Experiments in UNIX Command Prediction

Brian D. Davison and Haym Hirsh
Department of Computer Science
Rutgers, The State University of New Jersey
{davison,hirsh}@cs.rutgers.edu

August 1997

Abstract

A good user interface is central to the success of most products. Our research is concerned with improving an interface by making it adaptive — changing over time as it learns more about the user. In this paper we consider the task of modifying a UNIX shell to learn to predict the next command executed as one sample adaptive user interface. To this end, we have collected command histories (some extensive) from 77 people, and have calculated the predictive accuracy for each of five methods over this dataset. The algorithm with the highest performance produces an average online predictive accuracy of up to 38%.

1 Introduction

A good user interface is central to the success of operating systems, applications, and in general, most consumer products. Our research is concerned with improving an interface by making it adaptive — changing over time as it learns more about the user. To this end, we have collected and analyzed a sizable set of command usage histories from which we may determine appropriate settings and mechanisms to implement a command prediction system. This paper discusses how the data was collected and how well a sampling of algorithms perform on it.

Our motivation is to understand the principles of adaptive command prediction. While there are other human/computer interfaces, the command line interface provides a convenient testbed to explore such issues. It is in widespread use and allows easy data collection and has strong potential for technology integration into the interface. We believe that the same techniques are applicable to voice interfaces, and to menu selection in graphical user interfaces.

The prediction of a user's next command by the shell requires some mechanism to consider the user's current session and from that generate a prediction

for the next command. We have used a "learning apprentice" approach [14] to gather data by recording the sequence of commands executed as well as any pertinent information about the state of the shell. This forms the basis for applying standard inductive-learning methods that extrapolate from such data a procedure that will make a prediction of the user's next command after examining the current state of the shell (and perhaps knowing about past performance). One nice property of this approach is that data are collected passively, during a user's regular interactions with the shell, rather than requiring special data or training by the user as is the case, for example, with much of the work in programming by demonstration [3].

In our present research [8, 5], we focus exclusively on command prediction. This means that we do not attempt to predict command parameters, and so our methods do not completely specify a user's next action. This is discussed further at the end of the paper.

We consider command prediction to be a different, and in general, a more complex problem than the traditional machine learning tasks. The reason is this — instead of learning a boolean membership function of some concept, command prediction requires that the function learned be one that selects the most likely next command. One possible implementation is to learn membership functions for all possible commands, and choosing the one with the highest confidence to produce the output for the system. This is not the approach we have taken.

There are many attributes of a user's session that may be relevant to what he or she is likely to do next. This includes a portion or all of the user's past commands, both in this session and from previous sessions. It may also include information like the name of the machine, the time or date, the type of terminal emulation used, or the current directory.

The next section provides an overview of related work. The rest of the paper describes our experimental method, followed by experimental results, including a critical analysis, a description of an simplified prototype shell, and finally some concluding thoughts.

2 Related Work

There has been a range of work developing systems that recognize regularities in the usage of a computer. Motoda and Yoshida's [15, 22, 21] investigation of the use of machine learning to predict a user's next command is the most similar to the work reported here, but does not consider standard algorithms and has a small dataset. Also similar is work in programming by demonstration [3], such as Cypher's Eager [2], which recognizes and automates simple loops in user actions in a graphical interface. Finally, Lesh and Etzioni [9] also consider UNIX commands in plan recognition.

Other examples of action prediction include interface agents that learn to provide assistance [11], the prediction of which link on a WWW page will be

traversed by a user [1], the time and place for meetings for an on-line calendar system [6, 13], the values of fields in a form-filling system [7], the next thing a user will write on a pen-based computer [19, 18], and the next key a user will press on a calculator [20] or computer keyboard [4, 12].

3 Approach and Methodology

3.1 Who the test subjects were

We captured command histories from a total of 77 people at Rutgers University. Two of these were faculty (including the second author), five were graduate students (including the first author), and the other 70 were undergraduates in an Internet programming course. All subjects were aware that their commands were being logged, and either had the option of turning the logging off or had explicitly turned it on to facilitate our research.

