A Software Agent for Note Taking

Jeffrey C. Schlimmer
schlimme@eecs.wsu.edu

Leonard A. Hermens
lhermens@eecs.wsu.edu

School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752

Introduction and Motivation

People like to record information for later consultation. Paper is easy to use, inexpensive, and durable. To its disadvantage, paper records do not scale well. As the amount of information grows, retrieval becomes inefficient, physical storage becomes excessive, and duplication and distribution become expensive. In contrast, digital media offers better scaling capabilities, and it is clear that our computing environments are evolving as several vendors are beginning to market inexpensive, hand-held, highly portable computers that can convert handwriting into text. We view this as the start of a new paradigm shift in how digital information will be gathered and used. It is in this paradigm that our research is inspired. Our primary goal is to combine the best of both worlds by making digital media as convenient as paper.

This document describes interactive note-taking software for computers with pen-based input devices. Our software has two distinctive features: first, it actively predicts what the user is going to write and provides a default that the user may select; second, it automatically constructs a graphical interface at the user's request. Viewed in a larger context, the interactive note-taking system is a type of self-customizing software—it uses machine learning techniques to automatically customize task-generic software to a specific user. Because the software learns to assist the user by watching them complete tasks, the software is also a learning apprentice. Similarly, because the user does not explicitly program the defaults or the user interface for the note taking system, it is a type of software agent. Agents are a new user interface paradigm that frees the user from having to explicitly command the computer. With our software, the user can record information directly and in a free-form manner. Behind the interface, the software is acting on behalf of the user, helping to capture and organize the information.

Performance Task

The primary function of the note-taking software is to improve the user's speed and accuracy as they enter notes about various domains of interest. A note is a short, sequence of descriptive terms that describe a single object of interest. Example 1 shows a note describing a particular personal computer (recorded by the first author from for-sale offerings in a Usenet newsgroup from 1992):

4096K PowerBook 170, 1.4MB and 40MB Int. Drives,
2400/9600 Baud FAX Modem (Example 1)

The user may enter notes from different domains at their convenience and may use whatever syntactic style comes naturally.

From the user's point of view, the software operates in one of two modes: a contextual prompting mode, and an interactive graphical interface mode. In the first mode, the software continuously predicts a likely completion as the user writes out a note. The completion is offered as a default for the user. To indicate a completion, a small, colored button follows to the left and below where the user is writing. The default text is displayed to the

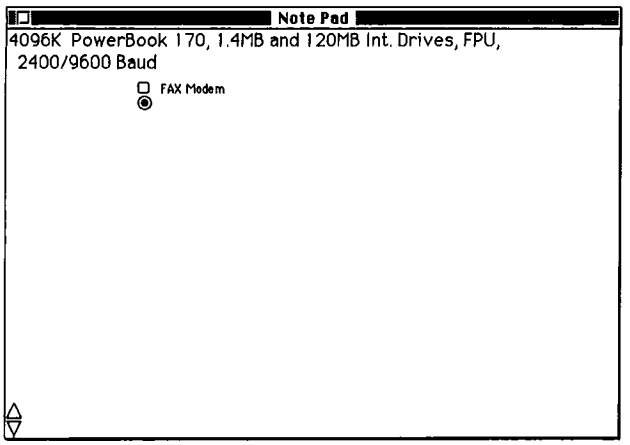

Figure 1: Screen snapshot of the note-taking software in contextual prompting mode for a PowerBook note. The two triangles in the lower left are scrolling buttons.

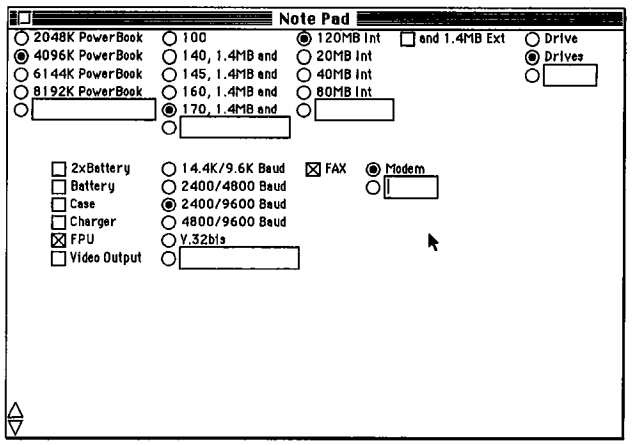

Figure 2: Screen snapshot of the note-taking software in button-box mode for a PowerBook note.

immediate right of this button in a smaller font. The button's saturation ranges from 1 (appearing green), when the software is highly confident of the predicted value, to 0 (appearing white), when the software lacks confidence. Figure 1 portrays a screen snapshot of the software operating in the contextual prompting mode for a PowerBook note.

