

# Online Learning

Nader H. Bshouty

Eyal Kushilevitz

## Abstract

Here we define the Online model and give some simple learnable classes.

## 1 On-Line Learning

The model of *on-line* learning is a very simple model which reflects situations such as weather forecasting. That is, learning is viewed as an ongoing process. In each step of this process the learning algorithm sees a certain situation (for example, various measures such as the temperature, the wind direction, etc.). The algorithm tries to make a prediction regarding this situation (for example, what will be the weather tomorrow). After making its prediction, and before the next step, it gets the actual value corresponding to the situation. In light of what it sees, the algorithm may update the way it makes its predictions. The goal is to minimize the number of mistakes the algorithm makes during this process.

We start by formally defining the model. Then, we present several learning algorithms and analyze their performance.

## 2 The Model

Let  $\mathcal{C}$  be a class of boolean functions  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ . Let  $f \in \mathcal{C}$  be the “target function”; that is, the function that we wish “to learn” (in a sense that will be formalized shortly). Let  $X = x^{(1)}, x^{(2)}, \dots, x^{(m)}$  be a (finite) sequence of examples, where each example  $x^{(i)}$  is an  $n$ -bit string (or alternatively an assignment to  $n$  variables).

An on-line learning algorithm  $\mathcal{A}$  works in *phases*. In its  $i$ -th phase the algorithm sees the example  $x^{(i)}$ ; it then produces a *prediction*  $p_i$ , which is intuitively what  $\mathcal{A}$  believes that the value of  $f(x^{(i)})$  is. This prediction is based on information that the algorithm collected in its memory during the first  $i - 1$  phases. The algorithm is said to make a *mistake* in its  $i$ -th phase if

$$p_i \neq f(x^{(i)}) .$$

We denote by  $\mathcal{MB}(\mathcal{A}, f, X)$  the number of mistakes that the algorithm  $\mathcal{A}$  does when getting  $X$  as its sequence of inputs and trying to learn the function  $f$  (where  $\mathcal{MB}$  stands for *Mistake Bound*). Define

$$\mathcal{MB}_{\mathcal{C}}(\mathcal{A}) = \max_{f \in \mathcal{C}, X} \mathcal{MB}(\mathcal{A}, f, X) .$$

That is,  $\mathcal{MB}_{\mathcal{C}}(\mathcal{A})$  is the worst case number of mistakes made by  $\mathcal{A}$  over all possible choices of a target function  $f \in \mathcal{C}$  and all possible choices of sequences. Finally, we define

$$\mathcal{MB}(\mathcal{C}) = \min_{\mathcal{A}} \mathcal{MB}_{\mathcal{C}}(\mathcal{A}) .$$

That is,  $\mathcal{MB}(\mathcal{C})$  is the worst case number of mistakes made by the *best* on-line learning algorithm for  $\mathcal{C}$ .

Note that we are concerned with a *worst case* measure. That is, the number of mistakes made in the worst possible situation. It is sometimes convenient in such a case to think of the target function  $f$  and the sequence  $X$  as if they were chosen by an *adversary* whose aim is to cause our learning algorithm to perform as bad as possible.

When designing an on-line learning algorithm  $\mathcal{A}$  there are several complexity measures that we try to optimize. First, the mistake-bound,  $\mathcal{MB}_{\mathcal{C}}(\mathcal{A})$  as defined above. That is, we try that our algorithm will make as few mistakes as possible in its predictions. Second, we try to make our algorithm efficient in terms of *time* (the amount of computation  $\mathcal{A}$  has to make when receiving the next example  $x^{(i)}$  until it produces its prediction  $p_i$ ; and the amount of time it takes  $\mathcal{A}$  to update its memory after seeing the actual value  $f(x^{(i)})$ ) and in terms of *space* (the amount of memory the algorithm needs to maintain in order to make its predictions). In the next section we present several very simple algorithms and see how to analyze their performance in view of the above complexity measures.

### 3 Basic (Non-efficient) Algorithms

Given a class of functions  $\mathcal{C}$  that we wish to learn, there are several natural strategies that can be applied to learn this class. We now present some of this strategies; all of them have some common properties: they are all *simple*; they are *generic* in the sense that they can be applied to any class  $\mathcal{C}$ ; and they are *not* efficient in at least one of the important measures.

