COMS 6998-2 Advanced Topics in Computational Learning Theory **Spring 2005**

Lecture 2 : January 27, 2005

Lecturer: Rocco A. Servedio *Scribe: David Goldberg*

1 Introduction to the Online Mistake Bound Learning Model

In the mistake bound learning model, the learning algorithm is told the concept class C to which the target concept c belongs. The learner works with a class of hypotheses H(not necessarily the same as C), and chooses an initial hypothesis $h \in H$. Since the learner knows nothing about c other than its general form (that it is from the class C), this initial hypothesis will not likely be a good approximation to C. Now the learner is given an instance x from the domain of c. The learner has no control over what instance is given. The learner calculates $h(x)$ and is then told $c(x)$. The learner then updates h. It is possible that the same instance x can be repeated, and thus the learner may or may not have already been given $c(x)$. In this model, the goal of the learner is to make the fewest mistakes possible. We say that the concept class the learning algorithm has mistake bound M if for any concept $c \in C$ the total number of mistakes made by the learner will never exceed M. The concept class C is learnable in time $T = M \cdot t$ if there exists a learning algorithm A with mistake bound M and runs for t time steps after each example.

2 The Halving Algorithm

The halving algorithm is an online learning algorithm that can be used to learn any finite concept class. The learner's initial hypothesis consists of a lookup table containing all of the concepts in the concept class C. To evaluate the hypothesis, the learner will evaluate an instance x on every concept in the lookup table, and return the majority vote. If $h(x) = c(x)$, the learner makes no update to h. If $h(x) \neq c(x)$, the learner updates h by removing all concepts h' from the lookup table such that $h'(x) \neq c(x)$.

Theorem 1 *For any finite concept class C, the Halving Algorithm has mistake bound $M = \log(|C|)$.*

Proof: Suppose the learner makes a mistake on some example x. Then at least half of the hypothesis h' in the lookup table predicted incorrectly on x and will be eliminated. In other words, the number of hypothesis in the lookup table after m mistakes is at most $\frac{|C|}{2^m}$. Since the table must have at least one target concept, we have that $\frac{|C|}{2^m} \geq 1$ or, equivalently, $m \leq \log(|C|)$■

Note that although the Halving Algorithm has a small mistake bound, the size of the table is initially $|C|$. Since every hypothesis must be evaluated in the first step, the running time is at least $|C|$.

3 The Elimination Algorithm

We consider a simple algorithm for learning the class of monotone disjunctions. By monotone we mean that there are no occurences of negative literals in the disjunction. The learning algorithm's initial hypothesis h is a monotone disjunction containing all n variables. After getting an instance x and evaluating $h(x)$, if $h(x) = c(x)$ the algorithm makes no update to h. If $h(x) = 1$ while $c(x) = 0$, then the algorithm removes all variables x_i that were set to 1 in x. It is easy to see that none of the x_i can be contained in the target disjunction or else $c(x)$ would have evaluated to 0. Thus the algorithm will never receive an example such that $h(x) = 0$ and $c(x) = 1$, since h will never remove any variables from h that are in the target disjunction.

Theorem 2 *The mistake bound of the elimination algorithm is n.*

Proof: We have already shown that the algorithm will never predict $h(x) = 0$ when $c(x) = 1$. We need now to show that after each mistake the algorithm will eliminate at least one variable. This is easy to see since in order for h to predict 1 while c predicts 1 on x, there must be some variable in h not in c that is set to 1 under x. Since there are n variables in the initial disjunction, the algorithm can never make more than n mistakes. ■

A slightly better bound comes from realizing that once all variables not in the target concept have been removed from the hypothesis, the hypothesis will be exactly the target concept. If the target concept contains r variables, then at most $n-r$ variables can be removed from the hypothesis. Since evaluating an n-variable disjunction and eliminating variables can be done in $O(n)$ time by simply individually examining each variable in the disjunction, the algorithm runs in time $O(n^2)$. It should be noted that for the class of monotone disjunctions, as demonstrated earlier, $|C| = 2^n$, and thus an algorithm that runs in time $O(n^2)$ is pretty good.

