Machine Learning, 9, 147–164 (1992)

Learning Conjunctions of Horn Clauses

DANA ANGLUIN ANGLUIN@CS.YALE.EDU
Computer Science, Yale University, New Haven, CT 06520

MICHAEL FRAZIER MFRAZIER@CS.UIUC.EDU
Computer Science, University of Illinois, Urbana, Illinois 61801

LEONARD PITT PITT@CS.UIUC.EDU
Computer Science, University of Illinois, Urbana, Illinois 61801

Abstract. An algorithm is presented for learning the class of Boolean formulas that are expressible as conjunctions of Horn clauses. (A Horn clause is a disjunction of literals, all but at most one of which is a negated variable.) The algorithm uses equivalence queries and membership queries to produce a formula that is logically equivalent to the unknown formula to be learned. The amount of time used by the algorithm is polynomial in the number of variables and the number of clauses in the unknown formula.

Keywords. Propositional Horn sentences, equivalence queries, membership queries, exact identification, polynomial time learning

1. The problem

Valiant (1984) introduced the distribution-free or "PAC" criterion for concept learning and focused attention on the question of what classes of Boolean formulas can be learned in polynomial time with respect to this criterion. He gave a polynomial-time algorithm to learn k-CNF or k-DNF formulas and posed the question, which still remains open, of whether general CNF and DNF formulas are learnable in polynomial time.

Positive results in this line of research include improved algorithms for k-CNF and k-DNF formulas (Haussler, 1988; Littlestone, 1988), and algorithms for larger classes of formulas, including linearly separable formulas (Littlestone, 1988), internal disjunctive formulas (Haussler, 1988), decision lists (Rivest, 1987), and rank k decision trees (Ehrenfeucht & Haussler, 1988). Kearns and Valiant (1989) give cryptographic evidence to the effect that the class of all Boolean formulas is not polynomial-time learnable.

Prediction-preserving reductions (Pitt & Warmuth, 1990) provide a powerful method for relating the difficulties of distinct learning problems. For example, one of the reductions given by Kearns, Li, Pitt and Valiant (1987) shows that PAC-learning monotone CNF and DNF formulas is as hard, modulo a polynomial time transformation, as PAC-learning general CNF and DNF formulas.

One approach to finding efficient learning algorithms has been to relax the problem by making membership queries available to the learning algorithm (Angluin, 1988). That is, the learning algorithm is allowed to query domain elements of its choice and receive their correct classifications with respect to the target concept.

Membership queries can make a difference—for example, monotone DNF and CNF formulas are PAC-learnable in polynomial time if membership queries are available (Valiant,

1984; Angluin, 1988), whereas by the remarks above their status is open without membership queries. Another example is provided by read-once formulas (Boolean formulas in which each variable occurs at most once). These are PAC-learnable in polynomial time if membership queries are available (Angluin, Hellerstein & Karpinski, 1989). However, by another reduction of Kearns, Li, Pitt and Valiant (1987), the PAC-learnability of read-once formulas is equivalent to that of general Boolean formulas in the absence of membership queries, and therefore as hard as certain apparently hard cryptographic problems, by the results of Kearns and Valiant (1989).

What are the limits of this approach? How much are membership queries likely to help with learning general Boolean formulas, or general CNF and DNF formulas? Angluin and Kharitonov (1991) give cryptographic evidence that the answer in both cases is: "not much." For general Boolean formulas there is cryptographic evidence of the same sort as given by Kearns and Valiant that they are not PAC-learnable in polynomial time even if membership queries are available. For general CNF and DNF formulas the situation is more complicated, but, in effect, there is cryptographic evidence that either general CNF formulas will be PAC-learnable in polynomial time without membership queries, or they won't be PAC-learnable in polynomial time even with membership queries—that is, the membership queries "won't help" in the case of general CNF and DNF formulas.

In this paper we consider the class of propositional Horn sentences, that is, CNF formulas in which each clause contains an arbitrary number of literals, at most one of which is positive. This class lies between general CNF and monotone CNF, being "nearly monotone." For the class of propositional Horn sentences there are polynomial-time algorithms to decide the satisfiability of a formula and the logical equivalence of two formulas (Jones & Laaser, 1977), in contrast to the situation for general CNF formulas.

A Horn clause such as $(\neg a \vee \neg b \vee \neg c \vee d)$ is logically equivalent to an implication $(a \wedge b \wedge c \Rightarrow d)$; thus, a propositional Horn sentence can be viewed as a conjunction of such implications. Of course, conjunctions of universally quantified Horn clauses in the predictate logic are familiar in the guise of Prolog programs, so one can also view this class as a "toy" model of Prolog. Angluin (1988) provides further discussion of propositional Horn sentences.

The main result of this paper implies that the class of propositional Horn sentences is polynomial-time learnable in the PAC model provided membership queries are available. Thus, in this case, membership queries do help.

Our learning algorithms are presented in the paradigm of exact identification with equivalence and membership queries, a more demanding setting than either PAC learning with membership queries or mistake-bounded prediction with membership queries. That is, our polynomial-time algorithm for exact identification of propositional Horn sentences using equivalence and membership queries can be transformed in a straightforward way into either (1) a polynomial-time PAC learning algorithm using membership queries (Angluin, 1988) or (2) a polynomial-time prediction algorithm with a polynomial mistake bound (Littlestone, 1988) for the same class of formulas.

Our main result may be stated formally as follows. For all natural numbers m and n our algorithm, HORN1, exactly identifies all the Horn sentences H_* with m clauses and n variables using equivalence and membership queries and runs in time[1] $\tilde{O}(m^2n^2)$, making $O(mn)$ equivalence queries and $O(m^2n)$ membership queries.

It is interesting to note that neither type of query may be eliminated. That is, there is no polynomial-time algorithm that exactly identifies all the Horn sentences using only membership queries (Angluin, 1988) or only equivalence queries (Angluin, 1990).

