Machine Learning: Foundations		**Fall semester, 2010/1**
Lecture 12	**Decision trees**	January 9, 2011
Lecturer: Yishay Mansour		*Scribe: Dedi Gadot Mor Sela Lior Shafir*

12.1 Decision Tree: Building

12.1.1 Introduction

We are interested in learning using decision trees as the classification method. We are given samples $< x_i, b_i >$ for some problem. We want to use them to build a decision tree. We would like our decision tree to have two qualities :
1.)We would like the tree classify most/all the sample points correctly.
2.)We would like the tree to be small.
By having these two qualities the tree is guaranteed to guarantee learning by Occam razor.

Preliminaries

There is a general predicates set $H = \{x_i \geq \alpha | \alpha \in R\}$. Each internal node in the decision tree is labeled with some predicate $h \in H$ and each leaf is labeled with 0 or 1, thus giving classification to the input.
To classify the input one starts at the root, going down the tree according to the values of predicate in the internal nodes until a leaf is reached and its label determines the input's classification (see Figure 12.1).

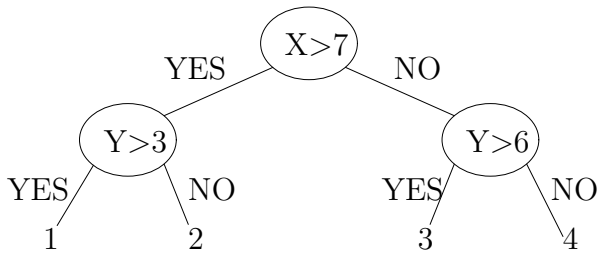


Figure 12.1: Example of a Decision tree

Some algorithms for building decision trees were suggested in the Machine Learning literature (C4.5, CART software packages). They have become very popular during the

90's because decision trees have an advantage to be easily interpretable by human beings, relatively simple and fast to compute. Today they are less popular then SVM or AdaBoost, but generally speaking their performance is comparable.

12.1.2 Top-Down Approach and Cost Function

The general idea how to build a decision tree is to grow it from root to the leaves by repeatedly splitting an existing leaf with a "good" internal node.
What do we consider as a "good" node ?
Knowing to answer that question will supply us a recursive algorithm for building a decision tree.
First we decide what predicate is "good" for the root then build the left and right subtrees recursively. Let's try to define "good". Our goal for the tree is to reduce the error. One simple option is to try to minimize the error at each node. Let the error at leaf v be $E(v)$, then the tree error is $E(T) = \sum_{v \in T} Prob[v]E(v)$
Still that doesn't give us any idea of what should be considered as a split. So we'll use a greedy algorithm that will minimize a local cost function $G(q)$ in each step.
The most intuitive local cost function $G(q)$ is the leaf error function: $G(q) = min\{q, 1-q\}$. where q is the fraction of the positive samples.
Lets consider the following example. In the above example we have a situation where a q fraction of the inputs that reach the root are 1 positive. With probability τ the predicate H is 0 (and $1-\tau$ it is 1). At each leaf p and r fraction of the inputs are positive. Clearly $q = \tau p + (1-\tau)r$.

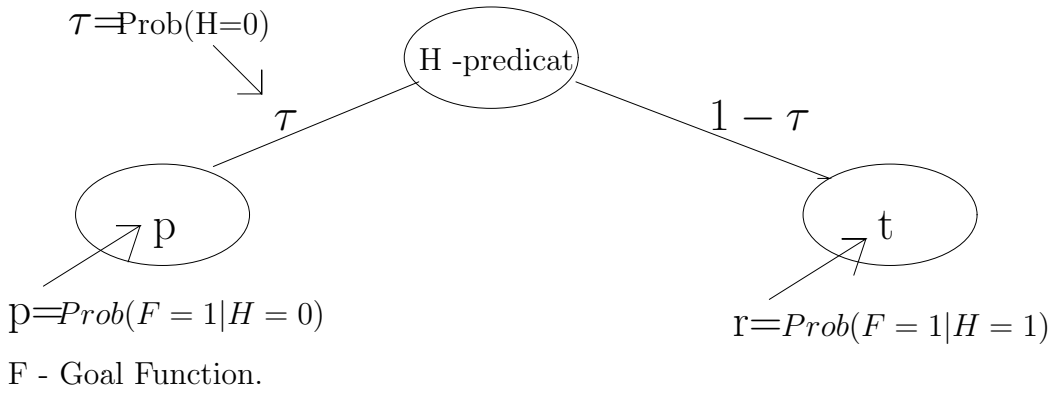
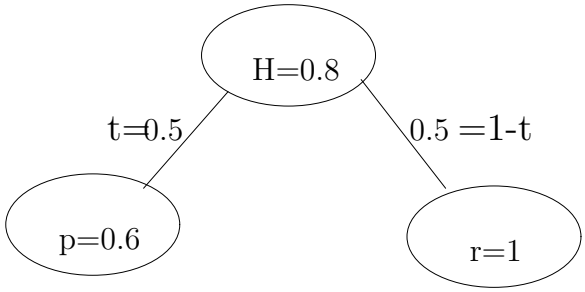


Figure 12.2: Split of Probability

As we shall see in the next example, the error function is not the right function for us to minimize locally since it does not give us progress, even in cases when progress clearly has

been made.

As we can see in the tree below the error at the root is 0.2.



H,p,r - Prob [F=1 The node has been Reached]

Figure 12.3: An example in which the error does not decrease although intuitively we made progress since half of input are classified perfectly.

