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.

Start with discrete (binary) features? Example?

Initially, let us assume that the features $x^{(i)}$ are all continuous-valued; we will discuss discrete-valued features in the sequel.

For example, a simple predictor

MORE; example function and tree and output

8.1 Predictor form

More generally, at each node of the tree, we perform a comparison based on the data point's features – typically, this depends only on one feature of $x^{(i)}$ – and branch on the result. For continuous-valued features, we can compare the feature's value to some threshold t.

For discrete-valued features, there are a number of possible variations on this template. If the decision tree is allowed to have more than two children per node, one option is to branch on each possible value of the feature. For example, a comparison of a discrete feature could have one child per possible value; however, this means that the branching and overall size of the tree can depend on the cardinality of the features selected, and generally complicates implementation.

For binary trees, another option is

A common choice to make the same procedure work easily with both continuous- and discrete-valued features is to apply the same thresholding comparisions previously discussed. If the discrete values have an ordinal nature (e.g., discrete categories corresponding to a user's age, so that higher categories indicate older users), this may make sense; alternatively, if the features are pre-processed and converted to a one-hot representation, a single-feature threshold will correspond to branching on one

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
```

Algorithm 8.2 FindBestSplit(D)

```
Input: A data set $D = (X, Y)$ of size $m$ and impurity function $H(\cdot)$.
Output: A split $j^*$, $t^*$ minimizing impurity $H$
Initialize $H^* = 0$
for each feature $j$ do
    Sort $\{x^{(i)}_j\}$ in order of increasing value
    for each $i$ such that $x^{(i)} < x^{(i+1)}$ do
        Compute $p^L_c = \frac{1}{i} \sum_{k \leq i} \mathbb{1}[y^{(k)} = c]$ and $p^R_c = \frac{1}{k-i} \sum_{k > i} \mathbb{1}[y^{(k)} = c]$
        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 $j^*$, $t^*$
```

8.2 Training decision trees

In this section, we turn our attention to how the parameterized form of a decision tree can be learned from data. Unfortunately, the threshold operation of a decision tree is discontinuous, making it difficult to apply gradient descent; and the nested branches make it similarly difficult to define a smooth surrogate (such as we used for linear classifiers) for training. Instead, we typically use a greedy selection process, described next.

8.3 Decision Tree Classifiers

A decision tree classifier consists of a sequence of ”comparison nodes”, at which a single feature of the data point is examined. For a continuous-valued feature, the decision node compares the feature value to a threshold, and depending on whether the value is above or below the threshold, recurses down the decision tree to the left or right. At some point, this process reaches a ”decision node”, at which one of the possible class categories is output.

For discrete-valued features, it does not really make sense to ”threshold” a value. Instead, there are a number of possible options. The most straightforward is to have one child node per possible feature value.

TBD

However, this results in a non-binary tree, possibly with a high branching factor, and can complicate the score function used in learning (see discussion in Duda and Hart). Another possibility is to keep the binary tree shape, in which case some discrete values are assigned to the ”left” and all others to the ”right” child.

8.4 Learning Decision Trees

Each comparison node of a decision tree consists of the selected feature index, and a threshold for comparison. Typically, we determine the values of these parameters by a simple exhaustive search, looping over all possible features and all possible thresholds and evaluating some score function, then picking the parameters that result in the best score. Note that, although the threshold is a continuous value, 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 given data points results in exactly the same rule on the training data, and thus are typically indistinguishable. We can thus enumerate the number of unique thresholds, and typically pick the mean of the two nearest data points as the value.

Score functions

The purpose of a score function is to decide how good any particular split is. You 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.

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.

How does entropy help us decide on a partitioning? We can use entropy to calculate the so-called ”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

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

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 S1, S2.

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 (see Figure 8.1) in their values, 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). 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.

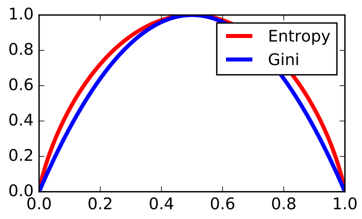


Figure 8.1: Shannon entropy and Gini impurity for binary classes. The two functions are nearly indistinguishable as a function of $p = p(y = 1)$, leading to broadly similar behavior during training.

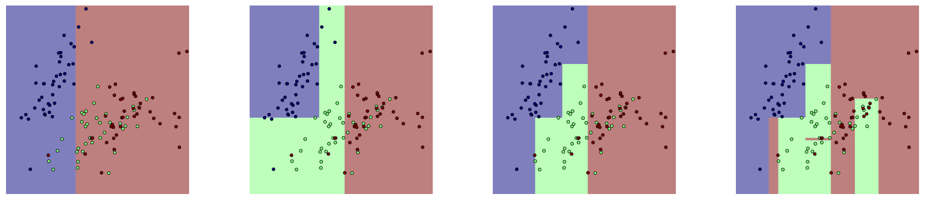
Computational complexity? $nm\log(m)$ **root; then two children are** $nm_L\log(m_L) + nm_R\log(m_R) < n(m_L + m_R)\log(m) = nm\log(m)$**. So max depth** d**:** $< O(dnm\log(m))$**.**

Complexity Control and Pruning

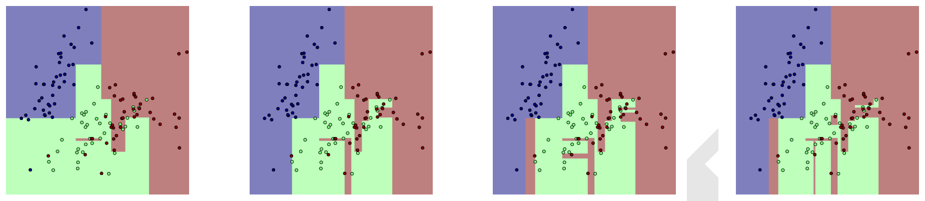
The complexity of a decision tree is essentially determined by its depth. (How many parameters can a binary decision tree of depth d have?) We may therefore want to control this complexity by reducing the depth. Common stopping rules include not proceeding past some maximum depth d, or not continuing to split nodes of the tree that have fewer than K training data points associated with them (since we may not trust our ability to learn a general rule based on so few data).

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 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.

(a)



(b)



(c)

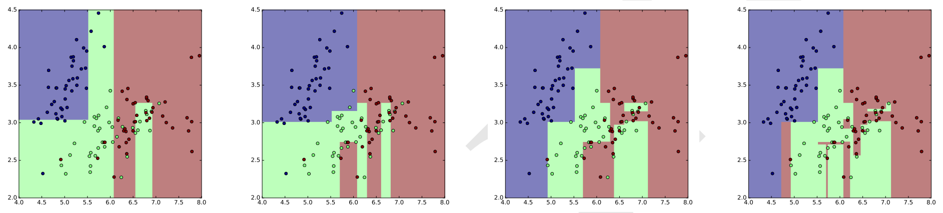


Figure 8.2: Complexity control of a decision tree using (a) maximum depth; (b) minimum number of data points required to form a parent (internal) node; (c) minimum number of data points required to form a leaf (decision) node.

8.5 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.