Why Programming by Demonstration Systems Fail: Lessons Learned for Usable AI

Tessa Lau

■ *Programming by demonstration systems have long attempted to make it possible for people to program computers without writing code. However, while these systems have resulted in many publications in AI venues, none of the technologies have yet achieved widespread adoption. Usability remains a critical barrier to their success. On the basis of lessons learned from three different programming by demonstration systems, we present a set of guidelines to consider when designing usable AI-based systems.*

The goal of programming by demonstration (PBD) is to enable ordinary end users to create programs without needing to learn the arcane details of programming languages, but simply by demonstrating what their program should do. If PBD were successful, the vast population of nonprogrammer computer users would be able to take control of their computing experience and create programs to automate routine tasks, develop applications for their specific needs, and manipulate information in service of their goals.[1] However, PBD has yet to achieve widespread adoption, partly because the problem is extremely difficult. How can any system successfully guess the user's intended program out of an infinite space of possible programs?

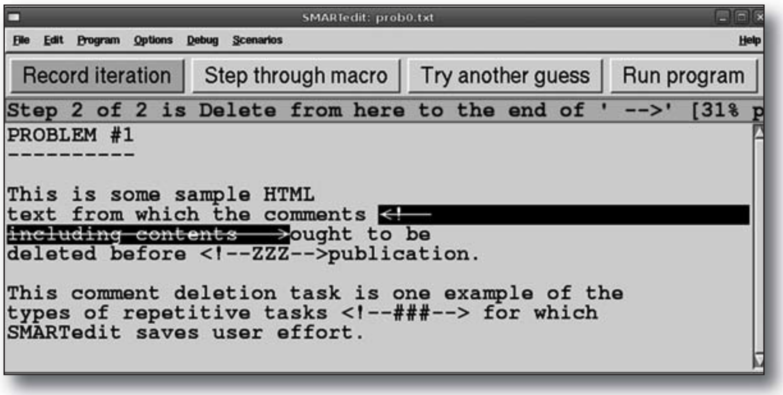
PBD is a natural match for artificial intelligence, particularly machine learning. By observing the actions taken by the user (training examples), the system can create a program (learned model) that is able to automate the same task in the future (predict future behavior). However, unlike most machine-learning systems that can rely on hundreds or thousands of training examples, users are rarely willing to provide more than a handful of examples from which the system can generalize. This constraint makes the design of machine-learning algorithms for PBD extremely challenging: they must learn accurately from an absurdly small number of user-provided training examples.

However, when designing machine-learning algorithms for use in a user-facing system, accuracy is not the only important factor. Researchers' experience designing and deploying machine-learning-based PBD systems reveals several factors that prevent users from wanting to use such systems. This paper presents some of the lessons researchers have learned about making AI systems usable.

Case Studies: Three Systems

This article presents case studies of three programming by demonstration systems that employ varying amounts of machine learning to intelligently predict user behavior.

SMARTedit (Lau et al. 2003) is a text editor that uses PBD to



```
Step 2 of 2 is Delete from here to the end of ' -->' [31% p
PROBLEM #1
----------

This is some sample HTML
text from which the comments ~~<!--~~
~~including contents -->~~ought to be
deleted before <!--ZZZ-->publication.

This comment deletion task is one example of the
types of repetitive tasks <!--###--> for which
SMARTedit saves user effort.
```

Figure 1. The SMARTedit User Interface

automate repetitive text-editing tasks (see figure 1). For example, when reformatting text copied and pasted from the web into a document, one can demonstrate how to reformat the first line or two of text, and the system learns how to reformat the remaining lines. The system is based on a novel machine-learning algorithm called version space algebra, which uses multiple examples incrementally to refine its hypotheses as to the user's intended actions.

SMARTedit was later reimplemented within the context of a word processor product (based on OpenOffice), though that feature was never released. During the development process, my colleagues and I solicited user feedback on the resulting system and learned that poor usability was the key barrier to acceptance.

Sheepdog (Lau et al. 2004) is a PBD system for learning to automate Windows-based system administration tasks based on traces of experts performing those tasks. For example, based on several demonstrations of experts fixing the configuration of a Windows laptop in different network environments (static IP, dynamic IP, different DNS servers), the system produced a procedure that could apply the correct settings, no matter what the initial configuration was. The system uses an extension to input-output hidden Markov models (Oblinger et al. 2005) to model the procedure as a probabilistic finite state machine whose transitions depend on features derived from the information currently displayed on the screen.

CoScripter (Little et al. 2007) is a PBD system for capturing and sharing scripts to automate common web tasks. CoScripter can be used both to automate repetitive tasks, as well as share instructions for performing a task with other users. For example, based on watching a user search for real estate using a housing search site, CoScripter automatically creates a script that can be shared with other users to replay the same search. The system uses a collection of heuristics to record the user's actions as a script. A script is represented as human-readable text containing a bulleted list of steps; users can modify the program and change its behavior simply by editing the text. A smart parser interprets each script step in order to execute the instruction relative to the current web page.

Design Guidelines for Usable AI

During the course of developing these systems, researchers conducted user studies and collected informal user feedback about each system's usability. This section summarizes some of the observations my colleagues and I made.

Detect failure and fail gracefully. SMARTedit's learning algorithm does not have a graceful way to handle noise in training examples. For example, if the user makes a mistake while providing a training example, or if the user's intent is not expressible within the system, the system collapses the version space and makes no predictions. The only action possible is to start over and create a new macro. Users who do not have a deep understanding of the workings of the algorithm, and who just expect the system to magically work, would be justifiably confused in this situation.

