Version Space Algebra and its Application to Programming by Demonstration

Tessa Lau TLAU@CS.WASHINGTON.EDU
Pedro Domingos PEDROD@CS.WASHINGTON.EDU
Daniel S. Weld WELD@CS.WASHINGTON.EDU
Dept. of Computer Science & Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350 USA

Abstract

Machine learning research has been very successful at producing powerful, broadly-applicable classification learners. However, many practical learning problems do not fit the classification framework well, and as a result the initial phase of suitably formulating the problem and incorporating the relevant domain knowledge can be very difficult and typically consumes the majority of the project time. Here we propose a framework to systematize and speed this process, based on the notion of version space algebra. We extend the notion of version spaces beyond concept learning, and propose that carefully-tailored version spaces for complex applications can be built by composing simpler, restricted version spaces. We illustrate our approach with SMARTedit, a programming by demonstration application for repetitive text-editing that uses version space algebra to guide a search over text-editing action sequences. We demonstrate the system on a suite of repetitive text-editing problems and present experimental results showing its effectiveness in learning from a small number of examples.

1. Introduction

More often than not, the most difficult and time-consuming part of developing a machine learning application is formulating the problem in terms amenable to currently-available learners (Langley & Simon, 1995). Machine-learning research has largely focused on classification, and although many sophisticated classifiers have been produced, many (perhaps most) problems are not easily cast in this framework. For example, in many domains (e.g., problem solving, vision, speech recognition, computational biology) the desired output of the learned function is not a discrete scalar, but a structured object (e.g., a sequence of actions, a description of a scene, the text corresponding to a speech stream, the 3D shape of a protein). As a result, applying machine learning is largely a "black art," with few formalized principles or domain-independent solutions available. In this paper we begin to address this problem by extending Mitchell's (1982) version space framework to more complex learning problems. We propose the notion of a *version space algebra* in which simple, restricted version spaces are combined into more complex ones using operations like union and join. This approach lets the application designer combine multiple strong biases to achieve a weaker one that is carefully tailored to the application, and thus to reduce statistical bias for the least increase in variance. We also begin to build a library of reusable version space components. We apply these components and the version space algebra in the construction of the SMARTedit system, a programming by demonstration application for the text-editing domain. In many cases, SMARTedit learns generalized macros that automate repetitive text transformations after just a single training example.

Version space algorithms have not been widely applied in practice, mainly because their extension to noisy data is non-trivial, and because the boundary-set representation they use is often not efficient enough (although see the work of Norton and Hirsh (1992), Hirsh et al. (1997), and others). We expect that similar problems will be felt in the broader range of applications we extend the version space approach to here. However, the introduction of version spaces was of enormous value in clarifying the concept learning problem, and in moving from a phase of building *ad hoc* systems with limited uses to the current generation of powerful, widely-applicable classifiers. The goal of this paper is to provide a first step in doing the same for a broader class of learning problems. Moreover, it may often be the case that some component version spaces in an application can be efficiently maintained while others require heuristic search. Combining the two should produce better results than approaching the whole problem in an *ad hoc* manner.

2. Version Space Algebra

In this section, we define our concept of a version space algebra: a method for composing together many simple version spaces using algebraic operations such that the whole is also a version space. We first extend version spaces to apply with any partial order (not just generality, as in Mitchell (1982)). We then define the union, intersection, join, and transform operators over these extended version spaces, and describe the conditions under which the combined version space may be efficiently maintained.

A *hypothesis* is a function that takes as input an element of its domain and produces as output an element of its range. A *hypothesis space* is a set of functions with the same domain and range. The *bias* determines which subset of the universe of possible functions is part of the hypothesis space; a stronger bias corresponds to a smaller hypothesis space. We say that a training example (i, o), for $i \in domain(h)$ and $o \in range(h)$, is *consistent* with a hypothesis h if and only if $h(i) = o$. A *version space*, $\mathrm{VS}_{H,D}$, consists of only those hypotheses in hypothesis space H that are consistent with the sequence D of examples. When a new example is observed, the version space must be *updated* to ensure that it remains consistent with the new example. We will omit the subscript and refer to the version space as VS when the hypothesis space and examples are clear from the context.