Collection periods ranged from two months to over six months. Most subjects had access to systems on which we were not capturing command histories. Nevertheless, these histories represent the primary work performed online by the authors and the participating graduate students. The undergraduate data was collected over two months on computer systems dedicated to their programming projects, and not their primary systems, and so the patterns of commands selected reflect the orientation of their use (i.e. editing and compilation rather than mail).

3.2 How we collected data

We collected data unobtrusively, by causing the `tcsh` shell (at its closing) to record the command history in a time-stamped file, along with the current hostname and terminal type. This method was used to minimize potential interference with the user's activities.

One drawback to this method is that for users who stayed logged in for days or weeks at a time (common for those with private offices), the command histories would not be recorded until they did finally log out, and would not be recorded at all if interrupted by a power outage or system crash. Another is that the commands executed are not necessarily recorded in order; while the commands are ordered properly within a single shell history, many users take advantage of modern user interfaces which allow multiple shells to be active simultaneously.

In early testing of data collection, we did, in fact, use a relatively obscure `tcsh` mechanism to record each command into a log after it had been executed. This caused occasional run-time problems which were never fully resolved. A small part of the command histories for the two authors was collected via this mechanism anyway.

```
...
96032013:05:23 athos.rutgers.edu xterm BLANK
96032013:05:23 athos.rutgers.edu xterm netscape2.0
96032013:05:23 athos.rutgers.edu xterm rm
96032013:05:23 athos.rutgers.edu xterm netscape2.0
96032013:05:23 athos.rutgers.edu xterm emacs
96032015:59:28 pei.rutgers.edu xterm BLANK
96032015:59:28 pei.rutgers.edu xterm im
96032015:59:28 pei.rutgers.edu xterm cd
96032015:59:28 pei.rutgers.edu xterm cd
96032015:59:28 pei.rutgers.edu xterm gunzip
96032015:59:28 pei.rutgers.edu xterm core3
96032015:59:28 pei.rutgers.edu xterm im
96032015:59:28 pei.rutgers.edu xterm lpr
96032015:59:28 pei.rutgers.edu xterm im
96032015:59:28 pei.rutgers.edu xterm ghostview
...
```

Figure 1: A portion of one user's history.

3.3 Format of collected data

We converted the command histories for each subject into single history files where each line contains the date/time of the start of the shell (as our estimate of the date/time of execution, used just to order the command history), the name of the machine used, the terminal emulation type for this session, and the command typed (parameters were stripped, and the commands may be user-defined aliases as well as executables in the user's path). When an attribute value was unknown (usually the terminal type), the string NONE was substituted. To facilitate the recognition of a new session, the first entry is the non-existent command BLANK. This generic history file was then converted to whatever format was necessary for the algorithm under consideration.

3.4 Our implementation

3.4.1 Algorithms

This paper describes the results of applying a total of five methods (along with some variations on the problem) to this data. We chose to run multiple methods both for comparison and verification of performance. The first is a fairly complex algorithm; the others are simplistic baselines or methods constructed specifically for this problem.

- The first was C4.5 [17], and was selected as a common, well-studied decision-tree learner with excellent performance over a variety of problems.
- An Omniscient predictor was devised that would correctly predict every command, providing that the desired command was present in the current training set. This provides an upper-bound on the potential predictive accuracy of any learner.
- At the other end we computed the usefulness of predicting the same command that was just executed (Most Recent Command). While this doesn't provide the same kind of strict lower bound that random guessing would provide (a bound at times near zero), it still provides a useful strawman.
- Another simplistic algorithm we used is one that predicts the Most Frequent Command in the training set.
- The last significant algorithm was constructed to operate similarly to a hardware cache or a string matching mechanism. It predicts the command (within the historical buffer) that has the longest matching Prefix to the current command. For example, the longest possible prefix is one in which all the commands in the current session (i.e. back to the starting 'BLANK' token) are matched.

With the exception of C4.5 (for which we used the default distribution), each of the algorithms was implemented in Perl and simple UNIX shell scripts.