Despite the "almost monotone" nature of propositional Horn sentences, the algorithm to learn them using membership and equivalence queries is quite different from the algorithm to learn monotone CNF and DNF formulas using membership and equivalence queries (Valiant, 1984; Angluin, 1988). A natural question to ask is whether the results can be extended by allowing "more" non-monotonicity in the clauses. If we define the k-quasi-Horn formulas to be conjunctions of clauses each of which has at most k positive literals, then the Horn sentences are also 1-quasi-Horn formulas. However, a reduction that preserves prediction with membership queries shows that learning even 2-quasi-Horn formulas with membership queries is as hard as learning general CNF formulas with membership queries. Therefore, if learning general CNF formulas with membership queries is indeed intractable, then Horn sentences are on the boundary of what can be learned in polynomial time with membership queries.

The research presented here also improves the results of Angluin (1988), where the class of Horn sentences is shown to be learnable by an algorithm that uses equivalence queries that return Horn clauses as counterexamples and "derivation queries"—a type of query that is significantly more powerful than a membership query.

The remainder of the paper is organized as follows. Section 2 gives basic definitions, notation, and lemmas that will be used throughout. In Section 3 we describe the first version of our algorithm, HORN, and give an example run. Section 4 provides a correctness proof and an analysis of the time and query complexity of this algorithm. In Section 5 we present a modified version of the algorithm, HORN1, which has improved time and query bounds. We conclude in Section 6 by discussing some related and open problems.

2. Preliminaries

2.1. Horn sentences

Definition 1. *The logical constant "true" is represented by* **T** *and the logical constant "false" is represented by* **F**.

Definition 2. *Let* $V = \{v_1, \ldots, v_n\}$ *be a set of Boolean variables. A* literal *is either a variable* v_i *or its negation* $\neg v_i$. *A* clause *over variable set* V *is a disjunction of literals. A* Horn clause *is a clause in which at most one literal is unnegated. A* Horn sentence *is a conjunction of Horn clauses.*

The class of Horn sentences over variable set V is a proper subclass of the class of conjunctive normal form (CNF) formulas over V.

Our presentation is simplified if we recall that a Horn clause has an equivalent representation as an implication. For example, the clause $(\neg a \vee \neg b \vee \neg c \vee d)$ is logically equivalent to the implication $(a \wedge b \wedge c \Rightarrow d)$. Similarly, the "degenerate" clauses $(\neg a \vee \neg b \vee \neg c)$ and (a) have implicative forms $(a \wedge b \wedge c \Rightarrow \mathbf{F})$ and $(\mathbf{T} \Rightarrow a)$ respectively. This is captured formally by the following definition.

Definition 3. *Let C be a Horn clause. Then antecedent(C) is the set of variables that occur negated in C. By convention, the constant* **T** *is also in antecedent(C), thus a clause with no negated variables has a nonempty antecedent.*[2] *If C contains an unnegated variable z, then consequent(C) is just z. Otherwise, C contains only negated variables and consequent(C) is* **F**.

2.2. Positive and negative examples

Definition 4. *Let H be any Horn sentence over V. An* example *is any assignment* $x : V \to \{\mathbf{T}, \mathbf{F}\}$. *A* positive *(respectively,* negative*) example for H is an assignment x such that H evaluates to* **T** *(respectively,* **F***) when each variable v in H is replaced by x(v).*

Definition 5. *Let x be an example; then true(x) is the set consisting of the constant* **T** *and the variables assigned the value* **T** *by x. Similarly, false(x) is the set consisting of the constant* **F** *and the variables assigned the value* **F** *by x.*

We now describe the relationships that may exist between an example and a Horn clause.

Definition 6. *An example x is said to* cover *a Horn clause C if* $antecedent(C) \subseteq true(x)$. *We say that x* does not cover *C if* $antecedent(C) \not\subseteq true(x)$. *The example x is said to* violate *the Horn clause C if x covers C and* $consequent(C) \in false(x)$.

Notice that if x violates C then x must cover C, but that the converse does not necessarily hold. It will be more convenient throughout the rest of the paper to consider a Horn sentence as a *set* of Horn clauses, representing the conjunction of the clauses. Our first observation is trivial, but it is helpful to state it formally.

Proposition 1. *If x is a negative example for the Horn sentence H, then x violates some clause of H.*

We next define the operation "$\cap$" denoting the intersection of a pair of examples.

Definition 7. *Let x and s be two examples, then* $x \cap s$ *is defined to be the example z such that* $true(z) = true(x) \cap true(s)$.

Note that this implies that $false(x \cap s)$ is $false(x) \cup false(s)$.

Lemma 1. *Let x and s be examples. If x violates C and s covers C, then* $x \cap s$ *violates C.*

Proof: If s covers C then $antecedent(C) \subseteq true(s)$. Also, if x violates C, then $antecedent(C) \subseteq true(x)$ and $consequent(C) \in false(x)$. Thus $antecedent(C) \subseteq true(s) \cap true(x) = true(s \cap x)$ and $consequent(C) \in false(x) \subseteq false(s) \cup false(x) = false(s \cap x)$. Thus, $s \cap x$ violates C. □

Corollary 1. *Let x and s be examples. If x violates C and s violates C, then $x \cap s$ violates C.*

Proof: Apply Lemma 1 after noting that if s violates C then it also covers C. □

Lemma 2. *If x does not cover C, then for any example s, $x \cap s$ does not violate C.*

Proof: If $antecedent(C) \not\subseteq true(x)$ then $antecedent(C) \not\subseteq true(x) \cap true(s) = true(x \cap s)$. Thus $x \cap s$ does not violate C. □

Lemma 3. *If $x \cap s$ violates C, then at least one of x and s violates C.*

Proof: If $x \cap s$ violates C then $consequent(C) \in false(x \cap s) = false(x) \cup false(s)$. Therefore, $consequent(C)$ is an element of at least one of $false(x)$ and $false(s)$. But since $x \cap s$ violates C, we also have $antecedent(C) \subseteq true(x)$ and $antecedent(C) \subseteq true(s)$. Thus at least one of x and s must violate C. □

2.3. The learning protocol

Let H_* denote the *target* Horn sentence to be learned. The goal of the learning algorithm is *exact identification*, that is, to succeed the algorithm must halt and output a Horn sentence that is logically equivalent to H_*. The learning algorithm may gather information about H_* using two types of queries, membership queries and equivalence queries.

