
Why PBD systems fail: Lessons learned for usable AI

Tessa Lau

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120 USA
tessalau@us.ibm.com

Abstract

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. Based on lessons learned from three different programming by demonstration systems, we present a set of guidelines to consider when designing usable AI-based systems.

Introduction

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 non-programmer 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. 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?

Copyright is held by the author/owner(s).
CHI 2008, April 5 – April 10, 2008, Florence, Italy
ACM 1-xxxxxx

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. Our 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 we have learned about making AI systems usable.

Case studies: Three systems

In the course of our research, we have developed three programming by demonstration systems that employ varying amounts of machine learning to intelligently predict user behavior.

SMARTedit [3] is a text editor that uses PBD to automate repetitive text-editing tasks. 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 our feature was never released. During the development process, we solicited user feedback on the resulting system and learned that poor usability was the key barrier to acceptance.

Sheepdog [2] 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* [5] 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 [4] 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, we conducted user studies and collected informal user feedback about each system's usability. This section summarizes some of our observations.

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 multi-step task, one wrong prediction in the middle of the process usually leads the entire script astray. When this happens, we 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 to directly modify the generated hypotheses (e.g., "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.

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 HMM, 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% might seem to be good

enough for most systems; however, if that remaining 1% could cause destructive behavior, users will quickly lose faith in the system.

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.

Consider the perceived 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 *perceived* to be useful only for

long or tedious tasks, even though it *could* have been applied to a broader range of tasks. Designers should take users' pain points into account when deciding where automation can be successfully applied.

Discussion and Conclusions

Based on our experience with several machine learning-based programming by demonstration systems, we 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 (e.g., [1]) but also improved 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 HCI communities, working together in concert.

References

- [1] Chen, J. and Weld, D. S. Recovering from Errors during Programming by Demonstration. In *Proc. IUI 2008*.
- [2] Lau, T., Bergman, L., Castelli, V., Oblinger, D. Sheepdog: Learning Procedures for Technical Support. In *Proc IUI 2004*, ACM Press (2004).
- [3] Lau, T., Wolfman, S., Domingos, P. and Weld, D.S. Programming by Demonstration using Version Space Algebra, *Machine Learning 53*, 1-2 (2003).
- [4] Little, G., Lau, T., Cypher, A., Lin, J., Haber, E., Kandogan, E. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proc. CHI 2007*, ACM Press (2007).
- [5] Oblinger, D., Castelli, V., Lau, T., Bergman, L. Similarity-based alignment and generalization. In *Proc. ECML 2005*, Springer (2005).