CHAPTER

8 Decision trees

Decision trees are another popular and powerful function type for supervised learning. One advantage of decision trees is that they produce very interpretable decision rules; they are easy to evaluate “by hand”, so that the factors that went into the class decision can be easily stated. In a decision tree, the functional form corresponds to a nested sequence of “if-then-else” decisions.

As an example, consider the logical “or” operator, whose truth table is shown in Figure 8.1(a). It is easy to see how this function can be implemented as a sequence of logical tests, for example the code listed in Figure 8.1(b). Then, we can visualize this nested sequence of tests as a tree, with the first test forming the root of the tree, and the leaves corresponding to the “return” statements that output the value of the function.

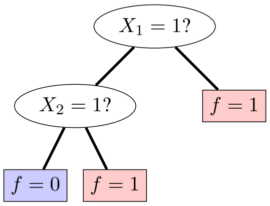
In general, at each node of the decision tree, we perform a comparison based on the features X – typically, this depends only on a single feature X_j, for reasons we will discuss in the sequel – and branch on the result. When the features X_j are non-binary but discrete-valued, there are a number of possible variations on this template, several of which are illustrated in Figure 8.2. A straightforward option is to create one child node for each possible value of X_j; in code, this would correspond to a “switch-case” style branching operation. However, this means that the structure and overall size of the tree can depend on the cardinality of the features selected, and generally complicates implementation. We can restrict our predictors to be representable by a binary tree by ensuring that each comparision is binary in nature. This could test whether X_j takes on a single, particular value, or one of the values in a set.

X_1	X_2	$f(X)$
0	0	0
0	1	1
1	0	1
1	1	1

(a)

```
if X₁ == 1:
    return 1;
else:
    if X₂ == 1:
        return 1;
    else:
        return 0;
```

(b)



(c)

Figure 8.1: (a) A truth table or other function, such as the “or” function shown here, can be represented as (b) a nested series of if-then-else operations, which can then be graphically depicted as (c) a decision tree diagram.

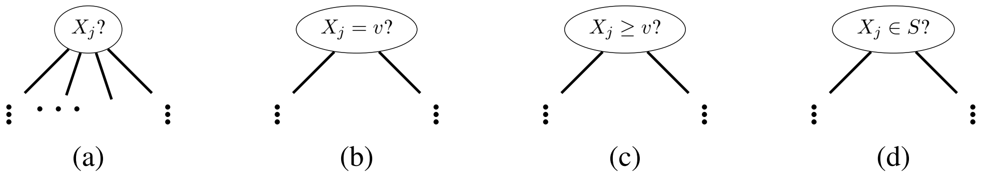


Figure 8.2: Various common architectures for decision trees on discrete features. (a) Branching on all values of a single feature leads to a non-binary tree with one outcome per possible value of X_j. (b) Testing X_j for a single value. (c) A binary test assuming an ordinal structure to the values of X_j. (d) A more general set-based binary test on X_j.

Assuming that the values of X_j can be assigned some ordinal structure (for example, mapped to the values $0 \ldots d-1$), a simple option is to make the binary comparison a threshold operation, $X_j \geq t$ for some t. This has the advantage that it also applies easily to continuous-valued features X_j. In the sequel, we will default to assuming this threshold comparison form for our decision trees.

It is worth noting that decision trees are a so-called *universal* learner, in the sense that given any function $f(X)$ over discrete-valued X, there exists a decision tree that implements the function f. For binary X, for example, at worst we could simply construct a tree that tested each X_j in sequence, with 2^n leaf nodes corresponding to each entry of $f(X)$ in the truth table. However, as Figure 8.1 illustrates, some functions may be more efficient to represent than others, requiring a much smaller tree than the worst case (for example, an "or" function on n features requires only $n+1$ leaves). Clearly, all of the variants in Figure 8.2 are universal in this sense; it is only that some functions may be more or less efficient to represent using one variant or another.

