CHAPTER

Eleven

Learning Repetitive Text-Editing Procedures with SMARTedit

Tessa Lau
University of Washington

Steve Wolfman
University of Washington

Pedro Domingos
University of Washington

Daniel S. Weld
University of Washington

S
R
L

Abstract

The SMARTedit system automates repetitive text-editing tasks by learning programs to perform them using techniques drawn from machine learning. SMARTedit represents a text-editing program as a series of functions that alter the state of the text editor (i.e., the contents of the file or the cursor position). Like macro recording systems, SMARTedit learns the program by observing a user performing her or his task. However, unlike macro recorders, SMARTedit examines the context in which the user's actions are performed and learns programs that work correctly in new contexts. Using a machine learning concept called *version space algebra,* SMARTedit is able to learn useful text-editing procedures after only a small number of demonstrations.

11.1 Introduction

Programming by demonstration (PBD) has the potential to allow users to customize their applications as never before. Rather than writing a program in an abstract programming language to automate a task, users demonstrate how to perform the task in the existing interface, and the system learns a generalized program that can perform it in new contexts.

In some domains, such as the Stagecast Creator and ToonTalk systems described elsewhere in this book, users' primary goals are to construct programs. In other domains, users are more interested in getting their work done than in programming the system. They are focused on their overall tasks, and constructing programs to automate repetitive subtasks are merely the means to an end.

Our work focuses on the latter class of users: those who don't want to, or don't know how to, construct a program to accomplish some task. Ultimately, the process of constructing a program (or recording a macro) boils down to the problem of choosing the correct action for the system to take at each step. A PBD system can make this programming process easier in two ways: by providing a visual representation of the program, thus eliminating the need to understand arcane syntax, and by inferring the user's intent based on demonstrations. In our work, we focus on the latter approach. We place the burden of inference on the system, rather than on the user; the system is responsible for figuring out what the user meant to do and for removing enough details to construct a reusable program. S

For a PBD inferencing system to be useful, it must be expressive enough to represent the types of programs users want to construct. On the other R L

hand, it must be able to perform its inference based on a very small number of examples—few users would be willing to train the system on hundreds or even tens of examples! These two goals are directly in conflict with each other: as the learning system's expressiveness (and thus the size of its search space) increases, so does the number of examples required to pick out the correct concepts in that space.

The approach we have taken, based on a machine learning concept called *version space algebra* (Lau, Domingos, and Weld 2000), allows us to get the best of both worlds: an expressive program representation language, combined with the ability to make useful inferences after a very small number of demonstrated examples. We do this by carefully crafting our search bias—the kinds of programs we can represent—so as to separate the wheat from the chaff. The version space algebra, which is an extension to Mitchell's (1982) concept of version spaces for machine learning, lets us represent only the programs that users might want to write, without being distracted by nonsensical or useless programs.

We have implemented and tested our framework in the text-editing domain. Examples abound of repetitive procedures for editing text: converting from one file format to another, reformatting address lists, manipulating bibliographic citations, and so on. Our system SMARTedit (Simple MAcro Recognition Tool) uses version space algebra to learn useful programs for editing text based on as little as a single demonstrated example.

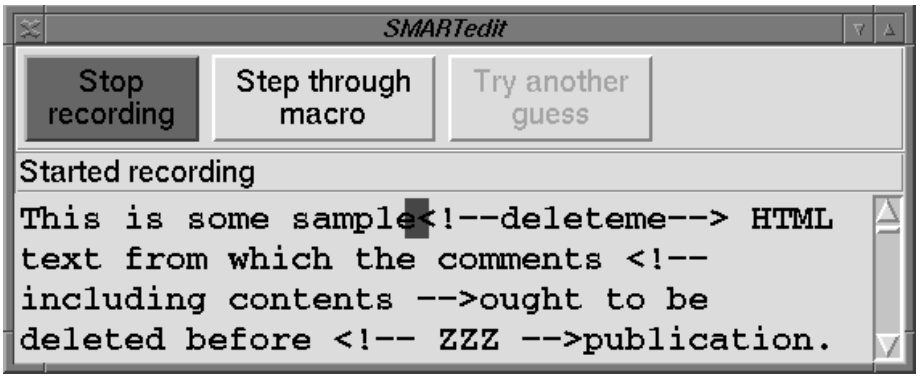
Beyond the consideration of a single domain, however, we are also concerned with a system's ability to scale both to more complex domains and to other domains entirely. The variety of systems described elsewhere in this book displays the breadth of application domains that could benefit from PBD. Given such a wide range, an important factor in the design of a PBD system is how domain knowledge is represented in the system. The answer to this question determines how easy it is to add new knowledge to the system and whether it can be easily applied to a different domain. Previous PBD systems have been built on domain-specific preference biases—knowledge about which actions are more likely than others—to force the system to come to the correct conclusion. Like expert systems, such techniques lead to brittle, poorly understood designs; extending the system to support a new concept requires delving into the depths of the code and understanding all of the ramifications of each change. In contrast, our version space algebra framework proposes a modular approach to the design of PBD systems: a domain-independent learning algorithm and a structured, high-level domain representation. Adding new features to the system, or applying it to a different domain, requires only changes to the domain representation. We hope that robust solutions such as ours will help PBD spread more quickly to more applications.

