Learning Interface Agents

Pattie Maes

MIT Media Laboratory
20 Ames Street Rm. 401a
Cambridge, MA 02139
pattie@media.mit.edu

Robyn Kozierok

MIT Media Laboratory
20 Ames Street Rm. 401c
Cambridge, MA 02139
robyn@media.mit.edu

Abstract

Interface agents are computer programs that employ Artificial Intelligence techniques in order to provide assistance to a user dealing with a particular computer application. The paper discusses an interface agent which has been modelled closely after the metaphor of a *personal assistant.* The agent learns how to assist the user by (i) observing the user's actions and imitating them, (ii) receiving user feedback when it takes wrong actions and (iii) being trained by the user on the basis of hypothetical examples. The paper discusses how this learning agent was implemented using *memory-based learning* and *reinforcement learning* techniques. It presents actual results from two prototype agents built using these techniques: one for a meeting scheduling application and one for electronic mail. It argues that the machine learning approach to building interface agents is a feasible one which has several advantages over other approaches: it provides a customized and adaptive solution which is less costly and ensures better user acceptability. The paper also argues what the advantages are of the particular learning techniques used.

Introduction

Computers are becoming the vehicle for an increasing range of everyday activities. Acquisition of news and information, mail and even social interactions become more and more computer-based. At the same time an increasing number of (untrained) users are interacting with computers. Unfortunately, these developments are not going hand in hand with a change in the way people interact with computers. The currently dominant interaction metaphor of *direct manipulation* [Schneiderman 1983] requires the user to initiate all tasks and interactions and monitor all events. If the ever growing group of non-trained users has to make effective use of the power and diversity the computer provides, current interfaces will prove to be insufficient. The work presented in this paper employs Artificial Intelligence techniques, in particular semi-intelligent semi-autonomous agents, to implement a complementary style of interaction, which has been referred to as *indirect management* [Kay 1990]. Instead of unidirectional interaction via commands and/or direct manipulation, the user is engaged in a cooperative process in which human and computer agent(s) both initiate communication, monitor events and perform tasks. The metaphor used is that of a *personal assistant* who is *collaborating with the user* in the same work environment.

The idea of employing agents in the interface to delegate certain computer-based tasks was introduced by people such as Nicholas Negroponte [Negroponte 1970] and Alan Kay [Kay 1984]. More recently, several computer manufacturers have adopted this idea to illustrate their vision of the interface of the future (cf. videos produced in 1990-1991 by Apple, Hewlett Packard, Digital and the Japanese FRIEND21 project). Even though a lot of work has gone into the modeling and construction of agents, currently available techniques are still far from being able to produce the high-level, human-like interactions depicted in these videos. Two approaches for building interface agents can be distinguished. Neither one of them provides a satisfactory solution to the problem of how the agent acquires the vast amounts of knowledge about the user and the application which it needs to successfully fulfill its task.

The first approach consists in making the end-user program the interface agent. Malone and Lai's *Oval* (formerly *Object-Lens*) system [Lai, Malone, & Yu 1988], for example, has "semi-autonomous agents" which consist of a collection of user-programmed rules for processing information related to a particular task. For example, the Oval user can create an electronic mail sorting agent by creating a number of rules that process incoming mail messages and sort them into different folders. Once created, these rules perform tasks for the user without having to be explicitly invoked by the user. The problem with this approach to building agents is that it requires too much insight, understanding and effort from the end-user. The user has to (1) recognize the opportunity for employing an agent, (2) take the initiative to create an agent, (3) endow the agent with explicit knowledge (specifying this knowledge in an abstract language) and (4) maintain the agent's rules over time (as work habits change, etc.).

