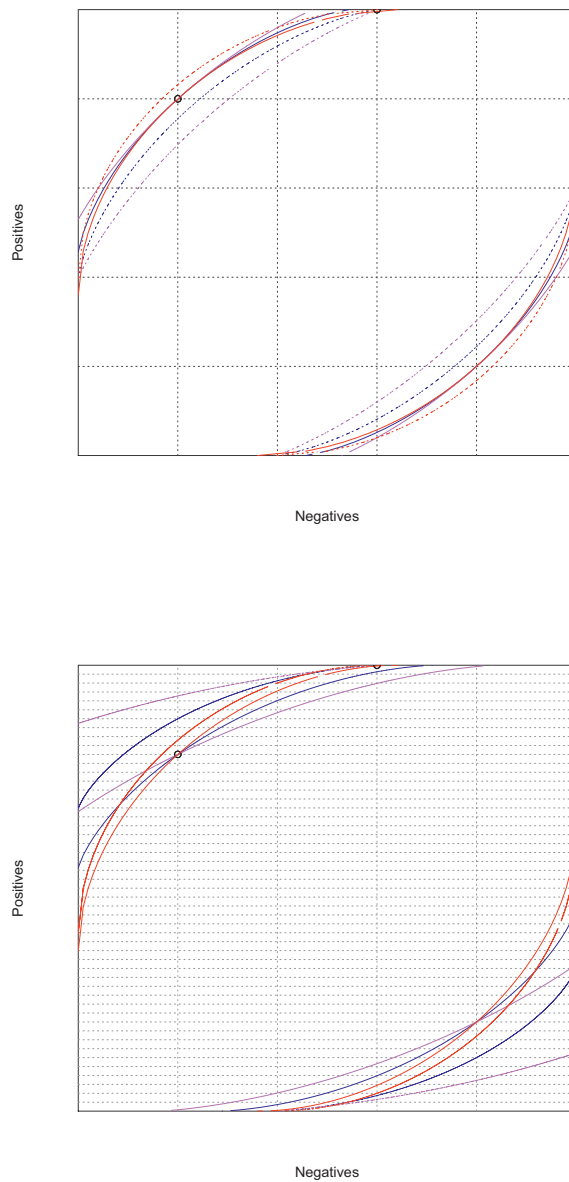


Figure 5.7. (top) ROC isometrics for entropy in blue, Gini index in violet and $\sqrt{\text{Gini}}$ in red through the splits $[8+, 2-][2+, 8-]$ (solid lines) and $[10+, 6-][0+, 4-]$ (dotted lines). Only $\sqrt{\text{Gini}}$ prefers the second split. (bottom) The same isometrics after inflating the positives with a factor 10. All splitting criteria now favour the second split; the $\sqrt{\text{Gini}}$ isometrics are the only ones that haven't moved.

and Gini index ‘mountains’ have rotated clockwise (Gini index more so than entropy), while the $\sqrt{\text{Gini}}$ mountain hasn’t moved at all.

The upshot of all this is that if you learn a decision tree or probability estimation tree using entropy or Gini index as impurity measure – which is what virtually all available tree learning packages do – then your model will change if you change the class distribution by oversampling, while if you use $\sqrt{\text{Gini}}$ you will learn the same tree each time. More generally, *entropy and Gini index are sensitive to fluctuations in the class distribution, $\sqrt{\text{Gini}}$ isn’t*. So which one should you choose? My recommendation echoes the ones I gave for majority class labelling and pruning: use a distribution-insensitive impurity measure such as $\sqrt{\text{Gini}}$ unless the training set operating conditions are representative.⁵

Let’s wrap up the discussion on tree models so far. How would *you* train a decision tree on a given data set, you might ask me? Here’s a list of the steps I would take:

1. First and foremost, I would concentrate on getting good ranking behaviour, because from a good ranker I can get good classification and probability estimation, but not necessarily the other way round.
2. I would therefore try to use an impurity measure that is distribution-insensitive, such as $\sqrt{\text{Gini}}$; if that isn’t available and I can’t hack the code, I would resort to oversampling the minority class to achieve a balanced class distribution.
3. I would disable pruning and smooth the probability estimates by means of the Laplace correction (or the m -estimate).
4. Once I know the deployment operation conditions, I would use these to select the best operating point on the ROC curve (i.e., a threshold on the predicted probabilities, or a labelling of the tree).
5. (optional) Finally, I would prune away any subtree whose leaves all have the same label.

Even though in our discussion we have mostly concentrated on binary classification tasks, it should be noted that decision trees can effortlessly deal with more than two classes – as, indeed, can any grouping model. As already mentioned, multi-class impurity measures simply sum up impurities for each class in a one-versus-rest manner. The only step in this list that isn’t entirely obvious when there are more than two classes is step 4: in this case I would learn a weight for each class as briefly explained in [Section 3.1](#), or possibly combine it with step 5 and resort to reduced-error pruning ([Algorithm 5.3](#)) which might be already implemented in the package you’re using.

⁵It should be noted that it is fairly easy to make measures such as entropy and Gini index distribution-insensitive as well: essentially, this would involve compensating for an observed class ratio $clr \neq 1$ by dividing all counts of positives, or positive empirical probabilities, by clr .

5.3 Tree learning as variance reduction

We will now consider how to adapt decision trees to regression and clustering tasks. This will turn out to be surprisingly straightforward, and is based on the following idea. Earlier, we defined the two-class Gini index $2\dot{p}(1 - \dot{p})$ of a leaf as the expected error resulting from labelling instances in the leaf randomly: positive with probability \dot{p} and negative with probability $1 - \dot{p}$. You can picture this as tossing a coin, prepared such that it comes up heads with probability \dot{p} , to classify examples. Representing this as a random variable with value 1 for heads and 0 for tails, the expected value of this random variable is \dot{p} and its variance $\dot{p}(1 - \dot{p})$ (look up ‘Bernoulli trial’ online if you want to read up on this). This leads to an alternative interpretation of the Gini index as a variance term: the purer the leaf, the more biased the coin will be, and the smaller the variance. For k classes we simply add up the variances of all one-versus-rest random variables.⁶

More specifically, consider a binary split into n_1 and $n_2 = n - n_1$ examples with empirical probabilities \dot{p}_1 and \dot{p}_2 , then the weighted average impurity of these children in terms of the Gini index is

$$\frac{n_1}{n} 2\dot{p}_1(1 - \dot{p}_1) + \frac{n_2}{n} 2\dot{p}_2(1 - \dot{p}_2) = 2 \left(\frac{n_1}{n} \sigma_1^2 + \frac{n_2}{n} \sigma_2^2 \right)$$

where σ_j^2 is the variance of a Bernoulli distribution with success probability \dot{p}_j . So, finding a split with minimum weighted average Gini index is equivalent to minimising weighted average variance (the factor 2 is common to all splits and so can be omitted), and learning a decision tree boils down to partitioning the instance space such that each segment has small variance.

Regression trees

In regression problems the target variable is continuous rather than binary, and in that case we can define the variance of a set Y of target values as the average squared distance from the mean:

$$\text{Var}(Y) = \frac{1}{|Y|} \sum_{y \in Y} (y - \bar{y})^2$$

where $\bar{y} = \frac{1}{|Y|} \sum_{y \in Y} y$ is the mean of the target values in Y ; see [Background 5.1](#) for some useful properties of variance. If a split partitions the set of target values Y into mutually exclusive sets $\{Y_1, \dots, Y_l\}$, the weighted average variance is then

$$\text{Var}(\{Y_1, \dots, Y_l\}) = \sum_{j=1}^l \frac{|Y_j|}{|Y|} \text{Var}(Y_j) = \sum_{j=1}^l \frac{|Y_j|}{|Y|} \left(\frac{1}{|Y_j|} \sum_{y \in Y_j} y^2 - \bar{y}_j^2 \right) = \frac{1}{|Y|} \sum_{y \in Y} y^2 - \sum_{j=1}^l \frac{|Y_j|}{|Y|} \bar{y}_j^2 \quad (5.4)$$

⁶This implicitly assumes that the one-versus-rest variables are uncorrelated, which is not strictly true.

The **variance** of a set of numbers $X \subseteq \mathbb{R}$ is defined as the average squared difference from the mean:

$$\text{Var}(X) = \frac{1}{|X|} \sum_{x \in X} (x - \bar{x})^2$$

where $\bar{x} = \frac{1}{|X|} \sum_{x \in X} x$ is the mean of X . Expanding $(x - \bar{x})^2 = x^2 - 2\bar{x}x + \bar{x}^2$ this can be written as

$$\text{Var}(X) = \frac{1}{|X|} \left(\sum_{x \in X} x^2 - 2\bar{x} \sum_{x \in X} x + \sum_{x \in X} \bar{x}^2 \right) = \frac{1}{|X|} \left(\sum_{x \in X} x^2 - 2\bar{x}|X|\bar{x} + |X|\bar{x}^2 \right) = \frac{1}{|X|} \sum_{x \in X} x^2 - \bar{x}^2 \quad (5.3)$$

So the variance is the difference between the mean of the squares and the square of the mean.

It is sometimes useful to consider the average squared difference from another value $x' \in \mathbb{R}$, which can similarly be expanded:

$$\frac{1}{|X|} \sum_{x \in X} (x - x')^2 = \frac{1}{|X|} \left(\sum_{x \in X} x^2 - 2x'|X|\bar{x} + |X|x'^2 \right) = \text{Var}(X) + (x' - \bar{x})^2$$

The last step follows because from Equation 5.3 we have $\frac{1}{|X|} \sum_{x \in X} x^2 = \text{Var}(X) + \bar{x}^2$.