Now again judging by intuition it seems that the predicate splits very well, since for half of the inputs there is no need to build any subtree.
Still the created subtree has error $E(T) = \sum_{v \in T} Prob[v]E(v) = 0.5 * 0 + 0.5 * 0.4 = 0.2$ which stays as before. So our greedy algorithm won't take this step if it will use the error function as the local minimizing function.
We need to search for a different local function. In general, we want that the cost function $G(q)$ will meet the following criteria:
1. $G(q)$ should be symmetric around $1/2$, that is $G(q) = G(1-q)$.
2. $G(T)$ should be convex.
This also guarantees $G(x) \geq x$ for $1/2 \geq x \geq 0$. Under these conditions $G(T) \geq error(T)$ and it will be sufficient to show that $G(T)$ is small.

12.1.3 The Idea of Top-Down Decision Tree building

We want to build a tree that minimizes the error. In general it is a NPC problem. However it is possible to construct a greedy algorithm that uses $G(T)$ to grow the tree stage by stage. If an algorithm is successful in building a tree with a small $G(T)$ then $error(T)$ is also small.

For the sake of simplicity, instead of talking about sampling to approximate probabilities, we assume that it is possible to compute exact probabilities of arriving to an internal node and a leaf.

In growing the tree the idea is to choose one of the leaves and turn it to the internal node by labeling the chosen leaf with hypothesis h. We suppose that there is the set of hypothesis functions (predicates) H that provides the possible hypotheses (predicates) for all nodes.

For some $h \in H$, the term $T(l,h)$ describes the tree that is created from T by labeling the leaf l with h and attaching two new leaves to the node l. The classification of each of the two new leaves is determined by the "most popular" classification value of the inputs that arrives at it. We choose hypothesis $h \in H$ that maximize the drop in the value of the cost function $G(T)$, that is $\max_{h_i \in H} G(T) - G(T(l,h))$. For the target function f let $q = Prob[f = 1|input\ arrived\ to\ l]$. After l became an internal node, then the q fraction is split into two fractions, p and r, one for each of the new leaves. The identity we have to maintain:

$$q = r\tau + p(1-\tau),$$

where τ is the probability that $h = 1$, given that we reached node l.

Because all the leaves except l remained unchanged, they are irrelevant in the difference $G(T) - G(T(l,H))$ and we get :

$$G(T) - G(T(l,H)) = G(q) - [\tau G(r) + (1-\tau)G(p)].$$

Our goal is to maximize the above expression. To achieve this goal we compute the maximal value of the drop for each leaf, and then choose the leaf with the maximum drop, i.e,
$\max_{h,l} G(T) - G(T(l,h))$.

12.1.4 Top-Down Algorithm

The algorithm is given the number of desired nodes t, the set of predicates H and the splitting function (cost function) $G(T)$.

During t steps, in each step i the algorithm does the following:
1. For T_i choose the pair (l,h) that maximizes $G(T_i) - G(T_i(l,h))$ [1]
2. $T_{i+1} \leftarrow T_i(l,h)$

The existing algorithms (such as C4.5 and CART) use a simple class $H = \{x_i \geq \alpha | \alpha \in \Re\}$. It is clear, that the more complex class H the more possibilities exist, but this is compensated by the computation complexity for maximizing the drop over functions that belong to H, the possibility of overfitting, and in addition the evaluation of tree on inputs may take a longer time.

For splitting there are several possibilities (see Figure 12.4) :
1. Ginn Index CART

$$G(q) = 2q(1-q)$$

[1] T_0 is the tree with only one node

2. Entropy C4.5

$$G(q) = \frac{1}{2}[q \cdot \log_2 \frac{1}{q} + (1-q) \cdot \log_2 \frac{1}{1-q}]$$

3. New function

$$G(q) = \sqrt{q(1-q)}$$

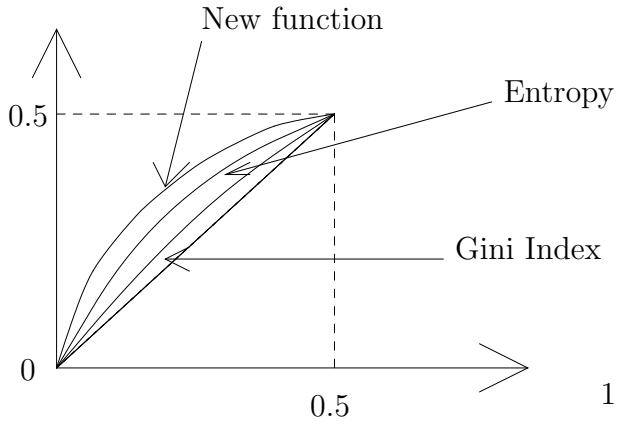


Figure 12.4: Relationships between different cost functions.

Why $G(q) = min(q, 1-q)$ is not a good choice? The reason is that this function behaves lineary. If function behaves lineary then split does not reduce the error of the tree.
If we recall the example we saw in the beginning, let $Prob[f = 1|\textit{input arrived to } l] = 0.8$, $\tau = 0.5$, $Prob[f = 1|h = 0] = 0.6$, $Prob[f = 1|h = 1] = 1$ (see Figure 12.3). Then, for $G(q) = min(q, 1-q)$ we have before splitting

$$G(0.8) = min(0.8, 0.2) = 0.2$$

and after the splitting

$$0.5G(0.6) + 0.5G(1) = 0.5 \cdot min(0.6, 0.4) + 0.5 \cdot min(0, 1) = 0.2.$$

That is, there is no drop in the cost of the tree after the splitting of the leaf. For $G = 2q(1-q)$ we have before splitting

$$G(0.8) = 2 \cdot 0.8 \cdot 0.2 = 0.32$$

and after the splitting

$$0.5G(0.6) + 0.5G(1) = 0.5 \cdot 2 \cdot 0.4 \cdot 0.6 = 0.24.$$

In this case, there is a significant drop in the cost of the tree after the splitting of the leaf. The drop is due to the fact that $G()$ is strictly convex. Similar drops would be observed for the other two optional cost functions (see figure 12.4).