11.2 The SMARTedit User Interface

We illustrate SMARTedit by showing how it automates a simple text-processing task. Suppose the user has some HTML with comments embedded in it and would like to remove the comment tags and all the text inside the comments. An HTML comment is a string delineated by the tokens <!— and—>. The comment may span multiple lines and contain arbitrary characters. To delete all such comments, a Microsoft Word user would have to enter a regular expression of the form \<!—*—\> into the search-and-replace dialog box. While this syntax may look straightforward to programmers, even we took several tries to figure out the correct syntax to use for that regular expression. In SMARTedit, however, no arcane syntax is required. The user simply demonstrates the desired functionality by deleting the first HTML comment, and the system is able to do the rest. Let's walk through this example to illustrate exactly what SMARTedit can do.

SMARTedit works like a simple macro recorder and follows the familiar macro recording interface. To begin creating a SMARTedit macro, the user pushes a *start recording* button. The button then turns red, indicating that her actions are being recorded. She then begins demonstrating what she wants SMARTedit to do. In this task, she first moves the cursor to the beginning of the next HTML comment, right before the <!—characters, using any

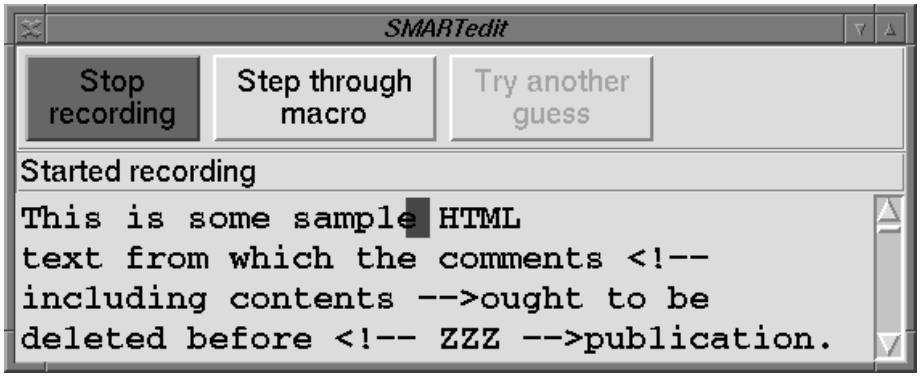
FIGURE 11.1



Recording a SMARTedit macro. The blue block is the position of the insertion cursor, the status bar indicates that the user is recording, and the record button has turned red to indicate recording is active.

S
R
L

FIGURE **11.2**



Deleting the HTML comment.

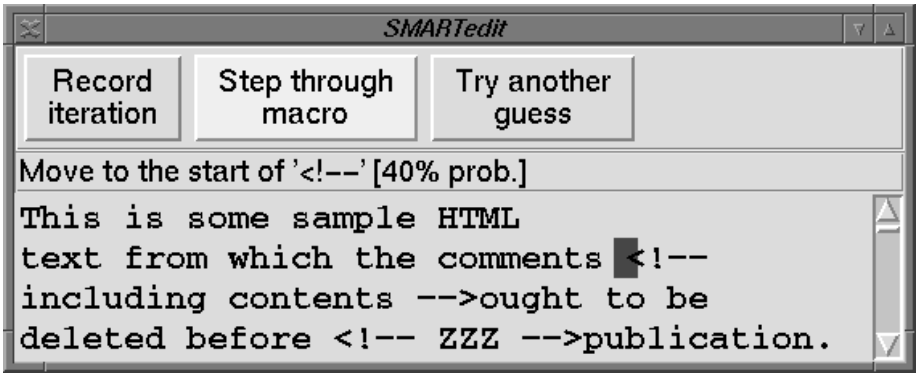
combination of cursor-motion keys or mouse clicks (Fig. 11.1). She then deletes the entire HTML comment (Fig. 11.2).