In Mitchell's (1982) original version space approach, the range of functions was required to be the Boolean set $\{0, 1\}$, and hypotheses in a version space were partially ordered by their generality. (A hypothesis h_1 is more general than another h_2 iff the set of examples for which $h_1(i) = 1$ is a superset of the examples for which $h_2(i) = 1$.) Mitchell showed that this partial order allows one to represent the version space solely in terms of its most-general and most-specific boundaries G and S (*i.e.*, the set G of most general hypotheses in the version space and the set S of most specific hypotheses). The consistent hypotheses are those that lie between the boundaries (*i.e.*, every hypothesis in the version space is more specific than some hypothesis in G and more general than some hypothesis in S). We say that a version space is *boundary-set representable* (BSR) if and only if it can be represented solely by the S and G boundaries. Hirsh (1991) showed that the properties of convexity and definiteness are necessary and sufficient for a version space to be BSR.

Mitchell's approach is appropriate for concept learning problems, where the goal is to predict whether an example is a member of a concept or not. We extend the approach to any supervised learning problem (*i.e.*, to learning functions with any range) by allowing arbitrary partial orders. We base this proposal on the observation that the efficient representation of a version space by its boundaries only requires that some partial order be defined on it, not necessarily one that corresponds to generality. The partial order to be used in a given version space is provided by the application designer, or by the designer of a version space library. The corresponding generalizations of the G and S boundaries are the least upper bound and greatest lower bound of the version space. As in Mitchell's approach, the application designer provides an update function $U(\mathrm{VS}, d)$ that shrinks VS to hold only the hypotheses consistent with example d.

We now introduce a version space algebra using these extended version spaces. We define an *atomic version space* to be a version space as described above, *i.e.*, one that is defined by a hypothesis space and a sequence of examples. We define a *composite version space* to be a composition of atomic or composite version spaces using one of the following operators.

Definition 1 (Version space union) *Let H_1 and H_2 be two hypothesis spaces such that the domain (and range) of functions in H_1 equals the domain (and range) of those in H_2. Let D be a sequence of training examples. The* version space union*, $VS_{H_1,D} \cup VS_{H_2,D}$, is equal to $VS_{H_1 \cup H_2,D}$.*

Hirsh proved that the union of two BSR version spaces is also BSR if and only if the union is convex and definite. In contrast, we allow unions of version spaces such that the unions are not necessarily boundary-set representable, by maintaining component version spaces separately; thus, we can efficiently represent more complex hypothesis spaces.

Proposition 1 (Efficiency of union) *The time (space) complexity of maintaining the union is a linear sum of the time (space) complexity of maintaining each component version space.*

Definition 2 (Version space intersection) *Let H_1 and H_2 be two hypothesis spaces such that the domain (and range) of functions in H_1 equals the domain (and range) of those in H_2. Let D be a sequence of training examples. The* version space intersection*,*

$VS_{H_1,D} \cap VS_{H_2,D}$, *is equal to* $VS_{H_1 \cap H_2,D}$.

The considerations made above for the version space union also apply to the version space intersection.

In order to introduce the next operator, let $C(h, D)$ be a consistency predicate that is true when hypothesis h is consistent with the data D, and false otherwise. In other words, $C(h, D) \equiv \bigwedge_{(i,o)\in D} h(i) = o$.