3.4.2 Variations in Testing

For each algorithm, we varied the size of the historical buffer. The buffer sizes were somewhat arbitrary, at n=100 commands, 500 commands, and 1000 commands. However, in earlier attempts, we found that some kind of limit was necessary, as the experiments required too much memory to run in a reasonable amount of time on Sun Ultrasparcs (for the subjects with the longest history files).

For C4.5, the attributes were the previous commands, varying in number from 1 to 4.

Where relevant, we performed the same experiment with and without the two additional attributes of terminal type and machine name.

3.4.3 Calculating Results

The resulting predictive accuracies for each method were performed in an online fashion (i.e. test on the current command and use the preceding n commands as the historical buffer for training), and with two variations. We computed the average accuracy for each subject over all the commands in the user's history,

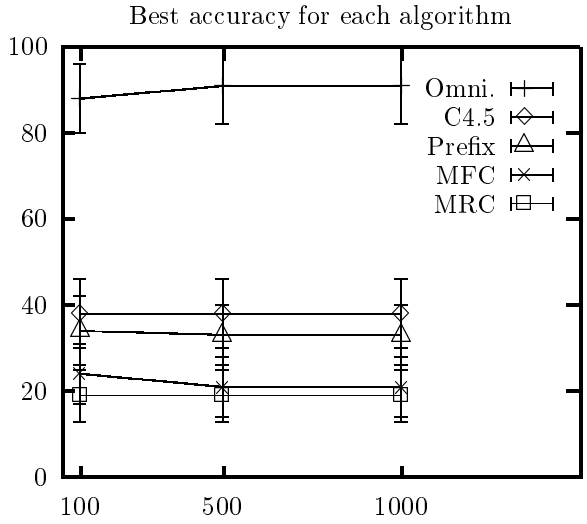

Figure 2: For each history buffer size, the highest macroaverage performance achieved for each algorithm is shown.

and we computed the same but without the first n commands in the history. While the latter means that a number of subjects were not able to produce results for certain tests, it may give a better estimate of asymptotic accuracy as it eliminates the startup period in which the algorithm is trained on less than n commands.

Finally, we both computed the average per person predictive accuracy (which we term macroaverage), and the average overall (i.e. over all commands in the study) predictive accuracy (microaverage).

4 The Experiments

4.1 Experimental Results

The 77 subjects ranged from a minimum of 15 commands to a maximum of 34940 commands, and had an average of 2184 ($\pm$ 4389) commands, each with an average length of 3.77 characters. The number of distinct commands per user varied widely. On average, each subject used 77 ($\pm$ 98) distinct commands. Overall, the most popular command was ls (occuring 13% of the time, followed by cd (7.5%), gcc (6.4%), emacs (5.8%), and more (4.6%). However, only 32 of the subjects had ls as their most frequent comand.

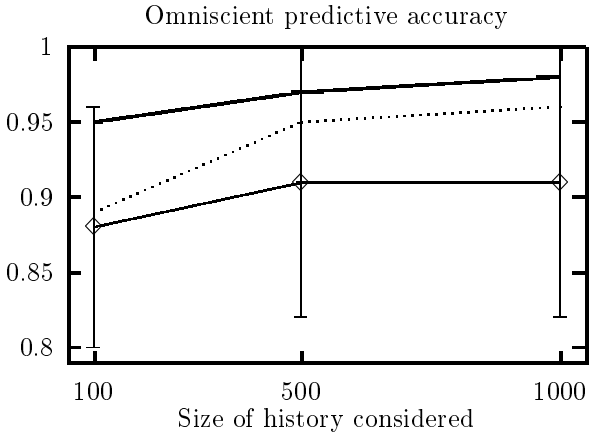

Figure 3: For each history buffer size, highest accuracy recorded, the microaverage accuracy, and the macroaverage accuracy (bounded by the standard deviation) are shown (in order, from highest to lowest curve).

