Inferring What a User is Not Interested In

Robert C. Holte
Computer Science Department
University of Ottawa
Ottawa, Canada K1N 6N5
holte@csi.uottawa.ca

John Ng Yuen Yan
BNR Ltd.
P.O. Box 3511, Station C
Ottawa, Canada K1Y 4H7
johnng@bnr.ca

Abstract

This paper describes a system to improve the speed and success rate with which users browse software libraries. The system is a learning apprentice: it monitors the user's normal browsing actions and from these infers the goal of the user's search. It then searches the library being browsed, uses the inferred goal to evaluate items and presents to the user those that are most relevant. The main contribution of this paper is the development of rules for negative inference (i.e. inferring features that the user is not interested in). These produce a dramatic improvement in the system's performance. The new system is more than twice as effective at identifying the user's search goal than the original, and it ranks the target much more accurately at all stages of search.

1 Introduction

"Browsing" is the searching of a computer library for an individual library item. The human doing the search (the "user") starts with a set of requirements and the aim of finding an item (the "target") that best meets these requirements. The user proceeds by assessing items currently being displayed and choosing an item (or multiple items) and an operation to apply. Typical operations are content-based retrieval, requesting additional information about a particular item, and navigating along links in the library that connect related items. The operation is executed by the browsing system and the display updated. This process is repeated until the user decides to abandon the search or deems an item to adequately satisfy the requirements. In assessing library items and choosing operations, the user is guided by his (or her) expectation of what the target item in the library will be like. The user's mental model of the target is called the "search goal". Because the user's knowledge of the content, organization, and descriptive language of the library is generally imperfect, browsing is fundamentally an uncertain and iterative process. Consequently it is slow, frustrating, and somewhat error prone.

To increase the speed and success rate of browsing we have developed a "learning apprentice" [Mitchell et al.,1985] that attempts to infer a user's search goal from his actions. The user browses as usual, unaware that the learning apprentice is monitoring his actions. From the sequence of user actions the learning apprentice infers an "analogue" representing what it believes to be the user's search goal. The analogue is converted into a "template" that can readily be used to measure the relevance of an individual library item to the user. The template is matched against each item in the library, the items are sorted according to their relevance, and displayed to the user in a special window (the "suggestion box").

Our first learning apprentice for browsing was very successful, inferring the target before it is found by the user about 40% of the time [Drummond et al.,1995]. The inference rules in this system are all "positive" in the sense that all draw conclusions of the form "the user is interested in items with feature X". In this paper we add to the system "negative" inference rules, which draw conclusions of the form "feature X is definitely not of interest to the user". We show experimentally that this dramatically improves the system's ability to infer the user's search goal.

2 A Browser for Object-Oriented Software Libraries

Our testbed browsing application is software reuse. The library is a collection of object-oriented software. An item in the library is a "class", in the object-oriented sense. Each class contains "instance variables" and "methods" that are locally defined, and inherits the variables and methods of its superclass in the inheritance hierarchy. The names of classes, variables, and methods very often consist of several words concatenated together. A class or variable name is typically a noun phrase with the noun at the end (e.g. "MenuItem"), whereas a method name tends to be a verb phrase with the verb at the beginning (e.g. "AdjustSizeForNewItem"). A class's functionality is determined by its methods (inherited and locally defined). The aim of browsing is to find the class whose functionality is closest to the required functionality.

Our browsing system works as follows. Initially, it presents the user with a list of all the classes in the library. As browsing proceeds additional class lists and method lists are created by the user's actions. To apply an operator to a class, the user selects the class, **C**, from any available class list and then specifies the operator to be applied. The main operators that can be applied to a class are:

Subclasses - creates a list of **C**'s subclasses.
Superclasses - creates a class list of **C**'s ancestors.
Defined Methods - creates a list of the methods **C** defines locally.
Similar Name - creates a list of classes ordered by how similar each class's name is to **C**'s. Similarity is based on the number of words the two names have in common.
Similar Functionality - creates a list of classes ordered by how similar each class's functionality is to **C**'s. Similarity is based on the similarity of the names of the methods defined in the two classes.