Definition 3 (Version space join)
Let $D_1 = \{d_1^j\}_{j=1}^n$ *be a sequence of* n *training examples each of the form* (i, o) *where* $i \in$ *domain*(H_1) *and* $o \in$ *range*(H_1)*, and similarly for* $D_2 = \{d_2^j\}_{j=1}^n$*. Let* D *be the sequence of* n *pairs of examples* $\langle d_1^j, d_2^j\rangle$*. The join of two version spaces,* $VS_{H_1,D_1} \bowtie VS_{H_2,D_2}$*, is the set of ordered pairs of hypotheses* $\{\langle h_1, h_2\rangle | h_1 \in VS_{H_1,D_1}, h_2 \in VS_{H_2,D_2}, C(\langle h_1, h_2\rangle, D)\}$.

Joins provide a powerful way to build complex version spaces, but a question is raised about whether they can be maintained efficiently. Let $T(\text{VS}, d)$ be the time required to update VS with example d. Let $S(\text{VS})$ be the space required to represent the version space VS (perhaps with boundary sets).

Proposition 2 (Efficiency of join)
Let $D_1 = \{d_1^j\}_{j=1}^n$ *be a sequence of* n *training examples each of the form* (i, o) *where* $i \in$ *domain*(H_1) *and* $o \in$ *range*(H_1)*, and let* d_1 *be another training example of the same type. Define* $D_2 = \{d_2^j\}_{j=1}^n$ *and* d_2 *similarly. Let* D *be the sequence of* n *pairs of examples* $\langle d_1^j, d_2^j\rangle$*. If* $\forall D_1, D_2\ [C(h_1, D_1) \wedge C(h_2, D_2) \Rightarrow C(\langle h_1, h_2\rangle, D)]$*, then* $\forall D_1, d_1, D_2, d_2$

$$\begin{aligned} & S(VS_{H_1,D_1} \bowtie VS_{H_2,D_2}) \\ & \quad = \ S(VS_{H_1,D_1}) + S(VS_{H_2,D_2}) + O(1) \\ & T(VS_{H_1,D_1} \bowtie VS_{H_2,D_2}, \langle d_1, d_2\rangle) \\ & \quad = \ T(VS_{H_1,D_1}, d_1) + T(VS_{H_2,D_2}, d_2) + O(1) \end{aligned}$$

In many domain representations, the consistency of a hypothesis in the join depends only on whether each individual hypothesis is consistent with its respective training examples, and not on a dependency between the two hypotheses in a pair. In this situation we say there is an *independent join* in which the consistency of a pair of hypotheses in the version space join follows from the consistency of each individual hypothesis relative to its respective training examples. If the join is independent, then the hypotheses in the version space join are exactly the hypotheses in the Cartesian product of the two component version spaces, and the join may be updated by updating each of the two component version spaces individually. For instance, given VS_1 containing hypotheses $\{A, B\}$, and VS_2 containing $\{X, Y\}$, even though A and X are consistent with their respective data, it is not always the case that $\langle A, X\rangle$ is consistent with the joint data. Although we have not yet formalized the conditions under which joins may be treated as independent, Section 4 gives several examples of independent joins in the PBD domain.

Note that our union and intersection operations are both commutative and associative, which follows directly from the properties of the underlying set operations. The join operator is neither commutative nor associative.

Definition 4 (Version space transform) *Let* τ_i *be a mapping from elements in the domain of* VS_1 *to elements in the domain of* VS_2*, and* τ_o *be a one-to-one mapping from elements in the range of* VS_1 *to elements in the range of* VS_2*. Version space* VS_1 *is a* transform *of* VS_2 *iff* $VS_1 = \{g | \exists_{f \in VS_2} \forall_i\, g(i) = \tau_o^{-1}(f(\tau_i(i)))\}$.

Transforms are useful for expressing domain-specific version spaces in terms of general-purpose ones; see section 4.3 for examples.