A bonus - learning non-monotone disjunctions: Now consider the problem of learning a concept from the class of all disjunctions over n variables, in which some of the variables may be negated. Consider the following transformation: for the negation of each variable x_i, create a new variable x_{n+i} which is implicitly defined by x_i by saying that in any instance in which $x_i = 1$, $x_{n+i} = 0$, and visa-versa. As far as evaluating the hypothesis goes, treat these special variables like standard variables: if any of them is set to 1, the disjunction evaluates to 1. Thus let your initial hypothesis contain all variables, and all the negations of these variables (in the form of these transformed x_{i+n}). Since the target concept can contain only these variables and special negated variables, running the same algorithm as before on these 2n variables will succeed by the same logic. Thus this transformation reduces the problem of learning non-monotone disjunctions over n variables to that of learning monotone disjunctions over 2n variables, which can be learned by the elimination algorithm with mistake bound O(2n) = O(n) and runtime O((2n)^2) = $O(n^2)$

4 An Algorithm for Learning Decision Lists

A generalized decision list is a decision list which, unlike normal decision lists in which each box of the decision list contains one rule (i.e. $x_1 \rightarrow 1$), each box is allowed to contain multiple rules. The generalized decision list is evaluated on an instance x by looking at the first box, and scanning the set of rules sequentially until a rule is reached that applies to the instance x. If no such rule is found, the next box is examined in the same way, and so on until a rule that applies is found. Since every such decision list has a default value rule, which applies in all situations, an applicable rule will always be found.

Example 1 *If the first box of the generalized decision list contained the rules $x_1 \rightarrow 1$, $\bar{x_3} \rightarrow 1$, $\bar{x_1} \rightarrow 0$, and the instance was 001, the first box would be scanned sequentially as follows. Since the instance x does not have $x_1 = 1$, the first rule does not apply and is skipped. Since the instance does not have $\bar{x_3} = 1$, this rule is also skipped. Since the instance does have $\bar{x_1} = 1$, this rule is applied and 0 is returned.*

The initial hypothesis is a generalized decision list that has every possible rule in the first box, and no other boxes. For example, on 2 variables, the set of rules would be $x_1 \to 1$, $x_1 \to 0$,$\bar{x_1} \to 1$, $\bar{x_1} \to 0$, $x_2 \to 1$, $x_2 \to 0$,$\bar{x_2} \to 1$, $\bar{x_2} \to 0$, 1, 0. The last two rules are the default rules, which always evaluate to 1 or 0 respectively. Thus for a set of n variables, there are a total of $4n + 2$ rules, since there are 4 rules for each variable ($x_i \to 1$, $x_i \to 0$, $\bar{x_i} \to 1$, $\bar{x_i} \to 0$), and two default rules (0,1). The order in which these rules are initially placed in the first box is not important. Given an instance x, the learning algorithm evaluates the hypothesis generalized decision list on x as described above. This will result in an applicable rule R being selected. If $h(x) = c(x)$, no modification is made to the hypothesis decision list. However, if $h(x) \neq c(x)$, the rule R is pushed down the list to the next box in the hypothesis decision list. If there is currently no next box, that box is created and rule R is made the first rule in the box. If the box already exists, rule R is placed at the bottom of the list of rules already in the box. We now prove the following claim which helps us determine the mistake bound of our algorithm.

Claim 1 *Let c be a decision list or length r made of rules $c_1, c_2, \ldots, c_r$. Then for $1 \leq i \leq r$, rule c_i is never pushed below box i in h.*

Proof: Let c_i be the first rule that would be pushed below box i in a run of the algorihtm. Then h maked a mistake on x using rule c_i. Since all rules began in box 1 and c_i is the first rule to be demoted below its correct level, rules $c_1, \ldots, c_{i-1}$ reside in boxes $1 \ldots i$ and c_i resides in box i. Then rules $c_1, \ldots, c_{i-1}$ must not have been satisfied by x in h. Furthermore, x must satisfy rule c_i for h to predict using it. But then $c_i(x)$ gives the correct output so our algorithm would not denote it. ■

Since c has r rules, we see that our algorithm will never demote any rule more than $r+1$ times(those rules not in c will be pushed to the $r+1$-st box). The hypothesis h has $(4n + 2)$ rules, and a rule is demoted each time the algorithm makes a mistake. This gives a mistake bound of $(4n + 2)(r + 1) = O(nr)$. Each update and prediction takes $O(n)$ time, so the running time of our algorithm is $O(n^2 r)$.