There are also operators that can be applied to methods. However, to apply an operator to a method is a two step process. First one must select the method in the method list produced by "Defined Methods". This "opens" the method in a window that is used for inspecting a method's details. To apply an operator, the user must select the method in this window and then specify the operator. The main operators that can be applied to a method are:

More details - each time this operator is applied to a method more details about its implementation are presented.
Mark - marks the method for use with the following operators.
Implemented In - creates a list of the classes ordered by the degree to which each implements all the currently marked methods. A class's score is based on the similarity of the marked methods' names to the names of the methods the class implements.
Used By - creates a list of classes ordered by the degree to which each uses all the currently marked methods. A class's score is based on the similarity of the marked methods' names to the names of the methods that are called by the class's own methods.

Table 1 shows the sequence of actions taken by a user searching for the class "Confirmer" in the experiment reported in [Drummond et al.,1995] (the user was not told the target's name). He first chooses a class (Prompter) from the initial class list and applies the "Defined Methods" operator to it. Three of these methods are opened for inspection. Additional details are requested for "WaitFor" and "WaitForUser", presumably in order to determine the difference between them. The user then marks "WaitForUser" and "PaintBackground" and requests a list of all classes that use these two methods (or methods with similar names). The target is on the resulting class list and is recognized by the user, ending the search.

We have observed that users do not apply operators in any fixed pattern. Different users favour different operators and an individual can use different search strategies for different search goals. However, the method-based operators ("Implemented In"

Table 1. Typical Sequence of Browsing Actions.

	Class or Method	Operator
1	Prompter	Defined Methods
2	PaintBackground	(open method)
3	WaitFor	(open method)
4	WaitForUser	(open method)
5	WaitFor	More details
6	WaitForUser	More details
7	WaitForUser	Mark
8	PaintBackground	Mark
9	(marked methods)	Used By
10	Confirmer	

and "Used By") are usually favoured over, and more effective than, the class-based operators. This fact is important because, as will be seen later, our negative inference strategy works only for the method-based operators.

3 The Original System

Our initial learning apprentice for browsing contained only positive inference rules, i.e. rules inferring that a particular feature is of interest to the user. The analogue representing the user's search goal is simply a list of the features in which the user has shown interest with an associated confidence factor. For example, this apprentice would make the following inferences during the search in Table 1. From the first action, it would infer that the user is interested in a class whose name is similar to "Prompter". From actions 2-4, it would infer that method names similar to those of the opened methods are of interest. Actions 5-8, and most significantly action 9, represent further interest exhibited by the user in specific methods and cause the confidence factors of those methods to be increased. After action 9 the analogue would contain the assertions

INTERESTED_IN (class name: Prompter) (confidence factor)
INTERESTED_IN (method name: PaintBackground) (confidence factor)
INTERESTED_IN (method name: WaitFor) (confidence factor)
INTERESTED_IN (method name: WaitForUser) (confidence factor)

For the purposes of matching the analogue is converted into a template that can be readily matched against each class in the library. The class's name is matched against the class names in the template, and its methods' names are matched against the method names in the template. Matching is a matter of degree. Identical names match with a score of 1.0, names that have no subterms (words) in common have a score of 0.0, and names having some subterms in common score an intermediate value. For example, a method named "PaintForeground" partially matches "PaintBackground". The matching process also takes into account methods that are inherited or used by a class. The overall match score for the class is the normalized weighted sum of the match scores for each name in the template. The weights used in this sum are derived from the confidence factors in the analogue. The use of subterms, not whole names, during matching produces some subtle effects. For example, because "Wait" is a subterm of two method names in the analogue its effective contribution to the overall score is greater than if it had only occurred in one.

4 Negative Inference

Positive inference in combination with partial matching proved very successful in our original experiments. The example in Table 1, however, illustrates a limitation of this system. The user's actions plainly indicate that he has deliberately decided method "WaitFor" is not of interest. The interest exhibited in opening and inspecting this method was tentative. Once "WaitFor" and "WaitForUser" have been compared and a decision between them made, only one ("WaitForUser") remains of interest. Merely retracting INTERESTED_IN (method name: WaitFor) would not entirely capture this information because "Wait" and "For" occur in "WaitForUser" and would therefore produce quite strong partial matches.