8.1 Predictor Form

Decision trees use a nested branching structure to represent a function of $x \in \mathcal{X}$; for continuous features, these branching decisions typically compare a single feature x_j to a threshold value. This means that the root of the tree can be viewed as splitting the feature space $\mathcal{X}$ into two partitions along some axis-aligned hyperplane, $\{x : x_j = t\}$. The left-hand child will only be reached by a point x with $x_j < t$, and thus its associated comparison partitions one of these two half-spaces with another axis-aligned split. Thus, each node of the tree can be associated with a region of the space defined by this recursive splitting process. This process is visualized in Figure 8.3. Each leaf node is thus associated with some hyper-rectangle of $\mathcal{X}$, and the decision tree's prediction function $f(x)$ outputs a constant value within this region. The "parameters" of the decision tree can thus be viewed as the tree structure itself, along with prediction values for each of the l leaves and a feature index and threshold for each of the $l-1$ non-leaf nodes of the tree.

Unfortunately, this parametric form is difficult to learn using gradient descent or other local search methods. In particular, the threshold operation of a decision tree is discontinuous, and although we could attempt to define a smooth surrogate (such as we used for learning linear classifier models), changes to either the feature index or tree structure would also produce signficant, discontinuous changes to the model's predictions. For example, changing the root feature from X_1 to X_2 in the tree in Figure 8.3 would produce a massive change in the overall prediction function. For that reason, most decision trees are trained using a recursive, greedy approach.

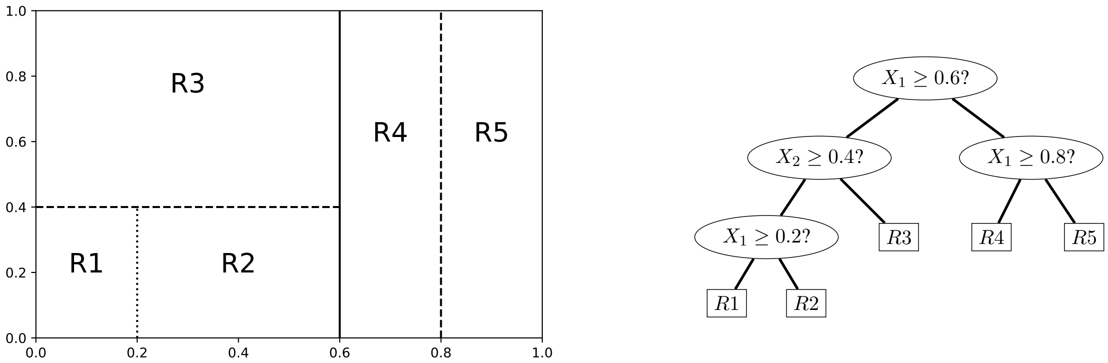


Figure 8.3: A decision tree defined over continuous features partitions the feature space into regions defined by successive axis-aligned hyperplanes.

8.2 Training decision trees

We can build up a decision tree based on a training data set D by constructing it recursively, from root to leaves. This process is outlined in Algorithm 8.1. At each level, we first decide whether the current node s of the tree will be a leaf ("LeafCondition"). If so, we can use the training data within its associated region R_s (i.e., the data that would be arrive at this node when filtered according to the current tree structure) to select the a prediction value for $f(x)$, $x \in R_s$ ("FindBestPrediction"). If the current node will not be a leaf, we instead select a criterion to split the data further ("FindBestSplit"), and then recursively train each of the descendant trees in the same manner.

We consider LeafCondition in more detail in the sequel, but a simple mechanism is to learn a fixed-depth tree, meaning that a node is a leaf if it is at depth d, and otherwise not. Similarly, FindBestPrediction is relatively easy to define. If our decision tree outputs a constant value for all data in the region R_s associated with node s, we can easily select a value based on the training data that fall in this region, $\{(x^{(i)}, y^{(i)}) \in D : x^{(i)} \in R_s\}$. For example, if learning a classifier, the minimum empirical error rate predictor is given by the majority class among these $y^{(i)}$; for regression problems, the minimum empirical mean-squared error is achieved by the average of the $y^{(i)}$.

Non-leaf nodes of the decision tree perform a test on the feature vector X; for our purposes, a threshold comparison on a single feature X_j. Each comparison node thus consists of the selected feature index j, and a threshold value t. We can determine values for these parameters by a simple exhaustive search, looping over all possible features and all possible thresholds and evaluating some score function, then