The first of these is the **lookup** algorithm, described in Figure 1. The algorithm just keeps in memory all the examples in  $\{0, 1\}^n$  it has already seen; upon receiving a new example if it already saw this example before then the algorithm has this example in its memory and so it knows the correct answer; otherwise, it arbitrarily predicts 0.

Algorithm lookup:  
**Phase  $i$  (input  $x^{(i)}$ ):**  
 If  $x^{(i)}$  is stored in the memory compute  $p_i = f(x^{(i)})$ .  
 Otherwise, predict  $p_i = 0$  and after receiving the actual value,  $f(x^{(i)})$ , store  $x^{(i)}$  and  $f(x^{(i)})$  in the memory.

Figure 1: Algorithm lookup

The analysis of this algorithm is also simple. Let  $f$  be any target function. The algorithm

`lookup` makes at most  $2^n$  mistake since on every  $n$ -bit string it may err only once (in fact, it may err only if  $f(x^{(i)}) = 1$ ). The computation is simple since all it does is storing information and looking it up. The only disadvantage of the algorithm `lookup` is that obviously it needs a memory of size  $2^n$  to store the value of  $f$  on each  $n$ -bit string. In a sense this algorithm does not really “learn”; it cannot do anything “smart” on examples it has never seen.

The next algorithm is called `consistent`. This algorithm is usually better in terms of memory but it may still make many mistakes and its running time is worse than the running time of the `lookup` algorithm. For a given class  $\mathcal{C}$  we order the functions of the class in some (arbitrary) order. That is

$$\mathcal{C} = \{f_1, f_2, \dots, f_{|\mathcal{C}|}\}.$$

The idea behind the algorithm is to use the functions in  $\mathcal{C}$  as hypotheses according to their order. Whenever the algorithm makes a mistake it looks for the next function (according to the fixed order) that is consistent with the value of  $f(x^{(i)})$ . The algorithm is described in Figure 2.

**Algorithm consistent:**  
**Initialize:**  $\ell = 1$   
**Phase  $i$  (input  $x^{(i)}$ ):**  
 predict  $p_i = f_\ell(x^{(i)})$ . If the prediction is wrong (i.e., the actual value,  $f(x^{(i)})$  is different than  $p_i$ ) then update  $\ell$  to be the first index  $\ell'$  larger than the current  $\ell$  such that  $f_{\ell'}(x^{(i)}) = f(x^{(i)})$ .

Figure 2: Algorithm `consistent`

The number of mistakes made by the algorithm `consistent`, that is  $\mathcal{MB}(\text{consistent})$ , is at most  $|\mathcal{C}|$  (which is typically larger than  $2^n$ , the mistake bound for algorithm `lookup`). This is because each mistake increases the value of  $\ell$  by at least one and because the target function  $f$  is in  $\mathcal{C}$  (i.e.,  $f = f_k$  for some  $k$ ). The memory used by `consistent` is at most  $\log |\mathcal{C}|$  bits since the only information stored by the algorithm is the index  $\ell$ . Note that every class  $\mathcal{C}$  of boolean functions on the domain  $\{0, 1\}^n$  is of size at most  $2^{2^n}$  and so the memory used by `consistent` is never more than  $2^n$ . However, typical classes of interest are much smaller and therefore so is the memory that the algorithm uses.

The next algorithm that we present in this section is called `halving` and it tries to avoid searching for the target function in  $\mathcal{C}$  one by one, as is done by algorithm `consistent`. Instead algorithm `halving` tries to “split” the set of consistent functions in each mistake it makes. This is described in Figure 3.

To analyze the number of mistakes made by algorithm `halving` we make the following simple observations. Obviously,  $|\mathcal{C}_i| \geq 1$  since at least the target function  $f$  is in  $\mathcal{C}_i$ . Moreover,  $\mathcal{C}_{i+1} \subseteq \mathcal{C}_i$  since any function which is not in  $\mathcal{C}_i$  cannot be in  $\mathcal{C}_{i+1}$ . Finally, note that any mistake made by algorithm `halving` means that the majority of functions in  $\mathcal{C}_i$  are not consistent with

Algorithm halving:

**Phase  $i$  (input  $x^{(i)}$ ):**

Let  $\mathcal{C}_i$  be the set of all functions in  $\mathcal{C}$  which are consistent with all the examples seen so far:

$$(x^{(1)}, f(x^{(1)})), \dots, (x^{(i-1)}, f(x^{(i-1)})).$$

Predict  $p_i$  which is the majority, over all functions  $g \in \mathcal{C}_i$ , of the value  $g(x^{(i)})$ .