5 Learning Decision Trees

Definition 1 *A k-Decision List is a decision list in which each box may hold a conjunction of size $\leq k$*

We can view a k-decision list as a normal decision list over $\{0,1\}^{n^k}$, where we have a variable for each possible conjunction of size at most k. It is easy to see that our algorithm above can learn any k-decision list over $\{0,1\}^n$ in $n^{O(k)}$ time. We may also use this algorithm to learn decision trees by converting the decision tree to a k-decision list, as we show in the following theorem.

Theorem 3 *Any s-leaf decision tree over n variables is expressible as a k-decision list with $k = \log s$.*

First we prove the following claim:

Claim 2 *:An s-leaf Decision Tree has at least one leaf at depth* $\leq \log(s)$

Proof: By definition, every internal node of a DT has two children. Assume that some s-leaf DT has all leaves at depth $> \log(s)$. Since all leaves are at depth $> \log(s)$, it must be that all nodes at depth $\log(s)$ and below in the tree are internal nodes, and thus each have two children. Thus the number of nodes at level 0 will be 1(just the root), which will have two children and thus the number of nodes at level 1 will be 2, at level 2 will be 4,...and at level k will be 2^k. Since every node at depth $< \log(s)$ has two children, the number of internal nodes at depth $\log(s)$ will be $2^{\log(s)} = s$. Since each of these nodes will have two children, the total number of leaves in the tree will be $\geq 2s$. But this is impossible since the DT has only s leaves. Thus a contradiction is reached, and it must be that an s-leaf DT has at least one leaf at depth $\leq \log(s)$ ■

Proof:

Setup: Consider an s-leaf descision tree T. Denote the root of T by ROOT. As was proven above, T must have at least one leaf at depth $\leq \log(s)$. Call this leaf X. Denote X's parent by P. Since all internal nodes of T have two children by the properties of DT, X must have a sibling, which we will call S.

Initial Step: Trace out the unique path through the DT leading to the leaf X. By the properties of DT, this path K represents a series of boolean conditions on the internal nodes that appear on that path, where each internal node represents some variable x_i and each edge on the path represents an internal node being set to 1(if the right edge leaving an internal node appears on the path) or 0 (if the left edge leaving an internal node appears on the path). Thus this leaf is reached on the DT $\leftrightarrow$ this path is traversed, which occurs $\leftrightarrow$ the variables that appear on the path are assigned the truth values necessary to create the path. In the case that the leaf is reached, the DT outputs the value of the leaf, which is a 0 or 1. Thus if the nodes appearing on the path K are x_1,x_2,...x_k, and if the path K leaves nodes x_1,x_2,...x_i to the right and nodes x_{i+1},x_{i+2},...$x_{i+3}$to the left, this path will be traced out in the DT, and the value of the leaf X returned, $\leftrightarrow$ x_1=x_2...=x_i=1 and x_{i+1} = x_{i+2}...=x_k=0, which is equivalent to requiring x_1=x_2=...x_i = $\bar{x_{i+1}}$ = $\bar{x_{i+2}}$...=$\bar{x_k}$ = 1. But this is equivalent to the conjunction C = $x_1x_2...x_i\bar{x_{i+1}}\bar{x_{i+2}}...\bar{x_k}$. Since if this conjunction C is true, the DT outputs the value of leaf X, val(X),the leaf X of the DT can be fully described by: if C, output val(X). Since the leaf X was at depth $\leq$ log(s), it must be that the number of nodes on the path to the leaf X was $\leq$ log(s), and thus the number of variables appearing in the conjunction C is $\leq$ log(s) and thus this is equivalent to a valid rule R in a log(s)-DL.

Followup step: Now since the DT will reach the leaf X and return val(X) if and only if that exact conjunction C is satisfied, this condition entirely treats the situation in which the conjunction C holds for some instance x and the DT is to be evaluated on x. Thus the DT is equivalent to the DT without leaf X combined with the rule R. However, all other paths and leaves on the DT must be unchanged. Consider the following transformation on the tree T: replace X's parent P with X's sibling S.