3. Application Design

In order to use the version space algebra to model an application, the application designer must specify a set of atomic and composite version spaces and designate a single target space, as shown in Table 1. The role of atomic spaces, composite spaces and target space in version space algebra is analogous to the role of terminal symbols, nonterminal symbols and start symbol in a context-free grammar. The simplest update function would examine each hypothesis in the version space individually, and discard the inconsistent hypotheses. A more efficient update function represents only the boundaries of the consistent set, and updates only the boundaries given each training example. The partial order is a means to this end. Similarly, the simplest execution function would separately compute the output for each hypothesis in the version space and assign votes to outputs accordingly, but it may be possible to find the vote for each output more efficiently; we show an example in the next section. When neither approach is feasible, approximate votes may be computed by sampling from the version space; such sampling is an area for future research.

4. Programming by Demonstration

Programming by demonstration (PBD) is one possible component of an adaptive user interface. In a PBD ap-

Table 1. Domain description to be provided by the application designer in order to use the version space algebra framework.

For each atomic version space:

1. Definition of the hypothesis space.
2. A partial order on the hypothesis space.
3. An update function that updates the version space to contain only those hypotheses consistent with a given example.
4. An execution function that computes a vote for each possible output given an input.

For each composite version space:

1. Formula expressing it in terms of atomic version spaces, previously-defined composite spaces, and version-space-algebraic operators.
2. A transformation function that takes an example for the composite version space and generates the corresponding examples for each component version space.
3. An execution function that takes the outputs (votes) of the component spaces' execution functions and produces a vote for each possible output of the composite space.

The target composite version space.
(One of the previously-defined composite spaces.)

plication, a user demonstrates how to perform a task, and the system learns an appropriate representation of the task procedure. The learned task model can then be executed on the user's behalf in order to automate repetitive tasks. In this section, we apply our version space algebra to the problem of learning procedural actions in a text-editing domain, and describe our SMARTedit (Simple MAcro Recognition Tool) PBD system that learns programs based on demonstrations of repetitive text-editing tasks.

We begin by sketching the SMARTedit user interface, then define SMARTedit's search space in terms of our version space algebra.

4.1 The SMARTedit User Interface

SMARTedit implements an editor that supports a subset of the Emacs command language. As the user is editing a file, when she notices that she is about to perform a repetitive task, she invokes the SMART recorder by clicking on a button in the user interface. SMARTedit then records the sequence of states that result from the user's editing commands, and learns functions that map from one state to another.

When the user has completed one instance of the repetitive task, she clicks another button to indicate that she has completed a single demonstration. At this point, SMARTedit initializes the version space using the recorded state sequence as the first training example. SMARTedit updates the version space lazily as the user provides training examples, which allows it to consider infinite version spaces that are only instantiated on receipt of a positive training example.

The learner is able to make useful predictions after just a single training example. When the user enters another state where the same repetitive task must be performed, she invokes the learned procedure step by step. The system chooses the most likely function in the version space, executes it, and presents the resulting state to the user. If the system's guess was incorrect, the user may press a button to switch to the next most likely state, and so on. At any point, she may choose to undo SMARTedit's last action, or override the system and perform edits manually. When the user chooses a state (either by selecting one of SMARTedit's choices or by performing the action manually), this state is interpreted as another example and used to update the version spaces appropriately.

4.2 Version Space Decomposition

We represent procedural knowledge as a function from one application state to another. In the text editing domain, the state is an ordered triple (T, L, P), where T is the contents of the text editing buffer, $L = (R, C)$ the row and column location of the insertion cursor, and P the contents of the clipboard. After an action is performed (*e.g.*, inserting a string at the current cursor position), the resultant state incorporates the changes made by that action.

At the highest level, our composite version space describes a set of functions mapping one text-editing program state to another. The set of functions in the version space represents all text-editing transformations we are able to learn. The goal of the learner is to induce a function from one state to another by generalizing from training examples (in the form of a sequence of states demonstrating the desired state changes). We compose the target version space out of smaller, component version spaces. Figure 1 shows the hierarchy of version spaces corresponding to the target function in the text-editing domain. Although we have presented it here as a tree for clarity, the complete version space has an equivalent representation as a formula in our algebraic notation.