A *membership query* is any assignment x to the variables, and the answer to the membership query is "yes" if x satisfies the target formula H_*, and "no" otherwise. Thus, a membership query is a request for a classification of a specific example with respect to the target concept.

In an *equivalence query* the learning algorithm may propose as a hypothesis any Horn sentence H. If H is logically equivalent to H_* then the answer to the query is "yes" and the learning algorithm has succeeded in the inference task, and halts with output H. Otherwise, the answer to the equivalence query is "no," and the learning algorithm receives a *counterexample*—an assignment $x : V \to \{\mathbf{T}, \mathbf{F}\}$ that satisfies H_* but does not satisfy H (a *positive* counterexample), or vice-versa (a *negative* counterexample). Note that the choice of counterexamples is arbitrary—the algorithm must work as advertised for the worst-case selection of counterexamples.

Because the problem of determining whether two Horn sentences are equivalent (and producing a counterexample if they are not) is solvable in polynomial time (Jones & Laaser, 1977), answers to both types of queries can be found in polynomial time given the target formula H_* and the input to the query (either an assignment x or a Horn sentence H.)

3. The algorithm HORN

The ideas behind our algorithm may be understood by considering the problems that arise with more straightforward approaches. After this motivation, we describe the algorithm HORN and give an example run. The correctness of HORN is demonstrated in the next section. In Section 5 we present a more efficient version of our algorithm, HORN1.

3.1. *Two unsuccessful approaches*

Let H_* be the target Horn sentence with respect to which equivalence and membership queries are answered. Every negative counterexample x violates some clause C of H_*. Given x, we would like to add the clause C to our current hypothesis, but we cannot exactly determine C from x alone. We know however that $antecedent(C) \subseteq true(x)$, and $consequent(C) \in false(x)$. Thus one approach would be to add to our current hypothesis H all elements of the set

$$clauses(x) = \left\{ \left[\bigwedge_{v \in true(x)} v \right] \Rightarrow z : z \in false(x) \right\}$$

whenever a new negative counterexample x is obtained. Each clause in this set is a possible explanation of why x is a negative example, since each one is falsified by x. However, there are two kinds of problems that arise—the new clauses may be incorrect (that is, not implied by H_*) or they may be correct, but too weak.

The problem of incorrect clauses is not a serious one because any clause that is not logically implied by the target formula H_* will eventually be discovered when a *positive* counterexample is produced that does not satisfy the clause. At this point, at least one incorrect clause will be eliminated.[3]

The problem of correct but weak clauses is more serious. To see that there is at least one correct clause, let C' be the clause from $clauses(x)$ with the same consequent as C; C' is subsumed by C, and thus is logically implied by H_*. However, the antecedent set of C' may be much larger than the antecedent set of C, with the result that there are numerous negative counterexamples that violate C but satisfy C'.

A simple scenario shows that an adversarial choice of counterexamples can force this approach to add exponentially many correct but weak clauses. Let n be an even number, let the variable set be $V = \{a, b_1, b_2, \ldots, b_n\}$, and suppose H_* is just a single clause, C, which is

$$a \Rightarrow \mathbf{F}.$$

Any example in which a is set to **T** is a negative example. In particular, the example with a set to **T** and also each variable b_i such that i is even set to **T** is a negative example. Among the clauses generated from this example is the correct but weak clause

$$(a \wedge b_2 \wedge b_4 \wedge \ldots \wedge b_n) \Rightarrow \mathbf{F}.$$

Suppose we next see a negative example which is identical to the previous negative example except that b_1 is **T** instead of b_2. This example does not violate any clause that we previously generated, so we must exclude this negative example by generating clauses for it. Among these clauses is the correct but weak clause

$$(a \wedge b_1 \wedge b_4 \wedge b_6 \wedge \ldots \wedge b_n) \Rightarrow \mathbf{F}.$$

The difficulty is now clear: there are exponentially many negative examples with a set to **T** and exactly half of the variables $\{b_i\}$ set to **T**—we are forced to exclude each one with its own clause.

In this scenario, because the antecedents of the clauses in *clauses*(x) are so much larger than the antecedent set of C, the negative examples that fail to satisfy the correct added clause are only a small fraction of those that fail to satisfy C. Thus the correct clause added to H is a very weak "approximation" of C. Consequently, very many such approximations to the target clause C are generated by the examples.

To counter this problem, a second approach might be to find smaller antecedents by using membership queries to set more of the variables to **F** in the negative counterexamples that we are given; this is the approach of the monotone DNF algorithm of Valiant (1984) and Angluin (1988). Given a negative counterexample x, we set some variable which is currently **T** in x to **F** and ask whether the result satisfies H_*. If not, then the result still violates some clause of H_* and we leave the variable set to **F**, otherwise we set the variable back to **T**. We repeat this process until we can set no more variables to **F**. This process can be done quickly and we are left with a minimal negative example. However, a second scenario will disclose a flaw in this approach.

Suppose the variable set is $V = \{a, b, c, d\}$ and H_* is

$$(b \wedge c \Rightarrow d) \wedge (b \Rightarrow a).$$

Further suppose that we minimize negative examples by trying to set the variables to **F** in alphabetical order. Let **TTTF** (that is, the variables a, b, and c set to **T** and the variable d set to **F**) be the first negative counterexample. We minimize this to **FTFF** and add *clauses*(**FTFF**) to our hypothesis, so H becomes

$$(b \Rightarrow a) \wedge (b \Rightarrow c) \wedge (b \Rightarrow d) \wedge (b \Rightarrow \mathbf{F}).$$

Next we see the positive counterexample **TTFF** which reduces H to

$$(b \Rightarrow a).$$

Thus we have indeed found a clause of H_*. Next we see the negative counterexample **TTTF**. But this is the same example that we saw at the beginning, and so our algorithm will never terminate; it is forced to find the same clause repeatedly.

