Introduction to Machine Learning **Fall Semester, 2016/7**

Lecture 8: Dec. 18, 2016

Lecturer: Yishay Mansour *Scribe: Yishay Mansour*

8.1 Decision Tree Learning Algorithms

8.1.1 What is a decision tree?

A decision tree is a tree whose nodes are labeled with predicates and whose leaves are labeled with the function values. In Figure 8.1 we have Boolean attributes. In this case we can have each predicate to be simply a single attribute. The leaves are labeled with $+1$ and -1. The edges are labeled with the values of the attributes that will generate that transition.

Given an input x, we first evaluate the root of the tree. Given the value of the predicate in the root, x_1, we continue either to the left sub-tree (if $x_1 = 0$) or the right sub-tree (if $x_1 = 1$). The output of the tree is the value of the leaf we reach in the computation. Any computation defines a path from the root to a leaf.

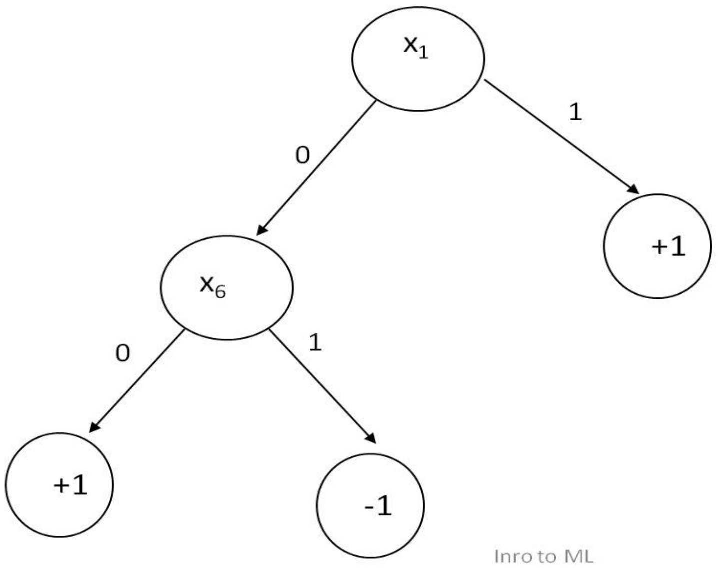

Figure 8.1: Boolean attributes decision tree

When we have continuous value attributes, we need to define some predicate class that will induce binary splits. A common class is *decision stumps* which compare a single attribute to a fixed value, e.g., $x_1 > 5$. The evaluation, again, starts by evaluating the root, and follows a path until we reach a leaf. (See Figure 8.2)

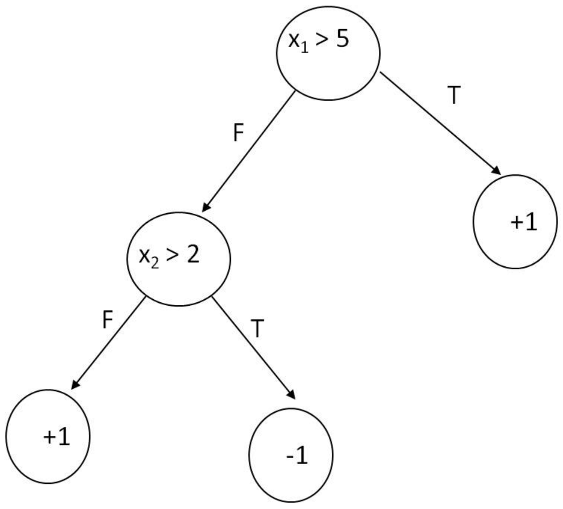

Figure 8.2: A decision tree with decision stumps

When we consider decision stumps, the boundary for decisions are axis parallel, as shown in Figure 8.3.

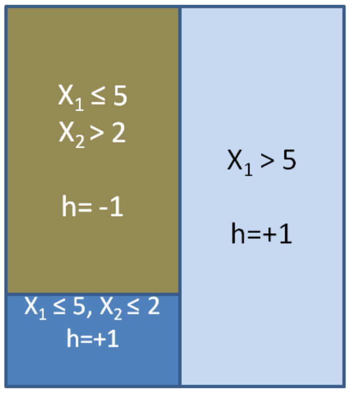

Figure 8.3: Decision boundary for decision stumps.

