Learning Using Divide and Conquer

Nader H. Bshouty **Eyal Kushilevitz**

Abstract

We give Online learning model.

1 Divide and Conquer

In this section we use another fundamental algorithmic technique, the so-called "divide and conquer" technique. The goal here is to "decompose" the problem (the learning task in our case) into several sub-problems such that each of the sub-problems can be solved separately and such that the solutions to the sub-problems can be later combined into a solution for the problem itself. The main question is how to find such a useful decomposition. For this, we apply again a technique that we already used in the monotone theory: we run learning algorithms that were designed for a certain class $\mathcal{C}$ on a wider class $\mathcal{C}'$. Either our algorithm succeeds or it fails with some useful information regarding the target function that can be used for the decomposition.

In order to demonstrate the divide and conquer method we first consider the class of *unate* DNF. That is, the class of all DNF formulae which are c-monotone (as defined in Exercise **??** and in Section **??**) for some c. In other words, this is the class of all DNF formulae where each variable either always appears un-negated (in which case it is called a monotone variable) or it always appears negated (in which case it is called anti-monotone variable). If we know which variable is monotone and which is anti-monotone we can run the algorithm $\texttt{LearnMonotone}_c$ of Exercise **??**. Here, however, we deal with the more difficult case where the type of each variable (i.e., the assignment c) is unknown.

We present the following algorithm, called `LearnUDNF`, that efficiently learns unate DNF formulae using membership queries and equivalence queries. This algorithm will later be used as a building block for a more powerful divide and conquer algorithm. The algorithm is described in Figure 1; here we give an informal description of it. We first assume that the target function f is monotone. We run the algorithm `LearnMonotone` that learns monotone DNF formulae (which is equivalent to the algorithm $\texttt{LearnMonotone}_c$ for $c = 00\ldots0$). If the function f is indeed monotone then the algorithm `LearnMonotone` expects to see only positive counterexamples to its equivalence queries. If it gets a negative counterexample b as an answer to a query $\texttt{EQ}(h)$ this indicates the existence of an anti-monotone variable. We remind the reader that the hypotheses used by algorithm `LearnMonotone` are of the form

$h = T_{a^{(1)}} \vee \cdots \vee T_{a^{(r)}}$ where the $a^{(i)}$'s are the local minterms of f that were found so far. Since b is a negative counterexample then $h(b) = 1$ and therefore $b > a^{(i)}$ for some i. This cannot happen for a monotone function f since we have $b > a^{(i)}$ but $0 = f(b) < f(a^{(i)}) = 1$. To find a variable which is "responsible" for this, we start flipping one-by-one bits of b in coordinates where b has a 1-bit and the corresponding bit in $a^{(i)}$ is 0; after each such flip we ask a membership query to find out the value of f on the modified assignment. Since $f(b) = 0$ and $f(a^{(i)}) = 1$ at some point, after flipping the j-th coordinate, the value of the function is changed from 0 to 1. Then, we know that the corresponding variable x_j is anti-monotone in the function f. We flip the j-th bit of c from 0 to 1, again assume that the function f is c-monotone (with respect to the new c) and repeat the above process. Since we have at most n anti-monotone variables in f the algorithm $\texttt{LearnMonotone}_c$ will be run at most n times.

Algorithm `LearnUDNF:`

$c \leftarrow 00\ldots0$.

While (TRUE) do

- Run $\texttt{LearnMonotone}_c$ until it asks an equivalence query $\texttt{EQ}(h)$ (for h of the form $h = \vee_i T_{a^{(i)}}$) that receives a negative counterexample b.
- Let i be an index such that $b >_c a^{(i)}$.
- Use membership queries to find assignments b' and b'' that differ in a single coordinate j where $b \geq_c b' >_c b'' \geq_c a^{(i)}$ and $f(b') = 0$ while $f(b'') = 1$.
- Flip the j-th bit of c.

Figure 1: Algorithm `LearnUDNF`

We now get to the divide and conquer method. We present an algorithm, called `DCLearn`, that tries to learn DNF formulae with no guarantee to be unate. We will try to run the algorithm `LearnUDNF` on DNF formulae f which are not necessarily unate. The main idea is that if we try to flip twice the same bit j then we can conclude that x_j is neither monotone nor anti-monotone in f (that is, both x_j and $\bar{x}_j$ appear in the formula). To simplify notations, we will assume $j = 1$. To get rid of this variable we can express f as:

$$f(x_1, x_2, \ldots, x_n) \;=\; \bar{x}_1 \cdot f(0, x_2, \ldots, x_n) \;\vee\; x_1 \cdot f(1, x_2, \ldots, x_n).$$