12.1.5 Theoretic Analysis

The Role of the Weak Learning Model

The following is a theoretic analysis of the model of using Weak Learning hypothesis. We assume that for every distribution D over the inputs there exists $h \in H$ such that

$$Prob[c_t \neq h] \leq \frac{1}{2} - \gamma.$$

It will be shown that heuristic decision tree algorithms, under this assumption, decrease the error (though they were not explicitly designed to work in this model). The performance of heuristic algorithms is less impressive than the performance of algorithms specially designed for Weak Learning (such as AdaBoost). Part of the problem is the decision tree representation that limits the algorithms.

Analysis using Weak Learning

At the first stage, in order to use the Weak Learning assumption, we must create a balanced distribution in the leaf. Meaning, half of the inputs that arrive at the leaf will be positive and half will be negative. On such distribution we find hypothesis $h \in H$ that is better than simple random guessing. It may be shown that such hypothesis reduces $G(T)$ considerably and does this locally in the splitted leaf. More specifically, it can be shown that there is a split of the leaf which reduces the cost of the leaf at least by

$$16\gamma^2[q(1-q)]^2.$$

This result may be used to find the number of steps until the algorithm reduces the error of the tree to at most ϵ. Let ϵ_t denote the prediction error of the tree at the stage t.

Lemma 12.1 *At any stage t there is a leaf l such that:*
1. *The probability for input to arrive at l is $Prob[l] \geq \frac{\epsilon_t}{3t}$.*
2. *$error(l) = min\{q_l, 1-q_l\} \geq \frac{\epsilon_t}{3}$.*

Proof: Leaves that for them $Prob[l] \leq \epsilon_t/3t$ (meaning they don't have property 1), contribute no more than $\epsilon_t/3$ to the prediction error of T(there are exactly $t+1$ leaves at step t).

Leaves that the error in them is less than $\epsilon_t/3$ (meaning they don't have property 2) also contribute no more than $\epsilon_t/3$ to the prediction error of T.
Therefore, if there is no leaf with properties **1** and **2** the total error of the tree T would be at most ϵ_t. ■

From this follows:

$$Prob[l]c_1\gamma^2[q_l(1-q_l)]^2 \geq c_2\gamma^2 Prob[l](min\{q_l, 1-q_l\})^2$$
$$\geq c_2\gamma^2\frac{\epsilon_t}{2t}(\frac{\epsilon_t}{2})^2 \geq c_3\gamma^2\frac{G_t^3}{t} ,$$

where G_t is the cost of the tree at the stage t and c_1, c_2, c_3 are constants (We used the fact, that $\epsilon_t \geq G_t/2 = \epsilon_t(1-\epsilon_t)$ for Gini Index). Since the above expression is the drop in the cost of the tree we may conclude that

$$G_{t+1} \leq G_t - c_3\frac{\gamma^2}{t}G_t^3$$

We want the total error of the tree to be less then ϵ. We can solve the above expression for G_t and get that if $t \geq 2^{c/\gamma^2\epsilon^2}$ then $G_t \leq \epsilon$. That is, if we run the algorithm at least $2^{c/\gamma^2\epsilon^2}$ steps then the error of the tree will be less than ϵ.

Using the results for the other cost functions (without proof) we may summarize :
For Ginn Index

$$t \geq 2^{c/\gamma^2\epsilon^2}$$

For Entropy

$$t \geq 2^{\frac{c}{\gamma^2}\frac{1}{\log^2\frac{1}{\epsilon}}}$$

For $G(q) = \sqrt{q(1-q)}$

$$t \geq (\frac{1}{\epsilon})^{c/\gamma}$$

12.2 Decision Trees Pruning

We previously introduced the model of *decision trees* for learning. As in other learning models we faced the problem of over-fitting, meaning: producing a predicator (in our case a decision tree) that would "over-fit" the seen samples and would fail to achieve good results with high probability on the real distribution.
In this part we'll introduce a different methodology:

- Grow the tree until receiving a decision tree that fits the samples exactly.

- start a procedure called **pruning** which will reduce the size of the tree based on the samples, by replacing some internal nodes by leaves (and pruning the sub-tree below them).

The pruning problem is actually of *model selection* - the model size parameter is the tree size and the trade off is with the accuracy of the tree on the given sample set.
We'll examine few pruning algorithms and finally show an efficient, linear time, algorithm for pruning.

12.2.1 Reduced Error Pruning

The first pruning algorithm we'll introduce is called *reduce error pruning*. The *reduce error pruning* algorithm splits the sample given S into two samples S_1 and S_2. The sample S_1 is used for building the decision tree T and S_2 is used for decisions of how to prune it. The pruning of an internal node v is done by replacing the sub-tree rooted at v by a leaf. In the first part of the algorithm we build the decision tree T using the afore-mentioned algorithm. This will produce a tree that reflects the sample S_1 accurately. In the second part we traverse the tree bottom-up and in each internal node v we check whether we want to replace the sub-tree originated by it with a leaf or not. The decision of whether to prune the node v or not is rather simple: we sum the errors of the leaves in the sub-tree rooted by node v and the errors in case we replace it with a leaf. If replacing the sub-tree rooted at v will reduce the total error on S_2 we prune v. In case the number of errors is the same we also prune the node in order to minimize the outputted model size. Notice that on each step in the second part of the algorithm we examine the sub-tree that is relevant to the current stage in the algorithm, meaning that if we pruned part of the sub-tree rooted in v previous iterations then when we check v we examine it against its current sub-tree and not the original one.