8.1.2 Decision Trees - Basic setup

The decision tree would have a class of predicates H. This class of predicates can be decision stumps, single attributes (in case of Boolean attributes) or any other predicate class. The predicate class H can be highly complicated (e.g., hyperplanes) but in most algorithms simple classes are used, since we like to build the decision trees from simple classifiers. Technically, we can have in each node even a large decision tree.

The input to the decision tree algorithm is the sample $S = \{(x, b)\}$, which as always includes examples of an input and its classification.

The output of the decision tree algorithm is a decision tree where each internal node has a predicate from H and each leaf node has a classification value.

The goal is to output a small decision tree which classifies all (or most) of the examples correctly. This is inline with Occam's Razor, which states that low complexity hypotheses have a better generalization guarantee.

8.1.3 Decision Trees - Why?

One of the main reasons for using decision trees is human interpretability. Humans find it much easier to understand (small) decision trees. The evaluation process is very simple and one can understand fairly simply what the decision tree will predict.

There is a variety of efficient decision tree algorithms, and there are many commercial software packages that learn decision trees, such as C4.5 or CART. Even Excel and MATLAB have a standard implementation of a decision tree algorithm.

The performance of decision trees is reasonable, probably slightly weaker than SVM and AdaBoost, but comparable on many data sets. Decision forests, which use many small decision trees have a general performance comparable to the best classification algorithms (including SVM and AdaBoost).

8.1.4 Decision Trees - VC dimension

We will compute the VC dimension as a function of the number of leaves s (which implies tree size $2s-1$) and the number of attributes n. We will denote it by $VCdim(s, n)$.

We start by computing the VC dimension in the case that the attributes are Boolean, i.e., the domain is $\{0, 1\}^n$, and the splits are using single attributes. First we show that $VCdim(s, n) \geq s$. For any set V of s examples, we can build a decision tree with s leaves such that each leaf has a unique example reaching it. (Why is it possible?) Let T_S be the tree, now for each assignment of labels to the set V we can set labels to the leaves and get the desire assignment. Therefore, $VCdim(s, n) \geq s$.

For the upper bound, we consider the number of trees, the possible labeling of internal nodes and leaves. The number of trees is the Catalan number which gives $\frac{1}{s+1}\binom{2s}{s} \leq 4^s$ as an upper bound on the number of trees. The number of assignments of attributes to internal nodes is n^{s-1} and the number of assignments of labels to the leaves is 2^s. This bounds the number of trees by $4^s n^{s-1} 2^s \leq (8n)^s$. Since the VC dimension is at most the logarithm of the size of the concept class, we have that $VCdim(s, n) \leq s \log(8n)$.

For the general case, where the splitting criteria are from a hypothesis class of VC dimension d, we will show that $VCdim(s,n) = O(sd\log(sd))$. The methodology is to bound the number of functions on m examples, denoted by $NF(m)$. For a shattered set of size m we have $2^m \leq NF(m)$, and this will allow us to upper bound the VC dimension.

We will first fix the tree structure. (There are at most 4^s such trees.) There are 2^s labeling to the leaves. For each tree and leaves labeling, we will build and $m \times s-1$ matrix, where the rows are labeled by inputs, and the columns are labeled by splitting functions. Note that given all the values of internal nodes we fix the path of an input. This implies that we need to count the number of different matrices. Using Sauer-Shelah lemma, for each column we have at most m^d different values. This bounds the number of matrices by m^{ds}. Assume we have $m = VC(s,n)$ points which are shattered, this implies that $2^m \leq 4^s 2^s m^{ds}$, and hence, $VCdim(s,n) = O(ds\log(sd))$.

8.1.5 Decision trees - Construction

There is a very natural greedy algorithm for constructing a decision tree given a sample. We first decide on a predicate $h_r \in H$ to assign to the root. Once we selected h_r, we split the sample S to two parts. In S_0 (S_1) we have all the examples where $h_r(x) = 0$ ($h_r(x) = 1$). Given S_0 and S_1 we can continue recursively to build a subtree for S_0 and a subtree for S_1.