Then we make a "divide" step: we recursively learn the two functions $f_0 = f(0, x_2, \ldots, x_n)$ and $f_1 = f(1, x_2, \ldots, x_n)$ (each has n variables but does not depend on x_1). More precisely, to learn f_0 and f_1 we run two copies of algorithm `LearnUDNF` in parallel; one copy for f_0 and one copy for f_1. Notice that each of the copies wants to ask queries regarding the corresponding function f_ξ ($\xi \in \{0, 1\}$) while we have available oracles for the original f only. To simulate a membership query $\texttt{MQ}(a_1, a_2, \ldots, a_n)$ for f_ξ, we ask $\texttt{MQ}(\xi, a_2, \ldots, a_n)$ (this gives us the value $f(\xi, a_2, \ldots, a_n)$ which is exactly what the algorithm needs). We proceed running both algorithms until each of them wants to ask an equivalence query $\texttt{EQ}(h_0)$ and $\texttt{EQ}(h_1)$, respectively. Then, algorithm `DCLearn` asks the equivalence query $\texttt{EQ}(h)$ where $h = \bar{x}_1 h_0 \vee x_1 h_1$ (these are the "conquer" steps of the algorithm; it combines information from the sub-problems to get information for

the problem itself). Either the answer to this query is YES, meaning that $h \equiv f$ or we get a counterexample b. Now, depending on the value of the first bit of b we find to which of the two hypotheses h_0 or h_1 the assignment b is a counterexample. Specifically, if $b_1 = 1$ then $f_1(b) = f(b) \neq h(b) = h_1(b)$ and therefore b is a counterexample for h_1. Similarly, if $b_1 = 0$ then b is a counterexample for h_0. Note that f_0 and f_1 themselves are learned recursively.

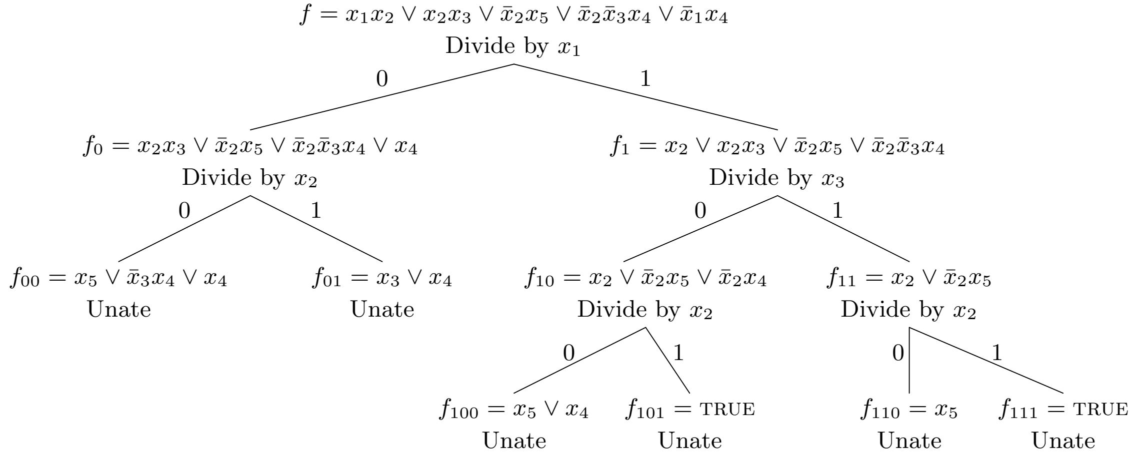

Figure 2: Divide and Conquer example

The execution of algorithm `DCLearn` can be described in a convenient way by a tree, as is done in Figure 2. The nodes of the tree are marked by the functions that the algorithm is trying to learn in each phase; in particular, the root of this tree is marked by the target function f. Each internal node is also marked by a variable name, x_j, according to which a "divide" step is taken. From each such node there are two outgoing edges labeled 0 and 1 that lead to nodes that describe the functions resulted by setting the value of x_j to 0 and 1 respectively. The leaves of the tree are unate functions to which no more "divide" steps are applied. The complexity of the algorithm depends on the number of "divide" steps that the algorithm performs; that is, the number of times that a variable x_j as above is found which results in two recursive calls. In the tree terminology, the complexity corresponds to the size of the tree.

Now, let us concentrate on the performance of algorithm `DCLearn` when it works on the class $O(\log n)$-TERM-DNF. We will show that in this case the algorithm learns in polynomial time. This was shown in Section **??** using a completely different algorithm. Let $f \in k$-TERM-DNF. Suppose that algorithm `DCLearn` finds a variable x_1 which is neither monotone nor anti-monotone. Then x_1 appears in some term of f and $\bar{x}_1$ appears in another term. Therefore, $f_0 = f(0, x_2, \ldots, x_n)$ and $f_1 = f(1, x_2, \ldots, x_n)$ both have at most $k-1$ terms. Therefore, if we look at the tree corresponding to the execution of algorithm `DCLearn` on such a function f then it immediately follows that its depth is at most k and hence its size is at most 2^k. This is polynomial for $k = O(\log n)$.

Exercise 1.0.1 *Analyze the complexity of the algorithm* `LearnUDNF`.

Exercise 1.0.2 *What is the complexity of algorithm* `DCLearn` *if at most* t *"divide" steps happen during the execution of the algorithm?*

Exercise 1.0.3 *Show that the output hypothesis of algorithm* `DCLearn` *on* $f \in O(\log n)$-TERM-DNF *is a polynomial size DNF.*