After receiving the actual value,  $f(x^{(i)})$  store  $x^{(i)}$  and  $f(x^{(i)})$  in the memory.

Figure 3: Algorithm halving

the value of  $f(x^{(i)})$ . Therefore,

$$|\mathcal{C}_{i+1}| \leq \frac{|\mathcal{C}_i|}{2}.$$

This implies that  $\mathcal{MB}(\text{halving})$ , the number of mistakes made by **halving**, is at most  $\log_2 |\mathcal{C}|$ , which is much better than the previous algorithms. Note, however, that the algorithm is required to store the examples it saw in its memory (which may require  $2^n$  memory) and that it is not clear how to compute the majority value,  $p_i$ , efficiently. It may be, though, that for certain classes  $\mathcal{C}$  both the space complexity and the time complexity can be improved.

**Exercise 3.1** Let  $\mathcal{C}$  be the class of all functions  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  (note that there are  $2^{2^n}$  such functions). Prove that  $\mathcal{MB}(\mathcal{C}) \geq 2^n$ ; that is, show that for every (deterministic) on-line learning algorithm there exist a function  $f$  and a sequence  $X$  on which the algorithm makes at least  $2^n$  errors. Conclude that for certain concept classes algorithm **halving** is optimal.

## 4 Example: Learning Disjunctions

In this section we show that in certain cases one can avoid the disadvantages of **halving**. We will start by considering the class  $\text{MDISJ}$  of all functions which are monotone disjunctions (that is, functions of the form  $f = x_i \vee x_j \vee \dots \vee x_k$ , where the disjunction refers only to variables and no negated variable is allowed). The algorithm uses the *elimination principle*. It starts with a set of all possible candidates (variables that may appear in the disjunction) and eliminate those who are surely not there when getting examples. The algorithm **eliminate** is described in Figure 4.

It is easy to note that the *space* complexity of the algorithm is only  $O(n)$ . All it needs to store is the set of candidates  $L$  (in particular, it does *not* need to keep any previous example). The time complexity is also  $O(n)$  because given an example all the algorithm need to check is whether  $L$  contains a variable with the value 1 in  $x^{(i)}$  and if  $f(x^{(i)})$  is 0 to remove all such variables from  $L$ . It remained to prove the mistake bound for algorithm **eliminate**. Denote by  $S$  the set of variables that appear in the target function  $f$ . We make the following observation:

**Algorithm eliminate:**

**Initialize:**  $L = \{x_1, \dots, x_n\}$

**Phase  $i$  (input  $x^{(i)}$ ):**

Denote  $f_L = \bigvee_{z \in L} z$ . Predict  $p_i = f_L(x^{(i)})$ .

If the actual value  $f(x^{(i)})$  is 0 remove from  $L$  all the variables satisfied by the assignment  $x^{(i)}$ .

Figure 4: Algorithm `eliminate`

**claim 4.1** *In every phase of algorithm `eliminate`,  $S \subseteq L$ .*

**Proof:** The proof is by induction. It is clearly true when the algorithm starts since  $L$  is initialized to be the set of *all* variables and  $S$  is a subset of the variables. Then, whenever a variable  $x_j$  is removed from  $L$  it must be that this variable is also not in  $S$ . This is because if  $x_j$  was in  $S$  then  $f(x^{(i)})$  is 1 and we would not remove this variable from  $L$ . The claim follows.  $\square$

It follows from the claim above that whenever  $f(x) = 1$  then also  $f_L(x) = 1$  (because if  $f(x) = 1$  then there is a variable in  $S$  which is satisfied and this variable is also in  $L$ ). Using this observation we claim:

**claim 4.2**  $\mathcal{MB}(\text{eliminate}) \leq n$ .

**Proof:** By the above discussion, mistakes can be made by `eliminate` only on examples  $x^{(i)}$  for which  $f(x^{(i)}) = 0$ . Note that a mistake on such an example means that the algorithm predicts  $p_i = 1$  which implies that there exists (at least) one variable in  $L$  which is satisfied by  $x^{(i)}$ . This variable is removed from  $L$  (and of course will never be added again). Since at the beginning  $|L| = n$  and every mistake removes at least one variable from it, the claim follows.  $\square$

**Exercise 4.1** *Let DISJ be the class of all (not necessarily monotone) disjunctions. This exercise shows the learnability of this class in various ways.*

1. *Modify the algorithm `eliminate` to get an efficient learning algorithm for the class DISJ. Hint: initialize  $L = \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$ .*
2. *Use algorithm `eliminate` as a subroutine to get an efficient algorithm for the class DISJ. Hint: think of the function  $f$  on the  $n$  variables  $x_1, \dots, x_n$  as a function  $g$  on  $2n$  variables  $x_1, \dots, x_n, y_1, \dots, y_n$ , where  $y_i = \bar{x}_i$ . Observe that  $g$  is a monotone disjunction.*

**Exercise 4.2** *Give an algorithm for learning the class MCONJ of all functions that are monotone conjunctions.*

## 5 A Composition Theorem

Composition theorems are an important tool for the design of learning algorithms. They allow using learning algorithms for simple classes of functions in order to construct algorithms for more complicated classes. We start with the following definition that shows how classes of functions can be combined.

**Definition 5.1** Let  $\mathcal{C}_1$  be a class of functions from  $\{0, 1\}^m$  to  $\{0, 1\}$  and let  $\mathcal{C}_2 = \{g_1, \dots, g_m\}$  be a class of  $m$  functions from  $\{0, 1\}^n$  to  $\{0, 1\}$ . Define the composition of  $\mathcal{C}_1$  on  $\mathcal{C}_2$  as:

$$\mathcal{C}_1(\mathcal{C}_2) \triangleq \{f : \{0, 1\}^n \rightarrow \{0, 1\} \mid f(x) = h(g_1(x), g_2(x), \dots, g_m(x)) \text{ for some } h \in \mathcal{C}_1\}.$$

For example, suppose that  $\mathcal{C}_1 = \text{MDISJ}_{2n}$  (the class of monotone disjunctions on  $2n$  variables) and that  $\mathcal{C}_2 = \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$  (the class of functions represented by a single literal), then their composition  $\mathcal{C}_1(\mathcal{C}_2)$  is just the class  $\text{DISJ}$  of (monotone and non-monotone) disjunctions. The intuition here is that we want to learn the function  $f$  by trying to learn the function  $h$ . Note that  $h$  is a function of  $m$  variables (and typically  $m$  will be larger than  $n$ ) but, by the way that  $f$  is defined, not all the  $2^m$  assignments to  $h$  are possible but only up to  $2^n$  assignments (corresponding to the  $2^n$  choices of  $x$ ).

**Theorem 5.1** Let  $\mathcal{C}_1$  be a class that is efficiently learnable using  $\mathcal{MB}(\mathcal{C}_1) = \alpha(n)$  mistakes. Let  $\mathcal{C}_2 = \{g_1, \dots, g_m\}$  be a class with  $m$  efficiently computable functions. Then, the class  $\mathcal{C}_1(\mathcal{C}_2)$  is learnable with at most  $\alpha(m)$  mistakes and time polynomial in  $m$  and  $n$ .

**Proof:** Let  $\mathcal{A}$  be the learning algorithm for  $\mathcal{C}_1$ . We will use this algorithm as a procedure in the algorithm  $\mathcal{B}$  that we construct. Denote the target function by

$$f(x) = h(g_1(x), g_2(x), \dots, g_m(x)).$$

Algorithm  $\mathcal{B}$  learns the function  $f$  using the algorithm  $\mathcal{A}$ . Since  $|\mathcal{C}_2|$  is “small”, algorithm  $\mathcal{B}$  can compute all the  $m$  values  $g_1(x), \dots, g_m(x)$  so as to get an  $m$ -bit string  $y$  from each  $n$ -bit example  $x$ . The algorithm  $\mathcal{B}$  is described in Figure 5.

