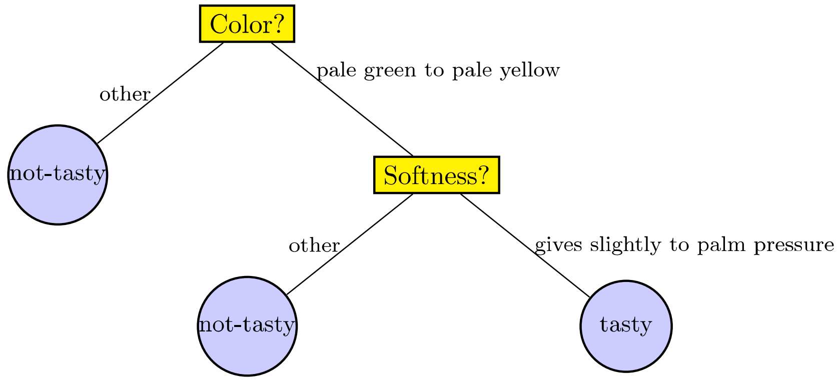
18 Decision Trees

A decision tree is a predictor, $h : \mathcal{X} \rightarrow \mathcal{Y}$, that predicts the label associated with an instance $\mathbf{x}$ by traveling from a root node of a tree to a leaf. For simplicity we focus on the binary classification setting, namely, $\mathcal{Y} = \{0, 1\}$, but decision trees can be applied for other prediction problems as well. At each node on the root-to-leaf path, the successor child is chosen on the basis of a splitting of the input space. Usually, the splitting is based on one of the features of $\mathbf{x}$ or on a predefined set of splitting rules. A leaf contains a specific label. An example of a decision tree for the papayas example (described in Chapter 2) is given in the following:



To check if a given papaya is tasty or not, the decision tree first examines the color of the Papaya. If this color is not in the range pale green to pale yellow, then the tree immediately predicts that the papaya is not tasty without additional tests. Otherwise, the tree turns to examine the softness of the papaya. If the softness level of the papaya is such that it gives slightly to palm pressure, the decision tree predicts that the papaya is tasty. Otherwise, the prediction is "not-tasty." The preceding example underscores one of the main advantages of decision trees – the resulting classifier is very simple to understand and interpret.

18.1 Sample Complexity

A popular splitting rule at internal nodes of the tree is based on thresholding the value of a single feature. That is, we move to the right or left child of the node on the basis of $\mathbb{1}_{[x_i<\theta]}$, where $i \in [d]$ is the index of the relevant feature and $\theta \in \mathbb{R}$ is the threshold. In such cases, we can think of a decision tree as a splitting of the instance space, $\mathcal{X} = \mathbb{R}^d$, into cells, where each leaf of the tree corresponds to one cell. It follows that a tree with k leaves can shatter a set of k instances. Hence, if we allow decision trees of arbitrary size, we obtain a hypothesis class of infinite VC dimension. Such an approach can easily lead to overfitting.

To avoid overfitting, we can rely on the minimum description length (MDL) principle described in Chapter 7, and aim at learning a decision tree that on one hand fits the data well while on the other hand is not too large.

For simplicity, we will assume that $\mathcal{X} = \{0,1\}^d$. In other words, each instance is a vector of d bits. In that case, thresholding the value of a single feature corresponds to a splitting rule of the form $\mathbb{1}_{[x_i=1]}$ for some $i = [d]$. For instance, we can model the "papaya decision tree" earlier by assuming that a papaya is parameterized by a two-dimensional bit vector $\mathbf{x} \in \{0,1\}^2$, where the bit x_1 represents whether the color is pale green to pale yellow or not, and the bit x_2 represents whether the softness is gives slightly to palm pressure or not. With this representation, the node Color? can be replaced with $\mathbb{1}_{[x_1=1]}$, and the node Softness? can be replaced with $\mathbb{1}_{[x_2=1]}$. While this is a big simplification, the algorithms and analysis we provide in the following can be extended to more general cases.

With the aforementioned simplifying assumption, the hypothesis class becomes finite, but is still very large. In particular, any classifier from $\{0,1\}^d$ to $\{0,1\}$ can be represented by a decision tree with 2^d leaves and depth of $d+1$ (see Exercise 1). Therefore, the VC dimension of the class is 2^d, which means that the number of examples we need to PAC learn the hypothesis class grows with 2^d. Unless d is very small, this is a huge number of examples.