The target space Program represents the class of all functions learnable in our domain. It is composed of an independent join of a fixed number of Action version spaces. (In practice, the number is determined lazily as the length of the first training example. Variable-length action sequences are a topic for future research.) Each Action function represents a simple command a user might perform in a text editor, such as moving the insertion cursor, inserting and deleting text at the current cursor location, and manipulating the clipboard (selecting text and copying it to and from the clipboard).

The leaf nodes in the version space hierarchy are the atomic version spaces. The ConstInt hypothesis space includes all functions of the form $f(int : x) = C$ for some integer constant C. We choose the partial order by the value of C; if $f(x) = C_1$ and $g(y) = C_2$, then $f \prec g$ iff $C_1 < C_2$. The ConstInt version space is trivially maintained; after two or more examples, the version space collapses to one or zero hypotheses. The LinearInt hypothesis space includes all functions of the form $f(int : x) = x + C$, for some integer constant C. Its partial order and update function are analogous to ConstInt.

The AbsRow and AbsCol version spaces transform the ConstInt atomic version spaces from integer functions into functions on row or column values (*i.e.*, into functions that change the cursor position to an absolute row or column). Similarly, the RelRow and RelCol version spaces transform LinearInt atomic version spaces into row and column functions (by changing the cursor position relative to its previous location). The Row composite version space consists of the union of AbsRow and RelRow version spaces, and likewise for the Column version space. The RowCol version space is the independent join of the Row and Column version spaces with a consistency predicate that is always true.

Besides row and column positioning, our domain representation supports positioning the cursor relative to the next occurrence of a string. If the cursor is positioned after (before) a string, we say that the user was finding the next *prefix* (*suffix*) match. Suppose the user has moved the cursor to the end of the next occurrence of the string "PBD". From the system's point of view, the user may have been searching for the prefix "PBD", the prefix "BD", or the prefix "D". The FindPrefix and FindSuffix version spaces represent these types of string-searching hypotheses.

More formally, the PrefStr and SuffStr hypothesis spaces include all functions of the form $f() = T$ for

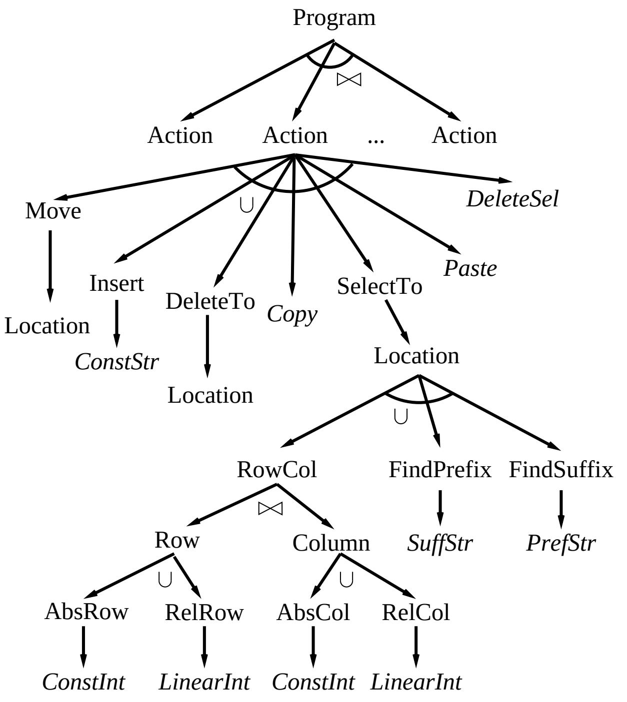

Figure 1. Version space structure for the text-editing domain. The upper tree shows the complete version space for a Program, expressed in terms of Action version spaces (middle tree). Action version spaces are in turn expressed in terms of Location version spaces (bottom tree). Italicized text denotes an atomic version space, while regular text denotes a composite version space.