The second approach, also called the "knowledge-based approach", consists in endowing an interface agent with a lot of domain-specific background knowledge about its application and about the user (called a domain model and user model respectively). This approach is adopted by the majority of people working on intelligent user interfaces [Sullivan & Tyler 1991]. At run-time, the interface agent uses its knowledge to recognize the user's plans and find opportunities for contributing to them. For example, UCEgo [Chin 1991] is an interface agent designed to help a user solve problems in using the UNIX operating system. The UCEgo agent has a large knowledge base about how to use UNIX, incorporates goals and meta-goals and does planning, for example to volunteer information or correct the user's misconceptions. One problem with this approach to building interface agents is that it requires a huge amount of work from the knowledge engineer: a large amount of application-specific and domain-specific knowledge has to be entered and little of this knowledge or the agent's control architecture can be used when building agents for other applications. A second problem is that the knowledge of the agent is fixed once and for all: it is possibly incorrect, incomplete, not useful, and can be neither adapted nor customized (e.g. to individual user differences or to the changing habits of one user). Finally, it can be questioned whether it is possible to provide all the knowledge an agent needs to always be able to "make sense" of the user's actions (people do not always behave rationally, unexpected events might happen, the organization might change, etc.).

A Machine Learning Approach

In our work we explore an alternative approach to building interface agents which heavily relies on Machine Learning. The scientific hypothesis that is tested is that under certain conditions, an interface agent can "program itself", i.e. it can acquire the knowledge it needs to assists its user. The agent is given a minimum of background knowledge and it learns appropriate "behavior" from the user. The particular conditions that have to be fulfilled are (1) the use of the application has to involve a lot of repetitive behavior, and (2) this repetitive behavior is very different for different users. If the latter condition is not met, i.e. the repetitive behavior demonstrated by different users is the same, a knowledge-based approach might prove to yield better results than a learning approach. If the former condition is not met, a learning agent will not be able to learn anything (because there are no regularities in the user's actions to learn about).

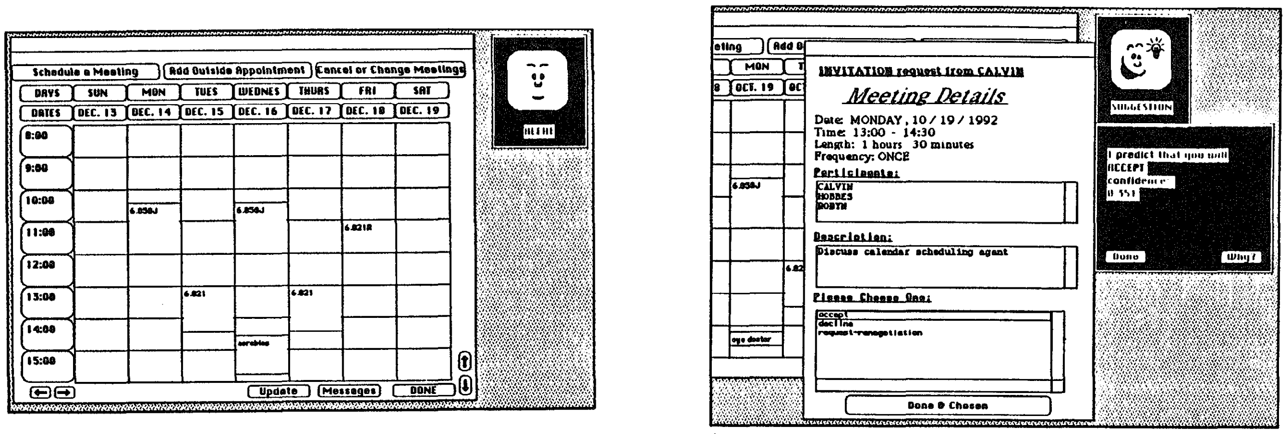
Our machine learning approach is inspired by the metaphor of a personal assistant. Initially a personal assistant is not very "customized" and may not even be very useful. Some amount of time will go by before the assistant becomes familiar with the habits, preferences and particular work methods of the person and organization at hand. However, with every experience, the assistant learns, and gradually more tasks that were initially performed by the person directly, can be taken care of by the assistant.The goal of our research is to demonstrate that a learning interface agent can in a similar way become gradually more "helpful" to its user. In addition, we attempt to prove that the learning approach has several advantages. First, it requires less work from the end-user and application developer. Second, the agent is more adaptive over time and the agent automatically becomes customized to individual user preferences and habits. The results described in a later section support all of the above hypotheses and predictions.