The difficulty lies in the fact that even though we were given an example that violated the first clause of H_*, our minimization produced an example that violated the second clause of H_*; and this fact led to non-termination. (One might object that it is foolhardy to decide *a priori* the order in which we will try to set the variables of our negative examples to **F**; rather we should dynamically decide the order in which to minimize a new negative counterexample. However, it appears to be a difficult problem to design a polynomial-time algorithm that is guaranteed to minimize a negative example and simultaneously avoid rediscovering any of the previously found correct clauses even if it is known which clauses in the current hypothesis are correct.)

3.2. *Description of HORN*

The first scenario above shows that we must reduce the number of variables set to **T** in the negative counterexamples we are given, and the second scenario rules out the obvious greedy approach. A data-driven approach solves the dilemma. A new negative example is used to attempt to "refine" previously obtained negative examples by intersection. Each such intersection, if it contains fewer true variables than the previously obtained negative example, is then tested to see whether it is negative (using a membership query). If so, it is a candidate to refine the previously obtained negative example.[4]

The algorithm maintains a sequence S of negative examples. Each new negative counterexample either is used to refine one element of S, or is added to the end of S. In order to learn all of the clauses of H_*, we would like the clauses induced by the (negative) examples in S to approximate *distinct* clauses of H_*. This will happen if the examples in S violate distinct clauses of H_*. Overzealous refinement may result in several examples in S violating the same clause of H_*. To avoid this, whenever a new negative counterexample could be used to refine several examples in the sequence S, only the first among these is refined.

Collecting all of these ideas, the learning algorithm HORN (Figure 1) can be described intuitively as follows. The sequence S of negative examples is used to generate new hypotheses. Each negative example x in the sequence can be explained by $O(n)$ different Horn clauses (*clauses*(x)), and each of these possible explanations is placed in the hypothesis.

```
1  Set S to be the empty sequence /*  s_i denotes the i-th element of S */
2  Set H to be the empty hypothesis
3  UNTIL equivalent(H) returns "yes" DO
4     BEGIN /* main loop */
5     Let x be the counterexample returned by the equivalence query
6     IF x violates at least one clause of H
7        THEN /* x is a positive example */
8           remove from H every clause that x violates
9        ELSE /* x is a negative example */
10          BEGIN
11          FOR each s_i in S such that true(s_i ∩ x) is properly contained in true(s_i)
12             BEGIN
13             query member(s_i ∩ x)
14             END
15          IF any of these queries is answered "no"
16             THEN let i = min{j : member(s_j ∩ x) = "no"}
17                refine s_i by replacing s_i with s_i ∩ x
18             ELSE add x as the last element in the sequence S
19          ENDIF
20          Set H to be ⋀_{s∈S} clauses(s),  where clauses(s) = {(⋀_{v∈true(s)} v) ⇒ z : z ∈ false(s)}
21          END
22    ENDIF
23    END /* main loop */
24 Return H
```

Figure 1. The algorithm HORN.

Any clause in the hypothesis which is not logically implied by the target will be exposed eventually by a positive counterexample. When a positive counterexample appears, the algorithm removes any clause that this example violates from the hypothesis.

On the other hand, the hypothesis may also contain some correct clauses which are too weak. Such a clause may be exposed eventually by a negative counterexample. When a negative counterexample occurs, the algorithm refines the first element of S it can (using an intersection and a membership query) or it appends the new negative example to the end of S. In either case, after modifying S the algorithm generates a new hypothesis from S. This process produces in polynomial time a hypothesis which is logically equivalent to the target.

3.3. An example run of HORN

A simulated run of HORN follows. Suppose the variable set is $V = \{a, b, c, d\}$ and H_* is

$$H_* : (a \wedge c \Rightarrow d) \wedge (a \wedge b \Rightarrow c).$$

Initially we set S to be the empty sequence and H to be the null hypothesis (true on all examples):

$$S : [\,],$$

$$H : \emptyset.$$

Suppose the first counterexample to our equivalence query for H is the negative example **TTTF**. There are no elements of S that we can attempt to refine with this negative example, so we simply append it to the end of the sequence. Since S has changed we generate a new hypothesis H by conjoining all of the clauses from *clauses*(s) for all $s \in S$. Thus,

$$S : [\mathbf{TTTF}],$$

$$H : (a \wedge b \wedge c \Rightarrow d) \wedge (a \wedge b \wedge c \Rightarrow \mathbf{F}).$$

Suppose the next counterexample to our equivalence query for H is the positive example **TTTT**. This eliminates an incorrect clause from H but does not change S, so we do not generate a new H from S. Thus we now have

$$S : [\mathbf{TTTF}],$$

$$H : (a \wedge b \wedge c \Rightarrow d).$$

For clarity, we will assume for the remainder of this simulated run that we are able to discard immediately any incorrect clauses by positive counterexamples returned by equivalence queries for H, and so we will only show the effect of receiving a (necessarily) negative

counterexample. Suppose the next negative counterexample is **TTFT**. We intersect this with the (first) element of S and get the example **TTFF**, which has strictly fewer variables set to **T** than the (first) element of S had. We then make a membership query with **TTFF** and discover that it is also a negative example, so we can replace the (first) element of S with the result of the intersection. Then, because S has changed, we generate a new hypothesis H from S. (Again assuming that all incorrect clauses have been eliminated) we have

$$S : [\mathbf{TTFF}],$$

$$H : (a \wedge b \Rightarrow c) \wedge (a \wedge b \Rightarrow d).$$

Suppose our next negative counterexample is **TFTF**. We intersect this with the (first) element of S to get the example **TFFF**, which a membership query shows to be a positive example for H_*. Thus, we cannot refine the (first) element of S with **TFTF**, so we add it to the end of S. This in turn mandates that we regenerate H from S, which (after elimination of incorrect clauses) leaves us with

$$S : [\mathbf{TTFF}, \mathbf{TFTF}],$$

$$H : (a \wedge b \Rightarrow c) \wedge (a \wedge b \Rightarrow d) \wedge (a \wedge c \Rightarrow d).$$

Our final equivalence query for H tells us that we have learned H_*; note that H is logically equivalent to H_*, though not identical to it.