Transformation Justification: Since X is a leaf, it has no children, and thus there is nothing to consider with regard to any children of X. Consider any leaf L that can be reached on a path going through the node S. Since S and X were siblings, they had a common parent P. Thus the path to get from the root to X differs only in one condition from the path to get from the root to S, namely that the edge traversed from P is in the opposite direction as it was to get to X. This must be since there is only one path from the root to any node, and thus there is only one path from the root to P, and thus this last edge which leaves P is the only way in which the two paths can differ. However, if all the variables are set so that P is reached AND P evaluates so that the edge needed to reach X is traversed, then the DL extracted in the previous step will output the correct value. Thus the DL will only reach the sibling S if the path to P is traversed and P evaluates so that the edge needed to reach X is not traversed, aka P has the opposite truth value as it did when X was reached. Now consider creating a DL, and placing the extracted rule R in the first box. If all the variables are set so that the path to P is traversed and P evaluates so X is reached, then since R is in the first box and R is satisfied for any such instance, the DL will correctly output the value the DT would have given. Now, since in this DL only one rule will be allowed to appear in each box, and once a rule is placed in a box it will not be moved into another box, this means that all other rules extracted from the DT will appear in later boxes of the DL. In light of this fact, reconsider the condition for the sibling to be reached in the DT. The path to K must be traversed, and P must evaluate to the opposite of the value needed to reach X. Since any rule R' either containing the variable in node S in its conjunction (if S is an internal node) or using val(S) as its output (if S is a leaf) must appear after the rule R in the DL, since R appears in the first box, it will be the case that R' is only reached if P evaluates so that S is reached instead of X, for if it evaluated so that X was reached then rule R would come into effect instead of rule R'. Thus the structure of the DL implicitly limits R' to the situation in which P evaluates so that S is reached instead of X. Thus by placing rule R' further down in the DL then the rule R, the fact that P evaluates to the S-condition is implicit in the DL having reached R', and thus need not be included in the conjunction describing R'. Thus after the initial rule R is extracted, the parent P of X need not be included in any conjunctions that appear in the DL, and thus need not be considered at all. Thus, assuming that all possible rules that define the DT are eventually extracted in this way, one can replace P by X's sibling S and still be certain that the DL consisting of these rules extracted in this order will be equivalent to the original DT.

Recurse: Now to consider the recursive step, examine the DT after the transformation. Since this tree has s-1 leaves, it must be that the tree has a leaf at depth $\leq \log(s-1) \leq \log(s)$. Thus one can turn to this leaf and repeat the same process by the same logic as above, extracting a second rule and placing it in the second box of the DL. Since this process always removes a leaf, at stage i there will be s-i leaves, and thus by the same logic there must exist a leaf at depth $\leq \log(s-i) \leq \log(s)$. Thus all rules extracted from the DT in this way will have associated paths of length $\leq \log(s)$ and thus associated conjunctions of length $\leq \log(s)$. Thus, repeating this process until all rules are extracted from the DT and placed into the DL will yield a DL in which each box contains a conjunction of length at most $\log(s)$ which evaluates all instances x in the same exact way as the original DT.

Conclusion: The conclusion is that the above algorithm converts any s-leaf DT into an equivalent $\log(s)$-DL, and thus every s-leaf DT must be equivalent to some $\log(s)$-DL.

■

Since this conversion involves, for every stage of the conversion, locating a leaf node of depth $\leq \log(s)$, walking the path to the leaf node, and performing a constant number of update operations which take constant time, the runtime will be dominated by locating shallow leaf nodes and walking the paths to the leaf nodes. Since the shallow leaf node must be at depth $\leq \log(s)$, and a binary tree has at most 2^d nodes at depth d as proven earlier, it must be that the first $\log(s)$ levels of the tree (collectively) contain at most $\sum_{i=0}^{\log(s)} 2^i = 2^{\log(s)+1} = 2s$ nodes, scanning these levels of the tree for a leaf node can be done in time $O(s)$. Since all leaves handled will be at depth $\leq \log(s)$, walking the path to each node can be done in time $O(\log(s))$, and thus the runtime is dominated by the time to find a shallow node, which can be done in O(s) time. Since a shallow node must be found at each stage of the conversion algorithm, and the algorithm has one stage for each of the s leaves, the runtime of the conversion algorithm is $O(s^2)$.