A particular additional advantage of the learning approach to building interface agents is that the user and agent can gradually build up a *trust relationship*. Most likely it is not a good idea to give a user an interface agent that is from the start very sophisticated, qualified and autonomous. Schneiderman has convincingly argued that such an agent would leave the user with a feeling of loss of control and understanding [Myers 1991]. On the other hand, if the agent gradually develops its abilities – as is the case in our approach – the user is also given time to gradually build up a model of how the agent makes decisions. A particular advantage of the machine learning technique we use, namely *memory-based learning* [Stanfill & Waltz 1986], is that it allows the agent to give "explanations" for its reasoning and behavior in a language that the user is familiar with, namely in terms of past examples which are similar to the current situation. ("I thought you might want to take this action because this situation is similar to this other situation we have experienced before, in which you also took this action.")

We have developed a generic architecture for building "learning interface agents". The following section discusses the design and implementation of this architecture. For more technical detail, the reader should consult [Kozierok & Maes 1993]. We also built concrete examples of interface agents using this generic architecture. These include (i) a "mail clerk", which learns how a specific user prefers to have electronic messages handled and (ii) a "calendar manager" which learns to manage the calendar of a user and schedule meetings according to his or her preferences. Figure 1 shows some screen snaps from the calendar agent implementation. The last section discusses the status of these prototypes and discusses the results obtained so far.

Learning Techniques

The interface agent uses several sources for learning (1) learning by observing the user, (2) learning from user feedback and (3) learning by being trained. Each of these methods for learning is described in more detail below. More detail on the learning algorithms used can be found in [Kozierok & Maes 1993].


Schedule a Meeting
Add Outside Appointment
Cancel or Change Meetings
DAYS SUN MON TUES WEDNES THURS FRI SAT
DATES DEC. 13 DEC. 14 DEC. 15 DEC. 16 DEC. 17 DEC. 18 DEC. 19
8:00 9:00 10:00 11:00 12:00 13:00 14:00 15:00
6.858J 6.858J 6.821R 6.821 6.821 aerobics
Update Messages DONE
ALERT
SUGGESTION
Done Why?
Done & Chosen

INVITATION request from CALVIN

Meeting Details

Date: MONDAY, 10 / 19 / 1992
Time: 13:00 - 14:30
Length: 1 hours 30 minutes
Frequency: ONCE
Participants:
CALVIN
HOBBES
ROBYN
Description:
Discuss calendar scheduling agent
Please Choose One:
accept
decline
request-renegotiation

I predict that you will
ACCEPT
confidence:
0.551

Figure 1: The alert agent observes and memorizes all of the user's interactions with the calendar application (left picture). When it thinks it knows what action the user is going to take in response to a meeting invitation, it may offer a suggestion (right picture), or even automate the action if its confidence is high enough (not shown).