Because the demonstration is now complete, the user presses the *stop recording* button to indicate that she is finished. SMARTedit has now learned a macro representing the procedure she has just performed. Looking more closely at the way the user demonstrated this task, she had to hit the delete key fifteen times to delete the extent of the comment. It's unlikely (though possible) that she wanted to delete exactly fifteen characters. What's more likely (and correct) is that she wanted to delete up until the comment closing tag. If she were writing a program to delete these HTML comments, she would have had to specify (in some programming language) when to stop deleting, using some abstract representation of the text, rather than the actual text sitting right there.

SMARTedit infers which meaning the user wanted, by using her demonstration as an example of the program she is trying to construct. The next section will show how SMARTedit represents these two possible actions (and others) as different hypotheses in its version space. But first let's see how SMARTedit's knowledge is used to help the user delete the next comment automatically.

SMARTedit learns procedures consisting of a sequence of actions—relatively high-level text-editing commands, such as moving the cursor to a new position; inserting a string; selecting or deleting a region of text; or manipulating the clipboard. The user can invoke SMARTedit's learned macro an action at a time by pressing the *Step through Macro* button. SMARTedit

FIGURE 11.3



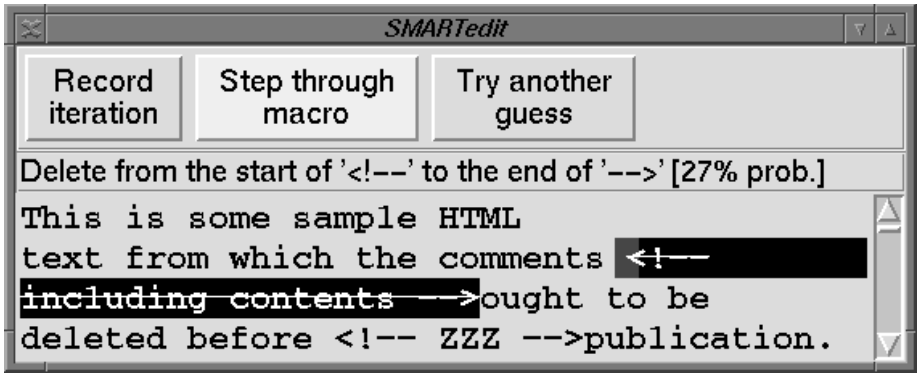
Invoking the macro by pressing the Step through Macro button.

will guess what action she's likely to take next. In this case, it correctly predicts that the next action she wants to take is to move the cursor to the beginning of the next HTML comment (Fig. 11.3). Given that's she's only demonstrated one example, however, SMARTedit could only predict this action with 40 percent probability. (For example, it's possible that she wanted to position the cursor after the word *sample* instead of before the HTML comment; this action and others like it also have nonzero probability.) If the result of this action is not what the user intended, she could use the *Try Another Guess* button to switch to SMARTedit's next most likely choice of action, and so on, until she finds the desired action. The user is always free to correct SMARTedit and perform the desired action herself.

After the user verifies that SMARTedit has performed the correct action, she steps to the next action by invoking the *Step through Macro* button again. This time, it correctly predicts with 27 percent probability that she will delete the extent of the HTML comment (Fig. 11.4). The action is visualized by striking through the region that is to be deleted, rather than deleting it without warning (which caused confusion in an earlier implementation).