To overcome this obstacle, we rely on the MDL scheme described in Chapter 7. The underlying prior knowledge is that we should prefer smaller trees over larger trees. To formalize this intuition, we first need to define a description language for decision trees, which is prefix free and requires fewer bits for smaller decision trees. Here is one possible way: A tree with n nodes will be described in $n+1$ blocks, each of size $\log_2(d+3)$ bits. The first n blocks encode the nodes of the tree, in a depth-first order (preorder), and the last block marks the end of the code. Each block indicates whether the current node is:

- An internal node of the form $\mathbb{1}_{[x_i=1]}$ for some $i \in [d]$
- A leaf whose value is 1
- A leaf whose value is 0
- End of the code

Overall, there are $d+3$ options, hence we need $\log_2(d+3)$ bits to describe each block.

Assuming each internal node has two children,[1] it is not hard to show that this is a prefix-free encoding of the tree, and that the description length of a tree with n nodes is $(n+1)\log_2(d+3)$.

By Theorem 7.7 we have that with probability of at least $1-\delta$ over a sample of size m, for every n and every decision tree $h \in \mathcal{H}$ with n nodes it holds that

$$L_{\mathcal{D}}(h) \leq L_S(h) + \sqrt{\frac{(n+1)\log_2(d+3) + \log(2/\delta)}{2m}}. \tag{18.1}$$

This bound performs a tradeoff: on the one hand, we expect larger, more complex decision trees to have a smaller training risk, $L_S(h)$, but the respective value of n will be larger. On the other hand, smaller decision trees will have a smaller value of n, but $L_S(h)$ might be larger. Our hope (or prior knowledge) is that we can find a decision tree with both low empirical risk, $L_S(h)$, and a number of nodes n not too high. Our bound indicates that such a tree will have low true risk, $L_{\mathcal{D}}(h)$.

18.2 Decision Tree Algorithms

The bound on $L_{\mathcal{D}}(h)$ given in Equation (18.1) suggests a learning rule for decision trees – search for a tree that minimizes the right-hand side of Equation (18.1). Unfortunately, it turns out that solving this problem is computationally hard.[2] Consequently, practical decision tree learning algorithms are based on heuristics such as a greedy approach, where the tree is constructed gradually, and locally optimal decisions are made at the construction of each node. Such algorithms cannot guarantee to return the globally optimal decision tree but tend to work reasonably well in practice.

A general framework for growing a decision tree is as follows. We start with a tree with a single leaf (the root) and assign this leaf a label according to a majority vote among all labels over the training set. We now perform a series of iterations. On each iteration, we examine the effect of splitting a single leaf. We define some "gain" measure that quantifies the improvement due to this split. Then, among all possible splits, we either choose the one that maximizes the gain and perform it, or choose not to split the leaf at all.

In the following we provide a possible implementation. It is based on a popular decision tree algorithm known as "ID3" (short for "Iterative Dichotomizer 3"). We describe the algorithm for the case of binary features, namely, $\mathcal{X} = \{0,1\}^d$,

[1] We may assume this without loss of generality, because if a decision node has only one child, we can replace the node by its child without affecting the predictions of the decision tree.

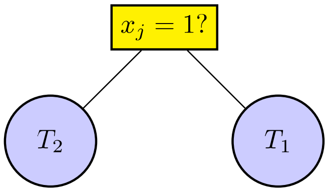
[2] More precisely, if NP≠P then no algorithm can solve Equation (18.1) in time polynomial in n, d, and m.

and therefore all splitting rules are of the form $\mathbb{1}_{[x_i=1]}$ for some feature $i \in [d]$. We discuss the case of real valued features in Section 18.2.3.

The algorithm works by recursive calls, with the initial call being $\texttt{ID3}(S, [d])$, and returns a decision tree. In the pseudocode that follows, we use a call to a procedure $\texttt{Gain}(S, i)$, which receives a training set S and an index i and evaluates the gain of a split of the tree according to the ith feature. We describe several gain measures in Section 18.2.1.

```
                    ID3($S, A$)

INPUT: training set $S$, feature subset $A \subseteq [d]$
if all examples in $S$ are labeled by 1, return a leaf 1
if all examples in $S$ are labeled by 0, return a leaf 0
if $A = \emptyset$, return a leaf whose value = majority of labels in $S$
else :
  Let $j = \text{argmax}_{i \in A}\, \texttt{Gain}(S, i)$
  if all examples in $S$ have the same label
    Return a leaf whose value = majority of labels in $S$
  else
    Let $T_1$ be the tree returned by $\texttt{ID3}(\{(\mathbf{x}, y) \in S : x_j = 1\}, A \setminus \{j\})$.
    Let $T_2$ be the tree returned by $\texttt{ID3}(\{(\mathbf{x}, y) \in S : x_j = 0\}, A \setminus \{j\})$.
    Return the tree:
```



18.2.1 Implementations of the Gain Measure

Different algorithms use different implementations of $\texttt{Gain}(S, i)$. Here we present three. We use the notation $\mathbb{P}_S[F]$ to denote the probability that an event holds with respect to the uniform distribution over S.