To make the correct inference from this sequence of actions, two changes are necessary to the learning apprentice. First, subterms must have their own entries in the analogue. This will permit the system to assign a higher confidence factor to "User", the subterm that actually discriminates between between the two method names in this example, than to the subterms "Wait" and "For". Secondly, rules are needed to do negative inference so that features that once seemed interesting can be removed from the analogue when they prove to be uninteresting.

Browsers sometimes have actions that directly indicate that the user is not interested in an item. For example, in the browsers for electronic news of [Lang,1995] and [Sheth and Maes,1993] the user explicitly indicates if a news article is or is not of interest. This is a form of relevance feedback [Harman,1992; Haines and Croft,1993]. In such cases, negative inference is as straightforward as positive inference.

In browsers, such as ours, that only have actions that a user applies to further explore items of interest, negative inference must be based on the missing actions – actions that could have been taken but were not. For example, the user could have applied the "Mark" action to "WaitFor", but did not. The difficulty, of course, is that there are a great many actions available to the user at any given moment of which only a few will actually be executed. It is certainly not correct to make negative inferences from all the missing actions.

What is needed to reliably make negative inferences is some indication that the user consciously considered an action and rejected it. In the example, the fact that the user opened "WaitFor" is a strong indication that he consciously considered its use in the subsequent "Used By" operation. This is because with our browser the main reason to open a method is so that it can be marked and used in conjunction with "Used By" or "Implemented By". The fact that "WaitFor" was not used in the culminating operation of this sequence is best explained by its being judged of no interest. To generalize this example, negative inference can be reliably performed when there is a definite culminating operation (in this case either "Used By" or "Implemented By") and a *two* step process for selecting the item(s) to which the operator is to be applied.

Specifically, negative inference is triggered by the "Used By" or "Implemented By" operations and is applied to the names of methods that are open but not marked. For each word, **W**, in these names the assertion

NOT_INTERESTED_IN(method subterm: **W**)

is added permanently to the analogue. This assertion is categorical (its certainty factor is 1.0); it overrides any previous or subsequent positive inference about the subterm **W**. In the above example, negative inference would produce

NOT_INTERESTED_IN (method subterm: Wait)
NOT_INTERESTED_IN (method subterm: For)

Only the positive assertions in the analogue are converted into the template. Class and method names are matched as before. A method subterm in the template is considered to match a class if the subterm appears in any of the class's own method names.

5 Experimental Method

The library and code for the browser in which our first learning apprentice was embedded are proprietary and no longer available. It was therefore necessary to re-implement the system and apply it to some other library. The browser is faithfully re-implemented, but several details of the learning apprentice and Rover (see below) differ in the re-implementation. The re-implemented version of the original learning apprentice is called Version1 below. Version2 is Version1 with analogue-subterms and negative inference added.

The advantage of negative inference is measured by comparing the performance of Version1 to Version2. In particular, we compare how the two learning apprentices rank the target as search proceeds. The higher the target is ranked, or, alternatively, the sooner it becomes highly ranked, the better the learning apprentice (the perfect apprentice would immediately assign the target a rank of 1). The rank assigned to the target by the learning apprentices is also compared to rank of the target in the user's most recently created class list. This comparison indicates if the learning apprentice is "ahead of" or "behind" the user.

[Drummond et al.,1995] reports the results of a small scale study with human users. But its main experiment employed an automated user, i.e., a computer program that played the role of the user. This enables large-scale experiments to be carried out quickly and also guarantees that experiments are repeatable and perfectly controlled. The experiment below is the same in its design. A similar experimental method is used in [Haines and Croft,1993] to compare relevance feedback systems.

We do not claim to have created a *simulated* user that perfectly mimics the rich behavioural patterns of a human user. Our aim in creating *automated* users for browsing has been to try to capture some of the more prominent general behavioural trends that a human might be expected to follow.

Our automated user, Rover, consists of two parts: a "fuzzy oracle" that represents the search goal, and a heuristic search strategy that consults the oracle and selects browsing actions. The fuzzy oracle contains a target class selected by the experimenter from amongst the classes in the library. The oracle gives YES/NO answers to questions about whether a given library item matches the target class in certain ways. The oracle is "fuzzy" because its answers are not always correct; it returns the incorrect answer with a certain probability. This noisiness represents the user's uncertainty in evaluating the degree of match between a library item and his requirements.