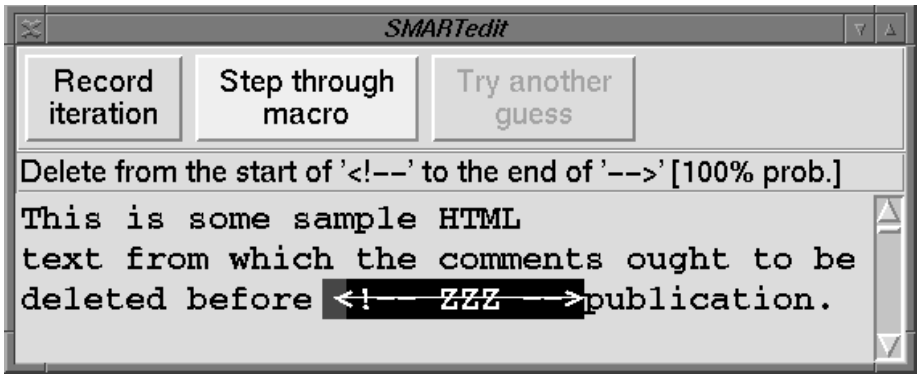
The user has now finished deleting the second HTML comment in the file. The next time she invokes the *Step* button, SMARTedit will again position the cursor at the beginning of the next HTML comment (Fig. 11.5). However, SMARTedit is adaptive, and it has been learning from her choices even while she was asking it to make predictions. It makes this prediction with 100 percent probability based on both the original example as well as the collaborative one.

FIGURE **11.4**



The macro predicting to delete the entire HTML comment.

FIGURE **11.5**



Pressing the Step button once more to start deleting the next HTML comment with 100 percent probability.

11.3 The Smarts behind SMARTedit

We view text editing as a sequence of changes in the state of a text editor application. The state includes the cursor position (as a row and column pair), the contents of the text editing buffer, and the contents of the clipboard.

Each text-editing action performed by the user in some state results in a new state. For instance, moving the cursor one row forward results in a new state, which differs from the initial state in that the cursor position is one row greater. Inserting a string by typing a few keys in one state results in a new state where the text-editing buffer includes the inserted string.

SMARTedit learns a macro as a sequence of functions that transform one state to another. For instance, the user's first action in the earlier example was to move the cursor to the beginning of the HTML comment; SMARTedit represented that action as a function that took a text state and output a new state in which the cursor was repositioned to lie in front of the next occurrence of the string <!—.

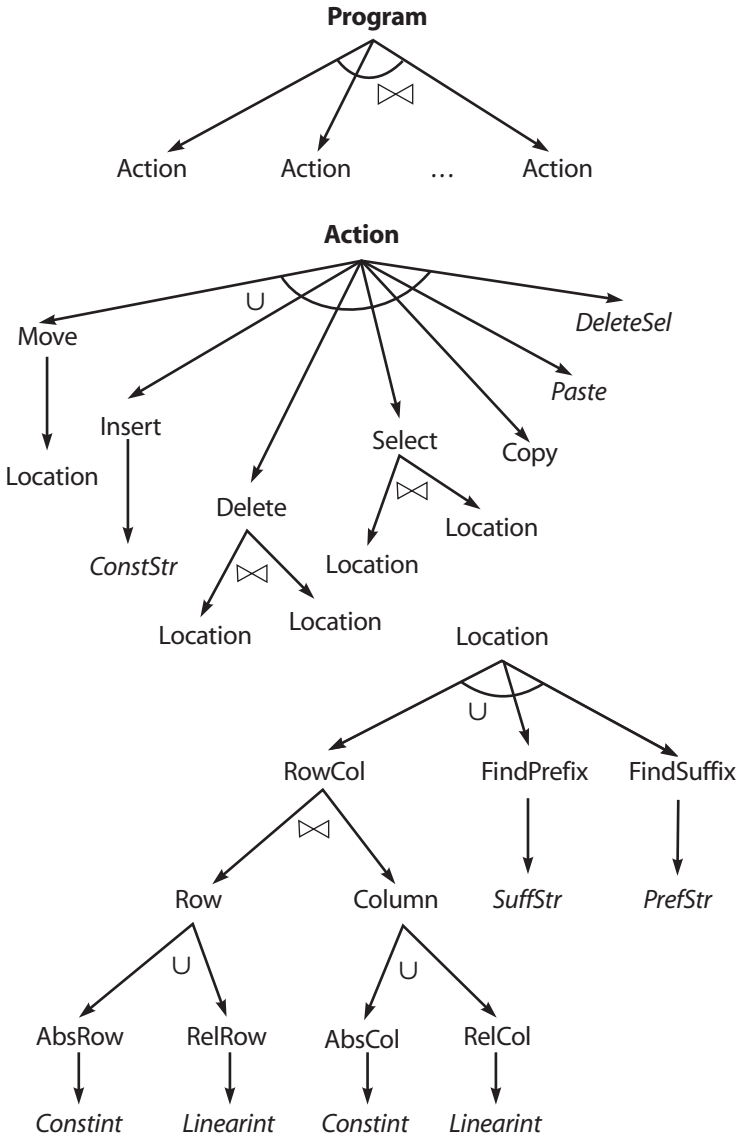
We have defined several classes of text-editing functions representing the types of text-editing operations people do in an editor. For example, the class of *Move* functions takes the input state with a particular cursor location into an output state in which only the cursor location is different. Each particular *Move* function moves the cursor for a different reason; possible reasons include moving to a particular row and column, moving to the next row or column, or moving the cursor before or after the next occurrence of a search string. For example, the correct *Move* function in the prior example is the one that moves the cursor before the next occurrence of the string <!—. The class of *Insert* functions models normal typing by mapping from an input state with some text buffer into a new state in which the text buffer includes an inserted string, but nothing else has changed. Each *Insert* function inserts a different string into the text buffer. Other basic functions include deleting text, selecting text, and manipulating the clipboard by cutting, copying, and pasting the selected text.