Algorithm  $\mathcal{B}$ :

**Initialize** algorithm  $\mathcal{A}$  (to inputs of size  $m$ )

**Phase  $i$  (input  $x^{(i)}$ ):**

Compute an  $m$ -bit string  $y^{(i)} = g_1(x^{(i)}), \dots, g_m(x^{(i)})$

Feed  $y^{(i)}$  to  $\mathcal{A}$  to get its  $i$ -th prediction  $p_i$ . Predict  $p_i$  and provide the value  $f(x^{(i)})$  to  $\mathcal{A}$  as the actual value of  $f(y^{(i)})$

Figure 5: Algorithm  $\mathcal{B}$  for the Proof of Theorem 5.1

To analyze the number of mistakes made by  $\mathcal{B}$  on a sequence  $X = x^{(1)}, x^{(2)}, \dots$ , consider the number of mistakes made by  $\mathcal{A}$  on the sequence  $Y = y^{(1)}, y^{(2)}, \dots$ . The assignments in

$Y$  are not arbitrary assignments in  $\{0, 1\}^m$  due to the way that they are produced. However, clearly the number of mistakes  $\mathcal{A}$  makes on such a sequence is not more than the corresponding mistake bound, that is  $\alpha(m)$ . As for the running time of  $\mathcal{B}$  – in each phase it evaluates the  $m$  functions in  $\mathcal{C}_2$  which are assumed to be efficiently computable and, in addition, it executes the efficient algorithm  $\mathcal{A}$ . All together the running time is polynomial in  $n$  and  $m$ .  $\square$

**Exercise 5.1** *Let  $k$  be a constant. This exercise shows the learnability of the class  $k$ -DNF in various ways.*

1. *Use the above theorem to prove that the class  $k$ -DNF is efficiently learnable.*
2. *Use the elimination principle presented in Section 4 to describe an explicit algorithm for the class  $k$ -DNF.*

**Exercise 5.2** *For a boolean function  $f$  denote its complement by  $\bar{f}$ . For a class  $\mathcal{C}$  denote by  $\bar{\mathcal{C}}$  the collection of all functions  $\bar{f}$  for  $f \in \mathcal{C}$ . Prove that if  $\mathcal{C}$  is efficiently learnable then so is  $\bar{\mathcal{C}}$ .*

**Exercise 5.3** *Let  $k$  be a constant. This exercise shows the learnability of the class  $k$ -CNF in various ways.*

1. *Use Exercises 5.1 and 5.2 to prove that the class  $k$ -CNF is efficiently learnable.*
2. *Use Exercises 4.2 and the above theorem to prove that the class  $k$ -CNF is efficiently learnable.*

**Exercise 5.4** *Prove that the class  $k$ -TERM-DNF is efficiently learnable.*

## 6 Example: XOR Functions

In this section we provide another simple example of a class of functions that can be efficiently learned. This is the class MXOR of all monotone exclusive-or function. That is, the class MXOR consists of all functions that can be represented by the sum, modulo 2, of some subset of the variables  $x_1, \dots, x_n$  (for example,  $f(x) = x_2 \oplus x_6 \oplus x_{11}$  is a function in this class).

The main observation here is that the  $x$ 's for which  $f(x) = 0$  form a linear space (over the field  $GF(2)$ ). Therefore the algorithm tries to collect vectors  $x$  which form a basis  $B$  for this linear space. The algorithm `learn-xor` is described in Figure 6.

**claim 6.1** *Denote by  $\text{span}(B)$  the set of all  $n$ -bit vectors which are spanned by the vectors in  $B$ , and denote by  $f^{-1}(0)$  the set of all  $n$ -bit vectors which are mapped by  $f$  to 0. Then, in every phase of the algorithm `learn-xor`,*

$$\text{span}(B) \subseteq f^{-1}(0).$$

**Algorithm learn-xor:**

**Initialize:**  $B = \emptyset$

**Phase  $i$  (input  $x^{(i)}$ ):**

If the vector  $x^{(i)}$  is spanned by the vectors in  $B$  then predict  $p_i = 0$ ; otherwise, predict  $p_i = 1$ .

If the actual value  $f(x^{(i)})$  is 0 but  $p_i = 1$  then add  $x^{(i)}$  to  $B$ .