Train Error: The simplest definition of gain is the decrease in training error. Formally, let $C(a) = \min\{a, 1-a\}$. Note that the training error before splitting on feature i is $C(\mathbb{P}_S[y = 1])$, since we took a majority vote among labels. Similarly, the error after splitting on feature i is

$$\underset{S}{\mathbb{P}}[x_i = 1]\, C(\underset{S}{\mathbb{P}}[y = 1|x_i = 1]) + \underset{S}{\mathbb{P}}[x_i = 0] C(\underset{S}{\mathbb{P}}[y = 1|x_i = 0]).$$

Therefore, we can define $\texttt{Gain}$ to be the difference between the two, namely,

$$\begin{aligned} \texttt{Gain}(S, i) := {} & C(\underset{S}{\mathbb{P}}[y = 1]) \\ & - \left(\underset{S}{\mathbb{P}}[x_i = 1]\, C(\underset{S}{\mathbb{P}}[y = 1|x_i = 1]) + \underset{S}{\mathbb{P}}[x_i = 0] C(\underset{S}{\mathbb{P}}[y = 1|x_i = 0])\right). \end{aligned}$$

Information Gain: Another popular gain measure that is used in the ID3 and C4.5 algorithms of Quinlan (1993) is the information gain. The information gain is the difference between the entropy of the label before and after the split, and is achieved by replacing the function C in the previous expression by the entropy function,

$$C(a) = -a\log(a) - (1-a)\log(1-a).$$

Gini Index: Yet another definition of a gain, which is used by the CART algorithm of Breiman, Friedman, Olshen & Stone (1984), is the Gini index,

$$C(a) = 2a(1-a).$$

Both the information gain and the Gini index are smooth and concave upper bounds of the train error. These properties can be advantageous in some situations (see, for example, Kearns & Mansour (1996)).

18.2.2 Pruning

The ID3 algorithm described previously still suffers from a big problem: The returned tree will usually be very large. Such trees may have low empirical risk, but their true risk will tend to be high – both according to our theoretical analysis, and in practice. One solution is to limit the number of iterations of ID3, leading to a tree with a bounded number of nodes. Another common solution is to *prune* the tree after it is built, hoping to reduce it to a much smaller tree, but still with a similar empirical error. Theoretically, according to the bound in Equation (18.1), if we can make n much smaller without increasing $L_S(h)$ by much, we are likely to get a decision tree with a smaller true risk.

Usually, the pruning is performed by a bottom-up walk on the tree. Each node might be replaced with one of its subtrees or with a leaf, based on some bound or estimate of $L_{\mathcal{D}}(h)$ (for example, the bound in Equation (18.1)). A pseudocode of a common template is given in the following.

```
Generic Tree Pruning Procedure

input:
  function $f(T, m)$ (bound/estimate for the generalization error
    of a decision tree $T$, based on a sample of size $m$),
  tree $T$.
foreach node $j$ in a bottom-up walk on $T$ (from leaves to root):
  find $T'$ which minimizes $f(T', m)$, where $T'$ is any of the following:
    the current tree after replacing node $j$ with a leaf 1.
    the current tree after replacing node $j$ with a leaf 0.
    the current tree after replacing node $j$ with its left subtree.
    the current tree after replacing node $j$ with its right subtree.
    the current tree.
  let $T := T'$.
```

18.2.3 Threshold-Based Splitting Rules for Real-Valued Features

In the previous section we have described an algorithm for growing a decision tree assuming that the features are binary and the splitting rules are of the form $\mathbb{1}_{[x_i=1]}$. We now extend this result to the case of real-valued features and threshold-based splitting rules, namely, $\mathbb{1}_{[x_i<\theta]}$. Such splitting rules yield decision stumps, and we have studied them in Chapter 10.