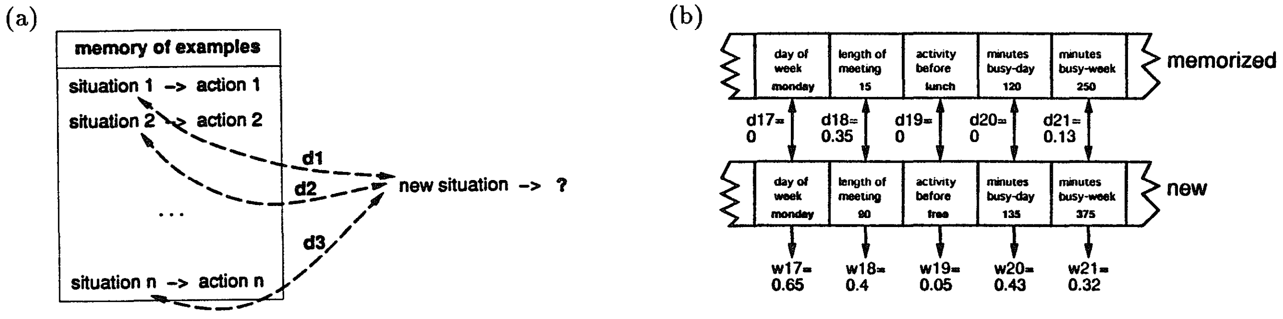


Figure 2:
(a) The agent suggests an action to perform based on the similarity of the current situation with previous (memorized) situations. d_i is the distance between the i^{th} memorized situation and the new situation.
(b) The distance between two situations is computed as a weighted sum over the features. The weight of a feature and the distance between the two values for it depend upon the correlation statistics computed by the agent. The figure shows some possible feature weights and distances from the calendar manager agent.

Learning by Observing the User

The interface agent learns by continuously "looking over the shoulder" of the user as the user is performing actions. The interface agent can monitor the activities of the user, keep track of all of his/her actions over long periods of time (weeks or months), find recurrent patterns and offer to automate these. For example, if an electronic mail agent notices that a user almost always stores messages sent to the mailing-list "intelligent-interfaces" in the folder `pattie:email:int-int.txt`, then it can offer to automate this action next time a message sent to that mailing-list has been read.

The main learning technique used in our implementation is a variant on nearest-neighbor techniques known as *memory-based learning* [Stanfill & Waltz 1986] (see illustration in Figure 2(a)). As the user performs actions, the agent memorizes all of the situation-action pairs generated. For example, if the the user saves a particular electronic mail message after having read it, the mail clerk agent adds a description of this situation and the action taken by the user to its memory of examples. Situations are described in terms of a set of features, which are currently handcoded. For example, the mail clerk keeps track of the sender and receiver of a message, the `Cc:` list, the keywords in the `Subject:` line, whether the message has been read or not, whether it has been replied to, and so on. When a new situation occurs, the agent compares it against the memorized situation-action pairs. The most similar of these memorized situations contribute to the decision of which action to take or suggest in the current situation.

The distance between a new situation and a memorized situation is computed as a weighted sum of the distances between the values for each feature as detailed in [Stanfill & Waltz 1986] (see Figure 2(b)). The distance between feature-values is based on a metric computed by observing how often in the example-base the two values in that feature correspond to the same action. The weight given to a particular feature depends upon the value for that feature in the new situation, and is computed by observing how well that value has historically correlated with the action taken. For example, if the `Sender:` field of a message has shown to correlate to the action of saving the message in a particular folder, then this feature (i.e. whether or not it is the same in the old and new situation) is given a large weight and thus has a high impact on the distance between the new and the memorized situation. At regular times, the agent analyzes its memory of examples and computes the statistical correlations of features and values to actions, which is used to determine these feature-distances and -weights.

Once all the distances have been computed, the agent predicts an action by computing a score for each action which occurs in the closest N (e.g. 5) memorized situations and selecting the one with the highest score. The score is computed as

$$\sum_{s \in S} \frac{1}{d_s}$$

where S is the set of memorized situations predicting that action, and d_s is the distance between the current situation and the memorized situation s.

Along with each prediction it makes, the agent computes a confidence level for its prediction, as follows:

$$\left(1 - \frac{\frac{d_{predicted}}{n_{predicted}}}{\frac{d_{other}}{n_{other}}}\right) \times \frac{n_{total}}{N}$$

where:

- N is, as before, the number of situations considered in making a prediction,
- $d_{predicted}$ is the distance to the closest situation with the same action as the predicted one,
- d_{other} is the distance to the closest situation with a different action from the predicted one,
- $n_{predicted}$ is the number of the closest N situations with distances less than a given maximum with the same action as the predicted one,
- n_{other} is the minimum of 1 or the number of the closest N situations with distances within the same maximum with different actions than the predicted one, and
- $n_{total} = n_{predicted} + n_{other}$, i.e. the total number of the closest N situations with distances below the maximum.

