CS 395T Computational Learning Theory

Lecture 3: September 10, 2007 Lecturer: Adam Klivans

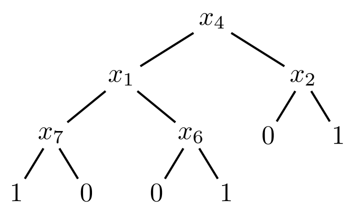
Scribe: Mike Halcrow

3.1 Decision List Recap

In the last class, we determined that, when learning a t-decision list, each update step could take time $n^{O(t)}$. The Mistake Bound (M.B.) is, at most, $n^{O(t)}$. When $t = 1$ – or any constant, for that matter – we get a polynomial-time learning algorithm.

3.2 Decision Tree

Definition 1 *In a decision tree, Each node has in-degree 1 and out-degree 2.*



$1000110 \to 1$
$1000100 \to 0$
$1001110 \to 0$
$0000000 \to 1$
$0111100 \to 1$
$0011100 \to 0$
$\ldots$

The decision is made by following the path in the tree to which the input corresponds. If the leaf is 0, output 0; if leaf is 1, output 1. This is a mapping function of the form $f : \{0,1\}^n \to \{0,1\}$. The *size* of decision tree is the number of nodes in the tree. A decision tree has many applications[1][2]

How can our analysis of the learnability of decision lists extend to the learnability of decision trees?

We start with the question, "Are polynomial-sized decision trees more powerful than polynomial-sized decision lists?"[3]

We begin our answer to this question by considering parity functions.

3.2.1 Parity Function

A parity function takes input $x \in \{0,1\}^n$ and outputs 1 if the number of 1's in x is odd; 0 otherwise.

[1]It can be a compact representation of probability of getting diabetes, for instance.

[2]There are such things as "Real-value Decision Trees." There are papers that demonstrate provably efficient learning bounds. This might be an area of research to pursue.

[3]For instance, a 3-decision list?

Let us consider the parity function on the first $\log n$ input bits of x: $PARITY_{\log n}$.

Question: What type of decision list can compute this function?

Answer: You need a $O(\log n)$ decision list to compute $PARITY_{\log n}$.

Question: Why can't a 1-decision list compute this parity function?

Answer: You have to look at all of the $\log n$ input bits to correctly decide.

To see why this is, we consider a conterexample. Suppose you had a 1-decision list that computed the parity function before having looked at all $\log n$ input bits. Now flip one of the bits in the input beyond the point of termination in the decision list. Flipping the bit should change the output, but that will not happen unless we read past this termination point to learn that the bit has been flipped. Therefore, we see that must read all $\log n$ bits in order to correctly decide $PARITY_{\log n}$. At best, we could hope for a $\log n$ decision list.

Fact 1 *Every polynomial-size decision tree on n variables can be computed by an $O(\log n)$-decision list.*

Both the running time and the mistake bound is $n^{O(\log n)}$ for a polynomial-sized decision tree. Whether or not there exists a polynomial-time algorithm is a major open problem.

What if you are the Learner, and you get to query the decision tree with any chosen input?

3.3 Rank of a Decision Tree

Definition 2 *A rank is a function mapping decision trees to integers. We specify $R(T)$ to be the rank of a tree T.*

$$R(T) = \begin{cases} 1 + R(T_1) & \textit{if } R(T_1) = R(T_2) \\ max(R(T_1), R(T_2)) & \textit{otherwise} \end{cases}$$

$$\begin{tikzcd}[column sep=small] & T \arrow[dl, dash] \arrow[dr, dash] & \\ T_1 & & T_2 \end{tikzcd}$$

For the examples in the remainder of these notes, let $R(0) = 0$.

Claim 1 *If a decision tree has size S, then rank $\leq \log S$.*

Proof:

We consider two cases; one where the ranks of the two subtrees under a given node are inequal and another where the ranks are equal.

Case 1: $R(T_1) < R(T_2)$. $R(T) = R(T_2)$ (according to the definition of *rank*). We know that T_2 has fewer than S nodes because it is a subtree, so $R(T_2) \leq \log S$. Therefore, $R(T) \leq \log S$.

Case 2: $R(T_1) = R(T_2)$. T_1 has $\leq \frac{S}{2}$ nodes. $R(T) = 1 + R(T_1)$ (according to the definition of *rank*). $R(T) = 1 + \log \frac{S}{2} = 1 + \log S - \log 2 = \log S$.

■

Claim 2 *A rank-t decision tree of size s has a* $(t+1)$*-decision list.*

Proof:

We induct on $(t+s)$.

We assume, without loss of generality, that the decision list never "falls off" the end. That is, of all the blocks in the decision list, at least one block "fires."

The base case is trivial; a rank-0 tree is size 1 and contains only a leaf node, so there will be only a single decision list element that outputs the decision.

Induction hypothesis: A rank-k decision tree of size l has a $(k+1)$-decision list for all (k+l) such that $1 \leq (k+l) < m$.

Inductive step: We prove that a a rank-k decision tree of size l has a $(k+1)$-decision list for $(k+l) = m$.

