Version Spaces and the Consistency Problem

Haym Hirsh * Nina Mishra † Leonard Pitt ‡

December 17, 2003

Abstract

A version space is a collection of concepts consistent with a given set of positive and negative examples. Mitchell [Mit82] proposed representing a version space by its boundary sets: the maximally general (G) and maximally specific consistent concepts (S). For many simple concept classes, the size of G and S is known to grow exponentially in the number of positive and negative examples. This paper argues that previous work on alternative representations of version spaces has disguised the real question underlying version space reasoning. We instead show that tractable reasoning with version spaces turns out to depend on the consistency problem, *i.e.*, determining if there is any concept consistent with a set of positive and negative examples. Indeed, we show that tractable version space reasoning is possible if and only if there is an efficient algorithm for the consistency problem. Our observations give rise to new concept classes for which tractable version space reasoning is now possible, *e.g.*, 1-decision lists, monotone depth two formulas, and halfspaces.[1]

1 Introduction

The problem of *inductive learning* is to extrapolate from a collection of *training data* — a set of examples each labeled as either positive or negative by some unknown target concept — a concept definition that accurately labels future, unlabeled data. Most inductive learning algorithms operate over some *concept class* — the set of concepts that the learner can potentially generate over all possible sets of training data.

Mitchell's [Mit82] introduction of the notion of a *version space* provided a useful conceptual tool for inductive learning. Given some concept class C, the version space for a set of examples is simply the set of concepts in C that are consistent with the data, *i.e.*, that correctly label the set of examples. Its name originates from the idea that all and only plausible versions of the unknown target concept are retained in this space.

To learn and reason using version spaces Mitchell proposed representing a version space by maintaining only the set of maximally general (G) and maximally specific (S) consistent concepts in the space. He called these the version space's *boundary sets* since they bound the set of all concepts in the version space. Maintaining boundary sets is clearly more efficient than explicitly maintaining all elements of a version space, since boundary sets include only a subset of the full version space. However, even the boundary-set representation has a number of serious limitations. For example, Haussler [Hau88] showed that for the concept class of positive-monotone terms over n variables (*i.e.*, simple conjunctions of subsets of n unnegated Boolean attributes) there are cases where after a number of examples linear in n, the boundary set G has size exponential in n. Further, for infinite concept classes, boundary sets can have potentially infinite size, and in some cases (for *inadmissible* concept classes [Mit82]) it is not even possible to represent a version space

*Rutgers University, hirsh@cs.rutgers.edu. Supported in part by NSF Grant IRI-9209795.

†HP Labs and Stanford University, nmishra@theory.stanford.edu. Supported in part by NSF Grant EIA-0137761.

‡University of Illinois at Urbana-Champaign, pitt@uiuc.edu. Supported in part by NSF Grant IIS-9907483.

[1]This paper expands and updates the results presented by Hirsh, Mishra, and Pitt (1997).

by its boundary sets. (See Section 3 for further discussion of concept classes that exhibit various pathologies of this sort.)[2]

A number of papers have proposed alternative representations for version spaces to overcome some of these limitations. Smith and Rosenbloom [SR90], for example, showed that it is possible to maintain the S and G boundary sets for all the positive data but only a selected subset of the negative data, along with a set N of the *unprocessed* negative examples to guarantee tractable version-space learning for a constrained class of conjunctive languages. Subsequent work [Hir92] took this further, doing away with the set G altogether, and instead using an $[S, N]$ representation that maintains only the boundary set S together with the set N of all negative examples. For many concept classes it was shown that most of what could be done with the $[S, G]$ representation could be accomplished with the $[S, N]$ representation. Moreover, tractability guarantees could be given due to the more modest representational requirement of maintaining a single boundary set. In contrast, both Lau, Wolfman, Domingos, and Weld [LWDW03] and Smith [Smi95] consider more markedly different representations of version spaces. Lau *et al.* present compositional approach that they call a "version space algebra" that can broaden the use of version spaces on more realistic tasks (for example, to automate repetitive text editing). Smith presents a more theoretical broadening of version spaces in which they are represented by grammars and operations over them.

This paper argues that the entire focus of such previous work on version-space representations disguises the real question underlying effective learning and reasoning with version spaces. We show that if the goal is tractable learning and reasoning with version spaces, the focus should be on finding efficient algorithms for the *consistency problem* — the problem of determining if *any* concept in a given concept class C correctly classifies a given set of positive and negative examples. We demonstrate that almost all operations performed on version spaces for any concept class C can be efficiently executed if and only if there is an efficient solution to the consistency problem for C. In version-space terminology, this says that almost all version-space operations can be efficiently executed if and only if there is an efficient way of determining whether a version space has *collapsed*, *i.e.*, is empty. The focus of past work on version-space representations has instead entangled questions about the tractability of computing various version-space operations within the representation with questions about representational adequacy (such as about the inadmissibility of a concept class). We will show that there are concept classes where boundary sets are intractable or ill-defined, yet because the consistency operation is tractable for these classes most version-space operations are also computable tractably. Thus, for example, the class of 1-decision lists [Riv87] can have both boundary sets grow exponentially large in the amount of data, yet because the consistency problem is tractable for 1-decision lists, so too are many of the most common version-space operations.[3] There are therefore problems in which version spaces are intractable if one is forced to use boundary-set representations, but where version-space operations are tractable when they are computed directly on the data using computations that can exploit tractable procedures for testing consistency. To motivate our work we present additional negative results of this sort concerning the use of boundary sets. We give numerous examples where manipulating boundary sets can be difficult because the number of elements or the size of the elements in the boundary sets may be large or the boundary sets may be undefined. We show cases where even determining how many elements are in the boundary sets can be extremely difficult.

Our positive results establish a number of key equivalences between various version-space operations. Clearly the question of whether there exists any concept c in a concept class C that is consistent with positive data P and negative data N is equivalent to the question of whether the version space for P and N has collapsed, *i.e.*, whether there are no concepts c consistent with the data. More interesting is that the operation of classifying a new example using a partially learned version space — determining if every element of a version space classifies the example identically and what the classification is — is computationally tractable if and only if the consistency problem is tractable.

Given that one of the main insights and sources of power concerning version spaces is that they provide a

[2]Version spaces also clearly face a number of other difficulties, such as in their limited capacity to handle noisy data or concept classes that do not contain the target concept [Utg86, Hir94], but we concern ourselves in this paper solely with limitations of version spaces that are present even when these are not at issue.

[3]It is interesting to contrast this with the case of conjunctions of existentially quantified predicates, which Haussler [Hau89] showed can have exponential boundary sets, but for which consistency is not computable tractably.

way to classify future, unseen examples even when insufficient data have been obtained to identify the unique identity of the target concept, we take a moment here to sketch the simple argument of the equivalence. To classify an example x according to the version space defined by positive examples P and negative examples N, we run the consistency algorithm twice. The first time, we run the algorithm assuming x is a positive example (*i.e.*, with positives $P \cup \{x\}$ and negatives N). If no consistent concept exists (*i.e.*, if the version space collapses), then every consistent concept must classify x negative. Otherwise, we run the consistency algorithm assuming x is a negative example (i.e., with positives P and negatives $N \cup \{x\}$). Similarly, if no consistent concept exists, then every consistent concept must classify x positive. If neither case holds, there must be at least one consistent concept that classifies x positive and at least one that classifies x negative — the label of x is not determined by the version space, and, consequently, x is labeled "?".

In addition to this equivalence, we show that determining whether the version space for one set of data is a subset of the version space for a second set of data is computationally equivalent to the consistency problem (and thus also equivalent to the problem of classifying examples using partially learned version spaces). Finally, a tractable solution to any of these operations also implies that it is possible to test if two version spaces are equivalent. The positive consequence of the above equivalences is that many classes previously not tractable using the boundary-set representation for version spaces can now be learned in the version-space framework. Such concept classes include, for example, 1-decision lists, half-spaces, and conjunctions of Horn clauses.

In Section 2 we give necessary preliminaries and notation, and formally define the problems addressed in the paper. After considering in Section 3 various boundary-set pathologies as discussed above, we prove in Section 4 that classifying an example with a partially learned version space is computationally exactly the consistency problem, and that almost all of the standard operations on version spaces can be performed using an efficient solution to the consistency problem. The main moral of this paper is that when dealing with version spaces, the operative question should *not* be "What is a good representation of the version space?", but rather "Is there an efficient consistency algorithm?" We make this more explicit in Section 5 by exhibiting concept classes where version-space manipulations turn out to be quite easy, since these classes have efficient consistency algorithms — whereas maintaining boundary sets for these concept classes can be intractable or even impossible.

2 Definitions and Notation

Concepts. Let $V = \{v_1, \ldots, v_m\}$ be a collection of Boolean variables. An *example* v is any assignment of the variables in V to 0 or 1, *i.e.*, $v \in \{0,1\}^m$. We use the terms example, vector, and assignment interchangeably. We denote by X the space of all examples $\{0,1\}^m$. A *concept* c is a subset $c \subseteq X$. An example $x \in X$ is said to be labeled *positive* by c if $x \in c$, and *negative* otherwise. A *concept class*, $C \subseteq 2^X$, is a collection of concepts. A concept c is not typically represented "extensionally" by explicitly listing its elements, but rather "intensionally" by giving a representation r for c from which one can efficiently compute whether a given example x is in c. More generally, concept classes C are defined by describing a collection of admissible representations R, where each $r \in R$ denotes some concept c, and where C is the set of representable concepts. For simplicity of exposition we will ignore this distinction, except where it helps to clarify a technical point.