The time complexity for such a procedure would be

$$Time(m) = O(|H|m) + Time(m_0) + Time(m_1)$$

where $m = |S|$, $m_0 = |S_0|$ and $m_1 = |S_1|$. The worse case time is $O(|H|m^2)$, and in most cases the running time is $O(|H|m\log m)$ since most of the splits will be approximately balanced.

The main issue that we need to resolve is how to select the predicate h_r given a sample S. Clearly, if all the samples have the same label we can select a leaf and mark it with that label. Note that we are building a decision tree that would classify all the samples correctly.

8.1.6 Selecting predicates - splitting criteria

Given the outline of the greedy algorithm, the main remaining task is to select a predicate, assuming that not all the examples have the same label. We would like a simple local criteria that would be based only on the parameters observed at the node.

We would like to use a potential function $val(\cdot)$ to guide our selection. First let us define val for a leaf v with a fraction of 1 equal to q_v as $val(q_v)$. Next for an inner node v which has two leaf sons, we define $val(v) = u \cdot val(p) + (1-u)val(r)$, where u, p and r are defined as follows. Let u be the fraction of examples such that $h(x) = 1$, let p be the fraction of examples for which $b = 1$ (the target function is labeled 1) out of the examples with $h(x) = 1$, and let r be the fraction of examples for which $b = 1$ out of the examples with $h(x) = 0$.

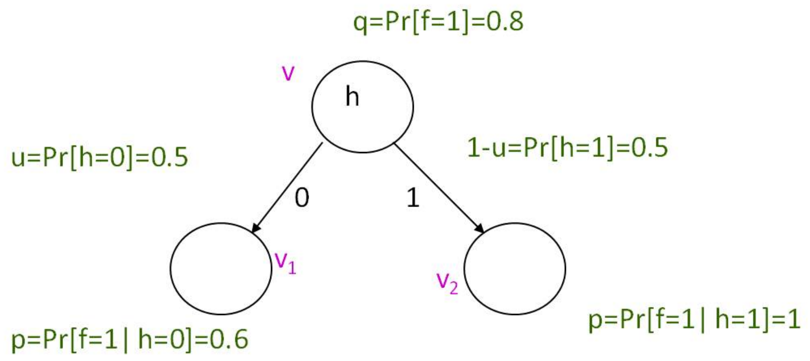

Figure 8.4: An example of a split for the potential equal to the observed error

Finally, for a decision tree T we define $val(T) = \sum_{v \in Leaves} p_v val(q_v)$, where p_v is the fraction of samples that reach v.

Given the function val we need to define how we select the predicate $h \in H$ to assign to node v. The idea is to consider the split induced by h, and to compute the value of the potential val if we stop immediately after that split. Namely, we have $val(v,h) = u \cdot val(p) + (1-u)val(r)$, where u, p, and r are defined as above. We select the predicate h that minimizes $val(v,h)$. (Note that the potential will never increase, since we can simply select not to split.)

What remains now is to define $val(q)$. Since we would like to minimize the overall error, it seems reasonable to select $val(q) = \min(q, 1-q)$. The reasoning is the following. When we fix v to be a leaf, then the label that would minimize the fraction of errors at v, namely the one that is more likely, and the fraction of errors would be $\min(q, 1-q)$. It is important to understand why the choice of a potential function which is the observed error is problematic.

Consider the example in Figure 8.4. In this example we have that before the split we have 0.20 fraction of errors and after the split we have $0.5 * 0.4 + 0.5 * 0 = 0.20$ fraction of errors, so the potential did not decrease. This implies that the potential of the observed error would prefer any other predicate over this split. On the other hand, if we look closely at the split, it looks like an excellent split. We have *half* of the examples perfectly labeled, and we are left with building a decision tree for only half of the sample.