Let T be a decision tree of rank $R(T) = k$ and size $S(T) = l$. Let $(k+l) = m$. We prove that T must have a $(k+1)$-decision list by considering all the possible ranks and sizes for the children of T.

Case 1: $R(T_1) = R(T_2)$. Without loss of generality, we make T_1 the left child and T_2 the right child. By the definition of *rank*, $R(T_1) = (R(T) - 1)$. It follows that $R(T) = (R(T_1) + 1)$. Note also that $S(T_1) < S(T)$. By the induction hypothesis, we have that the decision list for T_1 is size $(R(T_1)+1)$. We want to show that the decision list for T is of size $((R(T_1)+1)+1)$, which would mean that the size of the decision list for T is $(R(T)+1)$.

To illustrate that this is in fact the case, we represent the decision list for T_1 as a list of conjuncts C_i, and we represent the decision list for T_2 as a list of conjuncts D_i. We concatenate the list of D_i conjuncts onto the list of C_i conjuncts. Since the decision list for T_1 is size $(R(T_1)+1)$ and since the decision list for T_2 is size $(R(T_2)+1)$, no one conjunct in the entire list has more than $(R(T_1)+1)$ literals.

We assign the root node of tree T to correspond with some literal x_j. We then add the literal $\bar{x}_j$ to each C_i conjunct, making the first part of the decision list comprised of C_i conjuncts of size $((R(T_1)+1)+1)$, which then becomes the size of the entire decision list. Whenever x_j is 0, that "turns on" the list of C_i conjuncts, "selecting" the T_1 tree; whenever x_j is 1, then "turns off" the list of C_i conjuncts, and the D_i basic blocks representing the T_2 tree are used to evaluate the remaining input literals.

We end up with a decision list of this form:

$$\begin{tikzcd}
\boxed{\bar{x}_j C_1} \arrow[r] \arrow[d] & \boxed{\bar{x}_j C_2} \arrow[r] \arrow[d] & \boxed{\bar{x}_j C_3} \arrow[r] \arrow[d] & \boxed{D_1} \arrow[r] \arrow[d] & \boxed{D_2} \arrow[r] \arrow[d] & \boxed{D_3} \arrow[d] \\
1 & 1 & 1 & 1 & 1 & 1
\end{tikzcd}$$

Case 2: $R(T_1) < R(T_2)$. Without loss of generality, we make T_1 the left child and T_2 the right child. By the definition of *rank*, $R(T_2) = R(T)$. Since T_2 is a child of T, $S(T_2) < S(T)$. By the induction hypothesis, we have that the decision list for T_2 is size $(R(T_2) + 1)$, and the decision list for T_1 is $< (R(T_2) + 1)$. We want to show that the decision list for T is of size $((R(T_2) + 1)$, which would mean that the size of the decision list for T is $(R(T) + 1)$.

Using the same representation of the decision list as was given in Case 1, we observe that adding an extra literal x_j to the C_i conjuncts would give us a decision list of size (at most) $(R(T_2) + 1)$, since no conjunct C_i had a size greater than or equal to $(R(T_2) + 1)$ prior to adding the literal. Thus, the size of the entire decision list is still $(R(T_2) + 1)$.

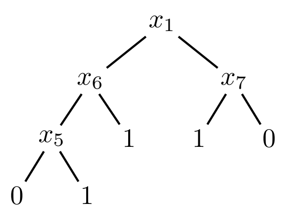
■

Thus we see that every decision tree is equivalent to a $\log n$ decision list. If a decision tree has size S, *rank* $\leq \log S$. A rank-t decision tree has a $(t + 1)$-decision list.

What is the next most interesting concept class that is more complicated than a decision tree?

3.3.1 DNF

A Disjunctive Normal Form (DNF) expression is comprised of "or's" of "and's." For instance:



$(\bar{x}_1 \wedge x_6) \vee (x_1 \wedge \bar{x}_7) \vee (\bar{x}_1 \wedge \bar{x}_6 \wedge x_5)$

In this case, the number of terms is 3 and the number of variables is 7. In general, we say that the number of variables is n, and the number of terms is s. The size of any given DNF is the number of terms in the formula; that is, s. s is n^2, n^3, ... In other words, s is polynomial according to n.

What is the mistake bound of learning DNF? Which is more general? Are DNF formulas more powerful than decision lists?[4]

Question: Why can every polynomial-size decision tree be computed as a DNF formula?

Answer: Take every leaf node and set that to a conjunction.

[4]Open problem: How can we PAC-learn DNF formula's?

3.3.2 For the next class

There are polynomial-size formula's that cannot be computed with a polynomial-size decision tree.

We will begin with a $2^{\sqrt{n}\log n}$-time algorithm and MB and then move on to show a $2^{n^{\frac{1}{3}}\log n}$-time and MB.

Claim 3 *When we have a DNF formula with n vars and s terms, with t being an arbitrary parameter, there is a decision tree such that the rank of the tree is $\frac{2n}{t}\log s$. At each leaf node, there is a DNF where all terms are of length $\leq t$.*