Boolean formulas, etc. A Boolean function $f(v_1, \ldots, v_m)$ is a function $f : \{0,1\}^m \to \{0,1\}$. A *monotone Boolean function* f has the property that if $f(x) = 1$ then for all $y \geq x$, $f(y) = 1$, where "$\geq$" is defined by the partial order induced by the n-dimensional Boolean hypercube $\{0,1\}^m$. Equivalently, $y \geq x$ means that each bit of the vector y is at least as large as the corresponding bit of x. The class of Boolean formulas F over variable set $V = \{v_1, \ldots, v_m\}$ is defined inductively as follows: (1) each of the symbols $\{0, 1, v_1, \ldots, v_m\}$ is a Boolean formula over V; (2) if F_1 and F_2 are Boolean formulas over V then so are $(F_1 \vee F_2)$, $(F_1 \wedge F_2)$; and (3) if F_1 is a Boolean formula over V then so is $\overline{F_1}$. The class of *monotone Boolean formulas* F is defined inductively in the same way, but using only rules (1) and (2). Thus, a monotone Boolean formula is one which contains no negation symbols. We use the terms formula and expression interchangeably. Each

Boolean formula over V describes a Boolean function of m Boolean variables in the usual way, with the standard interpretation of the logical connectives $\vee$ (OR), $\wedge$ (AND) and $\neg$ (NOT).

A monotone Boolean function f can be described by its *minimally positive assignments* (minimal models). A minimally positive assignment of f is a positive assignment with only negative assignments below it, *i.e.*, a vector $v \in \{0,1\}^m$ such that $f(v) = 1$ and for all $u < v$ $f(u) = 0$. A monotone Boolean function can also be described dually by its *maximally negative assignments*, *i.e.*, the vectors u such that $f(u) = 0$ and for all $v > u$, $f(v) = 1$.

A *term* t is the function represented by a simple conjunction (AND) $t = v_{i_1} \wedge v_{i_2} \wedge \cdots \wedge v_{i_s}$ of literals v_{i_j}, where a *literal* is either a variable x_i or its negation $\overline{x_i}$. A term is monotone if all literals in its representation are unnegated variables. Henceforth, we consider only monotone terms. The (monotone) term t evaluates to 1 if and only if each of the variables $v_{i_1}, v_{i_2}, \ldots, v_{i_s}$ have value 1. Similarly, a *monotone clause* c is the function represented by a disjunction (OR) $c = v_{j_1} \vee v_{j_2} \vee \cdots \vee v_{j_r}$ of variables. The clause c evaluates to 1 if and only if at least one of the variables $v_{j_1}, v_{j_2}, \ldots, v_{j_r}$ has value 1.

A *monotone DNF* expression is a disjunction (OR) of monotone terms $t_1 \vee t_2 \vee \cdots \vee t_a$, and evaluates to 1 iff at least one of the terms has value 1. If $T = \{t_1, \ldots, t_a\}$ is a set of terms, then $\vee T$ is the DNF expression $t_1 \vee t_2 \vee \cdots \vee t_a$. Similarly, a *monotone CNF* expression is a conjunction of monotone clauses $c_1 \wedge c_2 \wedge \cdots \wedge c_b$, and evaluates to 1 iff each of the clauses has value 1. If $C = \{c_1, \ldots, c_b\}$ is a set of clauses then $\wedge C$ is the CNF expression $c_1 \wedge c_2 \wedge \cdots \wedge c_b$.

A *decision list* is an ordered sequence of if-then-else statements. The sequence of if-then-else conditions are tested in order and the answer bit associated with the first satisfied condition is output. In the following decision list the answer bit $b_i \in \{0,1\}$, each c_i is a term, and p corresponds to some example.

if $c_1(p)$ then b_1
else if $c_2(p)$ then b_2
..
else b_q

Decision lists are typically written in the form $(c_1(p), b_1), (c_2(p), b_2), \ldots, b_q$. In a k-decision list (k-DL), each term c_i has no more than k variables.

A *Horn clause* resembles a rule $v_{j_1} \wedge \cdots \wedge v_{j_r} \rightarrow v_j$. A Horn clause evaluates to 0 if and only if the antecedent $(v_{j_1} \wedge \cdots \wedge v_{j_r})$ evaluates to 1 and the consequent v_j evaluates to 0. A *Horn sentence* is a conjunction of Horn clauses.

Version Spaces. For a set of positive examples, P, negative examples, N, and a concept class, C, the *version space* is the set of all concepts in C consistent with P and N [Mit82]. We use $C_{P,N}$ to denote the version space induced by P and N. Specifically, $C_{P,N} = \{c \in C : P \subseteq c, \text{ and } N \cap c = \emptyset\}$. If $C_{P,N}$ is empty we say that the version space has *collapsed*; if it contains a single concept we say that the version space has *converged*.

A version space can be viewed as a function. In particular, if every element of a version space labels an example x positive then that example x can be unambiguously labeled positive since the target concept is one of the elements in the version space and, independent of which element is the target, the example x is labeled positive. Similarly if every element of a version space labels an example x negative. We thus define, for a version space $C_{P,N}$, a corresponding function, *classify*$(C_{P,N})$, that, given an example, outputs a label that reflects the consensus of concepts in $C_{P,N}$:

$$\text{classify}(C_{P,N})(x) = \begin{cases} \text{“}\emptyset\text{”} & \text{if } C_{P,N} \text{ is empty} \\ \text{“}+\text{”} & \text{if } x \in c \text{ for all } c \in C_{P,N} \\ \text{“}-\text{”} & \text{if } x \notin c \text{ for all } c \in C_{P,N} \\ \text{“?”} & \text{otherwise} \end{cases}$$

We sometimes abuse notation and write $C_{P,N}(x)$ instead.

The tractable computation of the classify function is one of the key questions that we study here:

Definition 2.1 *A concept class C is (efficiently)* version-space predictable *if there exists an algorithm that,*

given a set P of positive examples, a set N of negative examples, and an example x, outputs $C_{P,N}(x)$ in time polynomial in $|P|$, $|N|$, and $|x|$.

Note that we are only interested in the question of the tractable computation of $C_{P,N}(x)$ for concept classes C that are "missing" some concepts, *i.e.*, $C \neq 2^X$. Otherwise, if $C = 2^X$ (*e.g.*, if C is the class of Boolean formulas in Disjunctive Normal Form (DNF)) then it is easy to see that C is trivially version-space predictable. In particular, since every Boolean function can be represented as a DNF, for any example x that is not in $P \cup N$, half of the concepts in $C_{P,N}$ classify x as positive, and half classify x as negative[4]. Consequently, $C_{P,N}(x) =?$. Note also that learning is impossible under such circumstances without additional assumptions, *e.g.*, the choice of a restricted space of hypotheses given by bounding the size of the DNF, or by other syntactic or semantic constraints. The choice of restriction is more commonly called the *bias* of the learning algorithm [Mit97].

The consistency problem for C can be summarized as the problem of determining if there is a concept in C that correctly labels a given set of positive and negative examples. More formally stated,

Definition 2.2 *The* consistency problem *for C is: Given a set P of positive examples and a set N of negative examples, is $C_{P,N} \neq \emptyset$?*

Notice that the consistency problem is exactly the question of whether a version space induced by P and N has not collapsed.

We'll say that the consistency problem for C is *efficiently* computable if there is an algorithm for the consistency problem for C that runs in time polynomial in $|P|$ and $|N|$. The definition of the consistency problem and its efficient computability is not as general as the one given in [AHHP98], but is sufficient for our purposes.

In many references [BEHW89, PV88, AHHP98], the consistency-problem definition requires that a consistent hypothesis be explicitly output if one exists. While our main results do not require that a hypothesis be output, all of the algorithms we exhibit in Section 5 actually output consistent hypotheses.

Other questions that we address in later sections include whether one version space is a subset of another, and whether two version spaces are equivalent.

Definition 2.3 *The version space for a concept class C is* subset-testable *if there exists an algorithm that, given two sets of positive examples, P_1 and P_2, and two sets of negative examples, N_1 and N_2, determines in time polynomial in $|P_1|$, $|P_2|$, $|N_1|$, and $|N_2|$, whether $C_{P_1,N_1} \subseteq C_{P_2,N_2}$.*

Definition 2.4 *The version space for a concept class C is* equivalence-testable *if there exists an algorithm that, given two sets of positive examples, P_1 and P_2, and two sets of negative examples, N_1 and N_2, determines in time polynomial in $|P_1|$, $|P_2|$, $|N_1|$, and $|N_2|$, whether $C_{P_1,N_1} = C_{P_2,N_2}$.*

Throughout the paper, "efficient" computation means "deterministic polynomial-time" in the relevant parameters.

Finally, we define the set of maximally general and maximally specific concepts. These sets have traditionally been the tools used to represent version spaces.

Definition 2.5 *For a concept class C, positive examples P, and negative examples N:*

- *A concept $c \in C$ is* maximally specific *if c is consistent with P and N and if for each $c' \in C$ such that c' is more specific than c, c' is not consistent with P and N. The* maximally specific set (S) *refers to the set of all maximally specific concepts.*

[4]Note that since $X = \{0,1\}^m$ and since there are 2^{2^m} different Boolean functions, the number of functions that classify a particular example x positive is 2^{2^m-1}. Thus exactly half of the consistent functions classify x as positive. Similarly if x is classified as negative.