To illustrate the algorithm we will use as an example the decision tree T in Figure **??**. As the tree is not pruned yet, the leaf label is the class of all the examples in S_1 that reach the leaf. Note that we can not assume anything like that for the examples in S_2.

We first examine the bottom right dotted subtree whose root is the node labeled X_4. In the test set there are 5 examples reaching the subtree, 2 of them to the left leaf and 3 to the right leaf. Assume that both examples that reach the left leaf have class label 1 and that among the three examples that reach the right leaf, two have class label 1 and one has class label 0 (This is indicated in the figure by the numbers below the subtree.) The subtree error is therefore 2, while the error of X_4 as a pruned leaf is 1 (The label of the pruned leaf X_4 is 1 since in S_1 3 of the examples reaching X_4 have class labeled 1 and only one example has class labeled 0). We can therefore prune T at X_4. A similar analysis is done for the subtrees whose roots are X_6 and X_3 (Note that when considering the X_3 subtree the counters are changed since we already pruned at X_4). The resulting tree is presented in Figure **??**.

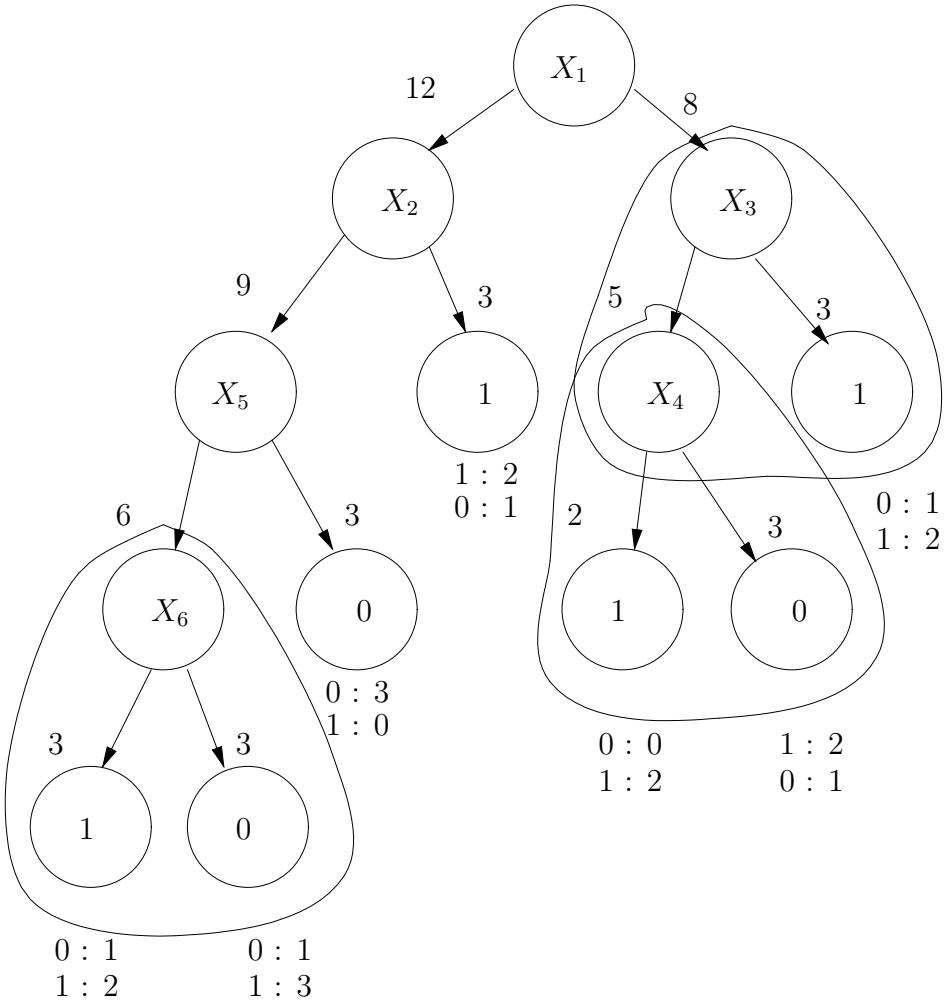


Figure 12.5: An initial decision tree T, about to be pruned. The numbers next to each edge are the number of examples that reach the edge when classified using T (on the test set). The X_i variable in each internal node is the attribute examined while making a decision in the node, and the number in each leaf is the classification made for examples reaching the leaf.

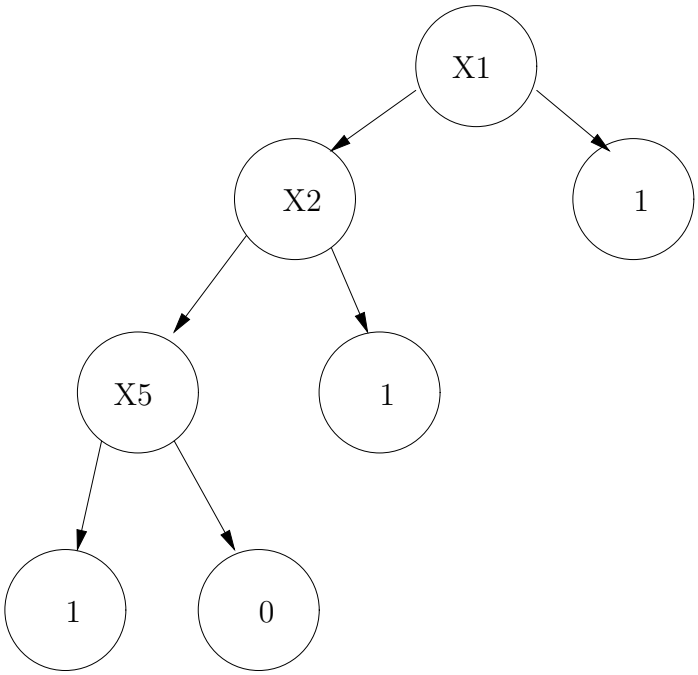


Figure 12.6: The resulting pruned decision tree

12.2.2 Global Structural Risk Minimization

In this section we'll present an algorithm that produces few pruning possibilities. The algorithm will be wrapped by Structural Risk Minimization and Cross Validation that chooses one of the pruning possibilities produced by the underline algorithm. Generally the core algorithm will be given a decision tree T and target number of errors ε and will produce the smallest pruned trees that has such errors rate.