4. Correctness and running time

We prove that the algorithm HORN given in Figure 1 correctly terminates in time $\tilde{O}(m^3n^4)$, making $O(m^2n^2)$ equivalence queries and $O(m^2n)$ membership queries. In the next section, a more efficient version of the algorithm, HORN1, improves these bounds to $\tilde{O}(m^2n^2)$, $O(mn)$, and $O(m^2n)$, respectively.

First observe that the algorithm terminates only if the hypothesis and the target Horn sentence H_* are logically equivalent. Therefore, if the algorithm terminates, it is correct. To show termination in polynomial time we first prove a couple of technical lemmas.

Lemma 4. *For each execution of the main loop of line 3, the following holds. Suppose that in step 5 of the algorithm a negative example x is obtained such that for some clause C of H_* and for some $s_i \in S$, x violates C and s_i covers C. Then there is some $j \leq i$ such that in step 17 the algorithm will refine s_j by replacing s_j with $s_j \cap x$.*

Proof: The proof is by induction on the number of iterations k of the main loop of line 3. If $k = 1$, then the lemma is vacuously true, since the sequence S is empty upon execution of step 5. Assume inductively that the lemma holds for iterations $1, 2, \ldots, k-1$ of the main loop, and assume that during the k-th execution of the loop, $\hat{H}$ is the value of H in the equivalence query in step 3, and at step 5 a negative counterexample x is obtained

such that for some clause C of H_* and for some $s_i \in S$, x violates C and s_i covers C. Since x is a negative counterexample, it must satisfy all the clauses of $\hat{H}$.

Clearly, if in step 17 of the k-th iteration, the algorithm refines some s_j where $j < i$, then we are done. Suppose that this does not happen. Now by Lemma 1, we know that $s_i \cap x$ violates C and therefore is a negative example of H_*. It only remains to be shown that $true(s_i \cap x)$ is properly contained in $true(s_i)$, for then s_i will be refined in step 17.

Suppose to the contrary that $true(s_i \cap x) = true(s_i)$. Since $s_i \cap x$ violates C, this implies that s_i also violates C, so $antecedent(C) \subseteq true(s_i)$ and $consequent(C) \in false(s_i)$. Consider the clause

$$\hat{C} = \left(\bigwedge_{v \in true(s_i)} v \right) \Rightarrow consequent(C).$$

Note that $\hat{C} \in clauses(s_i)$ and also C implies $\hat{C}$. We claim that x violates $\hat{C}$ and $\hat{C}$ is a clause of $\hat{H}$, contradicting the fact that x is a negative counterexample to $\hat{H}$.

To see that x violates $\hat{C}$, note that $true(s_i) \subseteq true(x)$, which implies that $antecedent(\hat{C}) \subseteq true(x)$. Also, since x violates C, $consequent(C) \in false(x)$, and therefore $consequent(\hat{C}) \in false(x)$.

To see that $\hat{C}$ is a clause of $\hat{H}$, observe that each time the sequence S is modified, step 20 of the algorithm discards the old hypothesis and constructs a new hypothesis H from the elements currently in S. Further observe that during each execution of the main loop of line 3, either S is modified (lines 10–21), or else a clause is removed from H (line 8). Let $k' < k$ be the last execution of the main loop of line 3 during which S was modified. Then, during the k'-th iteration, line 20 was executed and H was reconstructed from S. At this time the clause $\hat{C}$ was included in H because s_i was a member of S and $\hat{C} \in clauses(s_i)$. Now C logically implies $\hat{C}$, so $\hat{C}$ could not have been removed in line 8 during iterations $k' + 1, \ldots, k$ of the main loop. Thus $\hat{C}$ must be a clause of $\hat{H}$.

This yields the desired contradition; therefore $true(s_i \cap x)$ is properly contained in $true(s_i)$. Thus the algorithm will replace s_i by $s_i \cap x$ in line 17. □

Lemma 5. *Let S be a sequence of elements constructed for the target H_* by the algorithm. Then*

1. $\forall k \forall (i < k) \forall (C \in H_*)$ *if s_k violates C then s_i does not cover C.*
2. $\forall k \forall (i \neq k) \forall (C \in H_*)$ *if s_k violates C, then s_i does not violate C.*

Proof: We first show property 1 implies property 2, and then show inductively that property 1 is preserved under any modifications the algorithm makes to the sequence S.

Suppose S is any sequence of examples with property 1. To see that S must have property 2, assume to the contrary that k and i are distinct positive integers such that s_k and s_i both violate some clause C in H_*. Without loss of generality, assume that $i < k$. Then, since property 1 holds of S and s_k violates C, s_i does not cover C. By assumption s_i violates C and therefore must cover C. This contradiction shows that S must have property 2.

To see that property 1 is preserved, we argue inductively. Initially the sequence is empty, so property 1 holds vacuously. Now suppose that property 1 (and therefore also property 2) holds for some sequence, and suppose that the algorithm modifies the sequence in response to seeing the negative example x.

If the algorithm appends x to the sequence as, say, s_t, then suppose by way of contradiction that property 1 fails to hold. Inductively, the only way that property 1 could now fail to hold is if there is some $i < t$ such that s_i covers some clause C of H_* that $s_t = x$ violates. This together with Lemma 4 contradicts the fact that the algorithm did not replace s_j by $s_j \cap x$ for some $j \leq i$. Thus property 1 is preserved in this case.