CoScripter's parser does a heuristic parse of each textual step; because there is no formal syntax for steps, the heuristics could incorrectly predict the wrong action to take. When the system is used to automate a multistep task, one wrong prediction in the middle of the process usually leads the entire script astray. When this happens, users have observed that users are confused because the system says it has completed the script successfully, even though it diverged from the correct path midway through the script and did not actually complete the desired task. Few users monitor the system's behavior closely enough to detect when it has not done what it said it was going to do.

Make it easy to correct the system. Sheepdog's learning system takes as input a set of execution traces and produces a learned model. If the learned model fails to make the correct predictions, the only way to correct the system is to generate a new execution trace and retrain the system on the augmented set of traces. Similarly, SMARTedit's users complained that they wanted to be able directly to modify the generated hypotheses (for example, "set the font size to 12") without having to retrain the system with additional examples. One challenge for machine learning is the development of algorithms whose models can be easily corrected by users without the need for retraining.[2]

Encourage trust by presenting a model users can understand. The plain-text script representation used in CoScripter is a deliberate design chosen to let users read the instructions and trust that the system will not perform any unexpected actions. The scripting language is fairly close to the language people already use for browsing the web, unlike the language used in SMARTedit where users complained about arcane instructions such as "set the CharWeight to 1" (make the text bold). SMARTedit users also thought a higher-level description such as "delete all hyperlinks" would be more understandable than a series of lower level editing commands; generating such a summary description is a challenge for learning algorithms.

Sheepdog's procedure model is a black-box hidden Markov model, and the only way to see what a procedure would do is to run it. The system administrators who were the target audience for Sheepdog were uncomfortable with the idea that a procedure they created and sent to a client might accidentally wipe the client's hard disk. A prediction accuracy of 99 percent might seem to be good enough for most systems; however, if that remaining 1 percent could cause destructive behavior, users will quickly lose faith in the system.[3]

Enable partial automation. The naming of the Sheepdog system suggests that the users of the system are "sheep" who blindly follow the recommendations of the system. Yet users often have knowledge about their task that is not known to the system, and they often want to take advantage of partial automation while incorporating their own customizations. Early versions of Sheepdog assumed that all actions users performed were in service of the automated task, and would fail if (for example) an instant message popped up unexpectedly in the middle of the automation. Intelligent systems should be able to cope with interruptions and allow users to modify the automated system's behavior without derailing the automation.[4]

Consider the bottom-line value of automation. The benefits of automation must be weighed against the cost of using the automation. For PBD systems the cost includes invoking the system, teaching it the correct procedure, and supervising its progress.

For example, SMARTedit was originally implemented as a standalone text editor, rather than integrated into existing editors. The cost of switching to SMARTedit for the sake of a quick text edit was perceived as too high; for simple editing tasks, users felt they could complete the task more quickly by hand. With CoScripter, several users have complained that finding the right script to automate a repetitive task took longer than simply doing the task by hand. In both cases, automation was considered worthwhile by users only for long or tedious tasks, even though it could have been worthwhile for a broader range of tasks, if its costs had been reduced. Designers should take users' pain points into account when judging where automation is likely to be accepted.[5]

Discussion and Conclusions

Based on researchers' experience with several machine-learning-based programming by demonstration systems, I have characterized many of the usability issues that serve as barriers to widespread adoption of such systems. Ultimately the solution will require not only technical improvements to the underlying algorithms (for example, Chen and Weld [2008]) but also improved interaction designs that take into account the strengths and weaknesses of AI-based solutions. Building truly usable AI systems will require contributions from both the AI and human-computer interaction (HCI) communities, working together in concert.[6]

Notes

1. In the terms of Lieberman's theme article on usability benefits (Lieberman 2009) PBD is one of the ways in which AI can help realize the principle that interfaces should accomplish interactions in ways that make efficient user of user input.

2. These first three examples illustrate some of the many forms that can be taken by failures of system intelligence, their consequences, and the procedures by which users compensate for them (compare the usability side-effects theme article by Jameson, in this issue).

3. These two examples illustrate how a lack of comprehensibility can lead to a lack of controllability (compare the usability side-effects theme article). In this case, what needs to be comprehensible is not the machine learning that forms the core of the system but rather the product of the learning.

4. This partial automation can be seen as another way of giving the user some measure of control—though here what the user needs is not so much an understanding of the learned procedure as opportunities to influence the course of its execution.

5. These two examples illustrate two very different forms of the usability side effect of the user's having to make disproportionate effort in order to get the intelligent functionality to work properly.

6. The concluding remarks reflect the "binocular view" of the design of intelligent interactive systems discussed in the Introduction.

References

Chen, J., and Weld, D. S. 2008. Recovering from Errors during Programming by Demonstration. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*. New York: Association for Computing Machinery.

Lau, T.; Bergman, L.; Castelli, V.; Oblinger, D. 2004. Sheepdog: Learning Procedures for Technical Support. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI),* 327–329. New York: Association for Computing Machinery.

Lau, T.; Wolfman, S.; Domingos, P.; and Weld, D. S. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53(1–2).

Lieberman, H. User Interface Goals, AI Opportunities. *AI Magazine* 30(4).

Little, G.; Lau, T.; Cypher, A.; Lin, J.; Haber, E.; Kandogan, E. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI'07). New York: Association for Computing Machinery.

Oblinger, D.; Castelli, V.; Lau, T.; Bergman, L. 2005. Similarity-Based Alignment and Generalization. In *Proceedings of the 16th European Conference on Machine Learning*. Berlin: Springer.

Tessa Lau is a research staff manager at IBM's Almaden Research Center in San Jose, CA. Lau's research integrates techniques from AI and human-computer interaction to build systems that enhance human productivity and creativity; areas of interest include programming by demonstration, collaboration, and social software. She has served on organizing and program committees for major AI and HCI conferences and journals. She also serves on the board of CRA-W, the CRA committee on the status of women in computing. Lau holds a PhD in computer science from the University of Washington.