Given this representation of text-editing actions, we view the learning process as figuring out which functions are consistent with the actions the user performed. In a programming environment, the programmer would have to directly specify exactly which function to perform using programming language constructs. However, SMARTedit reduces user effort by inferring the correct function itself. It performs inference by using demonstrated actions to rule out functions successively until only a few are left.

Figure 11.6 shows the complete space of programs SMARTedit is able to learn. We call this SMARTedit's *version space,* so named because it contains all possible versions of the target function. Mitchell (1982) first introduced version spaces as a method for visualizing and efficiently representing the search for a concept in a space of hypotheses. However, Mitchell's version space contained a single set of hypotheses. In contrast, SMARTedit's version space (as shown in the figure) is organized hierarchically according to the important concepts in the domain, using version space algebra. The algebra defines how smaller, simpler sets can be combined together to construct a

S
R
L

FIGURE **11.6**



Version space algebra representation in which each node represents a set of functions transforming an input state into an output state. The contents of each node are computed from the contents of its children nodes by either union, cross product (the bowtie symbol), or transformation (unmarked).

S
R
L

single more complex set that retains a hierarchical structure. Functions with similar purpose (e.g., all movement functions) are grouped together into a single set, rather than being merged with all the other types of text-editing functions.

The correct function is somewhere in this version space, and SMARTedit's goal is to find it. Each node in the tree represents a set of functions that are consistent with the examples seen thus far; that set is constructed out of the sets beneath it in the tree. For instance, the root of the tree, labeled "Program," represents the set of all programs consistent with the actions the user has demonstrated. A program is made up of a sequence of *Action* nodes; each *Action* node is a union of all the different classes of text-editing functions. For example, the *Move* node represents the set of all movement functions consistent with the user's cursor repositioning activity.

Before the user records her actions, the version space initially contains the set of all possible functions. After she has made a recording, some of those functions are thrown out of the version space because they are not consistent with (do not explain) the observed actions. For instance, if the first action a user chooses is to reposition the cursor, all functions for inserting, deleting, or selecting text are inconsistent and can be removed from consideration. The version space for the first *Action* node is updated such that all actions other than *Move* contain the empty set. Moreover, the set of movement functions is constrained to be only those functions that move to locations consistent with the observed movement.

Locations in a text file are central to text-editing programs. The *Location* node in the version space represents this concept as a set of possible locations, specified in a variety of ways. Locations can be specified as row and column position (either relative to the previous position or absolute in the file), after a certain search string, or before a search string. For instance, in the prior example, the user's intent was to move the cursor before the string <!—. In our terminology, she was conducting a suffix search; the cursor position right before the next occurrence of <!—is one function in the *Location* version space. Locations are reused by several actions, such as determining the destination of a cursor movement action or specifying the extents of a region to be deleted.

Consider the HTML comment deletion example discussed previously. The program for that task contained two actions: a move and a deletion. Recall that the cursor started at the beginning of the line and was repositioned nineteen characters forward to lie between the word sample and the characters <!—. After the user has repositioned the cursor, SMARTedit's version space for the desired location is updated to be consistent with this example. The row and column version spaces are updated according to the observed

S
R
L

difference in row and column positions in this example. For example, a move to the absolute column 19 is consistent, as is a relative move forward nineteen columns (because the cursor started out on the zeroth column).