The heuristic search strategy combines depth-first search and hill-climbing. The first ten classes[1] in the most recent class list are scanned in order until a class is found whose name, according to the oracle, is similar to the target's. If no such class is found the strategy backtracks to the previous class list. If a class with a similar name is found, "Defined Methods" is applied to it. Methods in the resulting list are opened at random. Those that the oracle says have names similar to the target's methods are marked (by the "Mark" operation). When several methods have been marked the oracle is asked if this set of marked methods is collectively more similar to the target's methods than the best previous collection of marked methods. If the answer is YES, "Implemented By" is applied creating a new class list. Otherwise the strategy abandons this class and continues scanning the most recent class list.

For present purposes, a very important aspect of this strategy is that its search is almost exclusively based on the "Implemented By" operation, an operation that triggers negative inference. Therefore the effect of negative inference on the learning apprentice's ability to infer the target will be most evident with a search strategy of this kind. It must be said, however, that this strategy was not devised especially for this experiment. It is our standard strategy, the same as was used in [Drummond et al.,1995]. Furthermore, human users often heavily rely on the method-based operators. One of the human users in the experiment in [Drummond et al.,1995] exclusively used "Used By" (which is the same as "Implemented By" for the purpose of negative inference); another almost exclusively used "Implemented By", only occasionally adding a Subclass or Superclass operation.

The library used in the experiment is the Smalltalk code library, which contains 389 classes. Four of these classes have no defined methods; these are excluded from the study. Each of the remaining classes was used as the search target in a separate run of the experiment. Rover continues searching until it finds the target or 70 steps have been taken. A "step" in the search is the creation of a new class list by the operation "Implemented By". Rover's complete set of actions is recorded as is the rank of the target in Rover's most recent class list at each step. The resulting trace of Rover's search is then fed into each learning apprentice (Version1 and Version2) separately to determine the target's rank in the suggestion box at each step and the step at which the learning apprentice successfully identifies the target.

The learning apprentice is considered to have identified the target when its rank in the suggestion box is 10 or better for five consecutive steps. This definition precludes the learning apprentice from succeeding if Rover finds the target in fewer than 5 steps; this happens on 69 runs.

---

[1] the initial class list is scanned from first to last

6 Experimental Results and Discussion

The simplest summary of the experimental results is given in Table 2. Each run that is between 5 and 70 steps in length is categorized as a win (for the learning apprentice), a loss, or a draw, depending on whether the learning apprentice identified the target before Rover found it, did not identify the target at all, or identified it at the same time Rover found it. The row for Version1 indicates that 299 runs were between 5 and 70 steps in length and that of these, 70 (23.4%) were wins for Version1. This figure is considerably lower than the 40% win rate that the original learning apprentice obtained. The difference may in part be due to small differences in the re-implementation of the learning apprentice or Rover, but are probably mainly due to differences in the library. The Smalltalk library is, it seems, more difficult for the learning apprentice than the Objective-C library used in our original experiments.

Version2 performs very well, identifying the target before it is found by Rover over half the time. The addition of negative inference has more than doubled the number of wins. There are also 7 runs that required more than 70 steps by Rover and Version1 but which required fewer than 70 steps with Version2.

Table 3 is a direct comparison of the learning apprentices to each other. The first row summarizes the runs in which Version1 identified the target before Version2 did. This happened only 11 times, and on these targets Version1 was only 1.6 steps ahead of Version2. The second row summarizes the runs in which Version2 identified the target before Version1. This happened 120 times (almost 1/3 of the runs), and the reduction in search time on these runs was very considerable, 11.0 steps. This shows that negative inference is rarely detrimental, and never a significant impediment, and

Table 2. Wins-Losses-Draws

	Wins	Losses	Draws	Total
Version1	70 (23.4%)	207 (69.2%)	22 (7.4%)	299
Version2	161 (52.6%)	110 (36.0%)	35 (11.4%)	306

Table 3. Average Search Length when Version1 and Version2 differ.