Algorithm 8.1 BuildTree(D): Greedy training of a decision tree

```
Input: A data set $D = (X, Y)$.
Output: A decision tree.
if LeafCondition($D$) then
    $f_n$ = FindBestPrediction($D$)
else
    $j_n, t_n$ = FindBestSplit($D$)
    $D_L = \{(x^{(i)}, y^{(i)}) : x^{(i)}_{j_n} < t_n\}$   and   $D_R = \{(x^{(i)}, y^{(i)}) : x^{(i)}_{j_n} \geq t_n\}$
    Set left and right children to trees given by BuildTree($D_L$), BuildTree($D_R$), respectively.
end if
```

picking the parameters that result in the best score. Note that for continuous-valued features X, the threshold will be a continuous value; however, there are only a finite number of possible *decisions* to make on a given training set. In particular, when the training data are sorted along the feature being considered, any threshold falling between two consecutive data points results in exactly the same rule on the training data, and thus have indistinguishable performance. We can thus simply enumerate the number of unique decisions, and typically select the midpoint between of the nearest data points above and below the split to use for the threshold.

Score functions

The purpose of a score function is to decide how good any particular split is. We might think that the classification accuracy would make a good score function, since minimizing it is our true goal. However, as usual, classification accuracy is not particularly well behaved. It will often focus on selecting "very specialized" rules that try to get one more data point correct, rather than trying to split groups of data in a more holistic way. Also, among rules that do not get any additional data points correct, it provides no guidance whatsoever.

Simple example? (later?)

One useful score function is based on the entropy of the class values within each subtree. The empirical entropy, measured in bits, for a data set (S) is given by

$$H(p_S) = -\sum_y p_S(y) \log_2 p_S(y)$$

where $p_S(y)$ is the empirical distribution of the class value y, i.e., the fraction of data in set S that have class y. For convenience, we define $0 \log 0 := \lim_{p \to 0} p \log p = 0$.

Example 8-1 : Entropy

Suppose that we had a binary classification data set S consisting of seven examples, with four data points from class 1 and three from class 0. The empirical probability distribution of the data is then given by $\hat{p}_S(Y=1) = \frac{4}{7} = 1 - \hat{p}_S(Y=0)$, and the empirical entropy of the data set is

$$\begin{aligned} H(\hat{p}_S) &= -\hat{p}_S(Y=0)\log_2(p_S(Y=0)) - \hat{p}_S(Y=1)\log_2(p_S(Y=1)) \\ &= -\frac{3}{7}\log_2\left(\frac{3}{7}\right) - \frac{4}{7}\log_2\left(\frac{4}{7}\right) \\ &\approx .5239 + .4613 = .9852 \end{aligned}$$

On the other hand a more skewed data set, say $\hat{p}_S(Y=1) = \frac{6}{7}$, would have a lower entropy, $H([\frac{1}{7}, \frac{6}{7}]) \approx 0.5917$. In general, for a discrete-valued Y, the highest possible entropy is given by the uniform distribution over Y's values, with entropy $\log_2(c)$, where c is the number of values that Y can take on; the smallest possible entropy is zero, given by any deterministic distribution, in which one value has probability 1 and the others probability zero.

How does entropy help us decide on a partitioning? We can use entropy to calculate the (expected) *information gain*, which is the average reduction in entropy we see when we adopt some data split. In particular, suppose that we split a data set S into S_1, S_2 with $S = S_1 \cup S_2$. We compute the expected information gain as

$$IG(S_1, S_2) = \frac{|S_1|}{|S|}\Big(H(p_S) - H(p_{S_1})\Big) + \frac{|S_2|}{|S|}\Big(H(p_S) - H(p_{S_2})\Big)$$

The information gain is always positive, and has maximum value $H(p_S)$.

Example 8-2 : Empirical Information Gain

Consider the data set shown on the right. We have seven data points, each with three binary-valued features and a binary class Y.

X_1	X_2	X_3	Y
0	0	0	0
1	1	0	0
0	1	1	0
0	0	1	1
1	0	0	1
1	1	0	1
1	1	1	1

The empirical $\hat{p}_S(Y = 1) = \frac{4}{7}$, giving $H(\hat{p}_S) \approx .985$. Now, consider partitioning the data by measuring the value of X_1, so that $S_L = \{i : x_1^{(i)} = 0\}$ and $S_R = \{i : x_1^{(i)} = 1\}$. Then, we can see that the empirical distributions of these sets have entropy

$$\hat{p}_{S_L}(Y = 1) = \frac{1}{3} \qquad H(\hat{p}_{S_L}) \approx 0.918$$
$$\hat{p}_{S_R}(Y = 1) = \frac{3}{4} \qquad H(\hat{p}_{S_R}) \approx 0.811$$

and thus has information gain

$$IG(X_1) = \frac{3}{7}\Big(.985 - .918\Big) + \frac{4}{7}\Big(.985 - .811\Big) = 0.128$$