In addition, various string search functions may apply. In this case, the cursor ended up after the word *sample* and before the string <!—. SMARTedit can't tell whether the word *sample* is the important feature or whether perhaps the string *ample* would suffice or even just the string *le*. (It knows that the prefix *e* alone was not the important feature; otherwise, the user would have positioned the cursor after the *e* in the word *some*.) Any string, including the entire contents of the text file from the beginning to the current cursor position, may be the important feature. In fact, none of these prefix search hypotheses are correct, but SMARTedit hasn't seen enough examples to rule them out yet.

Similarly, a number of suffix search functions are consistent with the example: <, <!, <!-, <!—, and so on up to the entire contents of the text file after the cursor position.

When another example is observed (perhaps as the user is stepping through the macro on the second HTML comment), some of these hypotheses will be thrown out because they are not consistent with this new example. All of the row/column hypotheses are thrown out, because the second comment is not in the same column as the previous one (though in more structured editing tasks, the row/column hypotheses might be more important). All of the prefix search hypotheses are thrown out, because this HTML comment appears after the word *comments,* not after the word *sample*. In addition, the contents of this HTML comment differ from the previous one. Thus, the only hypotheses that remain are those that predict a suffix search of <, <!, <!-, and <!—.

Our version space algebra representation confers two advantages: an efficient representation of the set of consistent functions, and a method for structuring the function space such that related functions (e.g., all *Move* functions) are grouped together. Rather than explicitly enumerating every function hypothesis to determine which hypotheses are consistent with the examples, SMARTedit efficiently searches through this structured version space representation of consistent hypotheses to maintain its knowledge about the actions the user has performed.

11.4 Choosing the Most Likely Action

S
R
L

Using the version space representation, SMARTedit is able to maintain a large and complex space of function hypotheses. Given only a few

examples, however, it's rare for the version space to collapse to a single function. If the version space contains several functions, SMARTedit nonetheless picks a single action to perform for the user, as follows.

First, SMARTedit captures the current state of the application. It then treats this state as an input and executes each of the functions in the version space on this input, to produce a set of output states. Some functions may produce the same output state when applied to the same input state; for example, if the cursor is currently on row 4, moving the cursor to the next row has the same effect as moving the cursor to absolute row 5. In addition, some functions may be more likely than others. For instance, it's rare for users to be interested in an absolute row; relative row positioning is much more likely. Thus, each function has associated with it a probability based on two factors: its own prior probability and the prior probabilities of the version spaces which contain it.

If the version space execution produces only one output state, then SMARTedit presents this output state to the user, with 100 percent probability that this is what the user intended to do. Since this output state may have been produced by more than one function, it's possible that SMARTedit still doesn't know exactly what the correct function is. However, it knows enough to be able to predict the correct action.

If there is more than one output state predicted by the version space, then SMARTedit must choose between them. It does so by choosing the state that has the highest probability, where an output state's probability is the sum of the probabilities of the functions that produce it.

If the chosen state matches the user's expectations, she can continue executing the next action in the learned macro. If not, she can ask SMARTedit to switch to the next most likely output state, and so on, until she finds the correct one.

In the HTML comment deletion task described earlier, SMARTedit is, in fact, able to learn how to perform the task correctly after only a single demonstration (for a total of three HTML comments correctly deleted). We have also tested SMARTedit's ability in a number of repetitive text-editing scenarios, such as converting from one XML format to another, rearranging the order of columns in a structured text file, reformatting mailing addresses from single-line to multiline, converting C-language comments to C++-style comments, and converting among the Scribe, LaTeX, and HTML formatting languages. In all cases, SMARTedit requires between one and three demonstrations before it is able to perform the task correctly on the remainder of the examples in the scenario.

S
R
L

11.5 Making SMARTedit a More Intelligent Student

SMARTedit's current learning process—learning from observations, guessing a generalized concept, and collaborating with the user to refine those guesses—represents the first step toward "programming by teaching," as PBD is described in the introduction to this book: SMARTedit is an intelligent student poised to learn programs for a teacher. However, SMARTedit still falls short of fulfilling its role as a student; a truly intelligent student holds a much richer interaction with his teacher. Moreover, SMARTedit is not just a student but also an assistant; an automated assistant shares many traits with a good student but has the added concern of minimizing the user's burden. We see the next major steps for SMARTedit as making it a better student and assistant: taking the initiative to guide the learning process with its own questions, learning from gestures and discussion accompanying the demonstration, carrying knowledge forward from one learning episode to the next, and tempering this process with an awareness of the burden on the user.