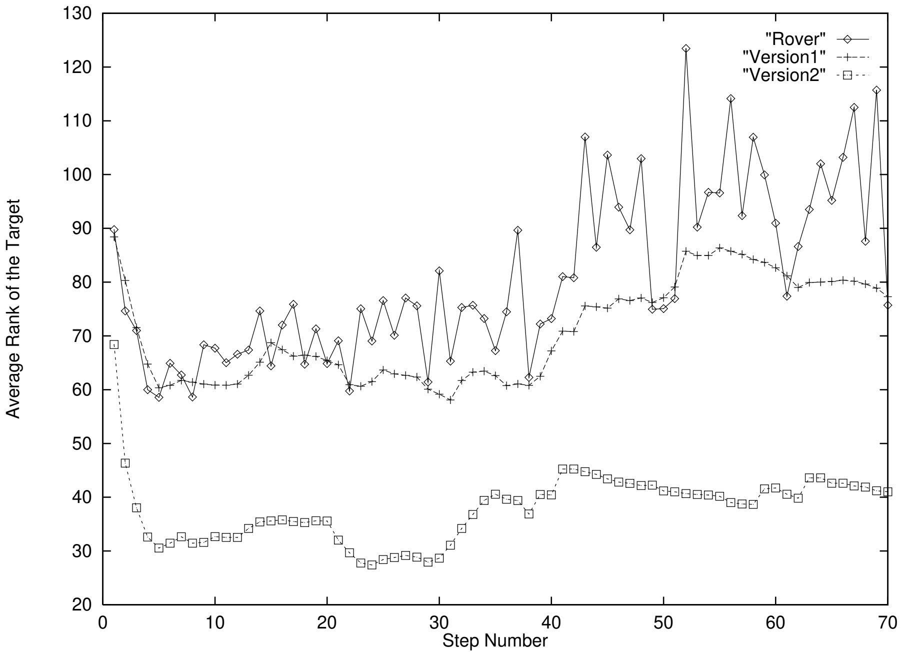
	Average Search Length			Number of
	Version1	Version2	Difference	Targets
Version1 finishes first	9.9	11.5	1.6	11
Version2 finishes first	23.6	12.6	11.0	120

that it frequently almost doubles the speed with which the learning apprentice identifies the target.

The preceding analyses address the question "how quickly is the target identified ?" but give no indication of how the target's rank evolved as the search progressed. If the user is to benefit from a learning apprentice in practice, it must be true that throughout the search the apprentice's rank for the target class is consistently significantly better than the user's own rank for the target. In other words, for the suggestion box to be useful the target's position in the suggestion box must be better (nearer the top) than its position in the user's own class lists.

Figure 1 plots the average rank of the target on each step. The average for step N is computed only for the targets that are still active at step N. For Rover "rank" refers to the position of the target in the user's most recent class list. For Version1 and Version2 it refers to the target's position in the suggestion box. The best rank is 1; the higher the average rank, the worse the system. From this perspective, Version1 is much better than Rover. Except for the first 5-10 steps of a search, the target is 10-20 positions higher in Version1's suggestion box than it is in Rover's most recent class list. This result is qualitatively identical to the corresponding result with the original learning apprentice (Figure 11 in [Drummond et al.,1995]).

Figure 1. Average Rank of the Target at each step



Version2 dramatically outperforms Version1. After just one step it ranks the target 20 positions higher than Rover and Version 1; after 3 steps this difference has increased to 30. The figure also shows that the target is almost always (on average) among the first 40 in the suggestion box. In a sense the user's search space has been reduced by 90%, from a library of almost 400 classes to a list of 40.

7 Related Work

The work presented in this paper is most closely related to research on "active assistants". An active assistant is a background process that monitors the user's actions and, in certain circumstances, interrupts the user and offers unsolicited advice. We categorize active assistants based on the nature of their internal inference mechanism.

"Daemons" are preprogrammed to recognize particular patterns of user behaviour, and, when a particular pattern of behaviour is detected, to issue the corresponding preprogrammed response. For example, the critics [Silverman and Mazher,1992] in Fischer's design environment [Fischer et al.,1990] are daemons: each critic recognizes certain types of flaws in the user's current design (e.g. violations of design constraints) and draws these to the user's attention (possibly also suggesting corrections). Finin's active help system [Finin,1983] is similar, consisting of a collection of rules each of which defines a particular situation (specified by a (generalized) sequence of actions) and the advice/help to give should that situation arise. Likewise, plan recognition systems are daemons because they simply match the user's action sequence against a given library of plans [McCalla et al., 1992; Cohen and Spencer, 1993; Hook et al., 1993] or "parse" the user's actions with a given set of plan schemas [Goodman and Litman,1990]. Most ‘‘programming by demonstration" systems that do inference employ daemons. For example, [Bos, 1992; Cypher, 1991; Witten and Mo, 1993] all use a preprogrammed notion of "similar action" to detect repeated sequences of actions.