Algorithm

- Build a tree T using S
- for each d, compute the minimal pruning size K_d (tree T_d) with at most d errors
- Select a pruning using **SRM** formula:

$$T^* = arg\min_d \{error(T_d) + \sqrt{\frac{k_d}{m}}\}$$

Where k_d is the size of the T and m the size of S.

Finding the minimum pruning

Given e, the number of errors, we want to find the smallest pruned version of T that has at most e errors. We give a recursive algorithm with exponential complexity. We use $\mathbf{T}[0]$ and $\mathbf{T}[1]$ to represent the left and right child subtrees of $\mathbf{T}$ respectively. root($\mathbf{T}$) is the root node of the tree $\mathbf{T}$. We also define $\mathbf{tree}(r, \mathbf{T_0}, \mathbf{T_1})$ to be the tree formed by making the subtrees $\mathbf{T_0}$ and $\mathbf{T_1}$ the left and right child of the root node r. For every node v of $\mathbf{T}$, $\mathbf{Errors}(v)$ is the number of classification errors on the sample set of v as a leaf.

prune(k:numErrors, $\mathbf{T}$:tree) $\Rightarrow$ (s:treeSize, $\mathbf{P}$:prunedTree)
    If $|\mathbf{T}| = \mathbf{1}$ then
      If $\mathbf{Errors}(\mathbf{T}) \leq k$ then
        $s = 1$
      Else
        $s = \infty$
      $\mathbf{P} = \mathbf{T}$
      return
    If $\mathbf{Errors}(\text{root}(\mathbf{T})) \leq k$ then
      $s = 1$
      $\mathbf{P} = \text{root}(\mathbf{T})$

return
For $i = 1 \dots k$
$(s_i^0, \mathbf{P}_i^0) \leftarrow \mathbf{prune}(i, \mathbf{T}[0])$
$(s_i^1, \mathbf{P}_i^1) \leftarrow \mathbf{prune}(k-i, \mathbf{T}[1],)$
$I = \arg\min_i \{s_i^0 + s_i^1\}$
$s = s_I^0 + s_I^1 + 1$
$\mathbf{P} = \mathbf{tree}(\text{root}(\mathbf{T}), \mathbf{P}_I^0, \mathbf{P}_I^1)$
return

The exponential complexity of the algorithm can be reduced by using dynamic programming method. Note that we can compute all the values for a node if we know all the values for its two child subtrees. The basic idea is therefore to do the computation from the leaves up towards the root of the tree. If we have all the computations done for the subtrees $\mathbf{T_0}$ and $\mathbf{T_1}$ (s_i and the suitable tree $\mathbf{P_i}$ were computed for every $i = 1 \dots k$) then we can compute for $\mathbf{T}$ by scanning the already computed values k times ($i = 1 \dots k$):

$I = arg\min_i \{s_i^0 + s_{k-i}^1\}$ ## time complexity O(k)
$s_i = s_I^0 + s_I^1 + 1$ ## time complexity O(1)
$\mathbf{P}_i = \mathbf{tree}(\text{root}(\mathbf{T}), \mathbf{P}_I^0, \mathbf{P}_I^1)$ ## time complexity O(1)

We also have to initialize all the values in the leaves of T. To do that we must find for every leaf the exact number of errors over the sample set (if we have l errors in a leaf then we set $s_l = 1$ and all other s_i to infinity). The total time complexity is therefore $O(k^2 * |T|) + O(m * depth(T))$ where m is the size of the sample set. Since $k = O(m)$ we have total time complexity $O(m^2 * |T|)$.

12.2.3 Bottom-Up SRM Pruning Algorithm

We want to use a pruning algorithm that will run in linear time. The algorithm that we shall describe will use a single bottom up pass on the tree. It will make its pruning decisions based on local information (of the subtree in question). Finally We shall prove a strong performance guarantee for the generalization error of the pruned tree achieved with this algorithm.

Description

The algorithm is given the random sample S and a tree T as input. We do not assume the independence of T and S, actually we may assume S was used to build T, therefore $T = T(S)$. The high-level structure of the algorithm is quite straightforward: the algorithm

makes a single "bottom-up" pass through T, and decides for every node v whether to leave the subtree currently rooted at v in place (at least for the moment), or whether to prune this subtree. The algorithm only considers pruning at a node v once it has first considered pruning at all nodes below v; this simply formalizes the standard notion of "bottom-up" processing. When moving up the tree the relevant information is passed from node v to its parent upon pruning. Thus when we reach a node we have all the relevant information for the decision making.

Two observations are in order here. First, the algorithm considers a pruning operation only once at each node v of T. Second, the subtree rooted at v at that time when considering pruning may be *different* than T_v (the original subtree of T rooted at v), because parts of T_v may have already been deleted. We shall use the notation T_v^* to denote the subtree that is rooted at v when considering pruning. It is T_v^* that our algorithm decides whether to prune.