Another useful property is that the average squared difference between any two elements of X is twice the variance:

$$\frac{1}{|X|^2} \sum_{x' \in X} \sum_{x \in X} (x - x')^2 = \frac{1}{|X|} \sum_{x' \in X} (\text{Var}(X) + (x' - \bar{x})^2) = \text{Var}(X) + \frac{1}{|X|} \sum_{x' \in X} (x' - \bar{x})^2 = 2\text{Var}(X)$$

If $X \subseteq \mathbb{R}^d$ is a set of d -vectors of numbers, we can define the variance $\text{Var}_i(X)$ for each of the d coordinates. We can then interpret the sum of variances $\sum_{i=1}^d \text{Var}_i(X)$ as the average squared Euclidean distance of the vectors in X to their vector mean $\bar{\mathbf{x}} = \frac{1}{|X|} \sum_{\mathbf{x} \in X} \mathbf{x}$.

(You will sometimes see sample variance defined as $\frac{1}{|X|-1} \sum_{x \in X} (x - \bar{x})^2$, which is a somewhat larger value. This version arises if we are estimating the variance of a population from which X is a random sample: normalising by $|X|$ would underestimate the population variance because of differences between the sample mean and the population mean. Here, we are only concerned with assessing the spread of the given values X and not with some unknown population, and so we can ignore this issue.)

Background 5.1. Variations on variance.

So, in order to obtain a regression tree learning algorithm, we replace the impurity measure **Imp** in Algorithm 5.2 with the function **Var**. Notice that $\frac{1}{|Y|} \sum_{y \in Y} y^2$ is constant for a given set Y , and so minimising variance over all possible splits of a given parent is the same as maximising the weighted average of squared means in the chil-

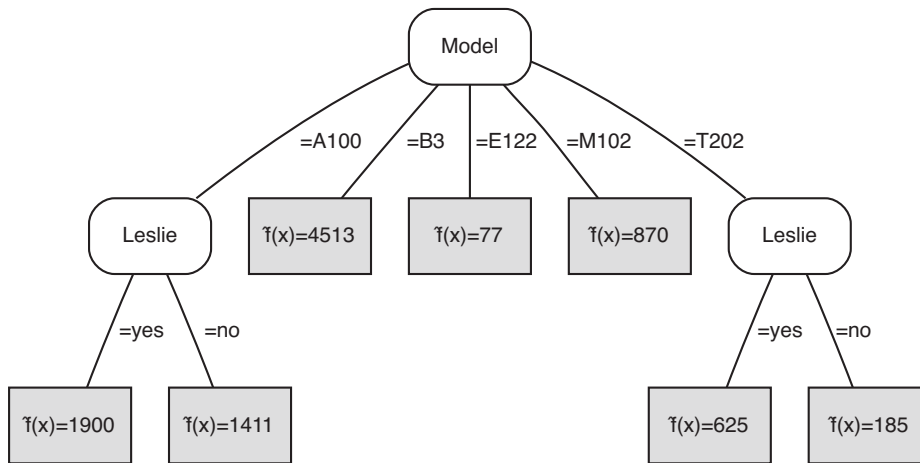


Figure 5.8. A regression tree learned from the data in Example 5.4.

dren. The function $\text{Label}(Y)$ is similarly adapted to return the mean value in Y , and the function $\text{Homogeneous}(Y)$ returns true if the variance of the target values in Y is zero (or smaller than a low threshold).

Example 5.4 (Learning a regression tree). Imagine you are a collector of vintage Hammond tonewheel organs. You have been monitoring an online auction site, from which you collected some data about interesting transactions:

#	Model	Condition	Leslie	Price
1.	B3	excellent	no	4513
2.	T202	fair	yes	625
3.	A100	good	no	1051
4.	T202	good	no	270
5.	M102	good	yes	870
6.	A100	excellent	no	1770
7.	T202	fair	no	99
8.	A100	good	yes	1900
9.	E112	fair	no	77

From this data, you want to construct a regression tree that will help you determine a reasonable price for your next purchase.

There are three features, hence three possible splits:

Model = [A100, B3, E112, M102, T202] [1051, 1770, 1900][4513][77][870][99, 270, 625]
 Condition = [excellent, good, fair] [1770, 4513][270, 870, 1051, 1900][77, 99, 625]
 Leslie = [yes, no] [625, 870, 1900][77, 99, 270, 1051, 1770, 4513]

The means of the first split are 1574, 4513, 77, 870 and 331, and the weighted average of squared means is $3.21 \cdot 10^6$. The means of the second split are 3142, 1023 and 267, with weighted average of squared means $2.68 \cdot 10^6$; for the third split the means are 1132 and 1297, with weighted average of squared means $1.55 \cdot 10^6$. We therefore branch on Model at the top level. This gives us three single-instance leaves, as well as three A100s and three T202s.

For the A100s we obtain the following splits:

Condition = [excellent, good, fair] [1770][1051, 1900][]
 Leslie = [yes, no] [1900][1051, 1770]

Without going through the calculations we can see that the second split results in less variance (to handle the empty child, it is customary to set its variance equal to that of the parent). For the T202s the splits are as follows:

Condition = [excellent, good, fair] [][270][99, 625]
 Leslie = [yes, no] [625][99, 270]

Again we see that splitting on Leslie gives tighter clusters of values. The learned regression tree is depicted in [Figure 5.8](#).

Regression trees are susceptible to overfitting. For instance, if we have exactly one example for each Hammond model then branching on Model will reduce the average variance in the children to zero. The data in [Example 5.4](#) is really too sparse to learn a good regression tree. Furthermore, it is a good idea to set aside a pruning set and to apply reduced-error pruning, pruning away a subtree if the average variance on the pruning set is lower without the subtree than with it (see [Algorithm 5.3](#) on p.144). It should also be noted that predicting a constant value in a leaf is a very simple strategy, and methods exist to learn so-called *model trees*, which are trees with linear regression models in their leaves ([linear regression](#) is explained in [Chapter 7](#)). In that case, the splitting criterion would be based on correlation of the target variable with the regressor variables, rather than simply on variance.

Clustering trees

The simple kind of regression tree considered here also suggests a way to learn clustering trees. This is perhaps surprising, since regression is a supervised learning problem while clustering is unsupervised. The key insight is that regression trees find instance space segments whose target values are tightly clustered around the mean value in the segment – indeed, the variance of a set of target values is simply the (univariate) average squared Euclidean distance to the mean. An immediate generalisation is to use a vector of target values, as this doesn't change the mathematics in an essential way. More generally yet, we can introduce an abstract function $\text{Dis} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ that measures the distance or *dissimilarity* of any two instances $x, x' \in \mathcal{X}$, such that the higher $\text{Dis}(x, x')$ is, the less similar x and x' are. The *cluster dissimilarity* of a set of instances D is then calculated as

$$\text{Dis}(D) = \frac{1}{|D|^2} \sum_{x \in D} \sum_{x' \in D} \text{Dis}(x, x') \quad (5.5)$$

The weighted average cluster dissimilarity over all children of a split then gives the *split dissimilarity*, which can be used to inform $\text{BestSplit}(D, F)$ in the *GrowTree algorithm* (Algorithm 5.1 on p.132).

Example 5.5 (Learning a clustering tree using a dissimilarity matrix).

Assessing the nine transactions on the online auction site from Example 5.4, using some additional features such as reserve price and number of bids (these features do not matter at the moment but are shown in Example 5.6), you come up with the following dissimilarity matrix:

0	11	6	13	10	3	13	3	12
11	0	1	1	1	3	0	4	0
6	1	0	2	1	1	2	2	1
13	1	2	0	0	4	0	4	0
10	1	1	0	0	3	0	2	0
3	3	1	4	3	0	4	1	3
13	0	2	0	0	4	0	4	0
3	4	2	4	2	1	4	0	4
12	0	1	0	0	3	0	4	0

This shows, for instance, that the first transaction is very different from the other eight. The average pairwise dissimilarity over all nine transactions is 2.94.

Using the same features from Example 5.4, the three possible splits are (now with transaction number rather than price):

Model = [A100, B3, E112, M102, T202] [3, 6, 8][1][9][5][2, 4, 7]

Condition = [excellent, good, fair] [1, 6][3, 4, 5, 8][2, 7, 9]

Leslie = [yes, no] [2, 5, 8][1, 3, 4, 6, 7, 9]

The cluster dissimilarity among transactions 3, 6 and 8 is $\frac{1}{3^2}(\mathbf{0}+\mathbf{1}+\mathbf{2}+\mathbf{1}+\mathbf{0}+\mathbf{1}+\mathbf{2}+\mathbf{1}+\mathbf{0}) = 0.89$; and among transactions 2, 4 and 7 it is $\frac{1}{3^2}(\mathbf{0}+\mathbf{1}+\mathbf{0}+\mathbf{1}+\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{0}) = 0.22$. The other three children of the first split contain only a single element and so have zero cluster dissimilarity. The weighted average cluster dissimilarity of the split is then $3/9 \cdot 0.89 + 1/9 \cdot 0 + 1/9 \cdot 0 + 1/9 \cdot 0 + 3/9 \cdot 0.22 = 0.37$. For the second split, similar calculations result in a split dissimilarity of $2/9 \cdot 1.5 + 4/9 \cdot 1.19 + 3/9 \cdot 0 = 0.86$, and the third split yields $3/9 \cdot 1.56 + 6/9 \cdot 3.56 = 2.89$. The Model feature thus captures most of the given dissimilarities, while the Leslie feature is virtually unrelated.