Now suppose that instead of appending x to the sequence, the algorithm replaces some s_k with $s_k \cap x$. Suppose by way of contradiction that property 1 fails to hold. There are two possibilities, either (a) there is some $i < k$ such that s_i covers and $s_k \cap x$ violates some particular clause C of H_* or (b) there is some $i > k$ such that $s_k \cap x$ covers and s_i violates some particular clause C of H_*.

If case (a) holds, then by Lemma 3 either x violates C or s_k violates C. If x violates C then (since s_i covers C) by Lemma 4 there must be some $j \leq i < k$ such that s_j was refined instead of s_k, a contradiction. On the other hand, if s_k violates C, then the fact that s_i and s_k both violate C contradicts the inductive assumption that property 1 (and therefore property 2) held before the modification.

In case (b), there is some $i > k$ such that $s_k \cap x$ covers and s_i violates some clause C of H_*. Since $s_k \cap x$ covers C, s_k covers C as well. Since s_i violates C and $i > k$, this contradicts the inductive assumption that property 1 held before the modification. Thus, in either case property 1 is preserved. □

Corollary 2. *At no time do two distinct elements in S violate the same clause of H_*.*

Proof: This is property 2 of Lemma 5. □

Lemma 6. *Every element of S violates at least one clause of H_*.*

Proof: Each of the elements in S is a negative example, thus by proposition 1, each of the elements violates some clause of H_*. □

Lemma 7. *If H_* has m clauses, then at no time are there more than m elements in the sequence S.*

Proof: This follows immediately from the fact that each of the elements in S violates some clause of H_* but no two elements violate the same clause of H_*. □

Finally, we have our theorem.

Theorem 1. *For all positive integers m and n, the algorithm HORN exactly identifies every Horn sentence with m clauses over n variables in time $\tilde{O}(m^3n^4)$ using $O(m^2n^2)$ equivalence queries and $O(m^2n)$ membership queries.*

Proof: The only changes to the sequence S during any run of the algorithm involve either appending a new element to S, or refining an existing element. Thus $|S|$ cannot decrease during any execution of the main loop of the algorithm. But Lemma 7 shows that there are at most m elements of S at any time. Thus line 18 is executed at most m times.

Now observe that whenever any element s_i of the sequence S is refined (line 17), the resulting new i-th element is $s_i \cap x$, which, by line 11, must contain strictly fewer variables assigned the value T than s_i. This can happen at most n times for each element of S. Thus line 17 is executed at most mn times.

Whenever the ELSE clause at line 9 is executed, either line 17 or 18 must be executed. It follows that lines 9–21 are executed at most $mn + m = m(n + 1)$ times. Note that this bounds the total number of membership queries made by $m^2(n + 1)$.

Next observe that for any element s of S, the cardinality of $false(s)$ is at most $n + 1$ (recalling that $\mathbf{F} \in false(s)$). Thus the cardinality of $clauses(s)$ is at most $n + 1$. Therefore, the number of clauses in any hypothesis H constructed in line 20 is at most $m(n + 1)$.

Now, since each positive counterexample obtained in line 5 necessarily causes at least one clause to be removed from H by line 8, the equivalence query can produce at most $m(n + 1)$ positive counterexamples between modifications to S. Therefore, line 8 is executed at most $m^2(n + 1)^2$ times.

Since each execution of line 3 that does not result in termination causes execution of line 8 or lines 9–21, the total number of executions of line 3 (and hence the total number of equivalence queries made) is at most $m^2(n + 1)^2 + m(n + 1) + 1$.

To complete the proof we need only show that the time needed for each execution of the main loop is $\tilde{O}(mn^2)$. Using the facts (above) that at any time during the execution of the algorithm $|S| \leq m$ and $|H| \leq m(n + 1)$, and that each element of H consists of at most $n + 1$ variables (antecedent + consequent), it is easily verified that the time needed to execute either of steps 8 and 20 is $\tilde{O}(mn^2)$, and that these steps dominate the time to execute one iteration of the main loop. □

5. Improvements to the algorithm

We now describe a more efficient version, HORN1, of our learning algorithm for Horn sentences. There is a natural shorthand notation for propositional Horn sentences obtained by gathering up all the clauses with the same antecedent set and conjoining the consequents. The conjunction of several clauses $C_1, \ldots, C_k$ with the same antecedent will be represented as a *meta-clause* whose antecedent is the common antecedent of the clauses and whose consequent is the conjunction of $consequent(C_i)$ for $i = 1, \ldots, k$. For example, the meta-clause

$$(a \wedge b \wedge d \Rightarrow c \wedge e)$$

is logically equivalent to, and will be used to represent, the conjunction

$$(a \wedge b \wedge d \Rightarrow c) \wedge (a \wedge b \wedge d \Rightarrow e).$$

The new version of the algorithm maintains the current hypothesis as a sequence of meta-clauses, one meta-clause corresponding to each negative example in the sequence S in the previous version. We assume that this representation is used both by the algorithm and for the equivalence queries. (If the equivalence queries require that the representation be strictly a conjunction of Horn clauses, further (straightforward) optimizations must be made to achieve the time bounds below.)

In addition to this shorthand representation, we make use of the observation that once a positive counterexample eliminates a clause, it eliminates any clause with a refined antecedent. For example, the positive counterexample **TTTF** eliminates the clause $(a \wedge b \wedge c \Rightarrow d)$, and also refinements like $(a \wedge b \Rightarrow d)$ and $(b \wedge c \Rightarrow d)$. Thus, when we refine the antecedent of a meta-clause, we do not need to re-introduce possible consequents that have been eliminated.