- *A concept $c \in C$ is* maximally general *if c is consistent with P and N and if for each $c' \in C$ such that c' is more general than c, c' is not consistent with P and N. The* maximally general set (G) *refers to the set of all maximally general concepts.*

The maximally specific set and maximally general set are commonly referred to as the boundary sets.

3 Boundary Set Pathologies

Whether various version space operations from boundary sets S and G can be tractably performed depends on the structure and representation of S and G. There are many circumstances where manipulating one or both of these sets is problematic. For example, the number of elements in both S and G can grow large quickly, S and/or G may have only one element but the representation of that element may be exponentially large, and S and/or G may not be well-defined. In addition, testing whether a given set of concepts C' is equal to G or S (induced from known P and N) can be hard. In some cases simply determining how many elements are in G or S can be hard.

In this section we concretely demonstrate such situations. The reader already convinced may skip this section. More specifically, we'll exhibit the following pathologies.

- For the class of 1-decision lists we give a set of $O(n)$ examples for which the corresponding sets G and S have size $\Omega(n!2^n)$.
- For the subclass of monotone depth-two formulas we'll give a set of $O(n)$ examples for which $|G| = |S| = 1$, but the size of the single element in each case is exponentially large.
- For the class of monotone terms we show that determining whether a given set of monotone terms F is equivalent to G is as hard as a well-known problem for which the best known running time is quasi-polynomial.
- For the class of monotone terms, we show that just determining $|G|$ is #P-complete[5] via a reduction from the problem of counting minimal vertex covers.
- We give some examples of ill-defined ("inadmissible") boundary sets.

Later in Section 5 we'll see that even though the classes discussed in this section are not version space predictable using boundary sets, all are (efficiently) version space predictable via efficient algorithms for the consistency problem.

Large Boundary Sets: 1-Decision Lists. Haussler [Hau88] showed for the class of terms that one boundary set can grow large very quickly. Single-sided boundary set representations of the type described in the introduction, where only one of the boundary sets is maintained, can be used to effectively deal with this problem. In addition, Haussler showed that both boundary sets can quickly grow large for a relational concept class. We show that both boundary sets can grow large even for a simple propositional class, 1-DLs. We next demonstrate that both G and S can grow to size at least $n!2^n$ after $O(n)$ examples. Thus, even single-sided boundary set representations like $[S, N]$ are not in general tractable to maintain.

Rivest [Riv87] proposed k-decision lists as an interesting class of Boolean functions that properly contains DNF (and CNF) formulas where each term (respectively, clause) has no more than k variables. The class of 1-Decision Lists (1-DLs) generalizes terms and clauses of (single) variables: A conjunction of literals $\ell_1 \wedge \ell_2 \wedge \ldots \wedge \ell_s$ can be represented as the 1-DL $(\overline{\ell_1}, 0), (\overline{\ell_1}, 0), \ldots, (\overline{\ell_s}, 0), 1$, and a disjunction of literals $\ell_1 \vee \ell_2 \vee \ldots \vee \ell_s$ can be represented as the 1-DL $(\ell_1, 1), (\ell_2, 1), \ldots, (\ell_s, 1), 0$.

To describe the positive and negative examples, we introduce some notation. Let $\vec{1}$ (resp, $\vec{0}$) be a vector of length $2n$ such that every bit position is 1 (resp, 0). Let $\vec{1_i}$ (resp, $\vec{0_i}$) be a vector of length

[5] A formal definition of #P-complete can be found in [GJ79]. Intuitively, #P is the counting analog of NP. For example, while the decision problem of determining whether a given graph has a vertex cover of size k is NP-complete, the problem of counting the number of minimal vertex covers is #P-complete.

$2n$ with all bit positions set on (resp, off) except for i and $i+1$. Let $p_i = 0_{2i}\vec{1}$ and $n_i = \vec{0}1_{2i}$ over the variables $u_1, v_1, \ldots u_n, v_n, x_1, y_1, \ldots, x_n, y_n$. The positive and negative examples that result in excessively large boundary sets are $P = \{p_i : i = 1, \ldots, n\}$ and $N = \{n_i : i = 1, \ldots, n\}$. We show in the appendix that for P and N as given that both boundary sets have size exponential in n.

Theorem 3.1 *For the concept class of 1-DLs and for P, N as given above, the size of both G and S is $\Omega(n!2^n)$.*

Large Singleton Boundary Sets: Monotone Depth-two Formulas. Even boundary sets with only one element can exhibit pathologies if that single concept has only exponentially large representations. We demonstrate this for the class of depth-two monotone formulas. Depth-two formulas are those whose tree representation has depth at most two, hence are identical to the class of CNF $\cup$ DNF. With the same set of training data given for the 1-decision list case, the boundary set S (respectively, G) contains only the monotone depth-two formula representing the Boolean function

$$f_S = ((u_1 \wedge v_1) \vee (u_2 \wedge v_2) \vee \cdots \vee (u_n \wedge v_n)) \wedge (x_1 \wedge y_1 \wedge \cdots \wedge x_n \wedge y_n)$$

(respectively, $f_G = (u_1 \vee v_1 \vee \cdots \vee u_n \vee v_n) \vee (x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \cdots \wedge (x_n \vee y_n)$).

As described, both f_S and f_G have polynomial-sized depth-three representations. In order to represent f_S (respectively, f_G) as a depth-two formula, i.e., in its unique reduced CNF (respectively DNF) form, the size of the representation will exponentially increase to $O(2^n)$.

Notice also that since the CNF and DNF for a function both provide lower bounds on the size of the smallest decision tree (the branches from the root to the "1" leaves give the DNF of a function) a lower bound of $\Omega(2^n)$ follows also for decision trees capturing these functions.

Testing Equivalence. Testing whether a given set of concepts F is equivalent to G may be important for algorithms that generate elements of the G set incrementally. Such a test may form the basis for algorithm termination.

However, determining whether F is equivalent to G can be problematic even for simple classes – we illustrate this difficulty for the class of monotone terms. We show that determining equivalence with G is as hard as determining if a given monotone CNF and DNF are equivalent. While the general complexity of the DNF/CNF equivalence problem is not known, it is unlikely to be NP-hard given Fredman and Khachyian's [FK96] quasi-polynomial time algorithm, *i.e.*, runs in time $O(\ell^{o(\log \ell)})$ where $\ell =$ |terms in DNF| + |clauses in CNF|.

Theorem 3.2 *Let C denote the class of monotone terms. Let N be a set of negative examples and G_N be the corresponding maximally general set. Given N and a set of terms F, determining if $F = G_N$ is as hard as determining if a given monotone CNF and monotone DNF are equivalent.*

Proof: We show how an algorithm for solving $F = G_N$ can be used to solve the CNF/DNF equivalence problem. Let C (respectively D) be the monotone CNF/DNF formulas that we want to test for functional equivalence. We transform C into a collection of negative examples as follows: for each clause c in C, create a negative example with 0s wherever there is a variable in c and 1s everywhere else. We transform D into G in the expected way, namely G is the set of terms in D.

We now show that G_N corresponds to the DNF description of C. By definition, the maximally general terms G_N for a set of negative examples N are those terms t for which t is consistent with N and for which any generalization of t is not consistent with N. (A term is made more general by dropping one or more literals.) Also, the reduced DNF description D_C of C (or any monotone function) can be obtained by identifying those terms t for which $t \rightarrow C$ and $t' \not\rightarrow C$ for any generalization t' of t. It is evident then that if we choose N to be the maximally negative examples of C that t is a term in D_C if and only if $t \in G_N$: Since $t \rightarrow C$, every negative example of C is also a negative example of t. Thus, if t is a term in D_C then by definition t is consistent with N. Further, if t is a term in D_C, then any generalization of t is not consistent

with N, since such a generalization t' of t consistent with N would imply that $t' \rightarrow C$, contradicting the assumption that t' is in D_C. The proof that if $t \in G_N$ then t is a term in D_C is similar.

Since F is essentially D and we've shown that G_N is essentially the DNF description of C, it follows that $F = G_N$ if and only if $C = D$: If $F = G_N$ then since G_N is the DNF description of C, then F must also equal the DNF description of C and since F is essentially D, we have that $C = D$. In the case that $F \neq G_N$, we know that the DNF description of C is not equivalent to F (D), and thus $C \neq D$. □

Cardinality of G. If the cardinality of G is very large, it may not be worth constructing since it will take too long to generate. Thus in some circumstances, having an algorithm that can determine the cardinality of G may save wasteful computation time.

However, beyond identifying G or determining if we have G, simply determining the cardinality of G can be problematic. We show that determining $|G|$ is #P-hard for the class of monotone terms via a reduction from counting the number of minimal vertex covers [GJ79] in a graph. The latter problem is known to be #P-hard [Val79].

Theorem 3.3 *Determining $|G|$ is #P-hard for the class of monotone terms.*

Proof: Given a graph (V, E), we construct a set of negative examples N such that the set of maximally general conjunctive concepts (*i.e.*, the set G) has a one to one correspondence with the set of minimal vertex covers, thus showing that determining the cardinality of $|G|$ is #P-hard.