The best macroaverage predictive performance was with C4.5, which achieved 38±8% accuracy. The microaverage was just a little worse, at 37%. The Prefix matching algorithm was just 4-6% behind at 34±8% and 31% respectively. Figure 2 shows the best scores for each algorithm investigated (given the best possible circumstances). The values shown represent averages over all commands — eliminating the predictions during startup increases the resulting accuracy by another percent or two.

To show a lower bound, we computed the accuracy of predicting the Most Recent Command. This was at minimum 4% accurate, up to 36% accurate. The macroaverage was 19±6%, with a microaverage of 22%.

As an upper bound, we found that an omniscient predictor had an average performance that ranged from 89% to 96% and never did worse than 47% (for a person with only 19 commands in his history file). This is the focus of Figure 3, which also suggests that a buffer size of at least 500 may be useful to capture more of the possible commands to be used.

C4.5 had the most variables of any of the algorithms. In addition to the extra attributes, the size of the training set, and whether or not to include startup performance, our C4.5 experiments also varied the number of previous commands used as attributes to the learner. The main liability in adding more attributes is that the more that are included, the more likely it is that baseless correlations will be found in the data, yielding inferior command predictions. This question

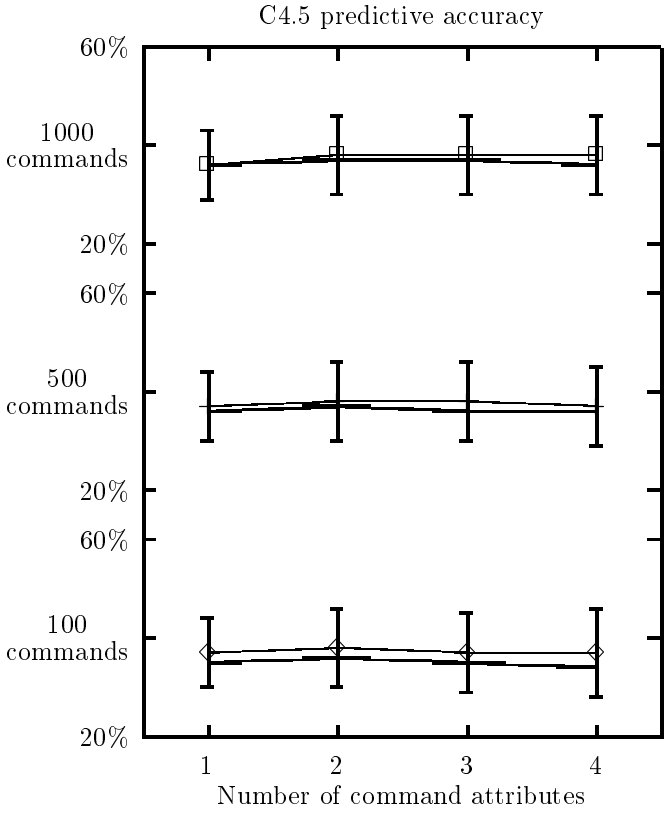

Figure 4: The top set, middle set, and bottom set reflect the training set sizes of 100, 500, and 1000 respectively. Bracketed curves show the macroaverages (± standard deviations); plain curves represent the microaverages.

is the focus of Figure 4, which suggests that 2 previous commands is slightly better than 1, 3, or 4. This graph shows scores without extra attributes, and includes the startup period. While the extra attributes do not change the graph much, eliminating the startup period increases average performance by 1-2%.

While not a strong performer, the Most Frequent Class classifier (shown in Figure 5) has similar curves as the other algorithms. This graph shows data without extra attributes and includes startup scores. If extra attributes were included, some points would be a few percent better. Eliminating startup scores did not significantly affect performance.