Thus we may conclude that since every s-leaf DT can be converted to an equivalent $\log(s)$-DL, and every $\log(s)$-DL is learnable in time $O(n^{O(\log(s)})$,every s-leaf DT is learnable in time $O(n^{O(\log(s)})$. This term dominates the $O(s^2)$ time necessary for the conversion, and thus s-leaf DT are learnable in $O(n^{O(\log(s)})$ time

5.1 The rank of a DT

Definition 2 *The rank of a DT, or for any binary tree for that matter, is defined recursively as follows:*

Example:

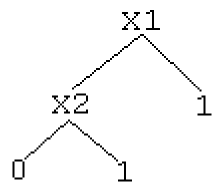


Figure 1: Here is the original DT

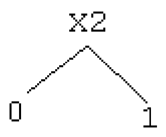


Figure 2: A shallow leaf, the right child of x_1, was extracted by the above algorithm. The leaf was deleted, and its parent x_1 was replaced by the leaf's sibling, x_2

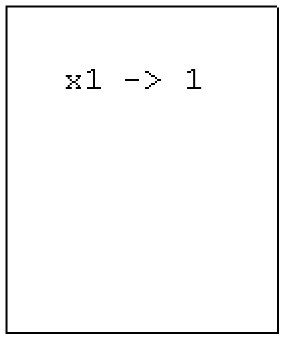


Figure 3: the path to get to the recently removed node was to move right from the x_1 node, which means the conjunction representing the path is just x_1. Since val(leaf) = 1, the extracted rule will be $x_1 \rightarrow 1$. This rule is put in the first box of a DL

Figure 4: A shallow leaf, the right child of x_2, was extracted by the algorithm. The leaf was deleted, and its parent x_2 was replaced by the leave's sibling, which happens to be a leaf with value 0

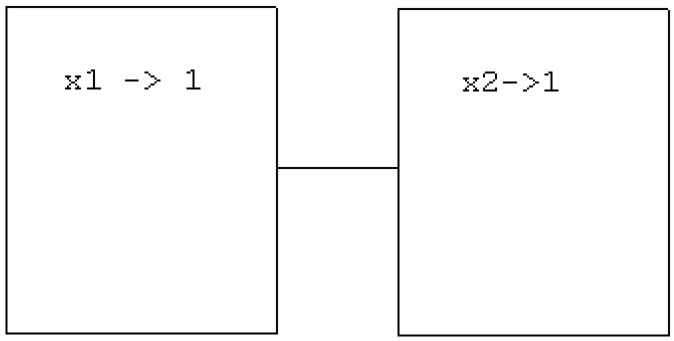


Figure 5: The path to get to the recently removed node was to move right from the x_2 node, which means the conjunction representing the path is just x_2. Since val(leaf) = 1, the extracted rule will be $x_2 \rightarrow 1$, and it will be placed in the second box of the DL

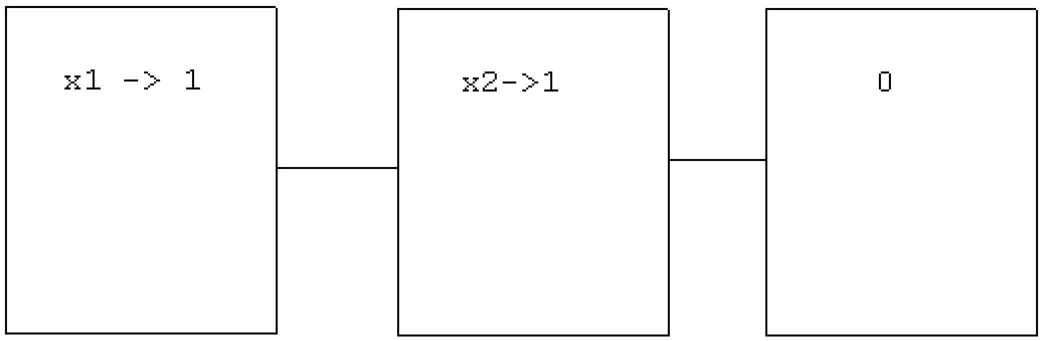
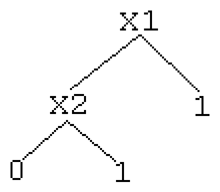


Figure 6: Since the only remaining node was a leaf, it is extracted. Since it was the root, there was no path needed to reach it, and thus the rule is simply 0, which is a default rule. This is placed in the third box of the DL, and the process is complete since the DT is empty



Example:

Figure 7: To compute the rank of this DT recursively, first observe that the rank of each of the subtrees rooted at a leaf = 0. Thus the rank of the subtree rooted at the x_2 node is 1 + 0 = 1 since both its left and right subtrees have equal rank (rank(R) = rank(L) = 0). Thus the rank of the tree rooted at x_1, which is the entire tree, is the max(1,0) = 1 since the tree rooted at node x_1's left child has rank 1, while the tree rooted at its right child has rank 0. Thus the tree has rank 1.