The reason why the observed error potential failed is that it was not able to quantify the progress we make when the observed error does not decrease. In order to overcome this difficulty we would like the following to hold:

1. Every non-trivial split reduces the potential. We will be able to achieve this by using a strictly concave function.

2. The potential is symmetric around 0.5, namely, $val(q) = val(1-q)$.

3. Potential zero implies perfect classification. This implies that $val(0) = val(1) = 0$.
4. We have $val(0.5) = 0.5$.

The important an interesting part is the strict concavity. The strict concavity will guarantee us that any split will have a decrease in the potential. The reason is that

$$val(q) > u \cdot val(p) + (1-u)val(r)$$

when $q = up + (1-u)r$, due to the strict concavity.

The other conditions are mainly to maintain normalization. Using them we can ensure that $val(T) \geq error(T)$, since at any leaf v we will have $val(v) > error(v)$.

8.1.7 Splitting criteria

We have reduced the discussion to selecting a splitting criteria which is a strictly concave function. Several potential functions are suggested in the literature (see Figure 8.5) :

1. Gini Index used in CART:
$$G(q) = 2q(1-q)$$
2. Entropy used in C4.5
$$G(q) = \frac{1}{2}\left[q \cdot \log_2 \frac{1}{q} + (1-q) \cdot \log_2 \frac{1}{1-q}\right]$$
3. Variance function
$$G(q) = \sqrt{q(1-q)}$$

We can now go back to the example of Figure 8.4, and consider the Gini index, i.e., $G = 2q(1-q)$. Before the split we have

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

and after the split we have

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

In this case, there is a drop in the potential, which is even significant. The drop is due to the fact that $G(\cdot)$ is strictly concave and not linear. Similar drops would be observed for the other two optional cost functions (see Figure 8.5).

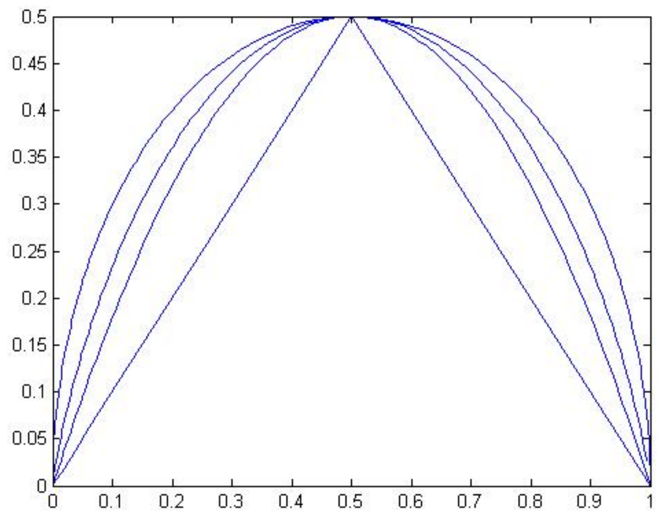

Figure 8.5: Relationships between different splitting criteria. The criteria, from inner to outer are observed error, Gini index, Entropy and Variance

8.1.8 Decision tree construction - putting it all together

We define a procedure $DT(S)$, where S is the sample. The procedure returns a tree that classifies S correctly.

```
Procedure DT(S) return a decision tree T
                If $\forall(x,b) \in S$ we have $b = 1$ Then
                   Create a Leaf with label 1 and Return.
                If $\forall(x,b) \in S$ we have $b = 0$ Then
                   Create a Leaf with label 0 and Return.
                For each $h \in H$ compute
                     $S_h = \{(x,b) \in S | h(x) = 1\}$.
                     $u_h = |S_h|/|S|$
                     $S_{h,1} = \{(x,b) \in S_h | b = 1\}$
                     $p_h = |S_{h,1}|/|S_h|$.
                     $S_{h,0} = \{(x,b) \in S - S_h | b = 1\}$
                     $r_h = |S_{h,0}|/|S - S_h|$.
                     $val(h) = u_h val(p_h) + (1 - u_h) val(r_h)$,
                Let $h' = \arg\min_h val(h)$.
                Let $S_{h'} = \{x \in S : h'(x) = 1\}$.
```

```
        Call $DT(S_{h'})$ and receive $T_1$.
        Call $DT(S - S_{h'})$ and receive $T_0$.
        Return a decision tree with root labeled by $h\prime$, right subtree $T_1$ and left subtree $T_0$.
```

In class, in the slides we have an example of running the algorithm with the Gini index.