Finally, the algorithm that predicted based on historical prefix matching

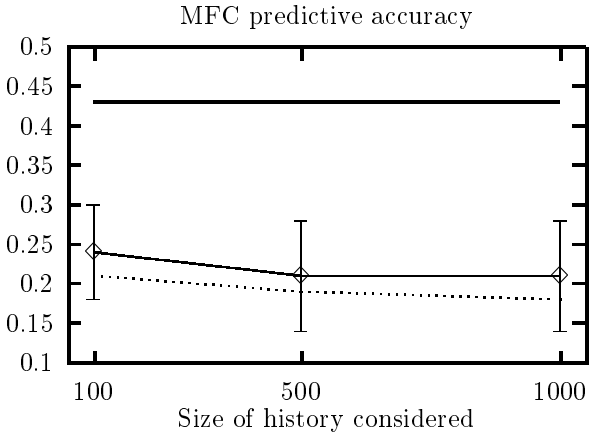

Figure 5: For each history buffer size, highest accuracy recorded, the macroaverage accuracy (bounded by the standard deviation), and the microaverage accuracy are shown (in order, from highest to lowest curve).

didn't perform quite as well as C4.5, but it does come fairly close (Figure 6). This graph shows data without extra attributes and includes startup scores. If extra attributes were included and eliminated startup scoring, some points would be up to 4% better.

4.2 Evaluation of Results

4.2.1 Online Evaluation

The data for this domain is inherently sequential. As a result, it does not lend itself to the random partitioning or sampling necessary for traditional cross-validation performance evaluation. While we could forego cross-validation and just split a user's data, e.g. the first two thirds for training, and the last third for evaluation (as [15, 22] does), we would lose two-thirds of our data points, and likely perform substantially worse on the last third.[1] Intuitively, this also makes sense; the patterns of use over the past day or even the past hour are often useful in predicting the next command. An adaptive method is particularly suited to situations like this, in which the 'target concept' changes over time. Finally, online predictive accuracy is an obvious measure – it measures exactly

[1] For comparison purposes, we note that on average C4.5 achieved 32±11% (macroaverage) accuracy on the last third of each user's data using this evaluation method.

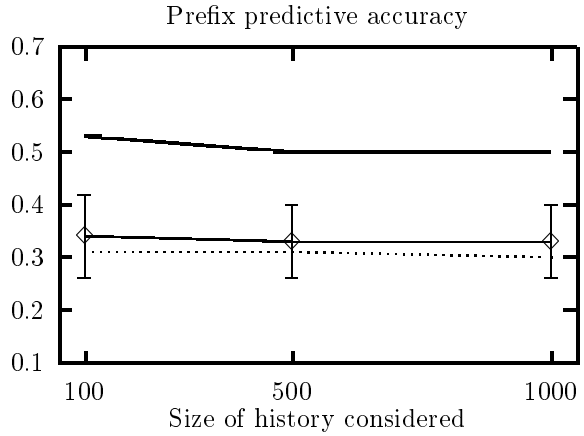

Figure 6: For each history buffer size, highest accuracy recorded, the microaverage accuracy, and the macroaverage accuracy (bounded by the standard deviation) are shown (in order, from highest to lowest curve).

the performance on an online algorithm, and retains the important characteristic of only reporting results on unseen data.

4.2.2 Predictive Accuracy

We found that the additional attributes of machine name and terminal type were generally not useful for increasing predictive accuracy. This was disappointing, as those attributes were selected as likely points upon which a learner could build, especially for the first author's data. However, we hypothesize that even for his data, the algorithms spent most of their effort on building structures to predict the most recent command.

For the average subject, the next command could have been correctly predicted 19% of the time by proposing the most recent command as what will be executed next. In a certain sense, then, the worst case usefulness (on average, per person) of a predictive shell would be to replace close to a quarter of the commands with a correct prediction. This assumes the user would use the 'previous command' keystroke of the shell to retrieve those commands rather than using the shell's prediction (and is a consequence of otherwise predicting 38% correctly).

Note that since our data are from recorded shell histories, the numbers do not show the effect of having the prediction visible and easily available to a user. It is likely that its presence would affect the command histories recorded (by

preventing some typos, for example, or even changing the idea of what the user wanted to do next), but the significance of such effects is not known.

With a best predictive performance of 38%, this methodology has strong potential. We also believe that it is possible to get even better performance. In particular, with learning algorithms with stronger primitives, it should be possible to use a variable representing, for example, 'the contents of attribute #3' as the output class. This would allow for learners to derive concepts that determine when the most recent command should be predicted, instead of learning that command x_1 often repeats itself.