For a Tree T with left subtree L and right subtree R

$$rank(T) = \begin{cases} 0 & \text{if T is a leaf} \\ rank(R) + 1 & \text{if rank(R) = rank(L)} \\ max(Rank(R), Rank(L)) & \text{if rank(R) } \neq \text{rank(L)} \end{cases}$$

Claim 3 *Any rank-r DT T has at least 2^r leaves*

Proof: Use a proof by induction

Base case: if rank(T) = 0 it must be that T is just a leaf, since if T has any internal nodes then it must be that some internal node has two leaf children, and thus the rank of that internal node will be 1 by the definition of rank. Since moving upward in the tree the parent of a node must have rank at least as great as the node itself by the way rank has been defined (since rank(parent) is either equal to rank(child)+1 or max(rank of its children)), and moving upward recursively parent to parent from this internal node with two leaf children will eventually lead to the root, it must be that the root and thus the tree itself has rank $\geq 1 > 0$. Thus, the only way for the rank of a tree to be zero is for that tree to consist of just a leaf. In this case, the tree has exactly one leaf. Since $2^0 = 1$, in this base case the number of leaves is $\geq 2^{rank}$

Inductive case: assuming that a rank-r DT has at least 2^r leaves, we must show that a rank-(r+1) DT has at least 2^{r+1} leaves. Assume the tree T has left subtree L and right subtree R. There are two cases to consider. If rank(L) = rank(R), then rank(T) = rank(L)+1 = r+1 and thus it must be that rank(L)=rank(R)=r. But by the inductive hypothesis, any tree of rank r has at least 2^r leaves. Thus each of the two subtrees rooted at T has at least 2^r leaves, and thus the entire tree rooted at T has at least $2^r + 2^r = 2^{r+1}$ leaves, and the induction is demonstrated. In the other case, it must be that one of T's children has rank r+1, and the other has rank <r+1. One can simply recurse on this child C with r+1 rank. Thus if the left and right subtrees of this child C have equal rank, the problem reduces to the solved case. If again the children have uneven rank, simply recurse again on the child with rank r+1. If this recursive step is performed 2^r times without termination, it must be that the tree has at least 2^r internal nodes and thus at least 2^r leaves since at least that many nodes have been examined by the algorithm. Also, if the recursion terminates, it must mean that some node had two children, each with rank r, and thus the induction hypothesis is similarly satisfied. It is impossible for the recursion to reach a leaf node since a leaf node has rank 0, and thus could not have been a child with rank r+1. Thus one of the above two conditions will always be satisfied, and thus either way the inductive hypothesis is proven. ■

Claim 4 *Any rank-r DT T is learnable in time $O(n^{O(r)})$*

Proof: Since any rank-r DT has at least 2^r leaves, and as shown earlier any DT with 2^r leaves is learnable in time $O(n^{O(\log(2^r))}) = O(n^{O(r)})$ time. ■

Claim 5 *Any rank-r DT has a leaf at depth $\leq r$ and is equivalent to some r-DL*

Proof: This follows directly from the laws proven earlier for s-leaf DT and the result guaranteeing that a rank-r DT has at least 2^r leaves ■