The software's second mode presents an interactive graphical interface. Instead of requiring the user to write out the text of a note, the software presents a radio-button and check-box interface. (What we call a *button-box* interface.) With this, the user may enter portions of notes by tapping on buttons with the pen interface device. Intuitively, check boxes are generated to depict optional descriptive terms, whereas radio-button panels are generated to depict alternate, exclusive descriptive terms. The software indicates its predictions by preselecting buttons. These default selections may be easily overridden by tapping the desired buttons. Figure 2 portrays a screen snapshot of the software operating in the interactive graphical interface mode for a PowerBook note.

Learning a Syntax

To implement the two modes of the note taking software, the system internally learns two structures. To characterize the syntax of user's notes, it learns finite state machines. To generate predictions, it learns decision tree classifiers situated at states within the finite state machines. In order to construct a graphical user interface, the system converts a finite state machine into a set of buttons.

Deterministic finite state machines (or FSMs) are one candidate approach for describing the syntax of a user's notes because they are well understood and relatively expressive. Moreover, Angluin (1982) and Berwick and Pilato (1987) present a straightforward algorithm for learning a specific class of FSMs called k-reversible FSMs. Given a list of tokens, the k-reversible FSM algorithm first constructs a prefix tree, where all token sequences with common k-leaders share a k-length path through the FSM. The algorithm uses state merging to create a FSM that accepts all token sequences in the prefix tree, as well as other candidate sequences (i.e., all sequences accepted by the k-reversible FSM). Angluin's merging rules are generalization operators that allow the FSM to accept previously unobserved sequences. These comprise a heuristic bias and may be too conservative. Table 1 lists specializations of Angluin's origi-

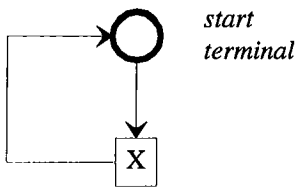

Figure 3: Simple finite-state machine with one state.

nal rules and adds a pair of rules we developed to make the FSM generalize more readily. Effectively, two states are merged by Rules 3a or 2′ if they share an incoming or outgoing transition. Rule 3b is a Kleene rule that encourages the FSM to generalize the number of times a token may appear in a sequence. If one state has a transition to another, then merging them will result in a transition that loops from and to the newly merged state.

Table 1. Extended FSM state merging rules.

Merge any two states if any of the following are true:

1. Another state transitions to both states on the same token; or

   (This enforces determinism.)

2′. Both states have a common 0-leader and

   a. Both states are accepting states, or

   b. Both states transition to a common state via the same token; or

3. Both states have a common 1-leader and

   a. Both states transition to a common state via any token, or

   b. One transitions to the other via any token.

Learning Embedded Classifiers

Embedding general classifiers in a FSM can alleviate some of the FSM's representational shortcomings. For example, in the FSM depicted in Figure 3, a decision tree embedded in this state easily tests whether the transition has already been taken and can advise against repeating it. Moreover, a classifier can predict based on previous transitions rather than just the frequency of the current state's transitions. To be precise, a classifier is trained for each state in the FSM which: (a) has more than one transition, or (b) is marked as a terminal state but also has a transition. The classifiers are updated incrementally after the user finishes each note. The classifier's training data are token sequences parsed at this state. The class value of the data is the transition taken from or termination at this state by the token sequences. The attributes of the data are the names of states prior to this one, and the values of the attributes are the transitions taken from those states.

Constructing a Button-Box Interface

In the button-box mode, the software presents an interactive graphical interface. Instead of writing out the note, the user may select note fragments by tapping buttons. To switch from contextual mode to button-box mode, a green radio-button indicator is displayed below the completion button. If the user taps this indicator, existing text is removed from view and the display is converted into a button-box interface. Buttons corresponding to the removed text are displayed as selected defaults.

The button-box interface is a direct presentation of a finite state machine. The mapping of a FSM into radio buttons and check boxes proceeds one state at a time. Any transition that starts a path, does not branch, and eventually returns back to the state (a loop in the FSM) is rendered as a check box. Other non-looping transitions are rendered as buttons in a single radio button panel along with an extra, unlabeled button.

Acknowledgments

We would like to thank Mike Kibler, Karl Hakimian, and the EECS staff for providing a consistent

and reliable computing environment. We are also thankful to Apple Cambridge for the Macintosh Common Lisp programming environment. Allen Cypher provided the tokenizer code. This work was supported in part by the National Science Foundation under grant number 92-1290, by NASA under grant number NCC 2-794, and by Digital Equipment Corporation.

REFERENCES

Angluin, D. (1982). Inference of reversible languages. *Journal of the Association for Computing Machinery, 29*, 741–765.

Berwick, R. C., & Pilato, S. (1987). Learning syntax by automata induction. *Machine Learning, 2*, 9–38.