8.1.9 Decision tree algorithms - Guaranteed performance

We do not have any guarantee about the decision tree size produced by the greedy algorithm. In fact, if we consider the target function $x_1 \oplus x_2$ and a uniform distribution over d binary attributes then the greedy algorithm would create a very large decision tree. The reason is that when it considers any single attribute, the probability that the target is 1 or 0 is identical (until we select either x_1 or x_2). This implies that the decision tree will select attributes randomly, until we select either x_1 or x_2.

In fact, we can show that finding the smallest decision tree is NP-hard, which means that it is unlikely we will have a computationally efficient algorithm for computing the smallest decision tree given a sample.

We can perform an analysis that is based on the weak learner hypothesis. The weak learner hypothesis assumes that for any distribution there is a predicate $h \in H$ which is a weak learner, i.e., has error at most $1/2 - \gamma$. (We will study the weak learner hypothesis extensively next lecture.) Assuming the weak learner hypothesis we can bound the decision tree size as a function of the parameter γ. Specifically,

1. For the Gini index,we have that the decision tree size is at most $e^{O(1/\gamma^2 1/\epsilon^2 log^2 1/\epsilon)}$.
2. For the Entropy index,we have that the decision tree size is at most $e^{O(1/\gamma^2 log^2 1/\epsilon)}$.
3. For the Variance index,we have that the decision tree size is at most $e^{O(1/\gamma^2 log 1/\epsilon)}$.

8.2 Decision Tree: Pruning

8.2.1 Why Pruning?

We saw the algorithm that builds a decision tree based on a sample. The decision tree is built until zero training error. As we discussed many times before, our goal is to minimize the testing error and not the training error.

In order to minimize the testing error, we have two basic options. The first option is to decide to do *early stopping*. Namely, stop building the decision tree at some point. Different alternatives to perform the early stopping are: (1) when the number of examples in a node is small, (2) The reduction in the splitting criteria is small, (3) a bound on the number of nodes, etc.

The alternative approach is to first build a large decision tree, until there is no training error, and then prune the decision. The net effect is the same. In both cases we end with a small decision tree. The major difference is what we observe on the way. When we first build a large tree, we may discover some important sub-structure that in case we stop early, we might miss. In some sense, we can always simulate the early stopping when we first build the large tree and then prune, but definitely cannot do the reverse.

8.2.2 Decision Tree Pruning: Problem Statement

The input to the decision tree pruning is a decision tree T. The output of the pruning is a decision tree T' which is derived from T. We will mainly look at the case where we can replace an inner node of T by a leaf. Another, more advanced option, is to replace an inner node by the sub-tree rooted at one of its children. At the extreme, if we have no restriction then we are essentially left with the same problem we started with, finding a good decision tree.

8.2.3 Reduced Error Pruning

In reduced error pruning we split the sample S to two parts S_1 and S_2. We use S_1 to build a decision tree and use S_2 for decisions of how to prune it. We do not assume anything about the building of the decision tree T using S_1. (We do not even assume that the decision tree T has zero error on S_1, although in our main application this will be the case.) Our pruning will use S_2 to modify the structure of T. The pruning of an internal node v is done by replacing the sub-tree rooted at v by a leaf.

In the pruning we traverse the tree bottom-up and for 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 size of the decision tree we output. Otherwise, we keep the node (and the sub-tree rooted at it).

Notice that on each step in the pruning 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 on previous iterations then when we check v we examine it against its current sub-tree and not the original one. The pruning of an internal node v is done by replacing the sub-tree rooted at v by a leaf at v.