The effects of counterexamples on meta-clauses can be exemplified as follows, starting with the metal-clause

$$(a \wedge b \wedge c \Rightarrow d \wedge e \wedge f).$$

A positive counterexample causes item(s) to be struck from the consequent, for example, a positive counterexample **TTTTFT** would result in the meta-clause

$$(a \wedge b \wedge c \Rightarrow d \wedge f).$$

A subsequent negative counterexample that refines the corresponding negative example moves variable(s) from the antecedent to the consequent. For example, the negative example **TTFFFF** then results in

$$(a \wedge b \Rightarrow c \wedge d \wedge f).$$

Using suitable data structures, this means that the total processing time spent modifying the hypothesis is $\tilde{O}(mn)$, since each variable can appear in the antecedent, be moved to the consequent, and be deleted, from each meta-clause. A more formal treatment follows.

We use the partial ordering $\leq$ on assignments defined by $x \leq y$ if and only if $x_i \Rightarrow y_i$ for $i = 1, \ldots, n$. (Equivalently, $x \leq y$ if and only if $x_i \leq y_i$, where we take $\mathbf{F} \leq \mathbf{T}$.) If C is a meta-clause, let $negex(C)$ denote the example that assigns **T** to all the variables in the antecedent of C and **F** to all the other variables. Then $negex(C)$ is the minimum example in the ordering by $\leq$ that violates C. In the new version of the algorithm, H is the conjunction of a sequence of meta-clauses C_i such that $negex(C_i) = s_i$.

We define three meta-clause operations: generating a new meta-clause from a negative example ($new(x)$), reducing a meta-clause with a positive counterexample ($reduce(C, x)$), and strengthening a meta-clause with a negative example ($refine(C, x)$.)

Given a negative example x, define $new(x)$ to be the meta-clause whose antecedent is the set $true(x)$ and whose consequent is **F**. Note that $negex(new(x)) = x$. For example,

$$new(\mathbf{TFTFT}) = (a \wedge c \wedge e \Rightarrow \mathbf{F}).$$

This operation is used to construct an initial meta-clause from a new negative example. It replaces the operation *clauses*(x). The consequent **F** is introduced first because the Horn clause with antecedent set A and consequent **F** logically implies every other Horn clause with antecedent set A. The other possible consequents are only introduced if and when the consequent **F** is eliminated by some positive counterexample.

Given a meta-clause C and an example $x > negex(C)$ for which C is false (intuitively, x is a positive counterexample) we define *reduce*(C, x) to be a meta-clause with the same antecedent as C and consequent defined as follows.

1. If the consequent of C is **F**, then the consequent of *reduce*(C, x) is the conjunction of those variables v_i such that $v_i \in false(negex(C)) \cap true(x)$.
2. If the consequent of C is not **F**, then the consequent of *reduce*(C, x) is the conjunction of those variables v_i in the consequent of C such that $v_i \in true(x)$.

For example,

$$reduce((a \wedge b \Rightarrow \mathbf{F}), \mathbf{TTTFT}) = (a \wedge b \Rightarrow c \wedge e).$$

Also,

$$reduce((a \wedge b \Rightarrow c \wedge d), \mathbf{TTTFT}) = (a \wedge b \Rightarrow c).$$

Given a meta-clause C and an example $x < negex(C)$ (intuitively, x is a negative counterexample) we define *refine*(C, x) to be a meta-clause whose antecedent is the set *true*(x) whose consequent is defined as follows.

1. If the consequent of C is **F**, then the consequent of *refine*(C, x) is **F**.
2. If the consequent of C is not **F**, then the consequent of *refine*(C, x) is the conjunction of all the variables in the consequent of C and all of the variables v_i such that $v_i \in false(x) \cap true(negex(C))$.

For example,

$$refine((a \wedge b \wedge c \Rightarrow \mathbf{F}), \mathbf{TFTFF}) = (a \wedge c \Rightarrow \mathbf{F}).$$

Also,

$$refine((a \wedge b \Rightarrow c \wedge e), \mathbf{TFFFF}) = (a \Rightarrow b \wedge c \wedge e).$$

Note that the possible consequent d is not re-introduced here, having been (presumably) eliminated by a previous positive example.

The new version of the algorithm, named HORN1 and shown in Figure 2, has the same line numbers as the previous version, for ease of comparison. As for the previous version, if the algorithm halts then its output is correct, so we need only give bounds on its running time.

```
1  /* H is a conjunction of meta-clauses C_i */
2  Set H to be the empty hypothesis
3  UNTIL equivalent(H) returns "yes" DO
4     BEGIN /* main loop */
5     Let x be the counterexample returned by the equivalence query
6     IF x violates at least one meta-clause of H
7        THEN /* x is a positive example */
8           replace C_i by reduce(C_i, x) for every C_i that x violates
9        ELSE /* x is a negative example */
10          BEGIN
11          FOR each C_i in H such that (negex(C_i)∩x) < negex(C_i)
12             BEGIN
13             query member(negex(C_i)∩x)
14             END
15          IF any of these queries is answered "no"
16             THEN let i = min{j : member(negex(C_j)∩x) = "no"}
17                replace C_i by refine(C_i, negex(C_i)∩x)
18             ELSE add new(x) as the last meta-clause in H
19          ENDIF
20          /* H is already updated */
21          END
22    ENDIF
23    END /* main loop */
24 Return H
```

Figure 2. The algorithm HORNI.

Since throughout the new version $negex(C_i)$ is equal to s_i in the old version, the same argument shows that if m is the number of clauses of the target H_*, then there are at most m meta-clauses in H at any time. Note that no meta-clause is ever deleted from H.

Consider the career of a particular meta-clause C_i. C_i is initially created in response to a negative counterexample. When C_i is first created, its antecedent consists of a conjunction of variables (and the constant **T**), and its consequent is **F**.

The antecedent of C_i then only changes in response to negative counterexamples, and the change must be to delete one or more variables from the antecedent. Thus, there can be at most n such changes. Since every negative counterexample must either cause the creation of a new meta-clause or refine an existing one, there can be at most $m(n + 1)$ negative counterexamples.

