Homework 1 TTIC 31250

Eddie Shim
(worked partially with Hanqi Zhang)
eddieshim@uchicago.edu

04/15/2020

Exercise 2

Show that decision lists are a special case of linear threshold functions. That is, any function that can be expressed as a decision list can also be expressed as a linear threshold function $f(x) = +$ iff $w_1x_1+...+w_nx_n \geq w_0$

Answer: For any given decision list, we can take a reverse recursive approach in order to translate it into a linear threshold function. Let's say we have are given any decision list described as:

- x_1 if $x_1 == \{0, 1\}$ then {positive, negative}
- x_2 if $x_2 == \{0, 1\}$ then {positive, negative}
- ...
- x_n if $x_n == \{0, 1\}$ then {positive, negative}

From here we can gradually recursively build up a equivalent linear threshold function working from the bottom up of this decision list. So, starting with x_n if $x_n == 0, 1$ then {positive, negative} we define the following base case:

Base case:

- Case 1: $x_n == 1$ then positive: $f(x) = x_n - 1 \geq 0$

- Case 2: $x_n == 1$ then negative: $f(x) = -1 * x_n \geq 0$
- Case 3: $x_n == 0$ then positive: $f(x) = x_n - 1 \geq 0$
- Case 4: $x_n == 0$ then negative: $f(x) = -1 * x_n \geq 0$

From the base case, we can recursively add onto the current linear threshold function we have by adding a $w_i x_i$ term with each iteration. We iterate backwards from term x_{n-1} to x_1 with the following rules:

- Case 1: $x_i == 1$ then positive: $w_i = -1 * (\sum(positiveWeights_i)) + 1)$
- Case 2: $x_i == 1$ then negative: $w_i = \sum(|negativeWeights_i|) + 1$
- Case 3: $x_i == 0$ then positive: $w_i * (1 - x_i)$ where $w_i = -1 * (\sum(positiveWeights_i)) + 1$
- Case 4: $x_i == 0$ then negative: $w_i * (1 - x_i)$ where $w_i = \sum(|negativeWeights_i|) + 1$

Here, the term $positiveWeights_i$ means all positive w_i in our current linear threshold function, and $negativeWeights_i$ means all negative w_i in our current linear threshold function. Intuitively, this guarantees that the linear threshold function will follow x_i's decision list rule because this means that if x_i's condition is true, then the magnitude of w_i forces the linear threshold function to decide in x_i's decision because all prior weight magnitudes cannot surpass w_i's magnitude. Below is an example I worked out:

Let's translate the following decision list:

1. If $x_0 == 0$ then negative
2. If $x_1 == 1$ then positive
3. If $x_2 == 1$ then negative
4. If $x_3 == 1$ then negative
5. If $x_4 == 1$ then positive

- Step1: $1 * x_4 \geq 0$
- Step2: $-2x_3 + x_4 \geq 0$
- Step3: $-2x_2 + -2x_3 + x_4 \geq 0$

- Step4: $5x_1 - 2x_2 - 2x_3 + x_4 \geq 0$
- Step5: $-7(1 - x_0) + 5x_1 - 2x_2 - 2x_3 + x_4 \geq 0$

Our final translated linear threshold function is $7x_0 + 5x_1 - 2x_2 - 2x_3 + x_4 \geq 7$.

Exercise 3

Prove that a decision tree with l leaves has rank at most $log_2(l)$

Answer: I will approach this answer using proof by induction:
Base case: From the given definition of rank, we know that a decision tree with a single leaf is rank 0.
Inductive hypothesis: Assume that our claim is true for all trees with leaves $l \leq t - 1$.
Inductive step: Take any tree with t leaves. Let r_L and r_R represent the two subtrees stemming from the root. We know that both subtrees must have less than t leaves. Using our inductive hypothesis, both subtrees will have rank at most $log_2(t - 1)$. One of our subtrees (let's say r_L WOLOG) will have $\leq t/2$ leaves and therefore will have rank $\leq log_2(t/2)$. From here we can define two possible cases:

1. $r_L == r_R$. In this case, we have $rank(r) = rank(r_L) + 1 \leq log_2(t/2) + 1$
2. $r_L != r_R$. In this case, we have $rank(r) = max(rank(r_L), rank(r_R)) \leq log_2(t))$

Thus, our inductive step holds and we've proved that the maximum rank of a tree with l leaves is $log_2(l)$.

Problem 4

Give an algorithm that learns the class of decision lists in the mistake-bound model, with mistake bound $O(nL)$ where n is the number of variables and L is the length of the shortest decision list consistent with the data. The algorithm should run in polynomial time per example. Briefly explain how your algorithm can be extended to learn the class of k-decision lists with mistake bound $O(n^k L)$, where k- decision list is a decision list in which each test is over a conjunction of k variables rather than just over a single variable.

Answer: In order to learn a decision list, take each step (node) of the list as one with $4n$ choices. Each node can be either be negated or not, and could decide 0 or 1. We start our algorithm with a list of k bags, where the first bag contains all possible $4n$ nodes, and other bags contain nothing. From here, we can set

our algorithm to learn from given data by iteratively checking each feature x_i against the 4 possible choices that are possible in order to have a consistent decision list. For example, an input of 1001 would start with a choice such as $(x_0, 1), (\neg x_1, 0), (\neg x_2, 1)$, or $(x_4, 1)$). If no such logical branch exists in order to continue a consistent decision list, then we move on to the next feature. If our guess is correct, then nothing changes. If it's incorrect then we should demote our node into the next available bag. We continue with this process until we have a consistent decision list that covers each feature. Since there are $4n$ possible nodes with each node being demoted at most k times, we can draw our mistake bound at $O(4nk)$. In a $k-$decision list, we could extend our algorithm where each node would instead now have $4n^k$ choices. Now we expand our first bag to contain $4n^k$ nodes, and iterate through the same process where with each iteration we demote a node into a new bag if the algorithm was incorrect. In this case, the mistake bound is $O(n^k L)$.

Problem 5

Show that the class of rank-k decision trees is a subclass of k-decision lists.

Answer: I will approach this answer using proof by induction:

Base case: As a trivial case, we can see that when $k = 1$, a rank-1 decision tree is equivalent to a 1-decision list. Furthermore, we know by definition of rank that a decision tree of two leaves must have rank 1.

Inductive step: Take any decision tree T with $rank = r$. This implies that there is a leaf l that is r distance away from the root node. In order to reach leaf l, we must take the path $n_1, ..., n_r$ which as a decision list D translates to: $rule_1 \wedge ... \wedge rule_r$, where each $rule_i$ is the logical if/then path held at n_i. We know that D must be consistent with T, and that n_r has two children in T, where I'll define l as one child and n_{r+1} is the other child. We know that since D must be consistent with T, an observation that did not pass D namely the n_{r+1} child, must have had $rule_1 \wedge ... \wedge rule_{r-1} \wedge \neg rule_r$. Here we can create a decision list off a new tree T', where T directly connects nodes n_{r-1}, r_{r+1}. We know T' is at most rank r because we still maintain node n_r as our max node rank. By induction, T' is now our wanted transformed k-decision list and thus we've shown that any rank-k decision tree is a subclass of k-decision lists.