If the result is < 0, the confidence is truncated to be 0. This occurs when $d_{predicted}/n_{predicted} < d_{other}/n_{other}$ which is usually the result of several different actions occurring in the top N situations. If every situation in the memory has the same action attached to it, d_{other} has no value. In this case the first term of the confidence formula is assigned a value of 1 (but it is still multiplied by the second term, which in this case is very likely to lower the confidence value as this will usually only happen when the agent has had very little experience). This computation takes into account the relative distances of the best situations predicting the selected action and another action, the proportion of the top N situations which predict the selected action, and the fraction of the top N situations which were closer to the current situation than the given maximum.

If the confidence level is above a threshold $T1$ (called the "tell-me" threshold), then the agent offers its suggestion to the user. The user can either accept this suggestion or decide to take a different action. If the confidence level is above a threshold $T2 > T1$ (called the "do-it" threshold), then it automates the action without asking for prior approval. The agent keeps track of all the automated actions and can provide a report to the user about its autonomous activities whenever the user desires this. The two thresholds are set by the user and are action-specific, thus the user may, for example, set higher "do-it" thresholds for actions which are harder to reverse. (A similar strategy is suggested for computer-chosen thresholds in [Lerner 1992].) The agent adds the new situation-action pair to its memory of examples, after the user has approved of the action.

Occasionally the agent "forgets" old examples so as to keep the size of the example memory manageable and so as to adapt to the changing habits of the user. At the moment, the agent deletes the oldest example whenever the number of examples reaches some maximum number. We intend to investigate more sophisticated "forgetting" methods later.

One of the advantages of this learning algorithm is that the agent needs very little background knowledge. Another advantage is that no information gets lost: the agent never attempts to abstract the regularities it detects into rules (which avoids problems related to the ordering of examples in incremental learning). Yet another advantage of keeping individual examples around is that they provide good explanations: the agent can explain to the user why it decided to suggest or automate a particular action based on the similarity with other concrete situations in which the user took that action – it can remind the user of these prior situations and point out the ways in which it finds them to be similar to the current situation. Examples provide a familiar language for the agent and user to communicate in. There is no need for a more abstract language and the extra cognitive burden that would accompany it.

One could argue that this algorithm has disadvantages in terms of computation time and storage requirements. We believe that the latter is not an issue because computer memory becomes cheaper and more available every day. The former is also less of a problem in practice. Computing the statistical correlations in the examples is an expensive operation ($O(n^2)$), but it can be performed off-line, for example at night or during lunch breaks. This does not mean that the agent does not learn from the examples it has observed earlier the same day: new examples are added to memory right away and can be used in subsequent predictions. What does not get updated on an example basis are the weights used in computing distances between examples. The prediction of an action is a less computation intensive operation ($O(n)$). This computation time can be controlled by restricting the number of examples memorized or by structuring and indexing the memory in more sophisticated ways. Furthermore, in a lot of the applications studied real-time response is not needed (for example, the agent does not have to decide instantly whether the user will accept a meeting invitation). In the experiments performed so far, all of the reaction times have been more than satisfactory. More details and results from this algorithm are described in a later section and in [Kozierok & Maes 1993].

Learning from User Feedback

A second source for learning is direct and indirect user feedback. Indirect feedback happens when the user neglects the suggestion of the agent and takes a different action instead. This can be as subtle as the user not reading the incoming electronic mail messages in the order which the agent had listed them in. The user can give explicit negative feedback when inspecting the report of actions automated by the agent ("don't do this action again"). One of the ways in which the agent learns from negative feedback is by adding the right action for this situation as a new example in its database.

Our agent architecture also supports another way in which the agent can learn from user feedback. The architecture includes a database of priority ratings which are relevant to all situations. For example, the calendar manager keeps a database of ratings expressing how important the user thinks other users of the system are, and how relevant the user feels certain keywords which appear in meeting descriptions are. These ratings are used to help compute the features which describe a situation. For example, there is an "initiator importance" feature in the calendar manager, which is computed by looking up who initiated the meeting, and then finding the importance rating for that person. When the agent makes an incorrect suggestion, it solicits feedback from the user as to whether it can attribute any of the blame to inaccuracy in these priority ratings, and if so in which ones. It can then adjust these ratings to reflect this new information, increasing or decreasing them as the difference in the "positiveness" of the suggested versus actual action dictates. The details of how this is done are described in [Kozierok & Maes 1993].