The consequent of C_i may change in response to negative or positive counterexamples. The first positive counterexample to C_i changes its consequent from **F** to the conjunction of a subset of the variables not in the antecedent of C_i. Negative counterexamples before the first positive counterexample do not change the consequent of C_i—it remains **F**. Subsequent negative counterexamples to C_i may move one or more variables from the antecedent to the consequent of C_i. Positive counterexamples to C_i, after the first, can only remove variables from the consequent of C_i. Every variable can be deleted at most once from the consequent of C_i, and, if the consequent is not **F**, at least one variable must remain in it. Thus, C_i can be changed by at most n positive counterexamples. Since every

positive counterexample must change one or more meta-clauses, there can be at most mn positive counterexamples.

Since each negative counterexample can cause at most m membership queries, no more than $m^2(n + 1)$ membership queries will be made. With a straightforward representation of meta-clauses and assignments as lists of length n, this algorithm can be implemented to run in time $\tilde{O}(m^2n^2)$ on a log-cost RAM.

Our analysis of the improved algorithm establishes the following theorem.

Theorem 2. *For all positive integers m and n, the algorithm HORN1 exactly identifies every Horn sentence with m clauses over n variables in time* $\tilde{O}(m^2n^2)$ *using* $O(mn)$ *equivalence queries and* $O(m^2n)$ *membership queries.*

6. Conclusions

Our main result is a polynomial-time algorithm for exact identification of Horn sentences using equivalence and membership queries. One corollary of the result is that the class of Horn sentences is polynomial-time PAC learnable if membership queries are available. Another is that the class of Horn sentences is polynomial-time predictable with a polynomial mistake bound if membership queries (excluding the elements to be predicted) are available.

If membership queries are not available, it is an open problem whether Horn sentences are PAC-learnable or polynomial-time predictable. By the reductions of Kearns, Li, Pitt, and Valiant (1987) PAC-learnability of Horn sentences would imply PAC-learnability of general CNF and DNF sentences, and similarily for polynomial predictability.

An interesting open problem is whether the algorithm here can be extended to handle restricted types of universally quantified Horn sentences (see the papers of Valiant (1985) and Haussler (1989) for related classes of formulas). This class is of significant interest due to its similarity to the language Prolog, and its use in logic programming and expert system design.

Acknowledgments

Financial support for this research from the Department of Computer Science of the University of Illinois at Urbana-Champaign and from the National Science Foundation under grants IRI-8718975, IRI-8809570, IRI-9014840 and CCR-9014943 is gratefully acknowledged. The authors thank Jonathan Gratch, David Haussler, Michael Kearns and the anonymous referees for their contributions to this paper. A preliminary version of this paper was presented at the 31st Annual Symposium on Foundations of Computer Science (Angluin, Frazier, & Pitt, 1990).

Notes

1. The $\tilde{O}(\ )$, or "soft-O," notation is similar to the usual $O(\ )$ notation except that $\tilde{O}(\ )$ ignores logarithmic factors.
2. Throughout the paper, we omit writing the "**T**" in an otherwise nonempty antecedent so as to avoid cluttering our examples.

3. This method of discarding overly restrictive pieces of a conjunctive hypothesis is a standard technique and is incorporated in step 8 of the learning algorithm in Figure 1.
4. The approach of intersections was used by Haussler in an alternative algorithm to learn monotone DNF (Haussler, private communication).

References

Angluin, D. (1988). Learning with hints. In *Proceedings of the 1988 Workshop on Computational Learning Theory* (pp. 167–181). Boston, MA: Morgan Kaufmann.

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, *2*, 319–342.

Angluin, D. (1988). *Requests for hints that return no hints* (Technical Report YALE/DCS/RR-647). Department of Computer Science, Yale University.

Angluin, D. (1990). Negative results for equivalence queries. *Machine Learning*, *5*, 121–150.

Angluin, D., Frazier, M., & Pitt, L. (1990). Learning conjunctions of Horn clauses. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science* (pp. 186–192). St. Louis, MO: IEEE Computer Society Press.

Angluin, D., Hellerstein, L., & Karpinski, M. (1989). *Learning read-once formulas with queries* (Technical Report, University of California at Berkeley, Report No. 89/528). (Also, International Computer Science Institute Technical Report TR-89-050. To appear, *JACM*.)

Angluin, D. & Kharitonov, M. (1991). When won't membership queries help? In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing* (pp. 444–454). New Orleans, LA: ACM Press.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. (1989). Learnability and the Vapnik-Chervonenkis dimension. *J. ACM*, *36*, 929–965.

Ehrenfeucht, A. & Haussler, D. (1988). Learning decision trees from random examples. In *Proceedings of the 1988 Workshop on Computational Learning Theory* (pp. 182–194). Boston, MA: Morgan Kaufmann.

Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, *36*, 177–221.

Haussler, D. (1989). Learning conjunctive concepts in structural domains. *Machine Learning*, *4*, 7–40.

Haussler, D., Littlestone, N., & Warmuth, M.K. (1988). Predicting {0, 1} functions on randomly drawn points. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, (pp. 100–109). Washington, D.C.: IEEE Computer Society Press.

Jones, N.D. & Laaser, W.T. (1977). Complete problems for deterministic polynomial time. *Theoretical Computer Science*, *3*, 107–113.

Kearns, M., Li, M., Pitt, L., & Valiant, L.G. (1987). On the learnability of Boolean formulae. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (pp. 285–295). New York, NY: ACM Press.

Kearns, M. & Valiant, L.G. (1989). Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing* (pp. 433–444). Seattle, WA: ACM Press.

Littlestone, N. (1988). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, *2*, 285–318.

Pitt, L. & Warmuth, M.K. (1990). Prediction-preserving reducibility. *J. of Computer and System Sciences*, *41*, 430–467.

Rivest, R.L. (1987). Learning decision lists. *Machine Learning*, *2*, 229–246.

Valiant, L.G. (1985). Learning disjunctions of conjunctions. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence* (pp. 560–566). Los Angeles, CA.

Valiant, L.G. (1984). A theory of the learnable. *Communications of the ACM*, *27*, 1134–1142.