The basic idea is to reduce the problem to the case of binary features as follows. Let $\mathbf{x}_1, \ldots, \mathbf{x}_m$ be the instances of the training set. For each real-valued feature i, sort the instances so that $x_{1,i} \leq \cdots \leq x_{m,i}$. Define a set of thresholds $\theta_{0,i}, \ldots, \theta_{m+1,i}$ such that $\theta_{j,i} \in (x_{j,i}, x_{j+1,i})$ (where we use the convention $x_{0,i} = -\infty$ and $x_{m+1,i} = \infty$). Finally, for each i and j we define the binary feature $\mathbb{1}_{[x_i<\theta_{j,i}]}$. Once we have constructed these binary features, we can run the ID3 procedure described in the previous section. It is easy to verify that for any decision tree with threshold-based splitting rules over the original real-valued features there exists a decision tree over the constructed binary features with the same training error and the same number of nodes.

If the original number of real-valued features is d and the number of examples is m, then the number of constructed binary features becomes dm. Calculating the `Gain` of each feature might therefore take $O(dm^2)$ operations. However, using a more clever implementation, the runtime can be reduced to $O(dm\log(m))$. The idea is similar to the implementation of ERM for decision stumps as described in Section 10.1.1.

18.3 Random Forests

As mentioned before, the class of decision trees of arbitrary size has infinite VC dimension. We therefore restricted the size of the decision tree. Another way to reduce the danger of overfitting is by constructing an ensemble of trees. In particular, in the following we describe the method of *random forests*, introduced by Breiman (2001).

A random forest is a classifier consisting of a collection of decision trees, where each tree is constructed by applying an algorithm A on the training set S and an additional random vector, θ, where θ is sampled i.i.d. from some distribution. The prediction of the random forest is obtained by a majority vote over the predictions of the individual trees.

To specify a particular random forest, we need to define the algorithm A and the distribution over θ. There are many ways to do this and here we describe one particular option. We generate θ as follows. First, we take a random subsample from S with replacements; namely, we sample a new training set S' of size m' using the uniform distribution over S. Second, we construct a sequence $I_1, I_2, \ldots$, where each I_t is a subset of $[d]$ of size k, which is generated by sampling uniformly at random elements from $[d]$. All these random variables form the vector θ. Then, the algorithm A grows a decision tree (e.g., using the ID3 algorithm) based on the sample S', where at each splitting stage of the algorithm, the algorithm is restricted to choosing a feature that maximizes Gain from the set I_t. Intuitively, if k is small, this restriction may prevent overfitting.

18.4 Summary

Decision trees are very intuitive predictors. Typically, if a human programmer creates a predictor it will look like a decision tree. We have shown that the VC dimension of decision trees with k leaves is k and proposed the MDL paradigm for learning decision trees. The main problem with decision trees is that they are computationally hard to learn; therefore we described several heuristic procedures for training them.

18.5 Bibliographic Remarks

Many algorithms for learning decision trees (such as ID3 and C4.5) have been derived by Quinlan (1986). The CART algorithm is due to Breiman et al. (1984). Random forests were introduced by Breiman (2001). For additional reading we refer the reader to (Hastie, Tibshirani & Friedman 2001, Rokach 2007).

The proof of the hardness of training decision trees is given in Hyafil & Rivest (1976).

18.6 Exercises

1. 1. Show that any binary classifier $h : \{0,1\}^d \mapsto \{0,1\}$ can be implemented as a decision tree of height at most $d+1$, with internal nodes of the form $(x_i = 0?)$ for some $i \in \{1,\dots,d\}$.
   2. Conclude that the VC dimension of the class of decision trees over the domain $\{0,1\}^d$ is 2^d.
2. **(Suboptimality of ID3)**
   Consider the following training set, where $\mathcal{X} = \{0,1\}^3$ and $\mathcal{Y} = \{0,1\}$:

$$((1,1,1),1)$$
$$((1,0,0),1)$$
$$((1,1,0),0)$$
$$((0,0,1),0)$$

Suppose we wish to use this training set in order to build a decision tree of depth 2 (i.e., for each input we are allowed to ask two questions of the form $(x_i = 0?)$ before deciding on the label).

1. Suppose we run the ID3 algorithm up to depth 2 (namely, we pick the root node and its children according to the algorithm, but instead of keeping on with the recursion, we stop and pick leaves according to the majority label in each subtree). Assume that the subroutine used to measure the quality of each feature is based on the entropy function (so we measure the *information gain*), and that if two features get the same score, one of them is picked arbitrarily. Show that the training error of the resulting decision tree is at least $1/4$.
2. Find a decision tree of depth 2 that attains zero training error.