Figure 6: Algorithm learn-xor

**Proof:** The claim holds upon initialization since  $B$  is initially empty. Then, every vector  $x^{(i)}$  added to  $B$  is in  $f^{-1}(0)$  and so  $\text{span}(B)$  remains a linear sub-space of  $f^{-1}(0)$ .  $\square$

**claim 6.2**  $\mathcal{MB}(\text{learn-xor}) \leq n$ .

**Proof:** By the previous claim, whenever the prediction of the algorithm is  $p_i = 0$  this prediction is correct. Therefore, mistakes can be made by learn-xor only on examples  $x^{(i)}$  for which  $f(x^{(i)}) = 0$ . Each such mistake, results in adding a new vector  $x^{(i)}$  to  $B$ . This new vector is not spanned by the previous vectors in  $B$  and therefore all vectors in  $B$  are linearly independent. The number of linearly independent vectors in  $f^{-1}(0) \subseteq \{0, 1\}^n$  is at most  $n$ . The claim follows.  $\square$

Note that all the computations made by algorithm learn-xor are simple linear algebraic computations (mostly checking whether  $x^{(i)}$  is spanned by a set of at most  $n$  vectors). The memory used by the algorithm is of size  $O(n^2)$  bits (for storing the basis  $B$ ).

**Exercise 6.1** Let XOR be the class of all functions that can be represented by the sum, modulo 2, of some subset of the literals  $x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n$  (for example,  $f(x) = \bar{x}_1 \oplus x_6 \oplus \bar{x}_8$ ). Give an efficient learning algorithm for the class XOR.

*Hint: use the composition theorem.*

## 7 Example: Learning Decision Lists

In this section we present an algorithm for learning the class of decision lists. In fact, we will present an algorithm for the simpler class of *monotone decision lists*, MDL, consisting of all the decision lists where nodes are labeled with variables only (the more general definition of decision lists allow literals in the nodes). Then, in the exercises this is extended in various ways using Theorem 5.1 (the composition theorem).

The first step before we describe the learning algorithm is to introduce the notion of *canonical decision lists*. In these lists every node is labeled by a disjunction of variables and, in addition, the labels of the leaves alternate between 0 and 1 (the first leaf is labeled 0). For example, this

is a canonic decision lists

$$(x_1 \vee x_4, 0), (x_2 \vee x_9, 1), (x_5 \vee x_{10} \vee x_{11}, 0), (x_8, 1), 0 \quad (1)$$

which is equivalent to following monotone decision list:

$$(x_1, 0), (x_4, 0), (x_2, 1), (x_9, 1), (x_5, 0), (x_{10}, 0), (x_{11}, 0), (x_8, 1), 0 \quad (2)$$

Observe that every canonic decision list can be transformed into a monotone decision list (with at most  $n$  nodes), and every monotone decision list can be transformed into a canonic decision list (with at most  $n + 1$  nodes). The algorithm `learn-dl`, presented in Figure 7, can be viewed as an extension of the algorithm `eliminate`. It learns a function  $f \in \text{MDL}$  by finding its canonical representation, and it uses the elimination principle for each of the nodes to find the disjunction corresponding to that node.

**Algorithm learn-dl:**

**Initialize:**  $L_0 = L_1 = \dots = L_n = \{x_1, \dots, x_n\}$

**Phase  $i$  (input  $x^{(i)}$ ):**

Consider the canonic decision list  $D_i$  as follows:

$$(f_{L_0}, 0), (f_{L_1}, 1), (f_{L_2}, 0), \dots, (f_{L_n}, n \bmod 2), (n + 1) \bmod 2,$$

where  $f_L$  denoted the disjunction  $\bigvee_{z \in L} z$ . Predict  $p_i$  as the value of the decision list  $D_i$  on  $x^{(i)}$ .

If the actual value  $f(x^{(i)})$  is different than  $p_i$  and  $j$  is the smallest index such that  $f_{L_j}(x^{(i)}) = 1$  then remove from  $L_j$  all the variables satisfied by  $x^{(i)}$ .

Figure 7: Algorithm `learn-dl`

The analysis of the algorithm `learn-dl` is also a generalization of the analysis of algorithm `eliminate`. Denote by  $S_i$  ( $0 \leq i \leq n$ ) the set of variables that appear in node  $i$  of the target canonic decision list  $f$ .