For each vertex i of the graph we create a Boolean variable x_i. For each edge (i, j) in the edge set E of the graph, let n_{ij} be the assignment with bits i and j off and the remaining bits on. Let $N = \{n_{ij} : (i, j) \in E\}$ be all such negative examples. We show that c is a monotone conjunctive concept consistent with N if and only if $V_c = \{i : x_i \text{ appears in } c\}$ is a vertex cover of the graph. If c is consistent with N then for each point $n_{i,j} \in N$, c contains either x_i or x_j (recall c is monotone). So, by construction, for each edge $(i, j) \in E$, V_c contains either vertex i or j, and thus V_c is a vertex cover. Conversely, if $V' \subseteq V$ is a vertex cover, then the term $c = \wedge_{v \in V'} v$ does not cover any element of N since to avoid covering some n_{ij}, c contains either x_i or x_j.

In the construction, the variables of a consistent term correspond to the vertices in a vertex cover. Consequently, the variables in a maximally general term correspond to vertices in a *minimal* vertex cover. So counting the minimal vertex covers is exactly the problem of determining $|G|$. □

This theorem provides an interesting contrast to the results of Smith and Rosenbloom [SR90]. Their results imply that, when learning monotone terms, G must be a singleton set if learning from a set of data where every negative example is a "near miss" (each has only one negated literal and the rest are positive). Instead, if we call data where every negative example has exactly two negated literals and the rest positive "almost near misses", then even in the restricted case of learning monotone terms with only almost near misses, not only is the G set no longer singleton, computing G may require time exponential in $|P| + |N|$ (by the construction above, since the number of minimal vertex covers can be exponential in the size of the input), and, as just shown, even simply determining G's size is #P-hard.

Finally, since the overall size of the version space is an upper bound on the size of G, one might instead try determining the number of concept definitions in the overall version space. If this number is small, so, too, is the size of G. Unfortunately, counting the total number of all vertex covers (whether minimal or not) is also #P-hard, and thus the construction in the proof above also demonstrates that counting the number of elements in a version space is #P-hard.

Ill-Defined Boundary Sets: Halfspaces. The preceding concept classes had finite cardinality, hence infinite boundary sets were not an issue. Consider the concept class of (open or closed) halfspaces over two real-valued variables x and y. Initially, G is the halfspace that includes the whole space (representable by $x < \infty$) and S is the empty halfspace (representable by $x \geq \infty$).

Suppose one positive and one negative example are given, *e.g.*, $P = \{(0, 0)\}$ and $N = \{(1, 0)\}$. The S set for this data would have an infinite number of concepts, since there are an infinite number of (closed) halfspaces that are incomparable but pass through $(0, 0)$ and exclude $(1, 0)$ (each is of the form $y \geq mx$

for some $m > 0$ or $y \leq mx$ for some $m < 0$). A similar argument shows that there are an infinite number of concepts in G, since there are an infinite number of (open) halfspaces that are incomparable that pass through (1,0) that include (0,0) (namely, for every $b \neq 0$ the open halfspace that goes through $(1, 0)$ and $(0, b)$ with the orientation of the half space including $(0, 0)$).

The preceding examples gave cases where boundary sets would have an infinite number of concepts. The same argument yields cases where boundary sets need not even be well-defined. If the concept class was solely *open* halfspaces then S is undefined and if the concept class was *closed* halfspaces then G is undefined — these are examples of *inadmissible* concept classes [Mit82].

4 Consistency is Key

In this section we prove one of the main results of the paper - that the consistency problem is equivalent to a variety of other version-space problems (defined in Section 2).

Theorem 4.1 *For concept class C, the following are equivalent:*

(i) *The consistency problem for C is efficiently computable.*

(ii) *There is an efficient algorithm for testing whether the version space for C collapses.*

(iii) *C is efficiently version-space predictable.*

(iv) *The version space for C is efficiently subset-testable.*

(v) *The version space for C is efficiently equivalence-testable.*

Proof: We show that the consistency problem is equivalent to each of the others.
(i) iff (ii) Determining whether there exists a concept $c \in C$ consistent with P and N is exactly the problem of determining whether the version space $C_{P,N}$ has not yet collapsed.

(i) iff (iii) An algorithm for the consistency problem can be used to solve the version space prediction problem, *i.e.* to predict $C_{P,N}(x)$ as follows:

1. Run the consistency algorithm on inputs $P \cup \{x\}, N$ and then again on $P, N \cup \{x\}$.
2. If (a) Both fail then output $\emptyset$ (b) Only the first succeeds then output "+" (c) Only the second succeeds then output "-" (d) Both succeed output "?".

To see why this works note that: (a) If both fail then no concept consistent with P and N classifies x positive, since $C_{P\cup\{x\},N} = \emptyset$. Further, no concept consistent with P and N classifies x negative, since $C_{P,N\cup\{x\}} = \emptyset$. This can only happen if there are no consistent concepts, *i.e.*, if $C_{P,N}$ is itself empty. (b) If only the first succeeds then since the second failed, that is $C_{P,N\cup\{x\}} = \emptyset$, we know that no concept in $C_{P,N}$ classifies x negative. As a result, every concept in $C_{P,N}$ must classify x positive. Thus $C_{P,N}(x) =$ "+". (c) Similar to part b. (d) If both succeed then there is a concept in $C_{P,N}$ that classifies x positive and another concept in $C_{P,N}$ that classifies x negative. Thus $C_{P,N}(x) =$ "?".

Observe that if the consistency algorithm is efficient, *i.e.*, runs in time $p(|P|, |N|)$, where p is some polynomial, then the version-space prediction algorithm runs in time $p(|P \cup \{x\}|, |N|) + p(|P|, |N \cup \{x\}|)$, and is also efficient.

Conversely, if C is version-space predictable then, by definition, the consistency problem is efficiently computable. To determine if there is a concept in C consistent with P and N, use the version-space prediction algorithm to classify $C_{P,N}(x)$ for an arbitrary example x in X. There is no concept consistent with P and N if and only if the version-space prediction algorithm outputs "$\emptyset$", *i.e.*, if the version space has collapsed.

(i) iff (iv) Suppose we have an efficient consistency-testing algorithm. Given (P_1, N_1) and (P_2, N_2), we show how to test whether $C_{P_1,N_1} \subseteq C_{P_2,N_2}$. First, test if $C_{P_1,N_1} = \emptyset$. If so, then $C_{P_1,N_1} \subseteq C_{P_2,N_2}$. Otherwise,

check that for every $p \in P_2$ $C_{P_1,N_1}(p) =$ "+", and for every $n \in N_2$, $C_{P_1,N_1}(n) =$ "−". (Such a check can be performed since we have just shown (i) iff (iii).) If all of these hold, then every concept in C_{P_1,N_1} classifies P_2 and N_2 correctly, and $C_{P_1,N_1} \subseteq C_{P_2,N_2}$. If one of these tests fails, then for some $p \in P_2, C_{P_1,N_1}(p) =$ "−"or "?", or, for some $n \in N_2$, $C_{P_1,N_1}(n) =$ "+"or "?". In the first case (second case similar), either all $c \in C_{P_1,N_1}$ have $c(p) =$ "−", or for some c and c' in C_{P_1,N_1}, $c(p) =$ "−"and $c'(p) =$ "+". Regardless, some $c \in C_{P_1,N_1}$ classifies $p \in P_2$ incorrectly, and C_{P_1,N_1} is not a subset of C_{P_2,N_2}.

Conversely, to use an efficient test for $C_{P_1,N_1} \subseteq C_{P_2,N_2}$ as the basis for efficiently solving the consistency problem, note that $C_{P,N} = \emptyset$ if and only if for arbitrary $x \in X$, $C_{P,N} \subseteq C_{\{x\},\{x\}}$ since $C_{\{x\},\{x\}}$ is an empty version space.

(i) iff (v)

To see that (i) implies (v), observe that two version spaces C_{P_1,N_1} and C_{P_2,N_2} are equal if and only if both $C_{P_1,N_1} \subseteq C_{P_2,N_2}$ and $C_{P_1,N_1} \supseteq C_{P_2,N_2}$, which by part (iv) can be efficiently decided if the consistency problem admits an efficient algorithm.

To see that (v) implies (i), note that there is a concept in C consistent with P and N if and only if, for an arbitrarily chosen x, $C_{P,N}$ is not equivalent to $C_{\{x\},\{x\}}$ (which is an empty version space). □

The subset testing problem and the equivalence testing problems have applications in incremental version space merging. Version spaces are simply sets, and thus one can ask whether the version space for one set of data is a subset of, or is equal to, the version space for a second set of data. At first this may appear to be a problem that is harder than the consistency problem — for example, one version space may be a subset of a second even if they are based on disjoint sets of training data. Initially it may seem necessary to enumerate the elements in both version spaces, or at least their boundary sets, to do subset-testing or equivalence testing. Theorem 4.1 shows that this is not so, and that consistency testing suffices.

5 New Tractable Classes

In this section we give classes that are version-space predictable. These classes are problematic for traditional version space approaches, as the sets G and/or S become too large to deal with efficiently. We skirt these issues entirely by ignoring representation of G and S, and instead "represent" them only implicitly with the data P and N. Giving efficient consistency algorithms then demonstrates the version-space predictability of the classes.

We'll begin with two classes, conjunctions of variables and conjunctions of Horn clauses, that are troublesome for the $[S, G]$ representation, although not for single-sided representations. Then we'll revisit the three classes, 1-Decision Lists, monotone decision trees and halfspaces, that we saw exhibited various pathologies (even in the single-sided case) in Section 3.