To allow SMARTedit to take a more active role in guiding the learning process, we envision a system that takes the initiative, asking questions to clarify its knowledge. Currently, SMARTedit either learns from the user demonstrating an example or from the user's acceptance or rejection of SMARTedit's guess at the next action. In both cases, the user is in control of the interaction. SMARTedit might instead guide the course of the interaction based on its assessment of the state of its knowledge. As a first step, SMARTedit can inform the user when it needs more examples and when it believes (based on the probability of its hypotheses) that it has discovered the user's procedure. With this feedback, the user can make an informed decision whether to begin examining SMARTedit's guesses or continue demonstrating examples. SMARTedit could take an even more active role by asking for information about the current action (e.g., "Are you searching for a string?"), suggesting that the user demonstrate an example other than the next one in the text or proposing a series of actions for consideration by the user rather than just one (e.g., proposing both the move and deletion from Figs. 11.3 and 11.4 in one interaction).

Careful selection of the next question to ask the user will make SMARTedit's learning process zero in on the user's procedure more quickly. We can make these selections based on their discriminating power in SMARTedit's version space or based on the expected benefit (information gain) of each question given the current probabilities of SMARTedit's hypotheses. The selection of the next question for the user should also take

into account the burden that question places on the user. An important, open research question is how we can balance the needs of SMARTedit's learning process against the added burden of its queries to the user. How much and what kind of effort is the user willing to expend answering these queries? How do the different queries inconvenience different users?

We can also enrich the interaction between SMARTedit and its user/teacher by allowing the teacher to provide information outside the strict bounds of the demonstration. In informal observations, we have found that people find it quite natural to narrate their actions as they perform them. Much of the "teaching" content of the demonstrations may be in these narrations. One could imagine using these narrations in a variety of ways from simple keyword detection (e.g., the word *searching* in the narration might increase the probability of string search hypotheses) to full-fledged natural language understanding. In particular, we speculate that technologies used in information retrieval (e.g., Latent Semantic Indexing, Deerwester et al. 1990) might help connect utterances to the actions the users perform and the text they are modifying. The "hints" we get from the narrations could then be used to change the prior probabilities on each of the different version spaces, so as to prefer some functions over others.

SMARTedit should also carry aspects of its knowledge forward from one interaction to the next. Conceptually, we would like SMARTedit to learn how to deal with different users, texts, and tasks. Context in the form of user habits or document type can dramatically narrow the scope of reasonable hypotheses that SMARTedit need consider. Practically, SMARTedit can support this kind of adaptation by adjusting the initial probabilities of its various hypotheses based on the identity of the user and type of file the user is editing. For example, if SMARTedit detects that the user is editing a comma delimited table (e.g., a file with the .csv extension), it can increase the initial probability of string searches containing a comma.

Finally, as an intelligent assistant, SMARTedit should endeavor to reduce the burden it places on the user. Rather than asking the user to start and stop the macro recorder explicitly in between demonstrations, the system should figure out that she has demonstrated the same task twice in a row. Moreover, SMARTedit should be more robust to user errors, such as performing actions in a different order in subsequent examples or pressing the wrong key at the wrong time. A more robust SMARTedit implementation would discount the mistakes, allow the user to correct them, or even automatically correct for the errors.

Together, these directions form an ambitious plan for improving SMARTedit: asking questions and proposing examples to direct the learning process, taking advantage of the user's narration of her actions, accumulating knowledge about context (users, texts, and tasks), and balancing

the advantages of all of these against their burden on the user. Principled use of these techniques will result in fundamental improvements to SMARTedit—improvements that should carry over to other domains and PBD systems. A more intelligent student will ease the burden on a teacher in any domain.

11.6 Other Directions for SMARTedit

There are many other avenues of research to explore. In the immediate future, we plan to increase SMARTedit's expressiveness by elaborating SMARTedit's state representation (perhaps including features such as sentence, paragraph, and section numbers) and adding to the set of version spaces used to construct programs. Moreover, we plan to construct a larger corpus of repetitive text-editing scenarios with which to evaluate the system's performance. We expect these two avenues to complement each other, suggesting new capabilities to introduce into SMARTedit and highlighting its limitations.