**claim 7.1** *In every phase of algorithm `learn-dl`,  $S_j \subseteq L_j$ , for every  $j$ .*

**Proof:** The proof is by induction. It is clearly true when the algorithm starts since each  $L_j$  is initialized to be the set of *all* variables and  $S_j$  is a subset of it. Now, consider a phase  $i$  in which a variable  $x_k$  is removed from  $L_j$ . By the algorithm, this implies that  $f_{L_j}$  is the first function in our list that gets the value 1 on  $x^{(i)}$ . In particular,

$$f_{L_1}(x^{(i)}) = \dots = f_{L_{j-1}}(x^{(i)}) = 0.$$

This, by the induction hypothesis, implies that in the target function we also have

$$f_{S_1}(x^{(i)}) = \dots = f_{S_{j-1}}(x^{(i)}) = 0.$$

Since a variable  $x_k$  is removed from  $L_j$  only when a mistake was made it also implies that  $f_{S_j}(x^{(i)}) = 0$  and since  $x_k$  is satisfied by  $x^{(i)}$  it also implies that  $x_k$  is *not* in  $S_j$ . Hence, after  $x_k$  is removed from  $L_j$ , still  $S_j \subseteq L_j$  and the claim follows.  $\square$

**claim 7.2**  $\mathcal{MB}(\text{learn-dl}) \leq n(n+1)$ .

**Proof:** We claim that at every phase in which the algorithm makes a mistake one of the sets  $L_j$  is reduced by at least one variable  $x_k$ . Since initially each of the  $n+1$  sets contains at most  $n$  variables then the claim follows. To prove this, note that if for every  $j$  we have  $f_{L_j}(x^{(i)}) = f_{S_j}(x^{(i)})$  then the target list and the list  $D_i$  compute the same value and no mistake can be made; this is because the two lists are canonical decision lists and so their leaves are also labeled in the same way. Hence, there exists a first index  $j$  such that  $f_{L_j}(x^{(i)}) \neq f_{S_j}(x^{(i)})$ . Then, by the previous claim it must be that  $f_{L_j}(x^{(i)}) = 1$  and  $f_{S_j}(x^{(i)}) = 0$ . In particular there must exist a variable  $x_k$  in  $L_j$  that is satisfied by the assignment  $x^{(i)}$ , as desired.  $\square$

**Exercise 7.1** *Prove that every canonic decision list can be transformed into a monotone decision list with at most  $n$  nodes, and every monotone decision list can be transformed into a canonic decision list with at most  $n+1$  nodes.*

**Exercise 7.2** *Let  $k$  be a constant. Prove that the class  $k$ -DL is efficiently learnable.*

*Hint: Use the composition theorem.*

**Exercise 7.3** *Prove that the class DT of (polynomial size) decision trees is learnable in time (and number of mistakes)  $n^{O(\log n)}$ .*

**Exercise 7.4** *Prove that the class DNF of (polynomial size) DNF formulae is learnable in sub-exponential time (and sub-exponential number of mistakes).*

## 8 The WINNOW Algorithm

Consider the algorithm *eliminate* of Section 4. This algorithm makes at most  $n$  mistakes. Moreover, it may make about  $n$  mistakes even if the number of variables in the disjunction,  $k$ , is very small. It is sometimes important to make the number of mistakes depending mainly on  $k$  and having only a weak dependency on  $n$ . The reason is that in many cases in practice, when it is hard to tell which variables may be indeed relevant to the target function, we get the value of many measurements which are suspected to be relevant but most of them are irrelevant. This creates situations in which the number of variables  $n$  is very large, but out of them only  $k \ll n$  are indeed relevant. In this section we present a learning algorithm for learning the class MDISJ who makes only  $O(k \log n)$  mistakes. That is, it has a linear dependency on the actual number of variables in the target,  $k$ , and only logarithmic dependency on the total number

of variables,  $n$ . (The time complexity and space complexity are still polynomial in  $n$ .) The algorithm `winnow` is presented in Figure 8. The idea here is that each variable is assigned a *weight*. Each variable which is eliminated its weight is set to 0, while the weight of variables which are suspected as appearing in the formula increases.