Conjunctions of Variables (Terms). The consistency problem for conjunctions of variables (or terms) is easily solvable via the naive algorithm of finding the term c that most specifically covers P [KV94]. If no negative example satisfies c then c is consistent with P and N. If some example in N does satisfy c , then it can be easily shown that no term is consistent with P and N. The procedure takes time linear in $|P|$ and $|N|$.

Conjunctions of Horn clauses. For conjunctions of Horn clauses (Horn sentences), it is possible to show with a small training sample that the size of one boundary set can grow exponentially large [AP95]. Nonetheless, the consistency problem for Horn sentences is efficiently computable. We demonstrate how to construct a Horn sentence H consistent with a given P and N whenever one exists. The idea is to construct for each negative example q a set of clauses that falsify q, and then remove from this set the clauses that also falsify examples in P. After this removal, the Horn sentence will necessarily be consistent with P, and, if it is not consistent with N, then it can be shown that no Horn sentence exists that can satisfy P and falsify N.

For an example q let $q(x_i)$ denote the value that q assigns variable x_i, that is, 1 if the i^{th} bit of q is 1, and 0 otherwise. Then define *ones(q)* $= \{\wedge x_i : q(x_i) = 1\}$. Define *zeros(q)* $= \{x_i : q(x_i) = 0\} \cup \{False\}$.

If q is a negative example, then the following is a Horn sentence that excludes q:

$$\text{clauses}(q) = \bigwedge_{z \in zeros(q)} (ones(q) \rightarrow z)$$

For example, if $q = 11001$, then clauses(q)= $((x_1 \wedge x_2 \wedge x_5) \rightarrow x_3) \wedge ((x_1 \wedge x_2 \wedge x_5) \rightarrow x_4) \wedge ((x_1 \wedge x_2 \wedge x_5) \rightarrow$ *False*). Consider the Horn sentence H obtained by conjoining *clauses(q)* for each q in N, and then removing any "bad" clauses that exclude points in P:

$$H = \bigwedge \{\alpha : (\exists q \in N)\ \alpha \in \text{ clauses}(q) \text{ and } (\forall p \in P)\ p \text{ satisfies } \alpha\}.$$

We show that H is consistent with P and N if and only if there exists H' consistent with P and N. One direction is easy: clearly if H is consistent with P and N then there exists H' consistent since H is such a Horn representation. In the other direction, we show that if H is not consistent with P and N, then no Horn sentence is. By definition of H, every $p \in P$ is consistent with H. Suppose that some $q \in N$ was not consistent with H, *i.e.*, q satisfies all clauses in H. Since q violates every clause of clauses(q), it must be the case that every clause of clauses(q) is violated by some $p \in P$, otherwise a clause that q violates would be in H and q would be properly classified as negative.

Any Horn sentence H' consistent with P and N must call q negative, hence must contain a clause $\beta \rightarrow \gamma$, where $\beta \subseteq$ ones(q) and $\gamma \in$ zeros(q). But the clause ones(q) $\rightarrow \gamma$ is in clauses(q), and is thus violated by some $p \in P$ as described just above. But if p violates ones(q) $\rightarrow \gamma$, then it also violates $\beta \rightarrow \gamma$, contradicting the consistency of H'.

The above procedure thus constructs a Horn sentence consistent with P and N if and only if there is a Horn sentence consistent with P and N. The running time is $O(|N||P|\ell)$, where ℓ is the number of variables.

1-Decision Lists, Monotone Depth-Two Formulas, Halfspaces. For the remaining three classes, we briefly describe solutions to the consistency problem.

Given a set of positive P and negative N examples, the following algorithm checks for 1-decision list consistency, and also outputs a consistent concept if one exists. Let x_i (or $\overline{x_i}$) be such that the points that satisfy x_i are either completely contained in P or completely contained in N. If no such literal exists, then there is no consistent concept and the algorithm stops. If such a literal exists, then an appropriate statement is added to the decision list, *i.e.*, if the variable x_i satisfies examples only in P (respectively N) then add the statement $(x_i, 1)$ (respectively, $(x_i, 0)$). Similarly for $\overline{x_i}$. The algorithm repeats with newly covered points removed from P (respectively, N). For further details, refer to Kearns and Vazirani's book [KV94] or Rivest's original paper on learning k-decision lists [Riv87].

Any monotone function can be represented as a monotone depth-two formula since monotone depth-two formulas contain the class of monotone DNF formulas. Thus, a pair of sets of examples P and N are consistent with some monotone depth-two formula iff P and N are consistent with some monotone function. This is accomplished in time $O(|P||N|)$ by verifying that there is no positive in P that falls below a negative in N on the hypercube.

For halfspaces, the consistency problem is efficiently solvable via an algorithm that checks if a given linear program possesses a feasible solution. Each positive and negative example essentially forms a constraint in the linear program. In two dimensions, for example, we want to determine if there exists coefficients A and B such that for each positive example $p = (p_x, p_y)$, $p_y \geq A \cdot p_x + B$ and for each negative example $n = (n_x, n_y)$, $n_y < A \cdot n_x + B$. A feasible solution exists for such a linear program if and only if there exists a halfspace consistent with P and N.

6 Other Version-Space Operations

The previous section showed the equivalence of key version-space reasoning operations: consistency/collapse, predictability, subset testing, and equivalence testing. The results imply that all these operations can be

tractably performed using a version space "representation" that consists of simply the sets P and N, as long as one has a tractable algorithm for consistency. Moreover, this representation permits the tractable use of these operations in a superset of cases than was possible using the original boundary-set representation. If we consider maintaining solely the P and N sets as a "minimalist" representation of a version space, it becomes possible to discuss the tractability of a range of other operations that are relevant to reasoning with version spaces.

Concept Membership: Given a version space $C_{P,N}$ and a concept c, determine if $c \in C_{P,N}$. This is trivially computed from the P, N representation by using c to classify each element of P and N and checking whether c is consistent. Notice that this, too, broadens the tractable usage of version spaces, in that performing this operation with boundary sets requires reasoning about the relative generality of concepts,[6] which can be intractable, whereas here it again simply reduces to the question of consistency. (Consider, for example, the case of context-free languages [VB87], in which computing relative generality is undecidable.)

Retraction: Noisy data is a common problem in many learning problems. Once learned, if an example is found to be incorrectly labeled, it is unclear how one can "retract" the example when a version space is represented by S and G [IA89]. In contrast, example retraction is trivial when $C_{P,N}$ is represented by P and N — simply remove the retracted examples from the appropriate set.

Update: To update a version space given new examples, one faces the same simplicity as for retraction — simply add the example to the appropriate set, P or N. This is in contrast to the more complex manipulations that must be performed with boundary sets (the "candidate elimination algorithm" [Mit82]).

Intersection: Intersecting two version spaces is also easy with the P, N representation since $C_{P_1,N_1} \cap C_{P_2,N_2} = C_{P_1 \cup P_2, N_1 \cup N_2}$. The new P and N are thus simply the unions of the individual P and N sets.[7]

Minimality: Finally, although the size of the representation of $C_{P,N}$ is exactly just the (relatively modest) size of maintaining all training data, our results suggests a way to reduce these space requirements in some cases. If for some $p \in P$, $C_{P-\{p\},N}(p) = +$ then p can be removed from P. Similarly, if for some $n \in N$, $C_{P,N-\{n\}}(n) = -$, then n can be removed from N. In other words, if the example is classified correctly by the version space that resulted from the removal of this example, the example provides no new information and can be deleted. Thus, a greedy algorithm maintains a minimal pair of sets $P' \subseteq P$ and $N' \subseteq N$ and ignores any training example x as it is processed unless $C_{P,N}(x) =$ "?".

There are concept classes C and sets P and N such that finding a *minimum* (in total cardinality) pair P' and N' is NP-hard. Suppose we have an algorithm that given P and N, can find a P', N' where $C_{P,N} = C_{P',N'}$ and $|P'| + |N'|$ is minimized. If $C_{P,N}$ is empty, *i.e.*, there are no consistent concepts, then when $P' = N' = \{x\}$ for any instance x, $|P'| + |N'|$ is minimized. Thus, if we had an algorithm that could output the minimum $|P'| + |N'|$ we could use that algorithm to solve the consistency problem for k-term DNF: run the minimum P', N' algorithm on input P, N and if $P' = N' = \{x\}$ for some instance x, then

[6] You must test whether c is above every element of S and below every element of G.

[7] Version spaces are simply sets, and thus in addition to intersection one can also consider the union and set difference of two sets. Unfortunately, although these two operations can be useful in learning with version spaces [Hir94], version spaces are not closed under either union or set difference — there may be no set of examples that gives a version space that is equal to the union or difference of two given version spaces.

However, it is worth noting in passing that one can consider storing both (P_1, N_1) and (P_2, N_2) as a "disjunctive representation" in such cases. To test consistency, note that there is a concept consistent with some pair from $(P_1, N_1), (P_2, N_2), \ldots, (P_n, N_n)$, if and only if there is one consistent with some (P_i, N_i). Prediction is handled via the following rules: If every C_{P_i,N_i} predicts "$\emptyset$", then predict $\emptyset$. If any predict "?", or if some C_{P_i,N_i} predicts "+"and another predicts "−", then predict "?"(because there are two concepts in the union that disagree). Otherwise, either all predict "−"or $\emptyset$, in which case we predict "−", or all predict "+"or $\emptyset$, in which case we predict "+". However, testing subset and equality appears more difficult.

there are no consistent concepts. Otherwise, there must be at least one consistent concept. Since finding a consistent k-term DNF is NP-hard [PV88], so is the problem of finding P', N' where $|P'|+|N'|$ is minimized.