some constant string T. We choose the partial order of PrefStr according to a string prefix relationship in the string T; if $f() = T_1$ and $g() = T_2$, then $f \prec g$ iff T_1 is a proper prefix of T_2. (SuffStr is defined similarly.) For clarity, we omit the function symbol and simply refer to the function as the string it produces. The least upper bound (LUB) and greatest lower bound (GLB) boundaries of the PrefStr version space are initialized to be $\mathcal{S}$ and $\mathcal{C}$ respectively, where $\mathcal{S}$ is a token representing the set of all strings of length K (some constant greater than the maximum text buffer size) and $\mathcal{C}$ is a token representing the set of all strings of unit length. When the first example is seen, the LUB becomes the singleton set containing the contents of the text buffer following the cursor (a string), and the GLB becomes the singleton set $\{$"c"$\}$, where c is the character immediately following the cursor. After the first example the LUB and GLB will always contain at most one string each. Given a new training example in which the string T follows the cursor, and the LUB contains the string S, the LUB is updated to contain the longest common prefix of S and T. The GLB remains unchanged if the character immediately following the cursor is again c; otherwise the version space collapses to the null set.

The FindPrefix version space transforms each hypothesis in the SuffStr version space into a function from a state to a cursor position. For each function in the SuffStr version space, we create a corresponding function in FindPrefix that locates the first occurrence of this string, and returns the cursor position at the end of the matching occurrence. FindSuffix transforms PrefStr analagously, finding the beginning of each matching occurrence.

The various types of cursor-positioning functions are unioned together as the single Location version space, which is in turn transformed by many of the actions. For instance, the Move version space transforms Location to provide functions from one state to a new state with a different cursor location. The DeleteTo (SelectTo) version space transforms a Location version space to represent functions that delete (select) from the input cursor location to a new location, and output a new state in which the text between the two positions has been deleted (selected).

4.3 Transformation and Execution

The transformation functions for the composite version spaces are straightforward; they convert between state-state functions and functions over more primitive data types such as integers and tuples of integers. Although a complete description of the transformation functions used in SMARTedit is beyond the scope of this paper, we highlight two of the transformations used in our system.

The transformation function for the top-level Program space takes a sequence of states $s_0, s_1, ..., s_n$ and constructs n examples of the form $\langle s_{i-1}, s_i \rangle$, such that the i^{th} tuple is used to update the i^{th} Action version space. The RowCol transformation function takes a pair of input/output cursor locations $((r_i, c_i), (r_o, c_o))$ and constructs the input/output examples (r_i, r_o) and (c_i, c_o) that are appropriate for its component version spaces. Other version space transformations are straightforwardly defined.

A version space may be executed on a new input state in order to produce one or more output states. In general, executing a version space on an input i means letting each hypothesis h in it cast a vote for its output $h(i)$, and then choosing the output with the most votes. If possible, it is desirable to collect the votes without explicitly enumerating the hypotheses. For the purposes of the PBD application, we are interested in the ranked list of output states, with a preference for output states that are supported by larger numbers of hypotheses. In order to compute the output states for a given input state, the input state is propagated down to the atomic version spaces at the leaves of the tree using the same transformation function used to update the version space. The outputs of the leaves are then propagated up through the version space, using the transformation functions to convert them to the proper type. The set of outputs for the target space are then ranked according to the number of hypotheses that voted for each output, and the highest-ranked state is presented to the user as previously described.

The execution of the FindSuffix version space bears mentioning. The underlying Prefix version space represents a set of strings, the longest of which is the string in the LUB, the others being some prefix of the string in the LUB. The execution maps each string in Prefix to the cursor location corresponding to its first occurrence in the text file following the location of the cursor. We can perform this search efficiently in time $O(st)$ where s is the length of the string and t is the length of the text file, by comparing the string against the text starting at every position in the text file following the cursor position. The algorithm is as follows: find the first location where the first k_1 characters of the text and search string match. Cast k_1 votes for this location, one for each of the prefixes which matched. Continue searching from this occurrence for a match of at least length $k_2 > k_1$, casting $k_2 - k_1$ votes for the next match. Repeat until all prefixes have been matched or the end of text is reached, and return the set of output locations and their votes as the result of the execution. FindPrefix is handled in a similar fashion.