4.2.3 User productivity

The results given in the previous sections have all been in terms of predictive accuracy. While this is a common and useful metric, it has some limitations. It does not, for example, measure the extent to which user productivity is improved (or harmed). For example, if a user has defined many single-character aliases, then even high accuracy in prediction need not result in a significant improvement in a user's efficiency if it is measured in, say, number of keystrokes, since both the user's command and the command-inserting may require only a single keystroke. Additionally, the overhead of managing a history and computing predictions from it may require too much time for an online system (as is likely the case for C4.5, but not perhaps for a custom decision tree implementation).

In the real world, user productivity is the primary concern. Since the data collected was from unmodified, non-predictive command shells, we can only make speculation as to the improvements in productivity afforded by the methods presented here. However, in the best case C4.5 was able to correctly predict up to 38% of the time. Recall that the lengths of the histories on average were 2184 commands, and the average command was 3.77 characters long. Assuming that a correct prediction could be inserted with a single character, and that commands recorded in the study were all typed explicitly (not using any historical shortcuts or command completion), this method would have saved just under 27% of the keystrokes typed, which is very close to the 28% expected if predictability were independent of command length. Note that we are not considering the improvements that could also result from lessening the need for correcting typos.

5 A Prototype Shell: `ilash`

We have implemented a simple prototype that incorporates some of the ideas described in this paper. Called `ilash` (Inductive Learning Apprentice SHell), it is an extension to the UNIX shell `tcsh` that allows the user to present the predicted command in the prompt string, and to insert the predicted command with a single keystroke. It uses only two attributes — the two previous com-

```
...
[cd] paul: more results.out
[more] paul: mail john@cs
[ls] paul: cd papers
[ls] paul: <ctrl-g>ls_
```

Figure 7: A sample interface for `ilash`, showing the prediction in brackets and the automatic insertion of the prediction when ¡ctrl-g¿ is pressed.

mands issued. Learning is performed by the decision-tree learner C4.5. It is described further in [8].

While this initial version of `ilash` incorporates a strong learner, it does make other tradeoffs. Decision trees are not created or updated online — a new decision tree must be built explicitly by the user, commonly just once a day. This choice allows the shell to keep its natural responsiveness, but reduces performance — the most recent commands are often quite relevant to the command to be executed next.

Our user studies did not include use of this prototype, for a number of reasons. First, doing so would have required a commitment to a single prediction method (at the very least, for each user), and we wanted to explore a range of methods. Second, we wanted to be able to separate as much as possible effects of the interface from the quality of competing prediction methods. On the other hand, the system's predictions, when provided to the user through the interface, could influence the user's next action, potentially changing the sequence of commands executed.

6 Concluding Remarks

This paper has described a number of methods of predicting a user's next command and integrating it into the shell. We found that relatively straightforward, knowledge-free methods were able to correctly predict the next command (without arguments) that the user would execute 38% of the time, potentially reducing the keystrokes needed by close to one third. The best performance was achieved using C4.5 on two features (the two previous commands), and a training window of 500 or 1000 data points.

We are also exploring other forms of command line prediction. For example, one can consider software capable of predicting a user's command as a function of the user's history as well as the first letter of the current command. Such predictions could form the basis for user-tailored command completion — after the user types the first character of a command the assistant would make a context-sensitive prediction of what the expansion for the command will be. Not too surprisingly, preliminary results show that much more accurate command

prediction can be achieved in this case.

Another, more sophisticated form of command completion would not only predict commands, but also command arguments for those commands that take arguments. For example, it would be nice to have a shell that predicted "dvips foo" if the previous command was "latex foo". We are hopeful that recent learning methods developed in the area of inductive logic programming (e.g., [16]) may prove successful in making such predictions.

One intriguing idea that this could enable is to give the software a greater degree of autonomy by having it begin execution of its prediction during the execution of the previous command (at least for those commands that are "non-destructive", or with a suitably designed "undo" function), much in the fashion that WebWatcher prefetches WWW pages that it predicts the user will next visit [1].