It remains only to describe how our algorithm decides whether or not to prune T_v^*. For this we need some additional notations:

- T^* the final pruning of T output by our algorithm
- S_v will denote the set of all $x \in S$ that reach v
- $m_v = |S_v|$
- n_v denotes the number of nodes in T_v^*
- ℓ_v denotes the depth of the node v in T
- $\hat{\epsilon}_v(T_v^*)$ as the fraction of errors that T_v^* makes on the local sample S_v
- $\hat{\epsilon}_v(\emptyset)$ is the fraction of errors the best leaf (a constant 0 or 1 function replacing v) makes on S_v
- $\delta \in [0, 1]$ is a confidence parameter

Our algorithm will replace T_v^* by this best leaf if and only if

$$\hat{\epsilon}_v(T_v^*) + \alpha(m_v, n_v, \ell_v, \delta) \geq \hat{\epsilon}_v(\emptyset) \tag{12.1}$$

The exact choice of $\alpha(m_v, n_v, \ell_v, \delta)$ will depend on the setting, but in all cases can be thought of as a penalty for the complexity of the subtree T_v^*.

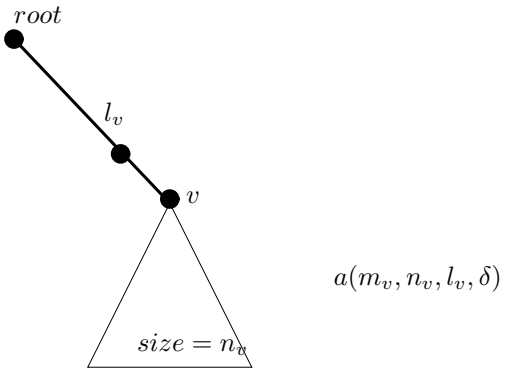


Figure 12.7: The basic setting for the algorithm

We use $\textsc{trees}(H, n)$ to denote the class of all binary trees with tests predicates from H and size at most n. And $\textsc{paths}(H, \ell)$ to denote the class of all conjunctions of at most ℓ predicates from H.

Let us first consider the case in which the class H of testing functions in the tree nodes is finite, in which case the class of possible path predicates $\textsc{paths}(H, \ell_v)$ leading to v and the class of possible subtrees $\textsc{trees}(H, n_v)$ rooted at v are also finite. In this case, we would choose

$$\alpha(m_v, n_v, \ell_v, \delta) = c\sqrt{\frac{\log(|\textsc{paths}(H, \ell_v)|) + \log(|\textsc{trees}(H, n_v)| + \log(m/\delta)}{m_v}} \tag{12.2}$$

for some constant specific $c > 1$.
Perhaps the most natural and common special case of this finite-cardinality setting is that in which the input space X is the Boolean hypercube $\{0, 1\}^d$, and the test class H contains just the d single-variable tests x_i ($H = \{x_i : 0 \leq i \leq d\}$). Then

$$|\textsc{paths}(H, \ell)| = 2^{l_v - 1} \binom{n}{l_v} \leq n_v^\ell$$

$$|\textsc{trees}(H, n)| \leq (c, d)^{n_v}$$

The $VCdim$ of PATHS and TREES can be used instead of the class size.

Let us first provide some brief intuition behind the algorithm: At each node v, the algorithm considers whether to leave the current subtree T_v^* or to delete it. The basis for

this comparison must clearly make use of the sample S provided. Furthermore, we could simply prefer whichever of T_v^* and the best leaf makes the smaller number of mistakes on S_v. This is clearly a poor idea, since T_v^* cannot do worse than the best leaf and may do considerably better but generalize poorly compared to the best leaf due to over-fitting. Thus, a penalty on T_v^* for its complexity seems reasonable, which is exactly the role of the additive term $\alpha(m_v, n_v, \ell_v, \delta)$ above.

The analysis will proceed as follows. We shall first argue that any time our algorithm chooses *not* to prune T_v^*, then (with high probability) this was in fact the "right" decision, in the sense that the current tree T^* would be degraded by deleting T_v^*. This allows us to establish that our final pruning will be a subtree of the optimal pruning, so our only source of additional error results from those subtrees of this optimal pruning that we deleted. Further analysis allows us to bound this additional error relatively to the error of the optimal pruning by a quantity related to the size of the optimal pruning.

12.2.4 Local Uniform Convergence

To complete the analysis we need a tool for estimating the relation between the local observed error $\hat{\epsilon}_v$ of a subtree T_v and its local true error ϵ_v. We can use the following results from Local uniform convergence:

Lemma 12.2 *let C_t be a target boolean function over x, and let D be a probability distribution over X. For any $path \in PATHS$ and $tree \in TREES$, let $\epsilon(path, tree) = \mathbf{Pr}_D[tree(x) \neq C_t(x) | path = 1]$, and for any labeled sample S of $C_t(x)$, let $\hat{\epsilon}(path, tree)$ denote the fraction of points in S_c on which tree errs, where $S_c = \{x \in S : path(x) = 1\}$. Then the probability that there exists a $path \in PATHS$ and an $tree \in TREES$ such that*

$$|\epsilon(path, tree) - \hat{\epsilon}(path, tree)| \geq \sqrt{\frac{\log(|PATHS|) + \log(|TREES|) + \log(1/\delta)}{m_{path}}} \qquad (12.3)$$

is at most δ, where $m_{path} = |S_c|$.