5. Experimental Results

We have applied the SMARTedit system to a representative collection of repetitive text-editing scenarios. Each scenario is a collection of training examples; a training example is a sequence of (T, L, P) states. Figure 2 lists the scenarios we used to evaluate SMARTedit, along with the total number of instances in each, and the number of instances the system needed to see before making the correct prediction on all remaining instances (*i.e.*, applying the correct transformation to them). In some cases, the system succeeds after just a single example.

The `columns` scenario operates on a text file containing data in whitespace-separated columns. The task in this scenario consists of moving the first column to the end of the line. The typical sequence of actions involved in this task is to select the text in the first

Scenario	Total # Exs.	# Train Exs.		
		U1	U2	U3
columns	8	2	2	2
boldface	4	1	1	1
addressbook	6	2	2	2
grades	7	1	1	1
commentstyle	5	1	2	2
HTML-to-LaTeX	7	2	2	2

Figure 2. List of scenarios used to test the SMARTedit system, total number of examples in each, and number of training examples (for each of three users) required by the system to induce a procedure that makes the correct predictions on the remaining examples.

column, copy it to the clipboard, delete the selection, move the cursor to the end of the line, and paste the contents of the clipboard.

The boldface scenario has the user take a paragraph of text containing the word "SMARTedit" and boldface each occurrence of that word by surrounding it with the HTML tags `<B>` and `</B>`.

The `addressbook` scenario operates on a text file containing a list of addresses, one per line. The task is to convert each address into a multi-line format suitable for printing on a mailing label by inserting carriage returns at appropriate locations in the address. This scenario is a simplified version of an example used to illustrate the TELS system (Mo, 1989).

The `grades` scenario operates on a text file containing a list of students and their grades in a class. The task in this scenario is to delete the student name that appears at the end of the line, leaving only the list of grades. The `commentstyle` scenario operates on a text file containing source code in the C programming language. The task in this scenario is to convert all C-style comments into C++-style comments (assuming all comments occur on a single line). The `HTML-to-LaTeX` scenario escapes the angle brackets in HTML formatting tags using LaTeX's $ math mode notation.

6. Related Work

Hirsh (1991) studied the algebra of boundary-set representable version spaces. We have extended his work beyond concept learning by allowing any partial order, defining the join operator, and allowing non-BSR unions and intersections.

A different extension of version spaces has been proposed by VanLehn and Ball (1987), for inducing context-free grammars from examples. Since generality is undecidable for context-free grammars, VanLehn and Ball approximate the generality relation using an alternative partial order.

Several prior systems have addressed programming by demonstration (Cypher, 1993) in the text-editing domain. The TELS system (Mo, 1989) learns programs given demonstrations of action sequences; TELS uses a collection of domain-specific heuristics to determine when two actions may be generalized. The Editing by Example system (Nix, 1985) took a different approach and induced a text-editing program using only the initial and final state of the transformation sequence; the system is able to learn a restricted subset of regular expressions.

The Cima system (Maulsby & Witten, 1997) employs a disjunctive rule learner to learn the arguments to an action. Although many knowledge representations suffice for classifying training data into positive and negative examples, they may be useless in acting on a novel example. Cima solved this problem by heuristically preferring rules that more fully describe its behavior on an unseen example. In contrast, SMARTedit's version space algebra approach makes the search bias explicit.

Work on automatic induction of wrappers for information resources (Kushmerick, 1997; Ashish & Knoblock, 1997) also generates procedural knowledge. In particular, the LR wrapper class described by Kushmerick (1997) defines a family of procedures, each of which is equivalent to a sequence of text-editing commands that, for each of the k attributes to be extracted: moves to the next attribute, selects the attribute's text, and copies it to the clipboard.