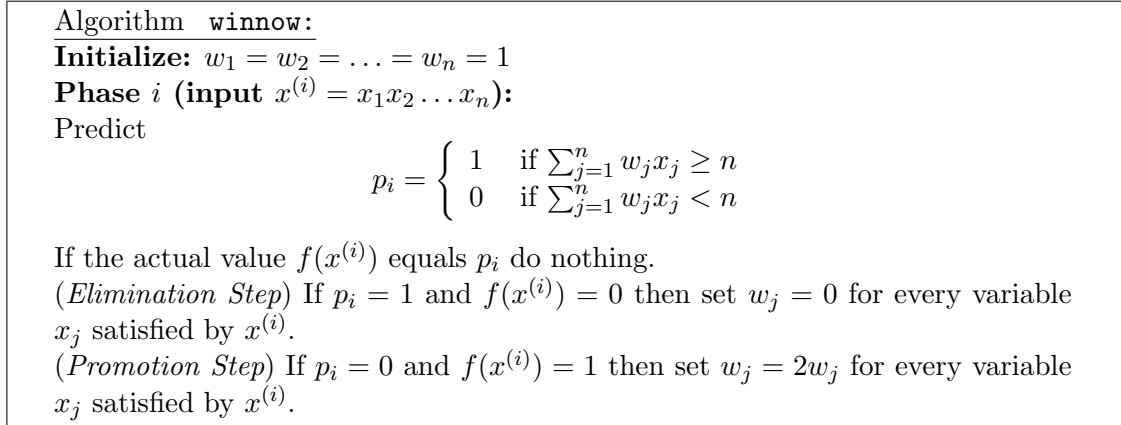


Figure 8: Algorithm `winnow`

Denote by  $E$  the number of elimination steps made by the algorithm and by  $P$  the number of promotion steps. We start by bounding the number  $P$ .

**claim 8.1**  $P \leq k \log n$ .

**Proof:** Consider the variables  $x_j$  that appear in the disjunction (there are at most  $k$  such variables) and the corresponding weights  $w_j$ . In each phase that we make a promotion mistake at least one of these variables is satisfied (since  $f(x^{(i)}) = 1$ ) and so its weight is doubled by 2. Since each  $w_j$  is initialized with the value 1 then, after  $\log n$  times that  $w_j$  is doubled, we will have  $w_j = n$ . From this point on, every example in which  $x_j$  is satisfied will be predicted as 1 (since  $\sum_{j=1}^n w_j x_j \geq n$ ) and there are no more mistakes on this type of examples. All together, we can make at most  $k \log n$  promotion steps, as claimed.  $\square$

Next, we claim that  $E$  is also not too large.

**claim 8.2**  $E \leq P + 1$ .

**Proof:** Consider the sum of weights  $W = \sum_{j=1}^n w_j$ . Upon initialization,  $W = n$ . Each elimination step reduces the value of  $W$  by at least  $n$ , since in such step we predicted  $p_i = 1$  which implies that  $\sum_{j:x_j=1} w_j \geq n$  and all these weights are set to 0. Each promotion step increases the value of  $W$  by less than  $n$ , since in such step we predicted  $p_i = 0$  which implies that  $\sum_{j:x_j=1} w_j < n$  and these are exactly the weights that we multiply by two. Finally, note that at any time every  $w_j \geq 0$  and so is  $W$ . Thus, we get that  $W$  satisfies

$$0 \leq W \leq n + (P \cdot n) - (E \cdot n) = (P - E + 1) \cdot n,$$

which implies that  $P - E + 1 \geq 0$ , as needed.  $\square$

**claim 8.3**  $\mathcal{MB}(\text{winnow}) = O(k \log n)$ .

**Proof:** The number of mistakes made by the algorithm is  $P + E$  which by the above two claims gives the result.  $\square$

Note that the time complexity of `winnow` is polynomial and so is the space complexity (it only keeps  $n$  weights).

**Exercise 8.1** Construct a function  $f \in \text{MDISJ}$  with  $k = O(1)$  literals, and a sequence  $X$  on which the algorithm `eliminate` makes almost  $n$  mistakes.

**Exercise 8.2** Show an efficient algorithm for the class `DISJ` that makes  $O(k \log n)$  mistakes. *Hint: use the composition theorem.*