Proof: Let us fix a $path \in PATHS$ and $tree \in TREES$. For these fixed choices, we have for any value λ,

$$\mathbf{Pr}_P[|\epsilon(tree, path) - \hat{\epsilon}(path, tree)| \geq \lambda] = \mathbf{E}_{m_{path}}[\mathbf{Pr}_{S_{path}}[|\epsilon(path, tree) - \hat{\epsilon}(path, tree)| \geq \lambda] \qquad (12.4)$$

Here the expectation is over the distribution on values of m_{path} induced by D, and the distribution on S_c is over samples of size m_{path} (which is fixed inside the expectation) drawn according to D_c (the distribution D conditioned on $path$ being 1). Since m_{path} is fixed, by standard Chernoff bounds we have

$$\mathbf{Pr}_{S_c}[|\epsilon(path, tree) - \hat{\epsilon}c(tree, path)| \geq \lambda] \leq e^{-\lambda^2 m_{path}} \qquad (12.5)$$

giving the bound

$$\mathbf{Pr}_P[|\epsilon_c(h) - \hat{\epsilon}_c(h)| \geq \lambda] \leq \mathbf{E}_{m_c}[e^{-\lambda^2 m_{path}}]. \tag{12.6}$$

If we choose

$$\lambda = \sqrt{\frac{\log(|PATHS|) + \log(|TREES|) + \log(1/\delta)}{m_{path}}} \tag{12.7}$$

■

12.2.5 Global Analysis of the Pruning Algorithm

Lemma 12.3 *With probability at least $1 - \delta$ over the draw of the input sample S, T^* is a subtree of T_{opt}.*

Proof: For the analysis, it will be convenient to introduce the notation

$$r_{v_i} = \log(|PATHS|) + \log(|TREES|) + \log(m/\delta). \tag{12.8}$$

Consider any node v that is a leaf in T_{opt}. It suffices to argue that our algorithm would choose to prune T_v^*, the subtree that remains at v when our algorithm reaches v. By Equation (**??**), our algorithm would *fail* to prune T_v^* only if $\hat{\epsilon}_v(\emptyset)$ exceeded $\hat{\epsilon}_v(T_v^*)$ by at least the amount $\alpha(m_v, s_v, \ell_v, \delta)$, but then using Lemma **??** and its application to the tree we can state that $\epsilon_v(T_v^*) < \epsilon_v(\emptyset)$ with high probability. In other words, if our algorithm fails to prune T_v^*, then T_{opt} would have smaller generalization error by including T_v^* rather than making v a leaf. This contradicts the optimality of T_{opt}. ■

Lemma **??** means that the only source of additional error of T^* compared to T_{opt} is through overpruning, not underpruning. Thus, for the purposes of our analysis, we can imagine that our algorithm is actually run on T_{opt} rather than the original input tree T (that is, the algorithm is initialized starting at the leaves of T_{opt}, since we know that the algorithm will prune everything below this frontier).

Let $V = \{v_1, \ldots, v_t\}$ be the sequence of nodes in T_{opt} at which the algorithm chooses to prune the subtree $T_{v_i}^*$; note that $t \leq s_{opt}$. We can now write:

$$\epsilon(T^*) - \epsilon_{opt} = \sum_{i=1}^{t} (\epsilon_{v_i}(\emptyset) - \epsilon_{v_i}(T_{v_i}^*)) p_{v_i} \tag{12.9}$$

where p_{v_i} is the probability under the input distribution P of reaching node v_i, that is, the probability of satisfying the path predicate $reach_{v_i}$. We can bound this difference:

$$\epsilon(T^*) - \epsilon_{opt} \leq \sum_{i=1}^{t} \left(|\epsilon_{v_i}(\emptyset) - \hat{\epsilon}_{v_i}(\emptyset)| + |\hat{\epsilon}_{v_i}(\emptyset) - \hat{\epsilon}_{v_i}(T_{v_i}^*)| + |\hat{\epsilon}_{v_i}(T_{v_i}^*) - \epsilon_{v_i}(T_{v_i}^*)| \right) p_{v_i}$$

$$\begin{aligned} &\leq \sum_{i=1}^{t} \left(\sqrt{\frac{(\ell_{v_i}+1)\log(n)+\log(m/\delta)}{m_{v_i}}} + \alpha(m_{v_i}, s_{v_i}, \ell_{v_i}, \delta) \right. \\ &\quad \left. + \sqrt{\frac{(\ell_{v_i}+s_{v_i})\log(n)+\log(m/\delta)}{m_{v_i}}} \right) p_{v_i} \\ &\leq 3\sum_{i=1}^{t} \left(\sqrt{\frac{r_{v_i}}{m_{v_i}}} \right) p_{v_i}. \end{aligned}$$

The first inequality comes from the triangle inequality. The second inequality uses two invocations of Lemma **??** in the special case form (equation **??**) on the first and third arguments, and applies on the second argument the fact that our algorithm directly compares $\hat{\epsilon}_{v_i}(\emptyset)$ and $\hat{\epsilon}_{v_i}(T^*_{v_i})$, and prunes only when they differ by less than $\alpha(m_{v_i}, s_{v_i}, \ell_{v_i}, \delta)$.

Thus, we would like to bound the sum $\Delta = \sum_{i=1}^{t}(\sqrt{r_{v_i}/m_{v_i}})p_{v_i}$. We shall use the following lemma to break up the sum.

