

---

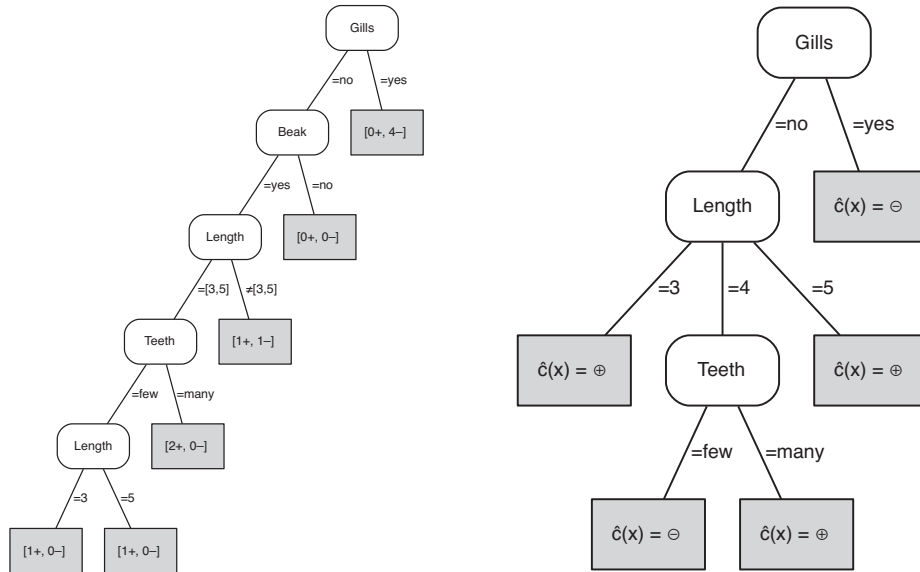
## Tree models

---

**T**REE 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:

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

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.<sup>1</sup> *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  $\Delta$  be the disjunction of all positive examples, then  $\Delta$  is in disjunctive normal form.  $\Delta$  clearly covers all positives – in fact,  $\Delta$ 's extension is exactly the set of positive examples. In other words, in the hypothesis space of DNF expressions (or of decision trees),  $\Delta$  is the LGG of the positive examples, but it doesn't cover any other instances. So  $\Delta$  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

<sup>1</sup>If we allowed the creation of new conjunctive features, we could actually represent this tree as the conjunctive concept  $Gills = no \wedge F = false$ , where  $F \equiv Length = 4 \wedge Teeth = few$  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.

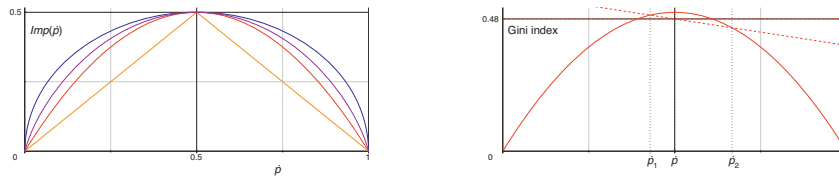
In the remainder of this chapter we will instantiate the generic Algorithm 5.1 to classification, ranking and probability estimation, clustering and regression tasks.

## 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  $D$ .<sup>2</sup> 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  $\dot{p} = n^{\oplus} / (n^{\oplus} + n^{\ominus})$ , 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  $\dot{p}$  with  $1 - \dot{p}$ . We also want a function that is 0 whenever  $\dot{p} = 0$  or  $\dot{p} = 1$  and that reaches its maximum

<sup>2</sup>If 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  $(1 - \dot{p})\dot{p}$ .<sup>3</sup>

**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  $\{D_1, \dots, D_I\}$  is then

<sup>3</sup>When 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:

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

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):