Most of the caveats of regression trees also apply to clustering trees: smaller clusters tend to have lower dissimilarity, and so it is easy to overfit. Setting aside a pruning set to remove the lower splits if they don't improve the cluster coherence on the pruning set is recommended. Single examples can dominate: in the above example, removing the first transaction reduces the overall pairwise dissimilarity from 2.94 to 1.5, and so it will be hard to beat a split that puts that transaction in a cluster of its own.

An interesting question is: how should the leaves of a clustering tree be labelled? Intuitively, it makes sense to label a cluster with its most representative instance. We can define an instance as most representative if its total dissimilarity to all other instances is lowest – this is defined as the medoid in [Chapter 8](#). For instance, in the A100 cluster transaction 6 is most representative because its dissimilarity to 3 and 8 is 1, whereas the dissimilarity between 3 and 8 is 2. Likewise, in the T202 cluster transaction 7 is most representative. However, there is no reason why this should always be uniquely defined.

A commonly encountered scenario, which both simplifies the calculations involved in determining the best split and provides a unique cluster label, is when the dissimilarities are Euclidean distances derived from numerical features. As shown in [Background 5.1](#), if $\text{Dis}(x, x')$ is squared Euclidean distance, then $\text{Dis}(D)$ is twice the average squared Euclidean distance to the mean. This simplifies calculations because both the mean and average squared distance to the mean can be calculated in $O(|D|)$ steps (a single sweep through the data), rather than the $O(|D|^2)$ required if all we have is a dissimilarity matrix. In fact, the average squared Euclidean distance is simply the sum of the variances of the individual features.

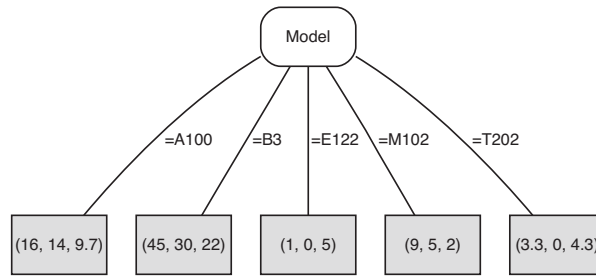


Figure 5.9. A clustering tree learned from the data in [Example 5.6](#) using Euclidean distance on the numerical features.

Example 5.6 (Learning a clustering tree with Euclidean distance). We extend our Hammond organ data with two new numerical features, one indicating the reserve price and the other the number of bids made in the auction. Sales price and reserve price are expressed in hundreds of pounds in order to give the three numerical features roughly equal weight in the distance calculations.

Model	Condition	Leslie	Price	Reserve	Bids
B3	excellent	no	45	30	22
T202	fair	yes	6	0	9
A100	good	no	11	8	13
T202	good	no	3	0	1
M102	good	yes	9	5	2
A100	excellent	no	18	15	15
T202	fair	no	1	0	3
A100	good	yes	19	19	1
E112	fair	no	1	0	5

The means of the three numerical features are (13.3, 8.6, 7.9) and their variances are (158, 101.8, 48.8). The average squared Euclidean distance to the mean is then the sum of these variances, which is 308.6 (if preferred we can double this number to obtain the cluster dissimilarity as defined in [Equation 5.5](#)). For the A100 cluster these vectors are (16, 14, 9.7) and (12.7, 20.7, 38.2), with average squared distance to the mean 71.6; for the T202 cluster they are (3.3, 0, 4.3) and (4.2, 0, 11.6), with average squared distance 15.8. Using this split we can construct a clustering tree whose leaves are labelled with the mean vectors ([Figure 5.9](#)).

In this example we used categorical features for splitting and numerical features for distance calculations. Indeed, in all tree examples considered so far we have only used categorical features for splitting.⁷ In practice, numerical features are frequently used for splitting: all we need to do is find a suitable threshold t so that feature F can be turned into a binary split with conditions $F \geq t$ and $F < t$. Finding the optimal split point is closely related to [discretisation](#) of numerical features, a topic we will look at in detail in [Chapter 10](#). For the moment, the following observations give some idea how we can learn a threshold on a numerical feature:

- ☞ Although in theory there are infinitely many possible thresholds, in practice we only need to consider values separating two examples that end up next to each other if we sort the training examples on increasing (or decreasing) value of the feature.
- ☞ We only consider consecutive examples of different class if our task is classification, whose target values are sufficiently different if our task is regression, or whose dissimilarity is sufficiently large if our task is clustering.
- ☞ Each potential threshold can be evaluated as if it were a distinct binary feature.

5.4 Tree models: Summary and further reading

Tree-based data structures are ubiquitous in computer science, and the situation is no different in machine learning. Tree models are concise, easy to interpret and learn, and can be applied to a wide range of tasks, including classification, ranking, probability estimation, regression and clustering. The tree-based classifier for human pose recognition in the Microsoft Kinect motion sensing device is described in [Shotton *et al.* \(2011\)](#).

- ☞ I introduced the feature tree as the common core for all these tree-based models, and the recursive [GrowTree](#) algorithm as a generic divide-and-conquer algorithm that can be adapted to each of these tasks by suitable choices for the functions that test whether a data set is sufficiently homogeneous, find a suitable label if it is, and find the best feature to split on if it isn't.
- ☞ Using a feature tree to predict class labels turns them into decision trees, the subject of [Section 5.1](#). There are two classical accounts of decision trees in machine learning, which are very similar algorithmically but differ in details such as heuristics and pruning strategies. Quinlan's approach was to use entropy as impurity measure, and progressed from the [ID3](#) algorithm ([Quinlan, 1986](#)), which

⁷ Categorical features are features with a relatively small set of discrete values. Technically, they distinguish themselves from numerical features by not having a scale or an ordering. This is further explored in [Chapter 10](#).

itself was inspired by [Hunt, Marin and Stone \(1966\)](#), to the sophisticated [C4.5](#) system ([Quinlan, 1993](#)). The [CART](#) approach stands for ‘classification and regression trees’ and was developed by [Breiman, Friedman, Olshen and Stone \(1984\)](#); it uses the Gini index as impurity measure. The $\sqrt{\text{Gini}}$ impurity measure was introduced by [Dietterich, Kearns and Mansour \(1996\)](#), and is hence sometimes referred to as [DKM](#). The geometric construction to find $\text{Imp}(\{D_1, D_2\})$ in [Figure 5.2 \(right\)](#) was also inspired by that paper.

- ☞ Employing the empirical distributions in the leaves of a feature tree in order to build rankers and probability estimators as described in [Section 5.2](#) is a much more recent development ([Ferri et al., 2002](#); [Provost and Domingos, 2003](#)). Experimental results demonstrating that better probability estimates are obtained by disabling tree pruning and smoothing the empirical probabilities by means of the Laplace correction are presented in the latter paper and corroborated by [Ferri et al. \(2003\)](#). The extent to which decision tree splitting criteria are insensitive to unbalanced classes or misclassification costs was studied and explained by [Drummond and Holte \(2000\)](#) and [Flach \(2003\)](#). Of the three splitting criteria mentioned above, only $\sqrt{\text{Gini}}$ is insensitive to such class and cost imbalance.
- ☞ Tree models are grouping models that aim to minimise diversity in their leaves, where the appropriate notion of diversity depends on the task. Very often diversity can be interpreted as some kind of variance, an idea that already appeared in ([Breiman et al., 1984](#)) and was revisited by [Langley \(1994\)](#), [Kramer \(1996\)](#) and [Blockeel, De Raedt and Ramon \(1998\)](#), among others. In [Section 5.3](#) we saw how this idea can be used to learn regression and clustering trees (glossing over many important details, such as when we should stop splitting nodes).

It should be kept in mind that the increased expressivity of tree models compared with, say, conjunctive concepts means that we should safeguard ourselves against overfitting. Furthermore, the greedy divide-and-conquer algorithm has the disadvantage that small changes in the training data may lead to a different choice of the feature at the root of the tree, which will influence the choice of feature at subsequent splits. We will see in [Chapter 11](#) how methods such as bagging can be applied to help reduce this kind of model variance.



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.