A similar computation on X_2 and X_3 give,

$$IG(X_2) = \frac{3}{7}\Big(.985 - .918\Big) + \frac{4}{7}\Big(.985 - 1.0\Big) = 0.02$$
$$IG(X_3) = \frac{4}{7}\Big(.985 - 1.0\Big) + \frac{3}{7}\Big(.985 - .918\Big) = 0.02$$

We can see that X_1 provides the most information gain of the three options. Intuitively, conditioning on X_1 is the most helpful in improving our ability to predict Y: without measuring X, both values of Y are approximately equally likely, but after measuring X_1, either we are somewhat confident that $Y = 0$ (when $X_1 = 0$), or that $Y = 1$ (when $X_1 = 1$). In contrast, a feature like X_2 is helpful if $X_2 = 0$, but we find that the value of Y is even more uncertain if $X_2 = 1$, so on average we get little (immediate) information out of measuring X_2.

A common alternative to entropy is the so-called *Gini index*, which measures the variance of the class variable, rather than its entropy. The Gini index equivalents of the above equations are:

$$H_{\text{gini}}(p_S) = \sum_y p_S(y)(1 - p_S(y)) \quad = \quad 1 - \sum_y p_S(y)^2$$
$$IG_{\text{gini}}(S_1, S_2) = \frac{|S_1|}{|S|}\Big(H_{\text{gini}}(p_S) - H_{\text{gini}}(p_{S_1})\Big) + \frac{|S_2|}{|S|}\Big(H_{\text{gini}}(p_S) - H_{\text{gini}}(p_{S_2})\Big)$$

Again, H is at its minimum (zero) when the variable y is deterministic (a single class) within the subset S, and IG measures the gain, or increase in determinism, caused by conditioning on the split into subsets S_1, S_2.

Example 8-3 : Empirical Gini Information

Using the same data as before, but evaluating the Gini information gain for X_1 gives:

$$H_{\text{gini}}(p_S) = 0.490 \qquad H_{\text{gini}}(p_{S_L}) = 0.444 \qquad H_{\text{gini}}(p_{S_R}) = 0.375$$

$$IG_{\text{gini}}(X_1) = \frac{3}{7}\Big(.490 - .444\Big) + \frac{4}{7}\Big(.490 - .375\Big) = 0.085$$

We can again compare this information gain to the other options, for example, $IG_{\text{gini}}(X_2) = 0.014$.

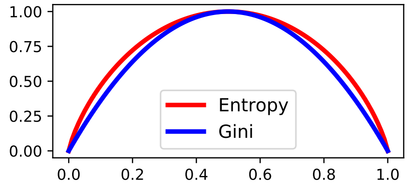


Figure 8.4: Shannon entropy and Gini impurity for binary classes. The two functions are nearly indistinguishable as a function of $p = p(y = 1)$, leading to broadly similar behavior during training.

Both the (Shannon) entropy and the Gini index are widely used impurity scores; which is used is largely a matter of taste. For binary-valued classes, both scores are extremely similar in their values (see Figure 8.4), ranging from zero at $p = 0$ or $p = 1$ (all one class), and reaching their maximum at $p = \frac{1}{2}$ (data equally distributed between the two classes).[1] Although even slight differences in impurity score functions can lead to different choices of which split to perform next, both score functions encourage the same general behavior: a preference for splits which lead to large sets S_i in which one class is heavily dominant. Which measure to use in practice is largely a matter of personal taste, although the Gini impurity does not require taking a logarithm and is thus slightly faster to evaluate.

The overall procedure for finding the best split is then shown in Algorithm 8.2. We loop over each possible feature j and each possible split of the data along feature j, and keep track of which feature and split provide the largest information gain. Since in decision tree learning, we are comparing possible splits S_L, S_R of a fixed set of data S, the entropy of the full set S, $H(p_S)$, is constant, and we can equivalently minimize the conditional entropy given the split, $\frac{|S_L|}{|S|}H(p_{S_L}) + \frac{|S_R|}{|S|}H(p_{S_R})$.

Computational Complexity of Decision Tree Learning

Although the brute-force nature of the decision tree training in Algorithms 8.1 and 8.2 seem inefficient, they are reasonably practical. Consider the total computational requirements for learning a tree of depth d from a training data set of size m, with n features per data point.