Learning by Being Trained

The agent can learn from examples given by the user intentionally. The user can teach/train the agent by giving it hypothetical examples of events and situations and showing the agent what to do in those cases. The interface agent records the actions, tracks relationships among objects and changes its example base to incorporate the example that it is shown. For example, the user can teach the mail clerk agent to save all messages sent by a particular person in a particular folder by creating a hypothetical example of an email message (which has all aspects unspecified except for the sender field) and dragging this message to the folder in question. Notice that in certain cases it is necessary to give more than one hypothetical example (e.g. if the user wants to train the system to save messages from different senders in the same folder).

This functionality is implemented by adding the example in memory, including "wildcards" for the features which were not specified in the hypothetical situation. The new situation-action pair will match all situations in which an email message has been received from a user with the same name. One of the unresolved questions is how such hypothetical examples should be treated differently both when selecting an action and when compiling statistics. [Kozierok 1993] explores this issue, and describes how both *default* and *hard-and-fast rules* can be implemented within the memory-based learning framework. This paper also discusses how rules may be used to compress the database, when either all or most of the situations the rule would represent have occurred.

Results

The generic architecture for a learning interface agent is implemented in CLOS (Common Lisp Object System) on a Macintosh. We have evaluated the design and implementation of this architecture by constructing agents for several application programs. We currently have a prototype of a mail clerk as well as a calendar manager agent. The application software (the meeting scheduling system and electronic mail system) was implemented from scratch, so as to make it easier to provide "hooks" for incorporating agents. Both applications have a graphical direct manipulation interface (also implemented in Macintosh CLOS). The agent itself is hardly visible in the interface: a caricature face in the corner of the screen provides feedback to the user as to what the current state of the interface agent is. Figure 3 lists some of these caricatures. They help the user to quickly (in the blink of an eye) find out "what the agent is up to".

We have performed testing of both agents with simulated users. We are currently testing the calendar agent on real users in our own office environment, and

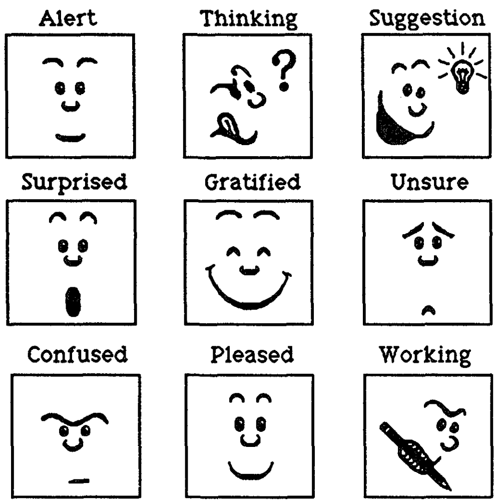


Figure 3: Simple caricatures convey the state of the agent to the user. The agent can be (a) alert (tracking the user's actions, (b) thinking (computing a suggestion), (c) offering a suggestion (when above tell-me threshold) (a suggestion box appears under the caricature), (d) surprised the suggestion is not accepted, (e) gratified the suggestion is accepted, (f) unsure about what to do in the current situation (below tell-me threshold) (suggestion box only shown upon demand), (g) confused about what the user does, (h) pleased that the suggestion it was not sure about turned out to be the right one and (i) working or performing an automated task (above do-it threshold).