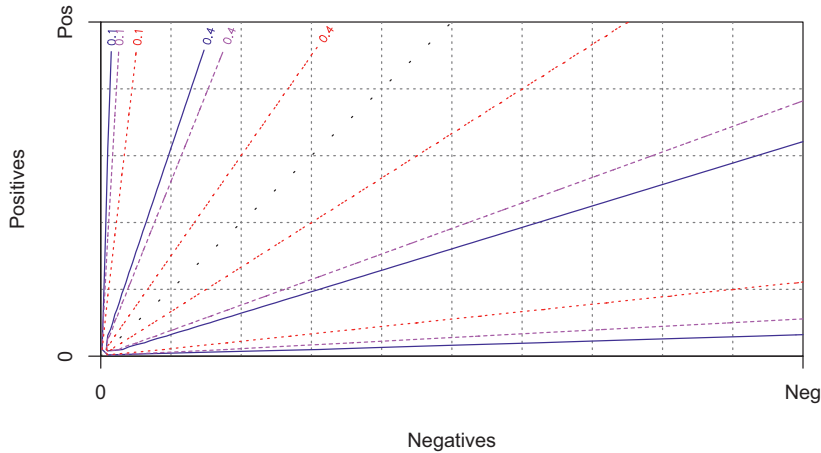


Figure 6.1. ROC isometrics for **entropy** (rescaled to have a maximum value of 1/2), **Gini index** and **minority class**. The grey dotted symmetry line is defined by $\hat{p} = 1/2$: each isometric has two parts, one above the symmetry line (where impurity decreases with increasing empirical probability \hat{p}) and its mirror image below the symmetry line (where impurity is proportional to \hat{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 \hat{p} (the relative frequency of covered positives) if $\hat{p} > 1/2$ and increases with \hat{p} if $\hat{p} < 1/2$; 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 $\min(\hat{p}, 1 - \hat{p})$ (or, if you prefer, $1/2 - |\hat{p} - 1/2|$), 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 \wedge Gills = no \wedge Beak = yes \wedge Teeth = many
- p2: Length = 4 \wedge Gills = no \wedge Beak = yes \wedge Teeth = many
- p3: Length = 3 \wedge Gills = no \wedge Beak = yes \wedge Teeth = few
- p4: Length = 5 \wedge Gills = no \wedge Beak = yes \wedge Teeth = many
- p5: Length = 5 \wedge Gills = no \wedge Beak = yes \wedge Teeth = few

and negatives

- n1: Length = 5 \wedge Gills = yes \wedge Beak = yes \wedge Teeth = many
- n2: Length = 4 \wedge Gills = yes \wedge Beak = yes \wedge Teeth = many
- n3: Length = 5 \wedge Gills = yes \wedge Beak = no \wedge Teeth = many
- n4: Length = 4 \wedge Gills = yes \wedge Beak = no \wedge Teeth = many
- n5: Length = 4 \wedge Gills = no \wedge Beak = yes \wedge 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 = \ominus ·

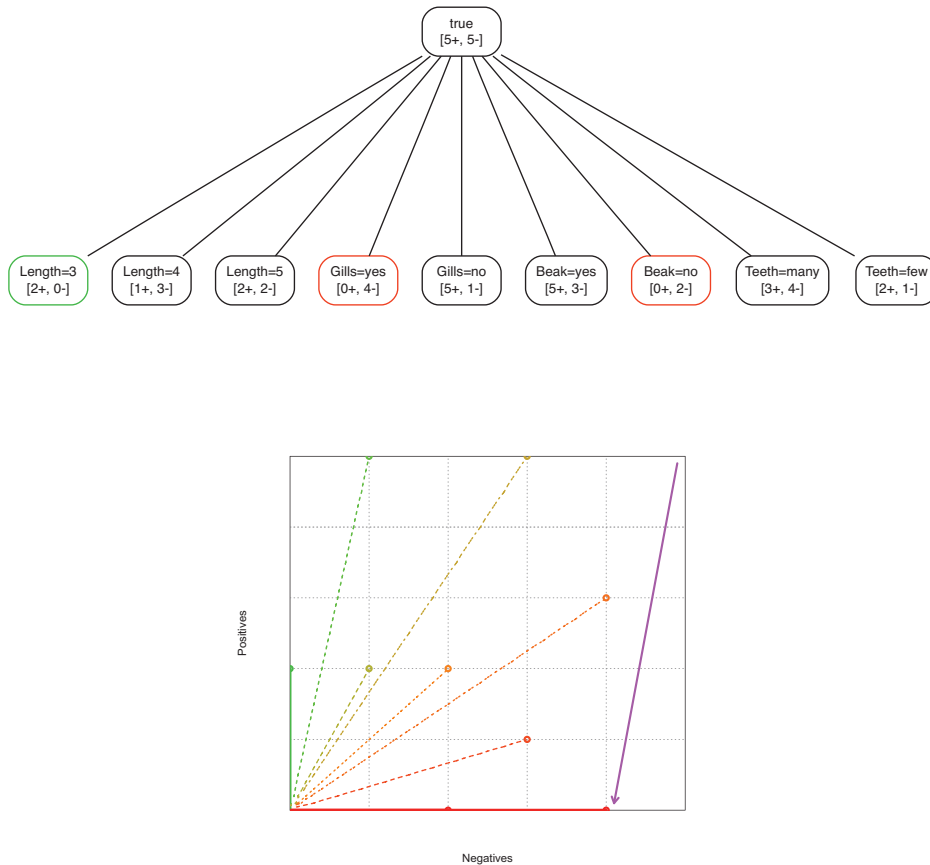


Figure 6.2. (top) All literals with their coverage counts on the data in Example 6.1. The ones in green (red) are pure for the positive (negative) class. **(bottom)** The nine literals plotted as points in coverage space, with their impurity values indicated by impurity isometrics (away from the ascending diagonal is better). Impurity values are colour-coded: towards green if $\hat{p} > 1/2$, towards red if $\hat{p} < 1/2$, and orange if $\hat{p} = 1/2$ (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

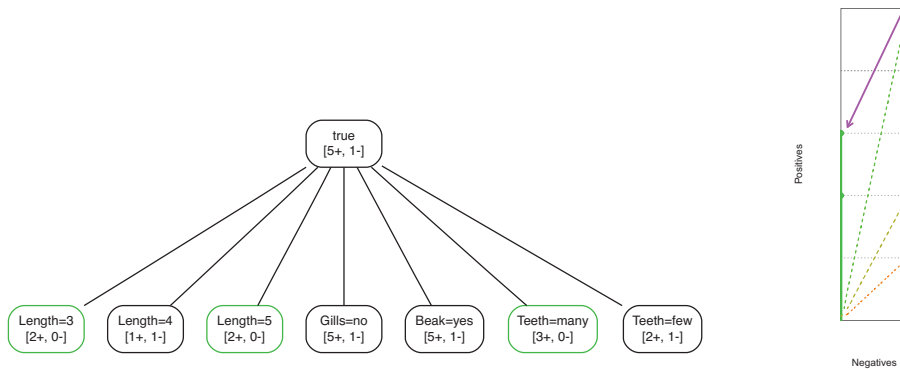


Figure 6.3. (left) Revised coverage counts after removing the four negative examples covered by the first rule found (literals not covering any examples are omitted). **(right)** We are now operating in the right-most ‘slice’ of Figure 6.2 on p.160.

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

·if Teeth = many then Class = ⊕·

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

·if Length = 4 then Class = ⊖·

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

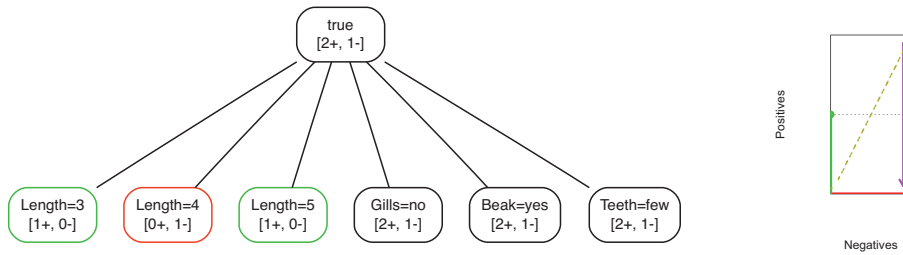


Figure 6.4. (left) The third rule covers the one remaining negative example, so that the remaining positives can be swept up by a default rule. **(right)** This will collapse the coverage space.

then as follows:

```
·if Gills = yes then Class = ⊖·
·else if Teeth = many then Class = ⊕·
·else if Length = 4 then Class = ⊖·
·else Class = ⊕·
```

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':

```
·if Gills = yes then Class = ⊖·
·if Gills = no ∧ Teeth = many then Class = ⊕·
·if Gills = no ∧ Teeth = few ∧ Length = 4 then Class = ⊖·
·if Gills = no ∧ Teeth = few ∧ Length ≠ 4 then Class = ⊕·
```

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:

```
·if  $P \wedge Q$  then Class = ⊕·
·else if  $R$  then Class = ⊖·
```

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

```
·if  $\neg(P \wedge Q) \wedge R$  then Class = ⊖·
```

or equivalently,

```
·if  $(\neg P \vee \neg Q) \wedge R$  then Class = ⊖·
```

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    $r \leftarrow \text{LearnRule}(D)$ ;           // LearnRule: see Algorithm 6.2
4   append  $r$  to the end of  $R$ ;
5    $D \leftarrow D \setminus \{x \in D \mid x \text{ is covered by } r\}$ ;
6 end
7 return  $R$ 
```

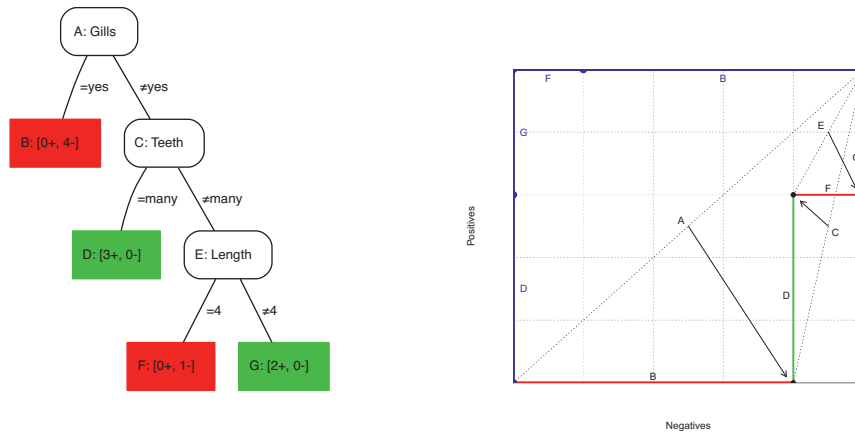


Figure 6.5. (left) A right-branching feature tree corresponding to a list of single-literal rules. (right) The construction of this feature tree depicted in coverage space. The leaves of the tree are either purely positive (in green) or purely negative (in red). Reordering these leaves on their empirical 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 .