For each of the n features, we sort the data, then run over the sorted data scoring each split. Since the probabilities p^L and p^R can be incrementally updated while running over the data, the cost of this procedure is dominated by the data sorting step, and thus has total cost $O(n \cdot m \log(m))$.

Having identified a split, we partition the data into left and right children, with m_L and m_R data each, $m_L + m_R = m$. We then repeat the same procedure on each child; however, the total work for the children is less than that of the parent node, since

$$n \cdot m_L \log(m_L) + n \cdot m_R \log(m_R) \leq n \cdot (m_L + m_R) \log(m) = n \cdot m \log(m).$$

Adding up the work for each level of the depth-d tree, we have that the total work is at most $O(d \cdot n \cdot m \log(m))$.

[1] Figure 8.4 technically shows $H(p)$ and $2H_{\text{gini}}(p)$, so that their values will also coincide at $p = \frac{1}{2}$.

Algorithm 8.2 FindBestSplit(D)

```
Input: A data set $D = (X, Y)$ of size $m$ and impurity function $H(\cdot)$.

Output: A feature $j^*$ and threshold $t^*$ whose split minimizes impurity $H$

Initialize $H^* = \infty$
for each feature $j$ do
    Sort $\{x_j^{(i)}\}$ in order of increasing value
    for each $i$ such that $x_j^{(i)} < x_j^{(i+1)}$ do
        Compute $p_y^L = \frac{1}{i} \sum_{k \le i} \mathbb{1}[y^{(k)} = y]$ and $p_y^R = \frac{1}{k-i} \sum_{k > i} \mathbb{1}[y^{(k)} = y]$
        Set $H' = \frac{i}{m} H(p^L) + \frac{m-i}{m} H(p^R)$
        if $H' < H^*$ then
            Set $j^* = j$, $t^* = (x^{(i)} + x^{(i+1)})/2$, $H^* = H'$
        end if
    end for
end for
Return $j^*$, $t^*$
```

Complexity Control and Pruning

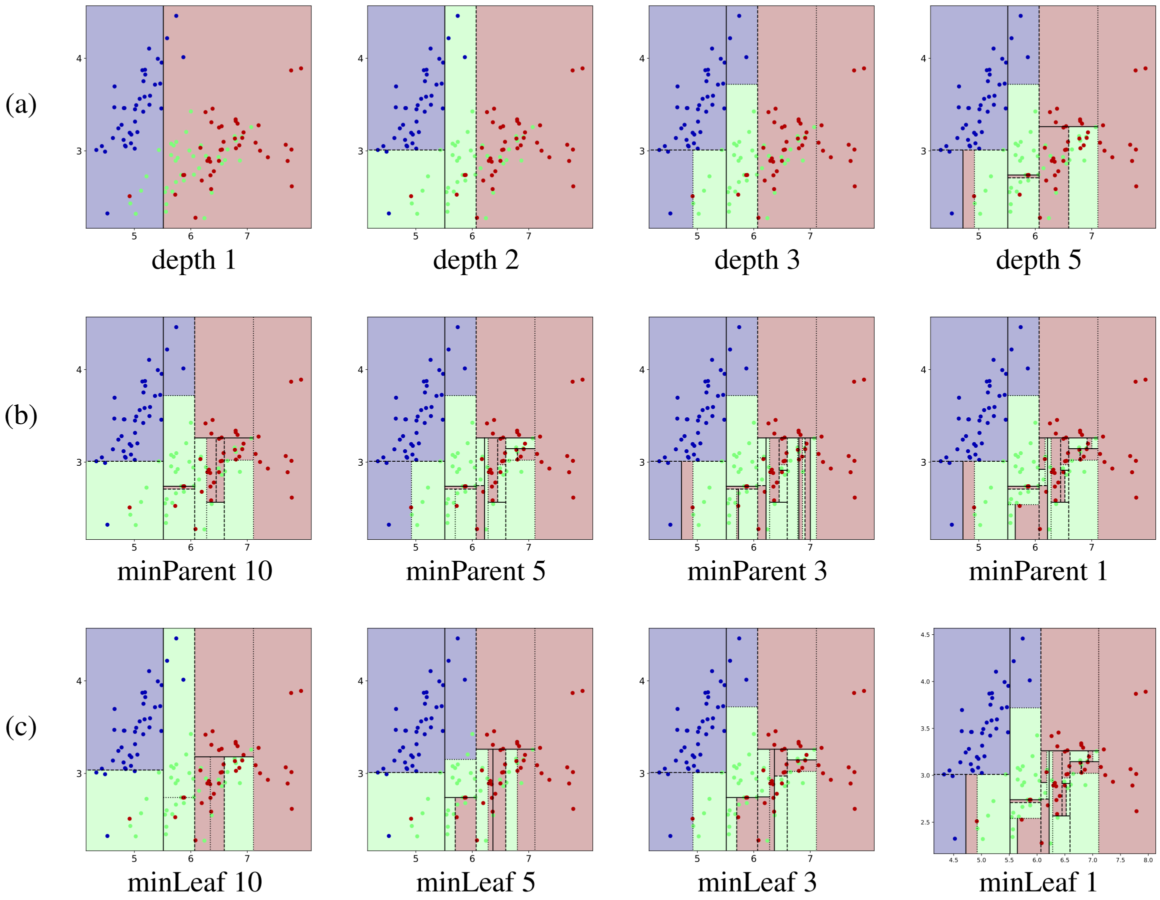
Perhaps the most major factor in the complexity of a function defined on a decision tree is the number of nodes (or number of leaf nodes) in the tree. Since each leaf s is associated with a region R_s with an independent prediction value, more leaf nodes means more flexibility in the function. As mentioned previously, a simple mechanism to control the number of nodes in the tree is to limit the depth of the tree; if a tree has depth d, it can have no more than 2^d leaves.[2]