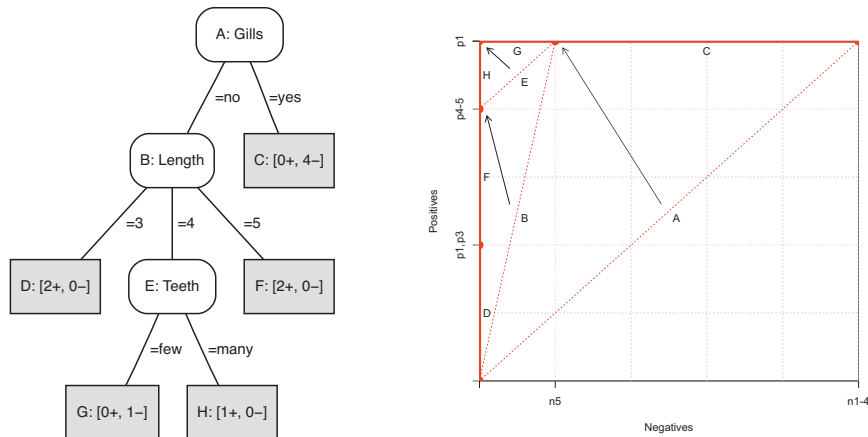
$$\begin{aligned}
\text{Length} & 2/10 \cdot 2 \cdot (2/2 \cdot 0/2) + 4/10 \cdot 2 \cdot (1/4 \cdot 3/4) + 4/10 \cdot 2 \cdot (2/4 \cdot 2/4) = 0.35; \\
\text{Gills} & 4/10 \cdot 0 + 6/10 \cdot 2 \cdot (5/6 \cdot 1/6) = 0.17; \\
\text{Beak} & 8/10 \cdot 2 \cdot (5/8 \cdot 3/8) + 2/10 \cdot 0 = 0.38;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

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.<sup>4</sup> 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

<sup>4</sup>Tie 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.

---

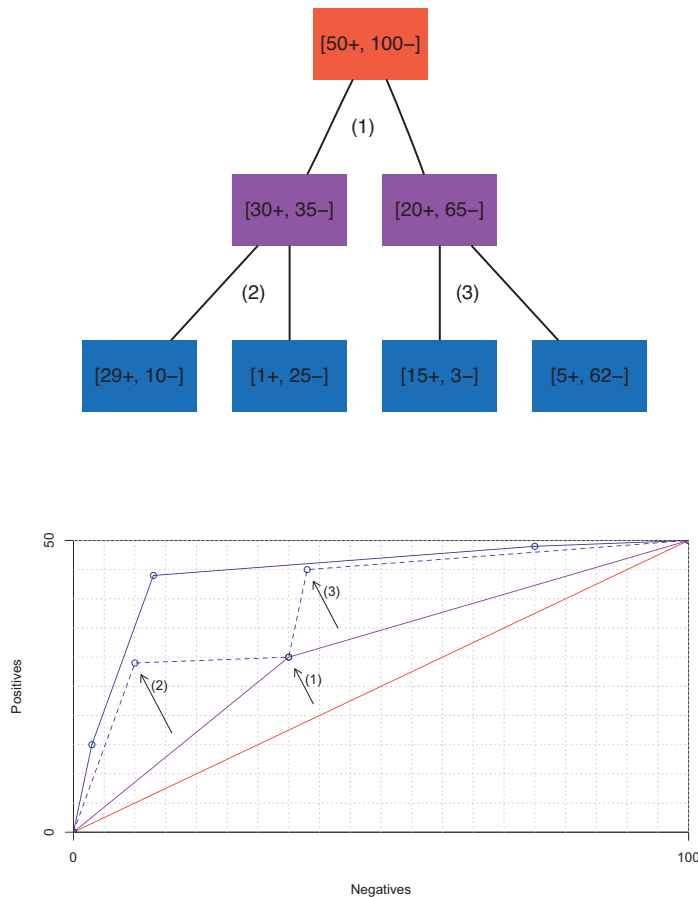
**Example 5.2 (Growing a tree).** Consider the tree in Figure 5.4 (top). Each node is labelled with the numbers of positive and negative examples covered by it: so, for instance, the root of the tree is labelled with the overall class distribution (50 positives and 100 negatives), resulting in the trivial 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:

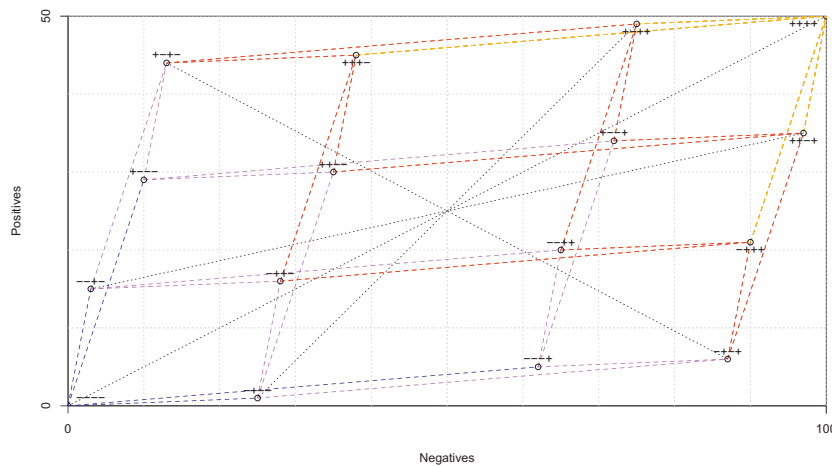
-  split the corresponding curve segment into two or more segments;
-  reorder the segments on decreasing slope.

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 reordering 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

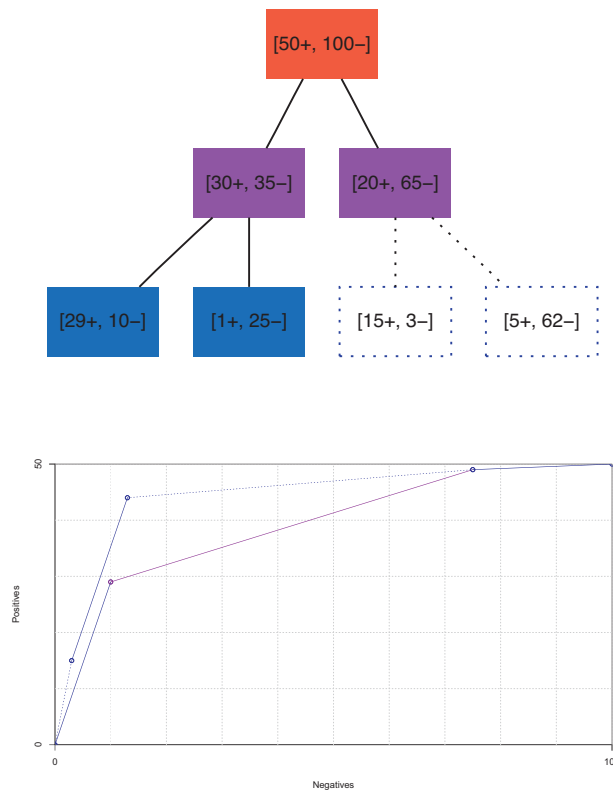
operating point that is optimal under those operating conditions.

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:** *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

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

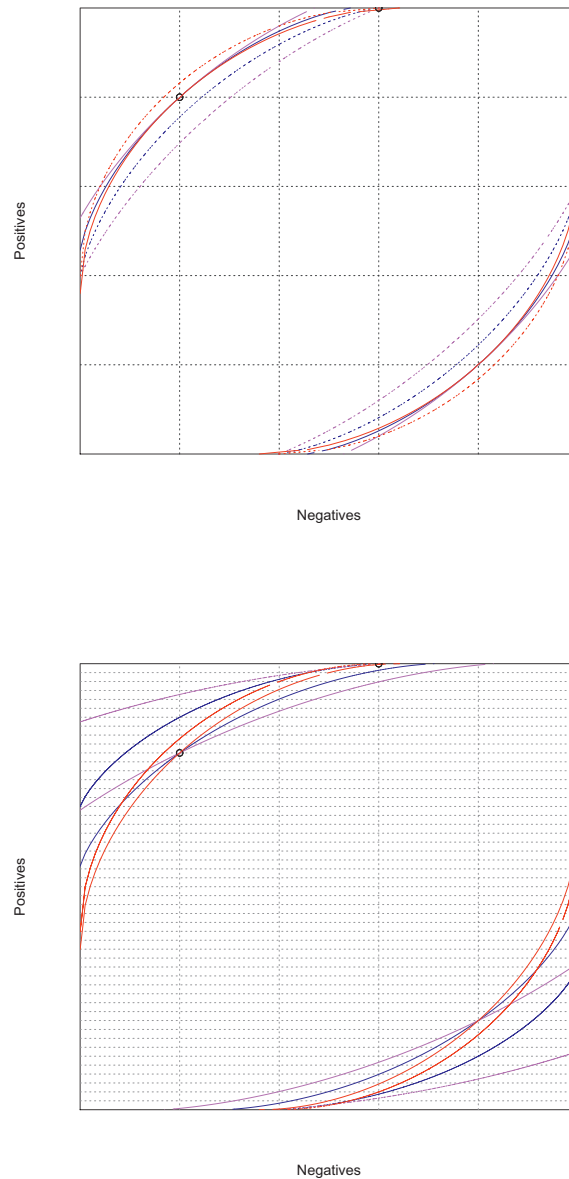
$$\text{weighted impurity of the child: } \frac{n_1}{n} \sqrt{\frac{n_1^{\oplus}}{n_1} \frac{n_1^{\ominus}}{n_1}};$$