```

1  $b \leftarrow \text{true}$ ;
2  $L \leftarrow$  set of available literals;
3 while not Homogeneous(D) do
4    $l \leftarrow \text{BestLiteral}(D, L)$ ; // e.g., highest purity; see text
5    $b \leftarrow b \wedge l$ ;
6    $D \leftarrow \{x \in D \mid x \text{ is covered by } b\}$ ;
7    $L \leftarrow L \setminus \{l' \in L \mid l' \text{ uses same feature as } l\}$ ;
8 end
9  $C \leftarrow \text{Label}(D)$ ; // e.g., majority class
10  $r \leftarrow \text{if } b \text{ then Class} = C$ ;
11 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 $B \setminus A$. 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: $A \setminus B$) 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,

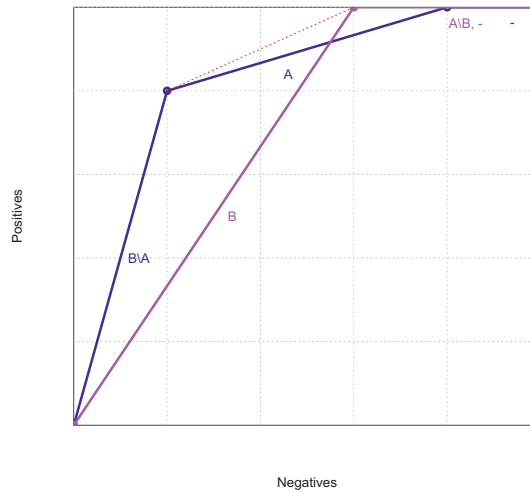


Figure 6.6. Coverage curves of two rule lists consisting of the rules from Example 6.2, in different order (AB in blue and BA in violet). $B \setminus A$ corresponds to the coverage of rule B once the coverage of rule A is taken away, and ‘-’ denotes the default rule. Neither curve dominates the other, 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 $A \wedge B$, 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 $A \wedge B$ – 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 $B \setminus A$ from rule list AB . It follows that this segment contains exactly those examples that are in B but not in $B \setminus A$, hence in $A \wedge B$. 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 $\min(\hat{p}, 1 - \hat{p})$, we can maximise \hat{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

·if Length = 3 then Class = \oplus ·

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:

·if Gills = no \wedge Length = 5 then Class = \oplus ·

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

·if Gills = no \wedge Teeth = many then Class = \oplus ·

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): ·if Gills = yes then Class = \ominus · first, followed by ·if Length = 4 \wedge Teeth = few then Class = \ominus ·. The final rule set with rules for

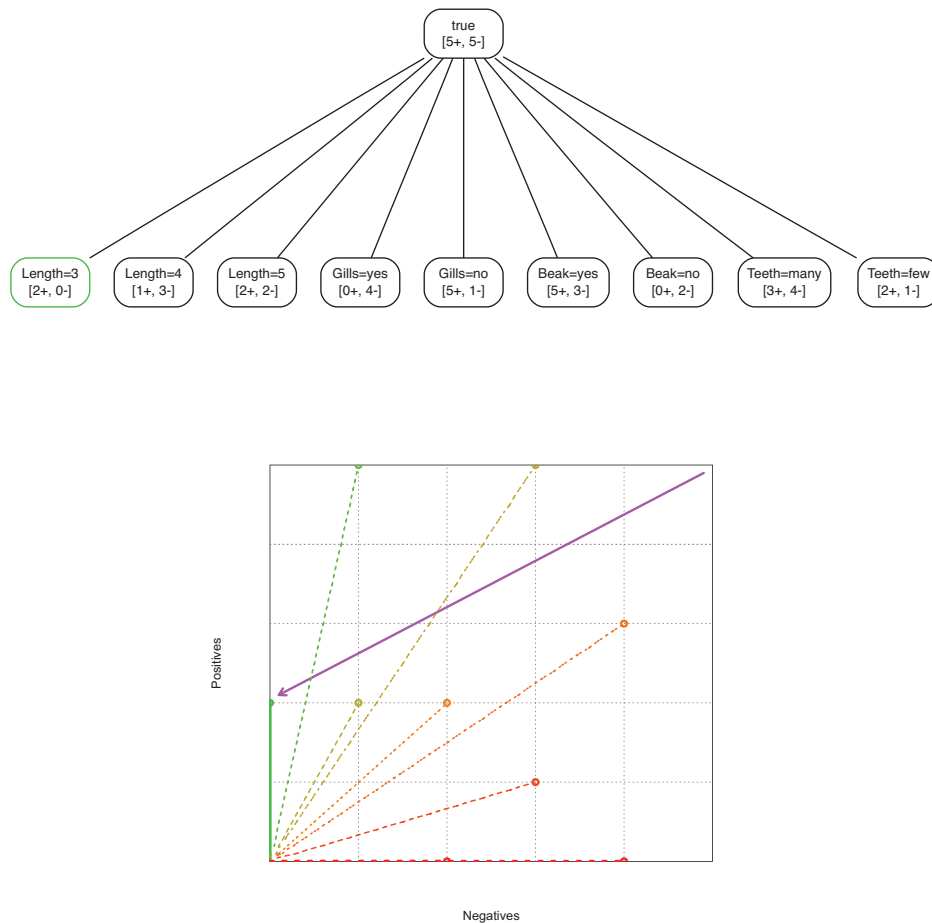


Figure 6.7. (top) The first rule is learned for the positive class. **(bottom)** Precision isometrics look identical to impurity isometrics (Figure 6.2); however, the difference is that precision is lowest on the x -axis and highest on the y -axis, while purity is lowest on the ascending diagonal and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

both classes is therefore

- (R1) ·if Length = 3 then Class = \oplus ·
- (R2) ·if Gills = no \wedge Length = 5 then Class = \oplus ·
- (R3) ·if Gills = no \wedge Teeth = many then Class = \oplus ·
- (R4) ·if Gills = yes then Class = \ominus ·
- (R5) ·if Length = 4 \wedge Teeth = few then Class = \ominus ·

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

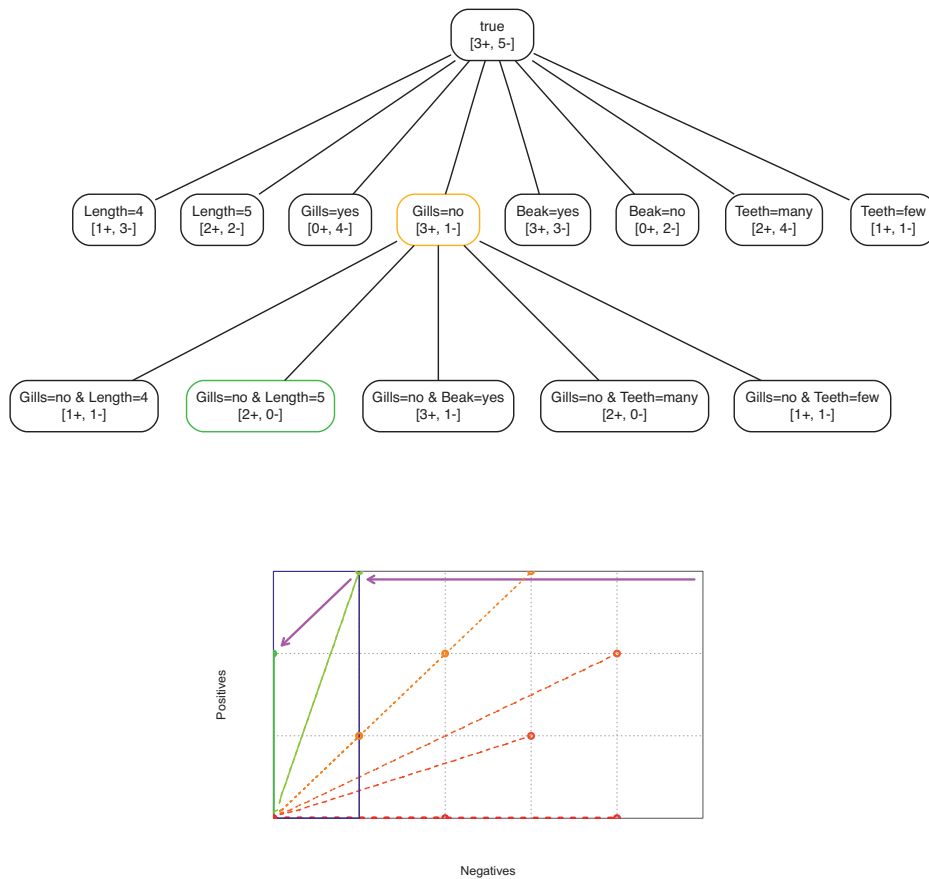


Figure 6.8. (top) The second rule needs two literals: we use maximum precision to select both. **(bottom)** The coverage space is smaller because the two positives covered by the first rule are removed. The blue box on the left indicates an even smaller coverage space in which the search for the second 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