Another simple rule to reduce the size of the tree is to refrain from splitting when doing so will not change the function $f(x)$. For example, if the remaining data are indistinguishable (all have the same feature vector x), no splitting can improve the prediction. Alternatively, if all the data are already correctly predicted, there is no reason to split further. The tree for $f(x) = (x_1 \text{ or } x_2)$ in Figure 8.1 illustrates this point; if $x_1 = 1$, there is no reason to split on x_2, even though the other branch of the tree requires depth two. This idea can be generalized to refrain from splitting a node when the prediction is already "sufficiently accurate" on the data at that node of the tree.

Finally, we may elect not to split a node because we do not believe we have enough data remaining to accurately estimate a more refined function. This could be implemented via a rule that enforced that no node would be split if it had fewer than k training points (enforcing a minimum number of data points to be a parent node, or `minParent`). Alternatively, we could enforce that any leaf node needs to have at least k' data points to ensure that f is accurately estimated on that region; this enforces a minimum number of data points to create a leaf (`minLeaf`). These rules lead to decision trees that can fit more complex functions (using more depth) in regions of the space with lots of data, while avoiding overfitting in regions of the space that have few data.

We can also decide to reduce the complexity of our decision tree representation *after* constructing it. Reducing complexity may be particularly desirable if we feel that the extra depth did not significantly improve our performance. It is often hard to tell whether a split will significantly improve performance when the tree is initially being constructed. For example, it is easy to make examples where one split provides no measurable gain in accuracy or score, but allows the next level's split to have significant

[2] A simple variant of this approach limits the number of leaf nodes of the decision tree; this requires that the tree be built in breadth-first fashion, with some sort of priority for splitting the nodes, but is otherwise straightforward.



depth 1 depth 2 depth 3 depth 5

minParent 10 minParent 5 minParent 3 minParent 1

minLeaf 10 minLeaf 5 minLeaf 3 minLeaf 1

Figure 8.5: Complexity control of a decision tree using (a) maximum depth; (b) minimum number of data points required to form a parent (internal) node; (c) minimum number of data points required to form a leaf (decision) node. The "simplest" models (least prone to overfitting) are shown on the far left, with less complexity control applied moving to the right.

gains. For this reason, one usually constructs the entire tree and then "prunes". Given the full decision tree, we start at the leaves and walk upward, checking whether each parent had an accuracy nearly equal to that given by its children. If the gain is below some threshold, we prune the children and continue upward; if not, we cease recursing for this node or its ancestors.

8.3 Decision Stumps

A decision ”stump” is a single-layer decision tree, i.e., a threshold value applied to a single feature. Although an extremely weak learner (it can only represent extremely simple decision boundaries), it is commonly used in techniques that leverage many weak learners to create a single more powerful learner, such as ensemble methods discussed in Chapter 9.