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

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.<sup>5</sup>

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.

---

<sup>5</sup>It 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.<sup>6</sup>

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  $\sigma_j^2$  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

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

<sup>6</sup>This 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-

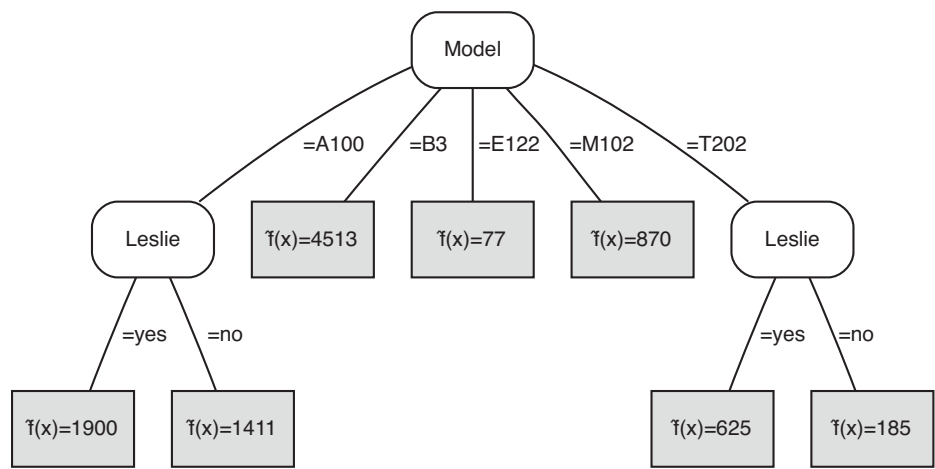


Figure 5.8. A regression tree learned from the data in Example 5.4.

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 (the *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	<b>0</b>	1	<b>1</b>	1	3	<b>0</b>	4	0
6	1	<b>0</b>	2	1	<b>1</b>	2	<b>2</b>	1
13	<b>1</b>	2	<b>0</b>	0	4	<b>0</b>	4	0
10	1	1	0	0	3	0	2	0
3	3	<b>1</b>	4	3	<b>0</b>	4	<b>1</b>	3
13	<b>0</b>	2	<b>0</b>	0	4	<b>0</b>	4	0
3	4	<b>2</b>	4	2	<b>1</b>	4	<b>0</b>	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]

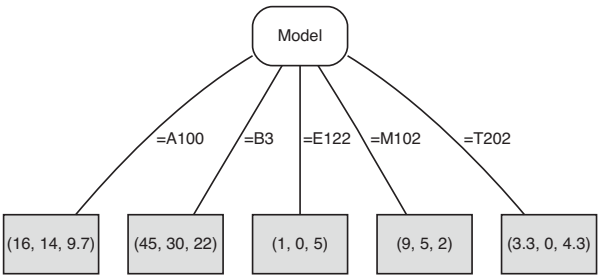
Leslie = [yes, no] [2, 5, 8][1, 3, 4, 6, 7, 9]

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.<sup>7</sup> 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

---

<sup>7</sup> 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.