will begin real-user testing of the email agent shortly as well. These users will be observed and interviewed over a period of time. The results obtained so far with both prototypes are encouraging. The email agent learns how to sort messages into the different mailboxes created by the user; when to mark messages as "to be followed up upon" or "to be replied to", etc. Current results on the meeting scheduling agent are described in detail in [Kozierok & Maes 1993]. A collection of seven such agents has been tested for several months worth of meeting problems (invitations to meetings, scheduling problems, rescheduling problems, etc). All seven agents learned over time to make mostly correct predictions, with high confidence in most of the correct predictions, and low confidence in almost all of the incorrect ones. Figure 4 shows the results for a representative agent. Again, the results obtained demonstrate that the learning interface agent approach is a very promising one. From this graph one can see that the correct predictions tend to increase in confidence level, while the incorrect ones tend to decrease. (We expect to see similar results in our real-user tests, but the inconsistencies and idiosyncracasies of real users will probably cause the agents to take longer to converge on such positive results. However, the memory-based learning algorithm employed is designed to allow for inconsistencies, so we have confidence that the agents will indeed be able to perform competently within a reasonable timeframe.) Providing the user with access to this type of performance information allows him to easily set the thresholds at reasonable levels, as shown in the figure.

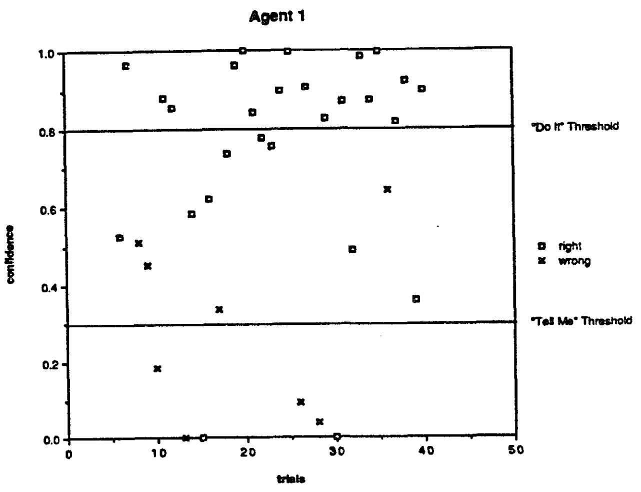


Figure 4: Results of a representative agent from the meeting scheduling application. The graph shows the right and wrong predictions made by the agent as plotted over time (X-axis). The Y-axis represents the confidence level the agent had in each of predictions. The picture also shows possible settings for the "tell-me" (T1) and "do it" (T2) thresholds.

Related Work

The work presented in this paper is related to a similar project under way at CMU. Dent et. al. [Dent et al. 1992] describe a personal learning apprentice which assists a user in managing a meeting calendar. Their experiments have concentrated on the prediction of meeting parameters such as location, duration and day-of-week. Their apprentice uses two competing learning methods: a decision tree learning method and a back-propagation neural network. One difference between their project and ours is that memory-based learning potentially makes better predictions because there is no "loss" of information: when suggesting an action, the detailed information about individual examples is used, rather than general rules that have been abstracted beforehand. On the other hand, the memory-based technique requires more computation time to make a particular suggestion. An advantage of our approach is that our scheduling agent has an estimate of the quality or accuracy of its suggestion. This estimate can be used to decide whether the prediction is good enough to be offered as a suggestion to the user or even to automate the task at hand.

The learning agents presented in this paper are also related to the work on so-called *demonstrational interfaces.* The work which is probably closest is Cypher's "eager personal assistant" for Hypercard [Cypher 1991]. This agent observes the user's actions, notices repetitive behavior and offers to automate and complete the repetitive sequence of actions. Myers [Myers 1988] and Lieberman [Lieberman 1993] built demonstrational systems for graphical applications. One difference between the research of all of the above authors and the work described in this paper is that the learning described here happens on a longer time scale (e.g. weekly or monthly habits). On the other hand a system like Eager forgets a procedure after it has executed it. A difference with the systems of Lieberman and Myers is that in our architecture, the user does not have to tell the agent when it has to pay attention and learn something.