Lesh and Etzioni (1995) used version spaces for goal recognition. In the BOCE system, each observed action caused the version space of goals to be updated to contain only goals consistent with the observed actions. The primary difference between BOCE and SMARTedit is that BOCE assumes that actions are modeled explicitly as having preconditions and effects in a STRIPS style; this representation allows BOCE to directly connect actions with their potential target goals. In contrast, SMARTedit learns action sequences themselves, rather than goals.

Langley and Simon (1995) discuss the difficulty of formulating a problem in a representation amenable to classification, identify several domains where non-classification learning may be applied, and describe a number of early attempts to do so.

7. Conclusion

We have described our version space algebra and applied it to the domain of programming by demonstration. We make the following contributions:

- We have generalized version spaces to learning input-output mappings of any kind, including outputs that are structured objects, not just Boolean values.
- We have developed a framework for application design using a version space algebra that provides union, intersection, join, and transformation operators to construct complex version spaces out of simpler ones.
- We have begun construction of a library of reusable component version spaces that may be applied to a variety of domains.
- Using our framework and library, we have built a programming by demonstration (PBD) system for the text-editing domain and evaluated its performance on several different repetitive text-editing problems.

Directions for future work include: extending the version space library; empirically evaluating alternative domain encodings; extending the algebra with useful new operators and studying their properties; combining boundary-represented version spaces with heuristically-searched ones; searching automatically through the space of version spaces to find the best structure for a domain; extending our algorithms to cope with noisy data probabilistically, including specifying prior probabilities on hypotheses; sampling from version spaces for efficient execution; and investigating the problems (including credit assignment) that arise when the parsing of an example into sub-examples is not given *a priori*. We also plan to enrich our domain model in order to learn more complex text transformation functions.

Acknowledgements

This research was funded in part by the Office of Naval Research Grant N00014-98-1-0147, by National Science Foundation Grants IRI-9303461 and IIS-9872128, and by a Microsoft Fellowship. We thank Corin Anderson, Pat Langley, Steve Wolfman, and the anonymous reviewers for their feedback.

References

Ashish, N., & Knoblock, C. (1997). Semi-automatic wrapper generation for Internet information sources. *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems* (pp. 160–169). Los Alamitos, CA: IEEE-CS Press.

Cypher, A. (Ed.). (1993). *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press.

Hirsh, H. (1991). Theoretical underpinnings of version spaces. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (pp. 665–670). San Francisco, CA: Morgan Kaufmann.

Hirsh, H., Mishra, N., & Pitt, L. (1997). Version spaces without boundary sets. *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (pp. 491–496). Menlo Park, CA: AAAI Press.

Kushmerick, N. (1997). *Wrapper induction for information extraction*. Doctoral dissertation, Department of Computer Science & Engineering, University of Washington, Seattle, WA.

Langley, P., & Simon, H. A. (1995). Applications of machine learning and rule induction. *Communications of the ACM*, *38*, 54–64.

Lesh, N., & Etzioni, O. (1995). A sound and fast goal recognizer. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (pp. 1704–1710). San Francisco, CA: Morgan Kaufmann.

Maulsby, D., & Witten, I. H. (1997). Cima: An Interactive Concept Learning System for End-User Applications. *Applied Artificial Intelligence*, *11*, 653–671.

Mitchell, T. (1982). Generalization as search. *Artificial Intelligence*, *18*, 203–226.

Mo, D. H. (1989). Learning text editing procedures from examples. Master's thesis, Department of Computer Science, University of Calgary, Calgary, AB.

Nix, R. P. (1985). Editing by example. *ACM Transactions on Programming Languages and Systems*, *7*, 600–621.

Norton, S. W., & Hirsh, H. (1992). Classifier learning from noisy data as probabilistic evidence combination. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 141–146). Menlo Park, CA: AAAI Press.

VanLehn, K., & Ball, W. (1987). A version space approach to learning context-free grammars. *Machine Learning*, *2*, 39–74.