The notion of minimality also has applications to function testing. For example, suppose our goal is to manufacture circuitry for a partially specified Boolean function. The specification is partial because there are many "don't care" input combinations for which the output is irrelevant. A good test suite P, N for the partial function is a pair such that $C_{P,N}$ contains all and only extensions of the function. The choice of a small test suite is a difficult one (typically NP-hard). Our results suggest some heuristics: Say that (P_1, N_1) dominates (P_2, N_2) as a test set if $C_{P_1,N_1} \subset C_{P_2,N_2}$. As we showed above, this can be tested when consistency is tractable. A reasonable greedy approach to finding good test suites would be to search for minimal P, N pairs, and among them, discard any that are dominated by any other pair.

7 Beyond Consistency

While the approach demonstrated in this paper is of broad use, not everything boils down to the consistency problem. For example, while it is easy to determine if a version space has converged with boundary sets, there is no obvious way to determine convergence with a consistency algorithm. In addition, we have no way of working with concept classes where solving the consistency problem is hard – although, by our observations, such classes are not version space predictable. We now explore what happens when we move beyond consistency.

Convergence. Recall that a version space $C_{P,N}$ is said to have *converged* if there is exactly one concept in C consistent with P and N, *i.e.*, if $|C_{P,N}| = 1$. Observe that the existence of an efficient algorithm for the consistency problem for C does *not* necessarily imply the existence of an efficient algorithm for the convergence problem for C. Intuitively, it is "harder" to determine if there is *exactly* one consistent concept (*i.e.*, convergence) than determine if there is *any* consistent concept.

To instantiate this intuition, we note that for the class C of monotone formulas, the consistency problem is efficiently computable. (Simply check if any positive example falls "below", in the Boolean hypercube, a negative, and vice versa.) However, it is possible to show that the convergence problem is equivalent to determining if a monotone DNF formula is equivalent to a monotone CNF formula. While there are efficient solutions to restricted versions of the equivalence of monotone DNF and CNF problem [EG95, JPY88, LLK80, DMP99, EGM03], the best known algorithm for the general problem runs in quasipolynomial time [FK96]. So while there is a polynomial-time algorithm for the consistency problem for monotone formulas, the convergence problem would appear to be harder.

That convergence appears to be difficult has much less import inasmuch as convergence testing does not play as important a role in learning with version spaces as it initially appeared to in Mitchell's work. In particular, convergence usually requires a very large number of examples: the smaller an unconverged version space is, the longer the wait for a random example that can distinguish them [Hau88]. Haussler also showed that under the PAC learning criteria, it is not necessary to generate a converged version space, since any element of a version space for some number of randomly chosen examples would perform comparably well on future data. It is thus unusual to wait until a version space becomes singleton, as opposed to, say, selecting a random element of the version space for classification purposes [NH92].

When Consistency is NP-hard. By applying Theorem 4.1 in the other direction, we have that if the consistency problem for C is NP-hard, then C is not version-space predictable, unless $P = NP$. For example, since results of Pitt and Valiant [PV88] show that the consistency problem for k-term DNF formulas is NP-hard, this class is not version-space predictable, unless $P = NP$. However (as their work goes on to suggest), we can still use version spaces for this concept class if we use the richer knowledge representation class of k-CNF formulas since it includes k-term DNF formulas and there is a tractable solution to the consistency problem for k-CNF formulas.

8 Final Remarks

Boundary set representations can be difficult to manipulate due to the fact that they can be large, undefined, infinite, and even hard to test for equality or cardinality. We make the simple but powerful observation that many of the common version space operations can be performed with a tractable consistency algorithm, and without boundary sets. We establish key equivalences between the consistency problem, version space collapse, version space predictability, concept class subset-testability and equivalence testability. We demonstrate that many new concept classes are version-space predictable due to efficient consistency algorithms, like 1-decision lists, monotone depth-two formulas, and halfspaces — classes that are problematic for boundary set representations.

One direction for the future is to explore how this work relates to other work on broadening version spaces. In particular, Lau *et al.* [LWDW03] consider a compositional approach to version spaces as well as their use on multi-class problems. It would be interesting to explore whether our insights concerning the centrality of consistency may also be relevant in their broader notion of version spaces. Also interesting is the relationship of version-space predictability to query-by-committee [SOS92] and co-training [BM98]. Both methods can be viewed as performing a noisy form of version-space predictability. In the case of query-by-committee the labels assigned by two randomly selected concepts are used to make a quick check whether labeling an example is likely to give any leverage in learning, as opposed to the more categorical assessments version-space prediction makes. In the case of co-training, different "views" of the data lead to separate learning tasks for the same problem. If an unlabeled example is likely to be labeled with an unambiguous class by one view's learning, it is labeled and provided to the other learning view. Version-space prediction could be used with either view to categorically determine cases where an unlabeled example has an unambiguous label, to be labeled and provided as labeled data for the other learning view. We leave these as questions for future work.

References

[AHHP98] H. Aizenstein, T. Hegedus, L. Hellerstein, and L. Pitt. Complexity theoretic hardness results for query learning. *Computational Complexity*, 7(1):19–53, 1998.

[AP95] H. Aizenstein and L. Pitt. On the learnability of disjunctive normal form formulas. *Machine Learning*, 19(3):183–208, 1995.

[BEHW89] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *J. ACM*, 36(4):929–965, 1989.

[BM98] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT-98)*, pages 92–100, New York, July 24–26 1998. ACM Press.

[DMP99] C. Domingo, N. Mishra, and L. Pitt. Efficient read-restricted monotone CNF/DNF dualization by learning with membership queries. *Machine Learning*, 37(1):89–110, 1999.

[EG95] T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278–1304, December 1995.

[EGM03] T. Eiter, G. Gottlob, and K. Makino. New results on monotone dualization and generating hypergraph transversals. *SIAM Journal on Computing*, 32(2):514 – 537, 2003.

[EIM98] T. Eiter, T. Ibaraki, and K. Makino. Decision lists and related boolean functions. *IFIG Research Report 9804*, April 1998.

[FK96] M. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, November 1996.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.

[Hau88] D. Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[Hau89] D. Haussler. Learning conjunctive concepts in structural domains. *Machine Learning*, 4(1):7–40, 1989.

[Hir92] H. Hirsh. Polynomial-time learning with version spaces. In William Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 117–122, San Jose, CA, July 1992. MIT Press.

[Hir94] H. Hirsh. Generalizing version spaces. *Machine Learning*, 17(1):5–46, 1994.

[IA89] P. Idestam-Almquist. Demand networks: An alternative representation of version spaces. SYS-LAB Report 75, Department of Computer and Systems Sciences, The Royal Institue of Technology and Stockholm University, 1989.

[JPY88] D. Johnson, C. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[KV94] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Cambridge, Massachusetts, 1994.

[LLK80] E. Lawler, J. Lenstra, and A. Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[LWDW03] T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[Mit82] T. M. Mitchell. Generalization as search. *Art. Int.*, 18:203–226, 1982.

[Mit97] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[NH92] S. W. Norton and H. Hirsh. Classifier learning from noisy data as probabilistic evidence combination. In William Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 141–146, San Jose, CA, July 1992. MIT Press.

[PV88] L. Pitt and L. Valiant. Computational limitations on learning from examples. *J. ACM*, 35:965–984, 1988.

[Riv87] R. L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[Smi95] B. Smith. Induction as knowledge integration. In *Ph.D. thesis*. University of Southern California, Computer Science Department, December 1995.

[SOS92] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In David Haussler, editor, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 287–294, Pittsburgh, PA, July 1992. ACM Press.

[SR90] B. D. Smith and P. S. Rosenbloom. Incremental non-backtracking focusing: A polynomially bounded generalization algorithm for version spaces. In *Proceedings of the National Conference on Artificial Intelligence*, pages 848–853, Boston, MA, August 1990.

[Utg86] P. E. Utgoff. *Machine Learning of Inductive Bias*. Kluwer, Boston, MA, 1986.

[Val79] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[VB87] K. VanLehn and W. Ball. A version space approach to learning context-free grammars. *Machine Learning*, 2(1):39–74, 1987.