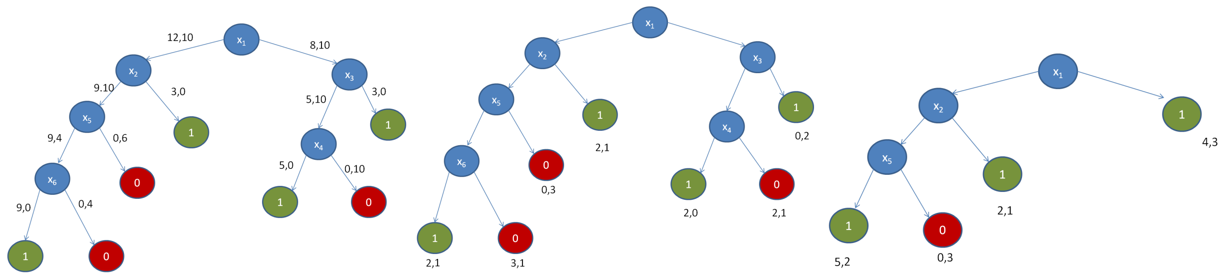

Figure 8.6: Example of data for decision tree Pruning. The left figure is the building of the decision tree using S_1. The middle is the set S_2 used for T. The right is the pruned decision tree. The numbers are the number of examples labeled by 1 followed by the number of examples labeled by 0.

8.2.4 Bottom-up pruning - SRM

We can perform a bottom up procedure, similar to REP, that rather than using a held out set of examples, will use a statistical criteria in order to prune. The main advantage is the saving in the held out set, that can now be used to enhance the training.

The basic idea is that we will use a conservative criteria, which will keep the subtree (i.e., not prune) only if we are rather certain that the additional subtree enhances the performance. We would like the criteria to be local to the node we examine and the performance of its subtree.

The notation we will use is v for the node, T_v for the subtree rooted at v (note that during the pruning this subtree is modified, T_v is the subtree when we are considering node v), ℓ_v the length of the path leading to v, and m_v the number of examples reaching node v. Let $\epsilon_v(T_v)$ and $\widehat{\epsilon}_v(T_v)$ be the true and observed error from v using the subtree T_v, and $\epsilon_v(\emptyset)$ and $\widehat{\epsilon}_v(\emptyset)$ be the true and observed error from v as a leaf. Using all the above we will define a penalty $\alpha(m_v, |T_v|, \ell_v, \delta)$. We will prune at node v if

$$\widehat{\epsilon}_v(T_v) + \alpha(m_v, |T_v|, \ell_v, \delta) > \widehat{\epsilon}_v(\emptyset)$$

We call a penalty *conservative*, if with probability $1-\delta$ we have that if

$$\widehat{\epsilon}_v(T_v) + \alpha(m_v, |T_v|, \ell_v, \delta) \leq \widehat{\epsilon}_v(\emptyset)$$

then

$$\epsilon_v(T_v) \leq \epsilon_v(\emptyset)$$

Namely, if we do not prune then with high probability T_v is an improvement over pruning.

An example for a conservative penalty function for a Boolean domain is,

$$\alpha(m_v, |T_v|, \ell_v, \delta) = \sqrt{\frac{(\ell_v + |T_v|)\log n + \log(1/\delta}{m_v}}$$

We can give a theoretical guarantee on this pruning (although we will give only the highlights). First we can show the following simple claim that follows from the conservativeness of the penalty. Let T_{opt} be the best pruning and T_{srm} the pruning using the SRM criteria. Then with high probability T_{srm} is a subtree of T_{opt}.

The main result is to bound the difference between the error of T_{srm} and T_{opt}. We can show the following relationship,

$$\epsilon(T_{srm}) - \epsilon(T_{opt}) \leq \sqrt{\frac{|T_{opt}}{m}}\, O(\log(|T_{opt}|/\delta) + h_{opt} \log(nm/\delta)),$$

where h_{optg} is the height of T_{opt}.

8.2.5 Model Selection

In this section we present an algorithm that produces few pruning possibilities. The algorithm will be wrapped by a model selection procedure (either Structural Risk Minimization, Hypothesis Validation, or any other) that chooses one of the pruning possibilities produced by the underlying algorithm. Generally the core algorithm will be given a decision tree T and target number of errors e and will produce the smallest pruned trees that has such errors rate. This algorithm is an instance of dynamic programming, where the main parameter is the number of errors.