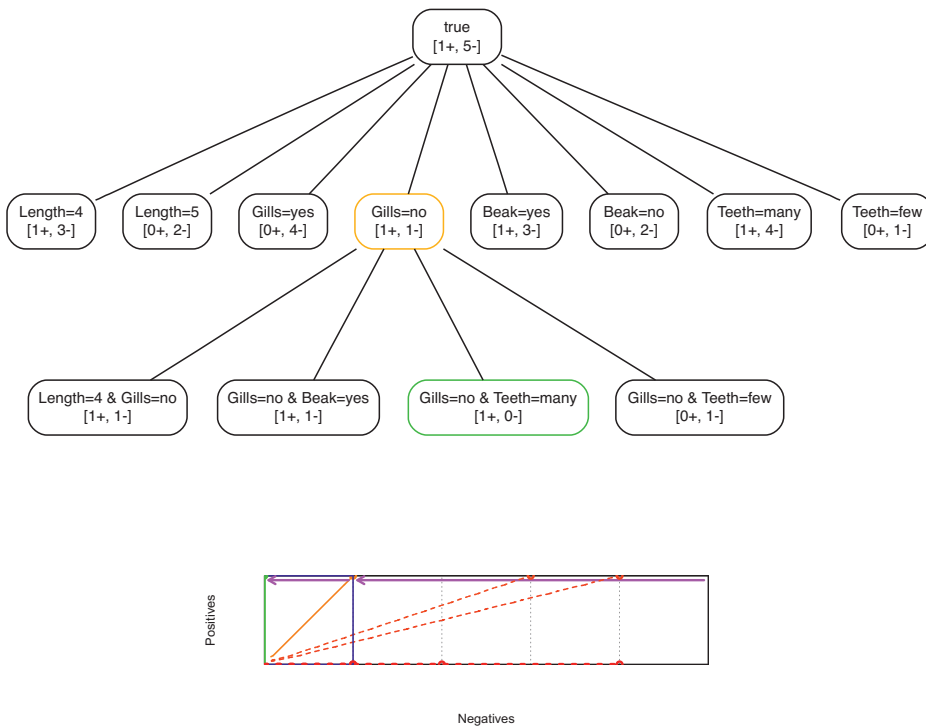


Figure 6.9. (top) The third and final rule again needs two literals. **(bottom)** The first literal excludes four negatives, the second excludes the one remaining negative.