Another dimension to pursue is to evaluate the system's ability to scale, both to larger and more expressive languages, and even to other domains. Will the version space approach suffice when the number of different functions grows to the hundreds or thousands? Is the version space approach appropriate for other domains besides text editing, such as spreadsheets or the desktop? We believe that careful construction of the version space hierarchy will allow our approach to scale; we envision component version spaces becoming part of a generally applicable, reusable library. Furthermore, we believe that the representation of procedures as functions over states is quite general, but undoubtedly we will find limitations in our approach when we consider different domains.

11.7 Comparison with Other Text-Editing PBD systems

Unlike most previous text-editing PBD systems, SMARTedit uses a formal machine learning technique to describe the generalization that is performed by the system. Witten and Mo (1993) describe the TELS system that records high-level actions similar to the actions used in SMARTedit and implements a set of expert rules for generalizing the arguments to each of the actions. TELS also uses heuristic rules to match actions against each other to detect loops in the user's demonstrated program; it outperforms

SMARTedit in this respect. However, TELS's dependence on heuristic rules to describe the possible generalizations makes it difficult to imagine applying the same techniques to a different domain, such as spreadsheet applications.

Nix (1985) describes the Editing by Example (EBE) system that looks not at recorded actions but at the input/output behavior of the complete demonstration. EBE attempts to find a program that could explain the observed difference between the initial and final state of the text editor. In this respect, SMARTedit is a refinement of EBE that uses not only the initial and final states but intermediate states as well. SMARTedit's approach has the drawback that it is sensitive to the order in which the user chooses to perform actions; on the other hand, it is making use of more information than EBE is given, and so SMARTedit is able to learn programs for more complex text transformations than EBE.

Masui and Nakayama (1994) describe the Dynamic Macro system for recording macros in the Emacs text editor. Dynamic Macro performs automatic segmentation of the user's actions, breaking up the stream of actions into repetitive subsequences, without requiring the user to invoke the macro recorder explicitly. Dynamic Macro performs no generalization, and it relies on several heuristics for detecting repetitive patterns of actions.

Maulsby and Witten's (1997) Cima system uses a classification rule learner to describe the arguments to particular actions, such as a rule describing how to select phone numbers in the local area code. (SMARTedit is able to learn a program to select all but one of the phone numbers given a single demonstration. The anomalous phone number lacks a preceding area code and is also difficult for Cima to classify correctly.) Unlike other PBD systems, Cima allows the user to give "hints" to the agent that focus its attention on certain features, such as the particular area code preceding phone numbers of interest. However, the knowledge gained from these hints is combined with Cima's domain knowledge using a set of hard-coded preference heuristics. As a result, it is never clear exactly which hypotheses Cima is considering or why it prefers one over another. In SMARTedit, these types of hints could be used to bias the probabilities on its different hypotheses.

11.8 Conclusion

We have described the SMARTedit PBD system that automates repetitive text-editing tasks. SMARTedit represents text-editing actions as functions

from one text-editing state to another and uses version space algebra to represent efficiently the set of functions that are consistent with the demonstrated examples. The system learns useful text-editing procedures based on a very small number of demonstrations.

SMARTedit's version space algebra representation allows it simultaneously to maintain different beliefs about the user's desired actions. It keeps a record of all functions that could possibly explain the observed series of state changes and throws out only those functions that are inconsistent with the observed data. This representation allows SMARTedit to fail gracefully—if its best guess for the user's next action is not correct, it can fall back to the next best guess, and so on, rather than failing completely.

We believe that the holy grail of an intuitive, intelligent, and flexible PBD system is within reach. The technologies behind SMARTedit are some of the first steps toward this goal.

References

Deerwester, S., S. Dumais, G. Furnas, T. Landauer, and R. Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, no. 6: 391–407.

Lau, T., P. Domingos, and D. S. Weld. 2000. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning.*

Masui, T., and K. Nakayama. 1994. Repeat and predict—Two keys to efficient text editing. In *Human factors in computing systems: CHI '94 Conference Proceedings.* Reading, Mass.: Addison-Wesley.

Maulsby, D. and I. H. Witten. 1997. Cima: An interactive concept learning system for end-user applications. *Applied Artificial Intelligence* 11, nos. 7–8: 653–671.

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

Nix, Robert P. 1985. Editing by example. *ACM Transactions on Programming Languages and Systems* 7, no. 4: 600–621.

Witten, I. H., and D. Mo. 1993. TELS: Learning text editing tasks from examples. In *Watch What I Do: Programming by Demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

S
R
L