Conclusion

We have modelled an interface agent after the metaphor of a personal assistant. The agent gradually learns how to better assist the user by (1) observing and imitating the user, (2) receiving feedback from the user and (3) being told what to do by the user. The agent becomes more helpful, as it accumulates knowledge about how the user deals with certain situations. We argued that such a gradual approach is beneficial as it allows the user to incrementally build up a model of the agent's behavior. We have presented a generic architecture for constructing such learning interface agents. This architecture relies on memory-based learning and reinforcement learning techniques. It has been used to build interface agents for two real applications. Encouraging results from tests of these prototypes have been presented.

Acknowledgments

Cecile Pham, Nick Cassimatis, Robert Ramstadt, Tod Drake and Simrod Furman have implemented parts of the meeting scheduling agent and the electronic mail agent. The authors have benefited from discussions with Henry Lieberman, Abbe Don and Mitch Resnick. This work is sponsored by the National Science Foundation (NSF) under grant number IRI-92056688. It has also been partially funded by Apple Computer. The second author is an NSF fellow.

References

Chin D. 1991. Intelligent Interfaces as Agents. In: J. Sullivan and S. Tyler eds. *Intelligent User Interfaces,* 177-206. New York, New York: ACM Press.

Crowston, K., and Malone, T. 1988. Intelligent Software Agents. *BYTE* 13(13):267-271.

Cypher, A. 1991. EAGER: Programming Repetitive Tasks by Example. In: CHI'91 Conference Proceedings, 33-39. New York, New York: ACM Press.

Don, A. (moderator and editor). 1992. Panel: Anthropomorphism: From Eliza to Terminator 2. In: CHI'92 Conference Proceedings, 67-72. New York, New York: ACM Press.

Kay, A. 1984. Computer Software. *Scientific American.* 251(3):53-59.

Kay, A. 1990. User Interface: A Personal View. In: B. Laurel ed. *The Art of Human-Computer Interface Design,* 191-208. Reading, Mass.: Addison-Wesley.

Kozierok, R., and Maes, P. 1993. A Learning Interface Agent for Scheduling Meetings. In: Proceedings of the 1993 International Workshop on Intelligent User Interfaces, 81-88. New York, New York: ACM Press.

Kozierok, R. 1993. Incorporating Rules into a Memory-Based Example Base, Media Lab Memo. Dept. of Media Arts and Sciences, MIT. Forthcoming.

Lai, K., Malone, T., and Yu, K. 1988. Object Lens: A "Spreadsheet" for Cooperative Work. *ACM Transactions on Office Information Systems* 5(4):297-326.

Laurel, B. 1990. Interface Agents: Metaphors with Character. In: B. Laurel ed. *The Art of Human-Computer Interface Design,* 355-366. Reading, Mass.: Addison-Wesley.

Lerner, B.S. 1992. Automated customization of structure editors. *International Journal of Man-Machine Studies* 37(4):529-563.

Lieberman, H. 1993. Mondrian: a Teachable Graphical Editor. In: A. Cypher ed. *Watch what I do: Programming by Demonstration.* Cambridge, Mass.: MIT Press. Forthcoming.

Dent, L., Boticario, J., McDermott, J., Mitchell, T., and Zabowski D. 1992. A Personal Learning Apprentice. In: Proceedings, Tenth National Conference on Artificial Intelligence, 96-103. Menlo Park, Calif.: AAAI Press.

Myers, B. 1988. *Creating User Interfaces by Demonstration.* San Diego, Calif.: Academic Press.

Myers, B. (moderator and editor). 1991. Panel: Demonstrational Interfaces: Coming Soon? In: CHI'91 Conference Proceedings, 393-396. New York, New York: ACM Press.

Negroponte, N. 1970. *The Architecture Machine; Towards a more Human Environment.* Cambridge, Mass.: MIT press.

Schneiderman, B. 1983. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer* 16(8):57-69.

Stanfill, C., and Waltz, D. 1986. Toward Memory-Based Reasoning. *Communications of the ACM* 29(12):1213-1228.

Sullivan, J.W., and Tyler, S.W. eds. 1991. *Intelligent User Interfaces.* New York, New York: ACM Press.