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* = \oplus , even though the near-pure literal *Gills* = no leads to the pure rule **if** *Gills* = no \wedge *Teeth* = many **then** *Class* = \oplus . 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 \wedge Gills = yes \wedge Beak = yes \wedge 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      $r \leftarrow \text{LearnRuleForClass}(D_i, C_i)$ ; // LearnRuleForClass: see Algorithm 6.4
6      $R \leftarrow R \cup \{r\}$ ;
7      $D_i \leftarrow D_i \setminus \{x \in C_i \mid x \text{ is covered by } r\}$ ; // 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  $b \leftarrow \text{true}$ ;
2  $L \leftarrow$  set of available literals; // can be initialised by seed example
3 while not Homogeneous( $D$ ) do
4    $l \leftarrow \text{BestLiteral}(D, L, C_i)$ ; // e.g. maximising precision on class  $C_i$ 
5    $b \leftarrow b \wedge l$ ;
6    $D \leftarrow \{x \in D \mid x \text{ is covered by } b\}$ ;
7    $L \leftarrow L \setminus \{l' \in L \mid l' \text{ uses same feature as } l\}$ ;
8 end
9  $r \leftarrow \text{if } b \text{ then Class} = C_i$ ;
10 return  $r$ 
```

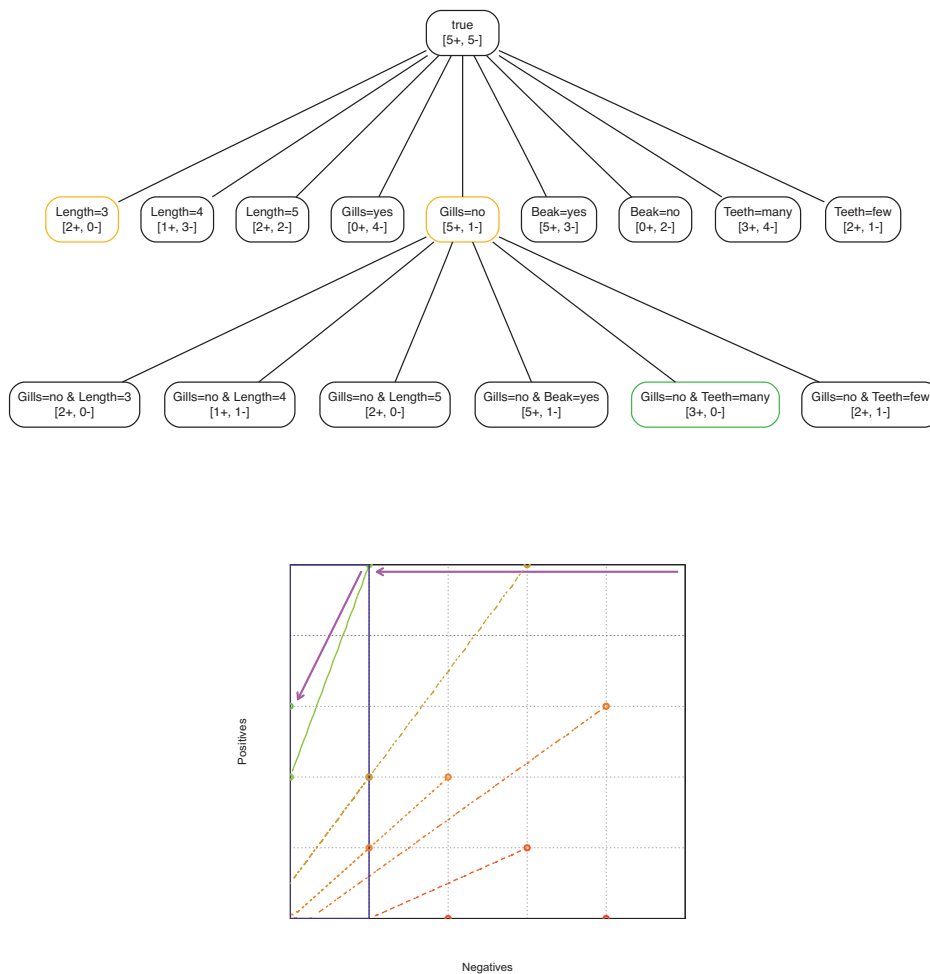


Figure 6.10. (top) Using Laplace-corrected precision allows learning a better rule in the first iteration. **(bottom)** Laplace correction adds one positive and one negative pseudo-count, which means that the isometrics now rotate around $(-1, -1)$ in coverage space, resulting in a preference for more 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 = \ominus · [1+, 3-]
- (B) ·if Beak = yes then Class = \oplus · [5+, 3-]
- (C) ·if Length = 5 then Class = \ominus · [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 $\hat{p}(A) = 1/4 = 0.25$, and similarly $\hat{p}(B) = 5/8 = 0.63$ and $\hat{p}(C) = 2/4 = 0.50$.

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 $\hat{p}(AB) = 3/6 = 0.50$. Similarly, $\hat{p}(BC) = 3.5/6 = 0.58$. 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 = \ominus · [1+, 3-]
- else if Beak = yes then Class = \oplus · [4+, 1-]
- else if Length = 5 then Class = \ominus · [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

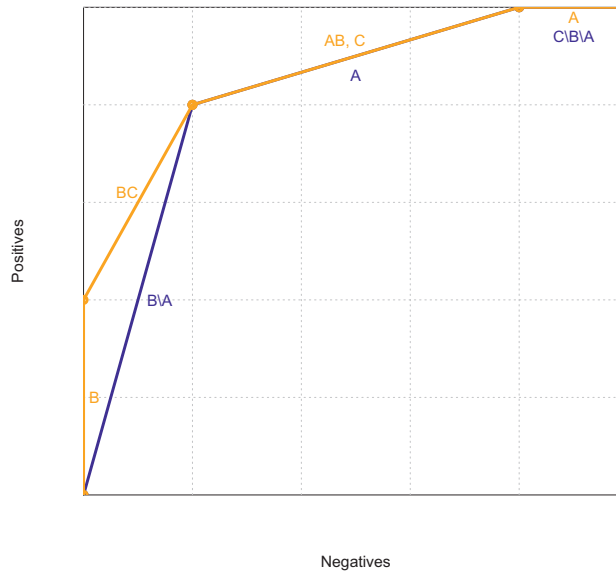


Figure 6.11. Coverage curves of the rule set in Example 6.4 (in orange) and the rule list ABC (in blue). The rule set partitions the instance space in smaller segments, which in this case lead to better ranking performance.

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 $r + 1$

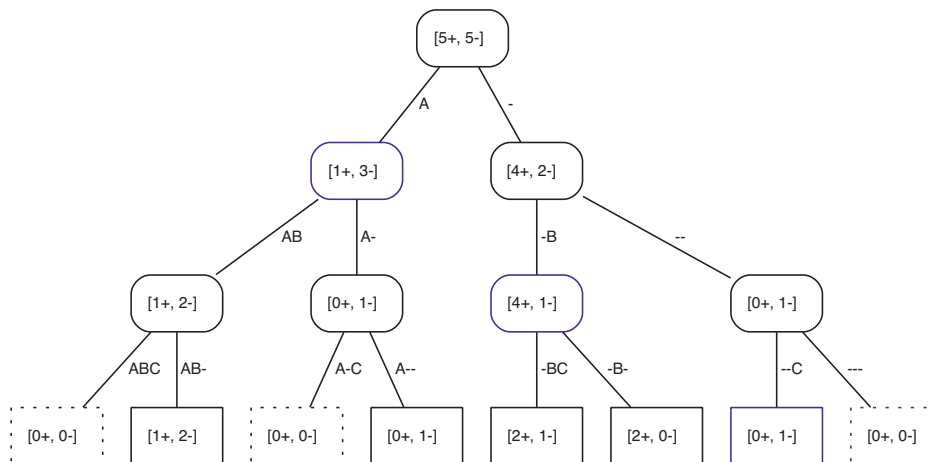


Figure 6.12. A rule tree constructed from the rules in Example 6.5. Nodes are labelled with their coverage (dotted leaves have empty coverage), and branch labels indicate particular areas in the instance space (e.g., $A-C$ denotes $A \wedge \neg B \wedge C$). The blue nodes are the instance space segments 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

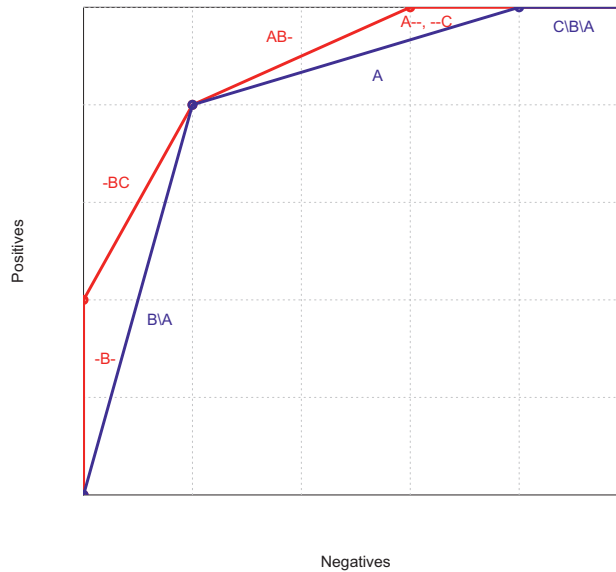


Figure 6.13. The blue line is the coverage curve of the rule list ABC in Example 6.4. This curve is dominated by the red coverage curve, corresponding to the rule tree in Figure 6.12. The rule tree also improves upon the rule set (orange curve in Figure 6.11), as it has access to exact coverage counts 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

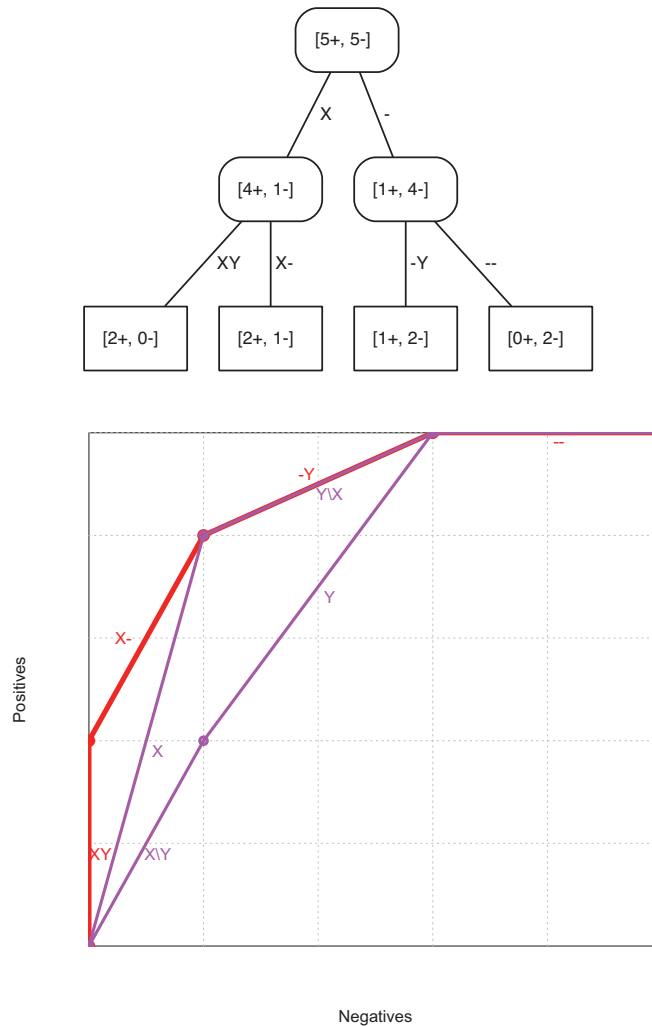


Figure 6.14. (top) A rule tree built on two rules X and Y . (bottom) The rule tree coverage curve strictly dominates the convex hull of the two rule list curves. This means that there is an operating point $[2+, 0-]$ that cannot be achieved by either rule list.

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 $\hat{g}: \mathcal{X} \rightarrow \{\text{true}, \text{false}\}$ – or alternatively, subsets of the instance space – that are learned from a set of labelled examples $(x_i, l(x_i))$, where $l: \mathcal{X} \rightarrow \mathcal{C}$ 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 $prec^L$, leading to the alternative measure $|prec^L - pos|$. 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 $|avg-rec - 0.5|$. Average recall can be written as $(1 + tpr - fpr)/2$, and thus $|avg-rec - 0.5| = |tpr - fpr|/2$. 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

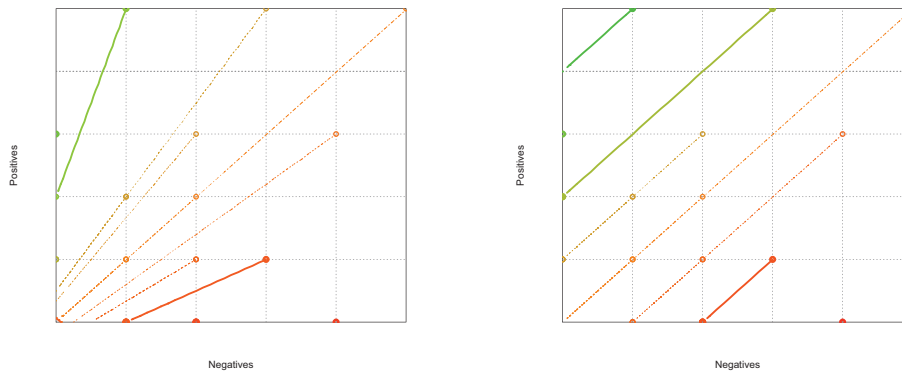


Figure 6.15. (left) Subgroups and their isometrics according to Laplace-corrected precision. The solid, outermost isometrics indicate the best subgroups. **(right)** The ranking changes if we order the subgroups on average recall. For example, $[5+, 1-]$ is now better than $[3+, 0-]$ and as good as $[0+,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

Subgroup	Coverage	$prec^L$	Rank	avg-rec	Rank
Gills = yes	$[0+, 4-]$	0.17	1	0.10	1–2
Gills = no \wedge 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 \wedge 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 \wedge Gills = yes	$[0+, 2-]$	0.25	3–9	0.30	4–11
Length = 5 \wedge Gills = no	$[2+, 0-]$	0.75	3–9	0.70	4–11
Length = 5 \wedge 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 $|avg-rec - 0.5|$. 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.

<i>Subgroup</i>	<i>Coverage</i>	<i>avg-rec</i>	<i>Wgtd coverage</i>	<i>W-avg-rec</i>	<i>Rank</i>
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 \wedge Teeth = many	[3+, 0-]	0.80	[2.5 +, 0-]	0.78	3
Length = 5 \wedge 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 \wedge 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 \wedge 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 \wedge 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 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 \wedge 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

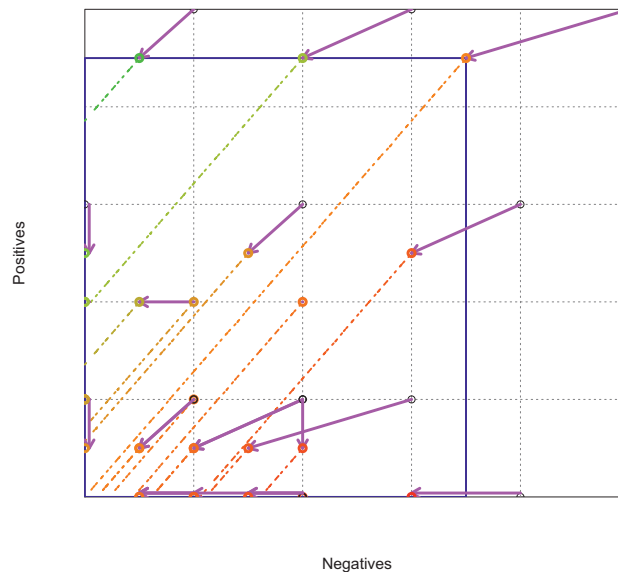


Figure 6.16. Visualisation of the effect of weighted covering. If the first subgroup found is $\text{Length} = 4$, then this halves the weight of one positive and three negatives, shrinking the coverage space to the blue box. The arrows indicate how this affects the weighted coverage of other subgroups,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

found is $\text{Length} = 4$, 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 $\text{Length} = 4$, as indicated by the arrows. Some subgroups end up closer to the diagonal and hence lose importance: for instance, $\text{Length} = 4$ itself, which moves from $[3+, 1-]$ to $[1.5+, 0.5-]$. Others move away from the diagonal and hence gain importance: for example $\text{Length} = 5 \wedge \text{Gills} = \text{yes}$ 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 {beer, 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    $r \leftarrow \text{LearnRule}(D)$ ;           // 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  $R$ 
```