Lemma 12.4 *The probability, over the sample S, that there exists a node $v_i \in V$ such that $p_{v_i} > 12\log(t/\delta)/m$ but $p_{v_i} \geq 2\hat{p}_{v_i}$, is at most δ.*

Proof: We will use the relative Chernoff bound

$$\mathbf{Pr}[\hat{p}_{v_i} < (1-\gamma)p_{v_i}] \leq e^{-mp\gamma^2/3} \tag{12.10}$$

which holds for any fixed v_i. By taking $\gamma = 1/2$ and applying the union bound, we obtain

$$\mathbf{Pr}[\exists v_i \in V : p_v \geq 2\hat{p}_v] \leq te^{-p_{v_i}m/12}. \tag{12.11}$$

Now we can use the assumed lower bound on p_{v_i} to bound the probability of the event by δ. ■

Let V' be the subset of V for which the lower bound $p_{v_i} > 12\log(t/\delta)/m$ holds. We divide the sum that describes Δ into two parts:

$$\Delta = \sum_{v_i \in V - V'} \left(\sqrt{r_{v_i}/m_{v_i}} \right) p_{v_i} + \sum_{v_i \in V'} \left(\sqrt{r_{v_i}/m_{v_i}} \right) p_{v_i} \tag{12.12}$$

The first sum is bounded by $12t\log(t/\delta)/m \leq 12s_{opt}\log(s_{opt}/\delta)/m$, since $\sqrt{r_{v_i}/m_{v_i}}$ is at most 1, and $t \leq s_{opt}$.

For the second sum, we perform a maximization. By Lemma **??**, with high probability we have that for every $v_i \in V'$, $p_{v_i} < 2\hat{p}_{v_i} = 2m_{v_i}/m$. Thus, with high probability we have

$$\sum_{v_i \in V'} \sqrt{\frac{r_{v_i}}{m_{v_i}}} p_{v_i} \quad < \quad \sum_{v_i \in V'} \sqrt{\frac{r_{v_i}}{m_{v_i}}} \left(2\frac{m_{v_i}}{m} \right) \tag{12.13}$$

$$= \frac{2}{m} \sum_{v_i \in V'} \sqrt{r_{v_i}} \sqrt{m_{v_i}} \tag{12.14}$$

$$\leq \frac{2}{m} \sqrt{(\sum_{v_i \in V'} r_{v_i})(\sum_{v_i \in V'} m_{v_i})} \tag{12.15}$$

To bound this last expression, we first bound $\sum_{v_i \in V'} r_{v_i}$. Recall that

$$r_{v_i} = (\ell_{v_i} + s_{v_i}) \log(n) + \log(m/\delta). \tag{12.16}$$

Since for any $v_i \in V'$, we have $\ell_{v_i} \leq \ell_{opt}$, we have that $\sum_{v_i \in V'} \ell_{v_i} \leq t\ell_{opt} \leq s_{opt}\ell_{opt}$, since $|V'| \leq t \leq s_{opt}$. Since the subtrees $T^*_{v_i}$ that we prune are disjoint and subsets of the optimal subtree T_{opt}, we have $\sum_{v_i \in V'} s_{v_i} \leq |T_{opt} - T^*| \leq s_{opt}$. Thus

$$\sum_{v_i \in V'} r_i \leq s_{opt}((1 + \ell_{opt}) \log(n) + \log(m/\delta)). \tag{12.17}$$

To bound $\sum_{v_i \in V'} m_{v_i}$ in Equation (**??**), we observe that since the sets of examples that reach different nodes at the same depth in the tree are disjoint (so their total in a given depth is at most m and there are at most ℓ_{opt} levels) , we have $\sum_{v_i \in V'} m_{v_i} \leq m\ell_{opt}$.

Putting it all together

Thus, with probability $1 - \delta$, we obtain an overall bound

$$\Delta < 12 \log(s_{opt}/\delta) \frac{s_{opt}}{m} + \frac{2}{m} \sqrt{s_{opt}((1 + \ell_{opt}) \log(n) + \log(m/\delta))(m\ell_{opt})} \tag{12.18}$$

$$= O\left(\left(\log(s_{opt}/\delta) + \ell_{opt}\sqrt{\log(n) + \log(m/\delta)}\right) \times \sqrt{\frac{s_{opt}}{m}}\right) \tag{12.19}$$

This gives the following important result.

Theorem 12.5 *Let S be a random sample of size m drawn according to an unknown target function and input distribution. Let $T = T(S)$ be any decision tree, and let T^* denote the subtree of T output by our pruning algorithm on inputs S and T. Let ϵ_{opt} denote the smallest generalization error among all subtrees of T, and let s_{opt} and ℓ_{opt} denote the size and depth of the subtree achieving ϵ_{opt}. Then with probability $1 - \delta$ over S,*

$$\epsilon(T^*) - \epsilon_{opt} = O\left(\left(\log(s_{opt}/\delta) + \ell_{opt}\sqrt{\log(n) + \log(m/\delta)}\right) \times \sqrt{s_{opt}/m}\right) \tag{12.20}$$

Roughly speaking, Theorem **??** ensures that the true error of the pruning found by our algorithm will be larger than that of the best possible pruning by an amount that is not much worse than $\sqrt{s_{opt}/m}$ (ignoring logarithmic and depth factors for simplicity). How

good is this? Since we assume that T itself (and therefore, all subtrees of T) may have been constructed from the sample S, standard model selection analysis indicate that ϵ_{opt} may be larger than the error of the best decision tree approximation to the target function by an amount growing like $\sqrt{s_{opt}/m}$. (Recall that ϵ_{opt} is only the error of the optimal *subtree of* T — there may be other trees which are not subtrees of T with error less than ϵ_{opt}, especially if T was constructed by a greedy top-down heuristic.) Thus, if we only compare our error to that of T_{opt}, we are effectively only paying an additional penalty of the same order that T_{opt} pays. If s_{opt} is small compared to m — that is, the optimal subtree of T is small — then this is quite good indeed.

12.3 Bibliographic Notes

In addition to the lecture notes, the following article was used in the preparation of this scribe.

1. *A Fast, Bottom-Up Decision Tree Pruning Algorithm with Near-Optimal Generalization by Michael Kearns and Yishay Mansour.*