Algorithm

1. Build a tree T using S.
2. For each e, compute the minimal pruning size k_e (and a pruned decision tree T_e) with at most e errors.
3. Select one pruning using some criteria.

We will first show how to find the pruning that will minimize the decision tree size for a given number of errors. (Or alternatively, find the pruning that for a given tree size minimizes the number of errors.)

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 dynamic programming algorithm. We use $\mathbf{T}[0]$ and $\mathbf{T}[1]$ to represent the left and right child subtrees of $\mathbf{T}$ respectively. We denote by $\mathrm{root}(\mathbf{T})$ 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                          /* Test if $T$ is only a leaf */
        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                /* Keep root as leaf */
        $s = 1$
        $\mathbf{P} = \text{root}(\mathbf{T})$
        Return
    For $i = 0 \ldots k$                        /* Check for $i$ and $k - i$ errors at children */
        $(s_i^0, \mathbf{P}_i^0) \leftarrow \mathbf{prune}(i, \mathbf{T}[0])$
        $(s_i^1, \mathbf{P}_i^1) \leftarrow \mathbf{prune}(k-i, \mathbf{T}[1],)$
    $I = \arg\min_i\{s_i^0 + s_i^1 + 1\}$
    $s = s_I^0 + s_I^1 + 1$
    $\mathbf{P} = \mathbf{MakeTree}(\text{root}(\mathbf{T}), \mathbf{P}_I^0, \mathbf{P}_I^1)$
    Return
```

Note that given the values for the two child subtrees, we can compute the values of the node. The basic idea is therefore to do the computation from the leaves up towards the root of the tree. The running time can be computed as follows. At each node we need to go over all values of $i \in [0, k]$, namely $k+1$ different values. For each value of i we need to compute the sum of the size of the pruned left subtree with i errors and the size of the pruned right subtree with $k-i$ errors. This is done in $O(1)$. Building the pruned decision tree is also $O(1)$ since it involves only pointer manipulation. (If we will duplicate the pruned tree then it would be at most $O(|T|) = O(m)$.)

Assume that we have a sample size of m. This implies that the decision tree is of size at most $O(m)$. The overall running time is $O(m^2)$, since $k \leq m$ and the running time of each invocation is $O(k) = O(m)$.

Model Selection using Hypothesis Validation

We use a held-out set S_2. We construct the different pruning P_i, where $i \in [1, m]$. We will select between the P_i using S_2. Namely, we will select the P_i which has the least number of errors on S_2.
How large should S_2 be?
Similar to selecting a hypothesis from a finite class of m hypothesis (in the PAC model), if

we set $|S_2| = \frac{1}{\epsilon^2} \log \frac{m}{\delta}$ then with probability $1-\delta$ we have for each P_i a deviation of at most ϵ in estimating the error of P_i. This implies that with probability $1-\delta$ the pruning P_i which minimizes the observed error of S_2 has true error of at most 2ϵ more than the true error of the best pruning P_j.

Model Selection using Structural Risk Minimization

We have a set of prunings P_i, for $i \in [1, m]$. We will select between them using the existing examples, and not use another set of examples. The Structural Risk Minimization (SRM) will use the following decision rule,

$$P_i^* = arg \min_{P_i} \left\{ error(P_i) + \sqrt{\frac{|P_i|}{m}} \right\}$$

Model selection - Summary

The main drawback of this approach is the running time, which is quadratic in the sample size. This implies that for a large data set the approach will not be feasible. The benefit is that in case of small data sets it allows to utilize much better the examples we have.

8.3 Bibliographic Notes

The following papers have been used for this leacture notes, and have additional proofs which are missing from this scribe notes.

1. Michael J. Kearns, Yishay Mansour: On the Boosting Ability of Top-Down Decision Tree Learning Algorithms. J. Comput. Syst. Sci. 1999 (preliminary version STOC 1996).

2. Yishay Mansour: Pessimistic decision tree pruning based Continuous-time. ICML 1997

3. Michael J. Kearns and Yishay Mansour: A Fast, Bottom-Up Decision Tree Pruning Algorithm with Near-Optimal Generalization. ICML 1998