A Proof of Theorem 3.1

Recall that $P = \{p_i : i = 1, \ldots, n\}$ and $N = \{n_i : i = 1, \ldots, n\}$. where $p_i = 0_{2i}\vec{1}$ and $n_i = \vec{0}1_{2i}$ over the variables $u_1, v_1, \ldots u_n, v_n, x_1, y_1, \ldots, x_n, y_n$. For example, in the case that $n = 3$, P and N are:

$$P = \left\{ \begin{array}{c|ccc|ccc} u_1v_1 & u_2v_2 & u_3v_3 & x_1y_1 & x_2y_2 & x_3y_3 \\ \hline 11 & 00 & 00 & 11 & 11 & 11 \\ 00 & 11 & 00 & 11 & 11 & 11 \\ 00 & 00 & 11 & 11 & 11 & 11 \end{array} \right\} N = \left\{ \begin{array}{c|ccc|ccc} u_1v_1 & u_2v_2 & u_3v_3 & x_1y_1 & x_2y_2 & x_3y_3 \\ \hline 00 & 00 & 00 & 00 & 11 & 11 \\ 00 & 00 & 00 & 11 & 00 & 11 \\ 00 & 00 & 00 & 11 & 11 & 00 \end{array} \right\}$$

We will argue that for P and N as given that for each permutation $[i_1, \ldots, i_n]$ of $[1, \ldots, n]$, the boundary sets are:

$$S = (\overline{x_1}, 0), (\overline{y_1}, 0), \ldots, (\overline{x_n}, 0), (\overline{y_n}, 0), (a_{i_1}, 1), (b_{i_1}, 0), (a_{i_2}, 1), (b_{i_2}, 0), \cdots (a_{i_{(n-1)}}, 1), (b_{i_{(n-1)}}, 0), (\overline{a_{i_n}}, 0), (\overline{b_{i_n}}, 0), 1$$

$$G = (u_1, 1), (v_1, 1), \ldots, (u_n, 1), (v_n, 1), (\overline{c_{i_1}}, 0), (\overline{d_{i_1}}, 1), (\overline{c_{i_2}}, 0), (\overline{d_{i_2}}, 1), \cdots, (\overline{c_{i_{(n-1)}}}, 0), (\overline{d_{i_{(n-1)}}}, 1), (c_{i_n}, 1), (d_{i_n}, 1), 0$$

where $a_{ik} \in \{u_{ik}, v_{ik}\}$ and $a_{ik} = u_{ik}$ (resp, v_{ik}) implies that $b_{ik} = v_{ik}$ (resp, u_{ik}) for $k = 1, \ldots, n$. Similarly, $c_{ik} \in \{x_{ik}, y_{ik}\}$ and $c_{ik} = x_{ik}$ (resp, y_{ik}) implies that $d_{ik} = y_{ik}$ (resp, x_{ik}). Observe that there are 2^n possible 1-decision lists for each permutation of $1, \ldots, n$. Thus,

$$\begin{array}{ll} S = & (\overline{x_1}, 0), (\overline{y_1}, 0), \ldots, (\overline{x_n}, 0), (\overline{y_n}, 0), \\ & (a_{i_1}, 1), (b_{i_1}, 0), \\ & (a_{i_2}, 1), (b_{i_2}, 0), \\ & \vdots \\ & (a_{i_{n-1}}, 1), (b_{i_{n-1}}, 0), \\ & (\overline{a_{i_n}}, 0), (\overline{b_{i_n}}, 0), \\ & 1 \end{array} \qquad \begin{array}{ll} G = & (u_1, 1), (v_1, 1), \ldots, (u_n, 1), (v_n, 1), \\ & (\overline{c_{i_1}}, 0), (\overline{d_{i_1}}, 1), \\ & (\overline{c_{i_2}}, 0), (\overline{d_{i_2}}, 1), \\ & \vdots \\ & (\overline{c_{i_{n-1}}}, 0), (\overline{d_{i_{n-1}}}, 1), \\ & (c_{i_n}, 1), (d_{i_n}, 1), \\ & 0 \end{array}$$

where $a_{i_k} \in \{u_{i_k}, v_{i_k}\}$ and $a_{i_k} = u_{i_k}$ (resp, v_{i_k}) implies that $b_{i_k} = v_{i_k}$ (resp, u_{i_k}) for $k = 1, \ldots, n$. Similarly, $c_{i_k} \in \{x_{i_k}, y_{i_k}\}$ and $c_{i_k} = x_{i_k}$ (resp, y_{i_k}) implies that $d_{i_k} = y_{i_k}$ (resp, x_{i_k}). Observe that there are 2^n possible 1-decision lists for each permutation of $1, \ldots, n$. Thus, the number of different functions in both S and G is at least $(n!\, 2^n)$.

Before explaining why S and G are in fact maximally specific/general, we describe some properties of 1-DLs that we'll use in our argument. In particular, since Boolean formula descriptions are easier to manipulate than 1-DLs, we describe Eiter et al's [EIM98] observation that the class of 1-DLs is equivalent to the class of linear read-once formulas. The class of *linear read-once* formulas F_{LR1} is defined as follows:

1. $\top, \bot \in F_{LR1}$ and
2. $\phi \in F_{LR1}$ and x_i is a variable not occurring in ϕ implies $x_i \vee \phi, \overline{x_i} \vee \phi, x_i \wedge \phi$ or $\overline{x_i} \wedge \phi \in F_{LR1}$.

Not every read-once formula is linear, *e.g.*, $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$. Note that any linear read-once formula can be converted into a 1-DL, and vice versa in linear time. For example, for $n = 3$, one element of the G set

$$g = (u_1, 1), (v_1, 1), (u_2, 1), (v_2, 1)(u_3, 1), (v_3, 1), (\overline{x_1}, 0), (\overline{y_1}, 1), (\overline{x_2}, 0), (\overline{y_2}, 1), (x_3, 1), (y_3, 1), 0$$

can be expressed as the linear read-once formula

$$g = u_1 \vee v_1 \vee u_2 \vee v_2 \vee u_3 \vee v_3 \vee (x_1 \wedge (\overline{y_1} \vee (x_2 \wedge (\overline{y_2} \vee x_3 \vee y_3))))$$

The characterization of 1-DLs as linear read-once formulas yields the following recursive method of recognizing a 1-DL. Let C_{LR1} denote the class of linear read-once functions. Let $f_{x \leftarrow 1}$ denote the projection of f where the variable x is forced to 1 (similarly for $f_{x \leftarrow 0}$).

Lemma A.1 (EIM) *A function f is in C_{1DL} iff (i) Either (a) $\overline{x_j} \rightarrow f$ (b) $\overline{x_j} \rightarrow \overline{f}$ (c)$x_j \rightarrow f$ or (d) $x_j \rightarrow \overline{f}$ for some j and (ii) $f_{x_j \leftarrow 1} \in C_{1DL}$ holds for all j satisfying (i)(a) and (i)(b), and $f_{x_j \leftarrow 0} \in C_{1DL}$ holds for all j satisfying (i)(c) and (i)(d).*

The lemma gives us a way of showing that elements of the G set indicated above are in fact maximally general (similarly for S). By assuming the existence of a more general formula g', we show by repeatedly projecting g and g' appropriately that a variable x implies g (resp, $\overline{g}$) iff x implies g' (resp, $\overline{g'}$). It will follow that $g' = g$, contradicting that g' is strictly more general than g.

We now proceed to prove the theorem.

Theorem A.1 *For the concept class of 1-DLs and for P, N given above, G/S are maximally general/specific.*

Proof: We show for one permutation g of $[i_1, \ldots, in]$, namely $[1, \ldots, n]$ and one c_i/d_i assignment that g is maximally general. A similar argument can be made for any other permutation and c_i/d_i assignment. In particular let g be:

$$g = (u_1, 1), (v_1, 1), \ldots, (u_n, 1), (v_n, 1), (\overline{x_1}, 0), (\overline{y_1}, 1), \ldots, (\overline{x_{n-1}}, 0), (\overline{y_{n-1}}, 1), (x_n, 1), (y_n, 1), 0$$

Our strategy for showing that g is maximally general is the following. Assume there exists a linear read-once formula g' strictly more general than g and consistent with P and N. We show that a variable x implies g (resp, $\overline{g}$) iff x implies g' (resp, $\overline{g'}$). Suppose there is an x such that x implies g iff x implies g'. Then for half of the points in the hypercube, namely the ones where the variable x is set on, both g and g' agree. In particular, both g and g' classify positive those points with bit x set on. In this case, we project the bit x off and repeat the argument in the other half of the hypercube, *i.e.*, we show that a variable y implies $g_{x \leftarrow 0}$ (resp, $\overline{g_{x \leftarrow 0}}$) iff y implies $g'_{x \leftarrow 0}$ (resp, $\overline{g'_{x \leftarrow 0}}$). On the other hand, suppose there is a variable x such that x implies $\overline{g}$ iff x implies $\overline{g'}$. Thus, again $\overline{g}$ and $\overline{g'}$ agree on the classification of half of the points in the hypercube, in particular $\overline{g}$ and $\overline{g'}$ classify any point with bit x set on negative. Thus we project the bit x off in both g and g' and repeat the argument in the other half of the hypercube. By Lemma A.1, we will always be able to find such a variable in one of $g, \overline{g}$ and also one of $g', \overline{g'}$. At the end of our argument, we will have shown that a variable x implies f (resp, $\overline{f}$) iff x implies f' (resp, $\overline{f'}$) where f is g and also every projection of g and likewise for f'. It will follow that $g' = g$, contradicting that g' is strictly more general than g.