6 Learning s-term DNF

6.1 A Beginning

Claim 6 *Any s-term DNF over n variables is equivalent to an n-DL of length s*

Proof: Since an s-term DNF over n variables is the disjunction of s conjunctions, each of length $\leq n$, simply creating a DL in which each box holds a different one of the s conjunctions and outputs 1 will suffice, with a default case of zero. To demonstrate this, assume that the s-term DNF returns 1 on some instance x. Then it must be that at least one of its terms was true in the instance. Since each of its terms appear in the DL, the DL will find the first such term that applies to the instance and return the output bit, which is one. Similarly, assume that the DL returns 1 on some instance x. Then it must be that some rule in one of its boxes applied to the instance, and thus evaluated to one on the instance, for if this was not the case the DL would reach the default case and output 0. Thus it must be that some rule's conjunction evaluated to 1 on the instance. But each rule's conjunction is a term of the DNF. Thus, since some term of the DNF evaluated to 1 on the instance, the DNF must have evaluated to 1. Similarly, assume that the DL returns 0 on some instance x. Then it must be that the default case was reached, and thus none of the conjunction's from any of its rules evaluated to 1 on the instance x, and since these are exactly the terms of the DNF it must be that none of the DNF's terms evaluated to 1, and thus the DNF evaluated to 0. If the DNF returns 0 on some instance x, it must be because none of its terms evaluated to 1 on x, and thus none of the conjunctions in the DL will evaluate to 1 and thus the default will be reached and 0 returned. Thus the DL returns 1 on an instance x $\leftrightarrow$ the DNF returns 1, and the DL returns 0 on x $\leftrightarrow$ the DNF returns 0, and the two are equivalent. ■

Claim 7 *Any n-DL over n variables of length s can be expressed as a* $O(\sqrt{n}\log(s)\log(n))$*-DL*

This will be proven later in the course. The proof relies on analyzing a DT-DL hybrid, in which a DT is allowed to have DL for leaf nodes.

Claim 8 *s-term DNF over n variables can be learned in time* $O(n^{O(\sqrt{n\log(s)\log(n)})}) = O(2^{\sqrt{n\log(s)}\log(n)^{3/2}}) \sim O(2^{\sqrt{n}})$

This follows directly from the above claims and the results already proven about learning DL.

7 Some odds and ends

7.1 Expressing DL as LTF

Claim 9 *The weight of the LTF for a DL over n variables is $\Omega(2^n)$*

Proof sketch: Due to the structure of a DL, the rule in the first box must take precedence over all other rules. Thus even if rules 2,3,4,...n all evaluate to true on the input and all return an output bit of negative 1, as long as that first rule also evaluates to true and returns an output bit of positive 1, the entire DL must evaluate to 1. By the same logic, even if rules 3,4,5,...n all evaluate to true on the input and all return an output bit of negative 1, as long as the first rule does not apply to the input (evaluates to false) and the second rule does evaluate to true and returns an output bit of positive 1, the DL must evaluate to positive 1. To simplify the analysis, assume that x_i appears in box i in the decision list, and that $\forall$odd i, the DL rule associated with x_i returns an output bit of negative 1, and that $\forall$ even i, the DL rule associated with x_i returns an output bit of positive 1. Let w_i be the magnitude of the coefficient of x_i in the LTF, and assume that the LTF has threshold 0. Since even if the only variables assigned a truth value of 1 are x_1,x_2,x_4,x_6,...x_n the LTF must still evaluate to true, it must be that $w_1 \geq \sum_{i=1}^{\frac{n}{2}} w_{2i}$. Similarly, since even if the only variables assigned a truth value of 1 are x_3,x_4,x_6,...x_n, the LTF must still evaluate to true, it must be that $w_3 \geq \sum_{i=2}^{\frac{n}{2}} w_{2i}$. By the same logic, $\forall k$,$w_{2k-1} \geq \sum_{i=k}^{\frac{n}{2}} w_{2i}$. But applying the same logic to the negative examples yields a symmetric situation for the even coefficients, namely that $\forall k$,$w_{2k} \geq \sum_{i=k}^{\frac{n}{2}-1} w_{2i+1}$. Thus any integer LTF representation must satisfy these restrictions on the weights. It can be shown that the smallest integer representation satisfying these conditions has weight $2^{\Omega(n)}$.

7.2 Conservative Learning Algorithms

Fact 1 *Any online learning algorithm with mistake bound M can be converted to an online learning algorithm with mistake bound M that never makes a change to its hypothesis during rounds in which the hypothesis agreed with the target concept. Such an algorithm is called conservative.*

8 Recommended Reading

For a good survey of Online Learning Algorithms in general, read ”Online Algorithms in Machine Learning” by Avrim Blum

For the original proofs regarding the conversion of DT to DL, read ”Rank-r Decision Trees are a Subclass of r-Decision Lists” by Avrim Blum

To learn about the DNF claims that went unproved, read ”A Subexponential Exact Learning Algorithm for DNF Using Equivalence Queries” by Nader Bshouty

For an excellent survey of graph and tree algorithms, check out my personal favorite algorithmic bible CLRS.