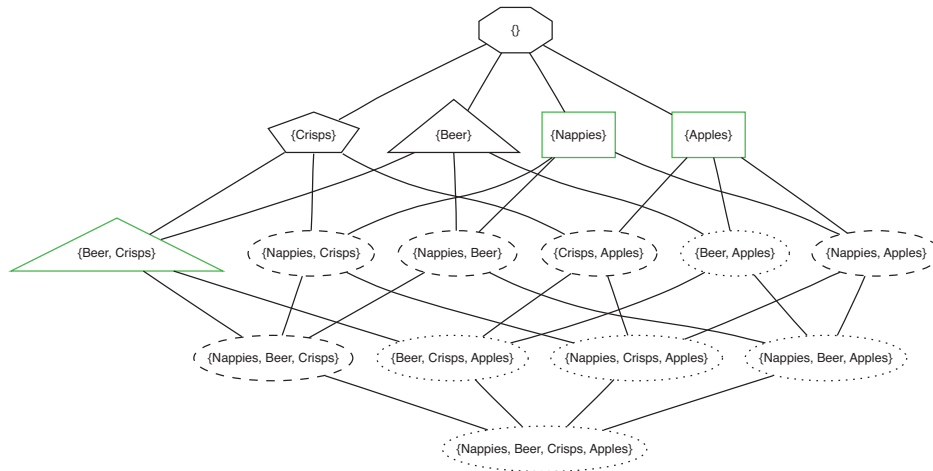


Figure 6.17. An item set lattice. Item sets in dotted ovals cover a single transaction; in dashed ovals, two transactions; in triangles, three transactions; and in polygons with n sides, n transactions. The maximal item sets with support 3 or more are indicated in green.

sets is *convex* and is fully determined by its lower boundary of largest item sets: in the example these maximal¹ 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 \wedge 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 $D \subseteq \mathcal{X}$; support threshold f_0 .
Output : set of maximal frequent item sets M .

```

1  $M \leftarrow \emptyset$ ;
2 initialise priority queue  $Q$  to contain the empty item set;
3 while  $Q$  is not empty do
4    $I \leftarrow$  next item set deleted from front of  $Q$ ;
5    $max \leftarrow \text{true}$ ; // flag to indicate whether  $I$  is maximal
6   for each possible extension  $I'$  of  $I$  do
7     if  $\text{Supp}(I') \geq f_0$  then
8        $max \leftarrow \text{false}$ ; // frequent extension found, so  $I$  is not maximal
9       add  $I'$  to back of  $Q$ ;
10    end
11  end
12  if  $max = \text{true}$  then  $M \leftarrow M \cup \{I\}$ ;
13 end
14 return  $M$ 

```

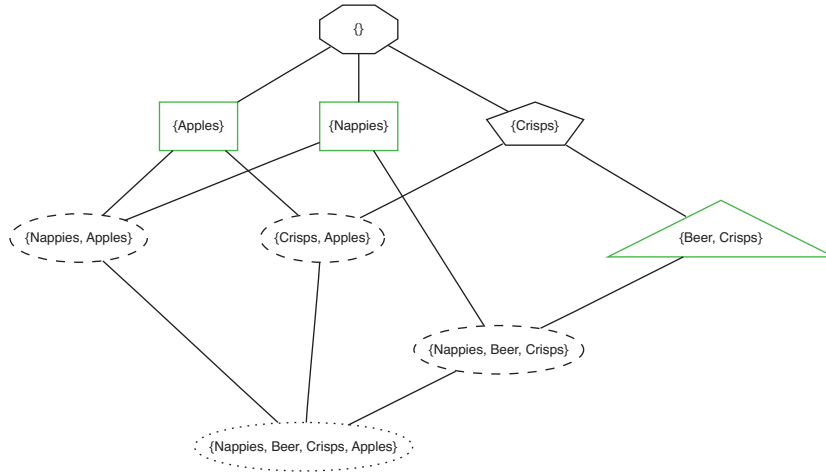


Figure 6.18. Closed item set lattice corresponding to the item sets in Figure 6.17. This lattice has the property that no two adjacent item sets have the same coverage.

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 $D \subseteq \mathcal{X}$; support threshold f_0 ; confidence threshold c_0 .

Output : set of association rules R .

```

1  $R \leftarrow \emptyset$ ;
2  $M \leftarrow \text{FrequentItems}(D, f_0)$ ;           // 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  $\text{Supp}(B \cup H) / \text{Supp}(B) \geq c_0$  then  $R \leftarrow R \cup \{\text{if } B \text{ then } H\}$ 
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

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

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 $\text{Lift}(\cdot\text{if } B \text{ then } H\cdot) = \text{Lift}(\cdot\text{if } H \text{ then } B\cdot)$. For the third rule we have $\text{Lift}(\cdot\text{if true then crisps}\cdot) = \frac{8 \cdot 5}{8 \cdot 5} = 1$. This holds for any rule with $B = \emptyset$, as

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

More generally, a lift of 1 means that $\text{Supp}(B \cup H)$ is entirely determined by the *marginal* frequencies $\text{Supp}(B)$ and $\text{Supp}(H)$ 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 $\text{Supp}(I)/n$ 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!).

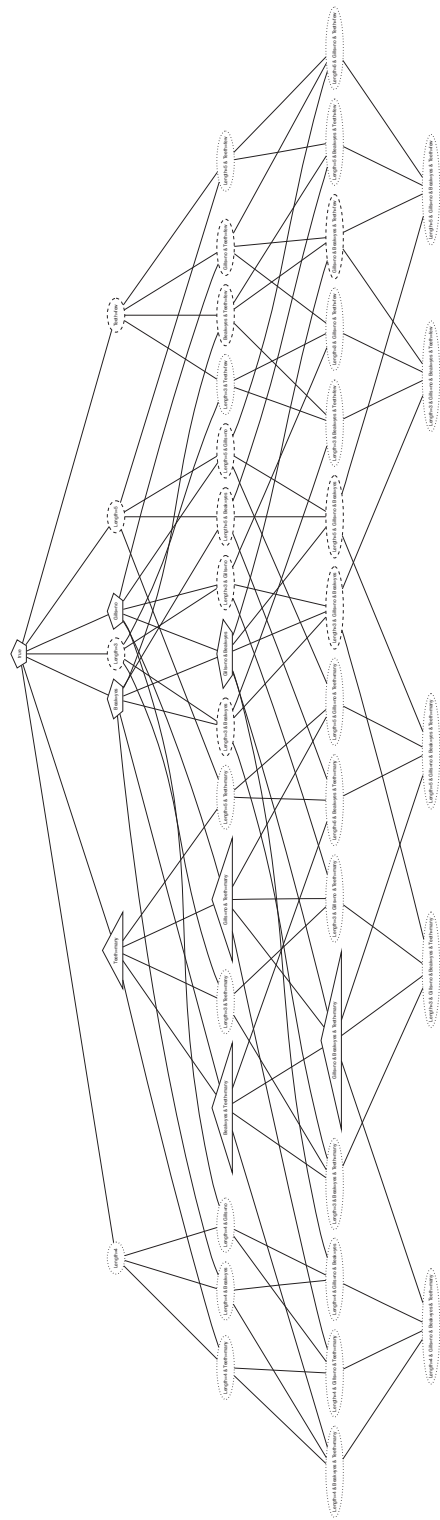


Figure 6.19. The item set lattice corresponding to the positive examples of the dolphin example in Example 4.4 on p.115. Each ‘item’ is a literal $\text{Feature} = \text{Value}$; each feature can occur at most once in an item set. The resulting structure is exactly the same as what was called the hypothesis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

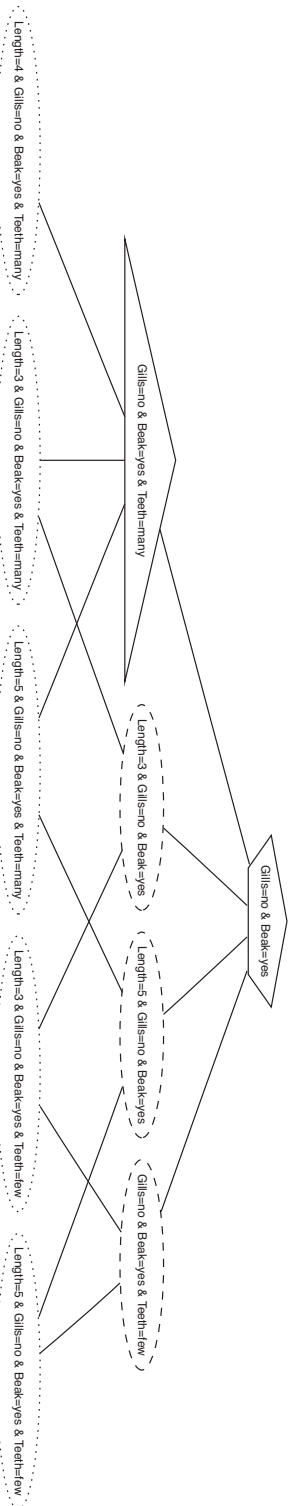


Figure 6.20. Closed item set lattice corresponding to the item sets in Figure 6.19.

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

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

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:

$$\begin{aligned} \forall x: \forall y: \text{BodyPart}(x, \text{PairOf}(y)) \rightarrow \text{SomeAnimal}(x) \\ \forall x: (\exists y: \text{BodyPart}(x, \text{PairOf}(y))) \rightarrow \text{SomeAnimal}(x) \end{aligned}$$

The first logical statement reads ‘for all x and y , if x has a pair of y s 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 y s 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).