In order to show that a variable x implies g or $\overline{g}$, we make reference to g's ($\overline{g}$'s) DNF (CNF) representation. By Eiter et al's [EIM98] linear read-once characterization of 1-DLs, we have that the DNF representation of g is:

$$\begin{aligned} g \quad = \quad & u_1 \vee v_1 \vee \cdots \vee u_n \vee v_n \vee (x_1 \wedge \overline{y_1}) \vee (x_1 \wedge x_2 \wedge \overline{y_2}) \vee \cdots \vee (x_1 \wedge \cdots \wedge x_{n-1} \wedge \overline{y_{n-1}}) \vee \\ & (x_1 \wedge \cdots \wedge x_n) \vee (x_1 \wedge \cdots \wedge x_{n-1} \wedge y_n) \end{aligned}$$

Note that a variable x implies a DNF formula f iff $f = x \vee \phi$ where ϕ is some reduced[8] DNF formula. By DeMorgan's Law, we have that the CNF representation of $\overline{g}$ is:

$$\begin{aligned} \overline{g} \quad = \quad & (\overline{u_1}) \wedge (\overline{v_1}) \wedge \cdots \wedge (\overline{u_n}) \wedge (\overline{v_n}) \wedge (x_1 \rightarrow y_1) \wedge ((x_1 \wedge x_2) \rightarrow y_2) \wedge \cdots \wedge ((x_1 \wedge \cdots \wedge x_{n-1}) \rightarrow y_{n-1}) \wedge \\ & ((x_1 \wedge \cdots \wedge x_n) \rightarrow \bot) \wedge ((x_1 \wedge \cdots \wedge x_{n-1} \wedge y_n) \rightarrow \bot) \end{aligned}$$

Note that a variable x implies a CNF formula f iff x appears in each conjunct of the reduced[9] CNF, *i.e.*, $f = (x \vee c_1) \wedge (x \vee c_2) \wedge \cdots \wedge (x \vee c_j)$ where c_i are disjunctions of variables.

From the DNF description of g, it is clear that the only variables that imply g are u_i and v_i. To see why the variables u_i and v_i imply g', note that $g \rightarrow g'$, thus if any variable implies g it also implies g'. To

[8] A DNF formula is reduced if no minterm is subsumed by another. It can be shown that any 1-DL has a unate DNF description and thus there exists a unique reduced DNF description.

[9] A CNF formula is reduced if no disjunction subsumes another. Since 1-DLs have unate DNF descriptions, their complements have unate CNF descriptions. Thus the complements of 1-DLs have unique reduced CNF descriptions.

see why the remaining variables don't imply g' note that if x_i, y_i implied g' then g' would not be consistent with n_j ($j \neq i$) and further if $\overline{x_i}$ or $\overline{y_i}$ implied g' then g' would not be consistent with n_i. Further, from the CNF description of $\overline{g}$, it is clear that no variable implies $\overline{g}$. It immediately follows that no variable implies $\overline{g'}$ since $g \rightarrow g'$ implies that $\overline{g'} \rightarrow \overline{g}$. Thus the fact that $x \not\rightarrow \overline{g}$ implies that $x \not\rightarrow \overline{g'}$.

We now project the variables $u_1, \ldots, u_n, v_1, \ldots, v_n$ to 0 in both g and g' and continue the argument in the projected space. We can now ignore the half of the hypercube where $u_1, \ldots, u_n, v_1, \ldots, v_n$ are set on since both g and g' classify all such points positive. Let $h = g_{u_1 \leftarrow 0, \ldots, u_n \leftarrow 0, v_1 \leftarrow 0, \ldots, v_n \leftarrow 0}$ and similarly for h'. We can project the descriptions of g and $\overline{g}$ accordingly.

$$\begin{aligned} h &= (x_1 \wedge \overline{y_1}) \vee (x_1 \wedge x_2 \wedge \overline{y_2}) \vee \cdots \vee (x_1 \wedge \cdots \wedge x_{n-1} \wedge \overline{y_{n-1}}) \vee \\ &\quad (x_1 \wedge \cdots \wedge x_n) \vee (x_1 \wedge \cdots \wedge x_{n-1} \wedge y_n) \\ \overline{h} &= (x_1 \rightarrow y_1) \wedge ((x_1 \wedge x_2) \rightarrow y_2) \wedge \cdots \wedge ((x_1 \wedge \cdots \wedge x_{n-1}) \rightarrow y_{n-1}) \wedge \\ &\quad ((x_1 \wedge \cdots \wedge x_n) \rightarrow \bot) \wedge ((x_1 \wedge \cdots \wedge x_{n-1} \wedge y_n) \rightarrow \bot) \end{aligned}$$

From the description of h, it is clear that no variable implies h. To see why no variable implies h' observe that: If this variable was x_i or y_i, then h' would not be consistent with n_j, $j \neq i$. If this variable was $\overline{x_i}$ or $\overline{y_i}$ then h' would not be consistent with n_i. From the description of $\overline{h}$, the only variable that implies $\overline{h}$ is $\overline{x_1}$. Thus, for all other variables x, $x \not\rightarrow \overline{h'}$ since $\overline{h'} \rightarrow \overline{h}$ and $x \not\rightarrow \overline{h}$. Since some variable must imply $\overline{h'}$ or h' in order for h' to be representable as a 1-DL and we've ruled out all the other variables, $\overline{x_1}$ must imply $\overline{h'}$.

We now project $x_1 \leftarrow 1$ in both h and h' and continue the argument in the projected space. The descriptions of $h_{x_1 \leftarrow 1}$ and $\overline{h_{x_1 \leftarrow 1}}$ are now

$$\begin{aligned} h_{x_1 \leftarrow 1} &= \overline{y_1} \vee (x_2 \wedge \overline{y_2}) \vee \cdots \vee (x_2 \wedge \cdots \wedge x_{n-1} \wedge \overline{y_{n-1}}) \vee \\ &\quad (x_2 \wedge \cdots \wedge x_n) \vee (x_2 \wedge \cdots \wedge x_{n-1} \wedge y_n) \\ \overline{h_{x_1 \leftarrow 1}} &= (y_1) \wedge (x_2 \rightarrow y_2) \wedge \cdots \wedge ((x_2 \wedge \cdots \wedge x_{n-1}) \rightarrow y_{n-1}) \wedge \\ &\quad ((x_2 \wedge \cdots \wedge x_n) \rightarrow \bot) \wedge ((x_2 \wedge \cdots \wedge x_{n-1} \wedge y_n) \rightarrow \bot) \end{aligned}$$

Note that $\overline{y_1}$ is the only variable that implies $h_{x_1 \leftarrow 1}$. $\overline{y_1}$ therefore also implies $h'_{x_1 \leftarrow 1}$ since $h'_{x_1 \leftarrow 1}$ is more general than $h_{x_1 \leftarrow 1}$. No other variable implies $h'_{x_1 \leftarrow 1}$: if that variable was x_i or y_i then $h'_{x_1 \leftarrow 1}$ would not be consistent with n_j, for $j \neq i$; if that variable was $\overline{x_i}$ or $\overline{y_i}$ then $h'_{x_1 \leftarrow 1}$ would not be consistent with n_i. From the CNF description of $\overline{h_{x_1 \leftarrow 1}}$, no variable implies $\overline{h_{x_1 \leftarrow 1}}$. Thus, since $\overline{h'_{x_1 \leftarrow 1}}$ implies $\overline{h_{x_1 \leftarrow 1}}$, no variable implies $\overline{h'_{x_1 \leftarrow 1}}$ either.

Let $f^i = h_{x_1 \leftarrow 1, \ldots, x_i \leftarrow 1, y_1 \leftarrow 1, \ldots, y_i \leftarrow 1}$ and $(f')^i = h'_{x_1 \leftarrow 1, \ldots, x_i \leftarrow 1, y_1 \leftarrow 1, \ldots, y_i \leftarrow 1}$. The argument given for h and h' can be repeated for f^i and $(f')^i$. In particular, it can be shown that: no variable implies f^i or $(f')^i$, the only variable that implies f^i is $\overline{x_{i+1}}$, and the only variable that implies $(f')^i$ is $\overline{x_{i+1}}$. Further, an argument similar to the one given for $h_{x_1 \leftarrow 1}$ and $h'_{x \leftarrow 1}$ can be used for $f^i_{x_{i+1} \leftarrow 1}$ and $(f')^i_{x_{i+1} \leftarrow 1}$, namely that: $\overline{y_{i+1}}$ is the only variable that implies $f^i_{x_{i+1} \leftarrow 1}$, $\overline{y_{i+1}}$ is the only variable that implies $(f')^i_{x_{i+1} \leftarrow 1}$, and no variable implies either $\overline{f^i_{x_{i+1} \leftarrow 1}}$, or $\overline{(f')^i_{x_{i+1} \leftarrow 1}}$.

After the above projections, we have the following DNF/CNF descriptions for f^{n-1} and $\overline{f^{n-1}}$.

$$f^{n-1} = x_n \vee y_n$$

$$\overline{f^{n-1}} = (x_n \rightarrow \bot)(y_n \rightarrow \bot)$$

From the DNF description of f^{n-1}, we have that both x_n and y_n imply f^{n-1}. Thus, since $f^{n-1} \rightarrow (f')^{n-1}$ we have that both x_n and y_n imply $(f')^{n-1}$. No other variable implies $(f')^{n-1}$ since if that variable was $\overline{x_n}$ or $\overline{y_n}$, $(f')^{n-1}$ would not be consistent with n_n. From the CNF description of $\overline{f^{n-1}}$ we have that no variable implies $\overline{f^{n-1}}$. Thus, since $\overline{(f')^{n-1}}$ implies $\overline{f^{n-1}}$, no variable implies $\overline{(f')^{n-1}}$.

After the above $2n$ projections, only one point remains, namely the negative example n_n. Clearly both g and g' must classify this point negative. As a result, we have that $g' = g$ contradicting that g' is strictly more general than g.

The above argument can be suitably modified to show that elements of the S set are indeed maximally specific. □