Our current interests include a slightly more sophisticated version of the Prefix strategy, which can be viewed as a simple form of nearest-neighbor classification (using largest matching prefix size as a distance metric); we plan to study using this method with other distance metrics which would provide a form of approximate history matching. More sophisticated still would be to do a weighted vote of multiple matches from each of a number of distance metrics, with the weights of each match learned using recently developed "weighted-majority" learning algorithms [10].

Automatic adaptation in any user interface is a welcome improvement. This paper has demonstrated that adaptation is feasible for command line interfaces; there are likewise other aspects of most user interfaces that can take advantage of the regularities present in most human interactions.

Acknowledgements

Mark Limotte implemented the modifications to `tcsh` that formed the prototype described here. We thank our colleagues at Rutgers and CMU for helpful discussions concerning this work.

References

[1] Robert Armstrong, Dayne Freitag, Thorsten Joachims, and Tom Mitchell. WebWatcher: A learning apprentice for the world wide web. In *Proceedings of the AAAI Spring Symposium on Information Gathering from Distributed, Heterogeneous Environments*, March 1995.

[2] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In Cypher [3], pages 204–217.

[3] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.

[4] J. J. Darragh, I. H. Witten, and M. L. James. The reactive keyboard: A predictive typing aid. *IEEE Computer*, 23(11):41–49, November 1990.

[5] Brian D. Davison and Haym Hirsh. Toward an adaptive command line interface. In *Proceedings of the Seventh International Conference on Human-Computer Interaction*, San Francisco, CA, August 1997. Elsevier Science Publishers.

[6] Lisa Dent, J. Boticario, John McDermott, Tom Mitchell, and David Zabowski. A personal learning apprentice. In *Proceedings of the National Conference on Artificial Intelligence*, pages 96–103, Menlo Park, CA, July 1992. AAAI Press.

[7] Leonard A. Hermens and Jeffrey C. Schlimmer. A machine-learning apprentice for the completion of repetitive forms. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence Applications*, Los Alamitos, CA, March 1993. IEEE Computer Society Press.

[8] Haym Hirsh and Brian D. Davison. An adaptive UNIX command-line assistant. In *Proceedings of the First International Conference on Autonomous Agents*. ACM Press, 1997.

[9] Neal Lesh and Oren Etzioni. A sound and fast goal recognizer. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1995.

[10] Nicholas Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108:212–261, 1994.

[11] Pattie Maes and Robyn Kozierok. Learning interface agents. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, CA, 1993. AAAI Press.

[12] T. Masui and K. Nakayama. Repeat and predict — two keys to efficient text editing. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 118–123, New York, April 1994. ACM Press.

[13] Tom Mitchell, Rich Caruana, Dayne Freitag, John McDermott, and David Zabowski. Experience with a learning personal assistant. *Communications of the ACM*, 37(7):81–91, July 1994.

[14] Tom Michael Mitchell, Sridhar Mahadevan, and Louis I. Steinberg. LEAP: A learning apprentice for VLSI design. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985.

[15] Hiroshi Motoda and Kenichi Yoshida. Machine learning techniques to make computers easier to use. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, August 1997.

[16] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[17] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[18] Jeffrey C. Schlimmer and Leonard A. Hermens. Software agents: Completing patterns and constructing user interfaces. *Journal of Artificial Intelligence Research*, 1:61–89, 1993.

[19] Jeffrey C. Schlimmer and Patricia Crane Wells. Quantitative results comparing three intelligent interfaces for information capture: A case study adding name information into an electronic personal organizer. *Journal of Artificial Intelligence Research*, 5:329–349, 1996.

[20] Ian H. Witten. A predictive calculator. In Cypher [3], pages 66–76.

[21] Kenichi Yoshida. User command prediction by graph-based induction. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 732–735, Los Alamitos, CA, November 1994. IEEE Computer Society Press.

[22] Kenichi Yoshida and Hiroshi Motoda. Automated user modeling for intelligent interface. *International Journal of Human-Computer Interaction*, 8(3):237–258, 1996.