"Learning agents" do not have a preprogrammed set of situation−action rules. Instead they learn from the user's actions when to interrupt the user and/or what advice to give. For example, [Sheth and Maes, 1993] describes a "personalized information filtering" agent, which assists a user by suggesting USENET news articles that might be of interest. The user directly states his actual interest in the articles and this feedback drives a form of artificial evolution that improves the agent's performance. Other news filtering agents [Jennings and Higuchi, 1993; Lang,1995] are similar but uses different techniques for learning. Unlike classical relevance feedback systems, which "adapt" to the user's immediate concerns, these systems learn a user model over an extended period.

Learning agents have also been developed to assist a user in filling in a form [Dent et al., 1992; Hermens and Schlimmer, 1993; Maes and Kozierok, 1993]. As the user fills in the various fields of the form, the agent suggests how the remaining fields should be completed. If the agent's predictions are correct, the number of keystrokes needed to complete the form will have been reduced, thereby speeding up the process. Each completed form is added to the set of "training examples" on which the learning agent's subsequent predictions will be based.

An important feature of all these learning agents, and of relevance feedback systems, is that they receive immediate feedback from the user directly indicating the correctness of their predictions. In the news reading and relevance feedback applications, the user indicates immediately whether the articles retrieved are or are not relevant. In the form-filling applications the correct entry for a field is immediately provided. This is crucial to the operation of these systems because this feedback provides new, highly informative training data that can be used to improve the agent's subsequent predictions.

By contrast, in our browsing task the correctness of the apprentice's predictions cannot be determined until the search has ended. Only then does the user know the library item that satisfies his requirements. If our learning apprentice's purpose was to learn in the long-term, feedback about its predictions after the search had ended would be useful. But its purpose is to speedup the search: feedback after the search has ended is of no use.

Our learning apprentice does get some feedback during search, but it is much lower quality than the feedback available to the above learning agents. It is "noisy" and only indirectly related to the correctness of the apprentice's predictions. It is noisy because the user is searching somewhat blindly. To some degree, the user will pursue deadends and circuitous routes, thus giving misleading feedback about which directions are "most promising". There are two reasons why the feedback is not directly related to the correctness of the apprentice's predictions. The first is simply that the user might choose to completely disregard the apprentice's suggestions. This would happen, for example, if the user is in the midst of pursuing his own search strategy. We expect the user to consult the suggestion box only occasionally, when he feels the need for assistance. A more subtle reason is that, unlike the learning agents described above, a learning apprentice for browsing is not attempting to predict or suggest the next action (or sequence of actions). It is trying to predict the final stopping point of the user's search, and this is only remotely related to the user's judgement about which action is leading in the most promising direction.

8 Conclusions

This paper has presented rules for negative inference (i.e. inferring features that the user is not interested in). When added to (a re-implementation of) our original learning apprentice [Drummond et al., 1995] these produce a dramatic improvement in the system's performance. The new system is more than twice as effective at identifying the user's search goal than the original, and it ranks the target much more accurately at all stages of search.

Acknowledgements

This research was supported in part by an operating grant from the Natural Sciences and Engineering Research Council of Canada. We wish to thank Chris Drummond, the developer of the first learning apprentice, for his assistance and advice.

References

Bos, E. (1992), "Some Virtues and Limitations Of Action Inferring Interfaces", 5th Annual Symposium on User Interface Software and Technology.

Cohen, R., and B. Spencer (1993), "Specifying and Updating Plan Libraries for Plan Recognition Tasks", *Proceedings of the Conference on Artificial Intelligence Applications (CAIA'93)*, pp. 27-33.

Cypher, A. (1991),"EAGER: Programming Repetitive Tasks by Example", *SIGCHI'91*, pp. 33-39.

Dent, L., J. Boticario, J. McDermott, T.M. Mitchell and D. Zabowski (1992), "A Personal Learning Apprentice", *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, pp. 96-102.

Drummond, C., D. Ionescu and R.C. Holte (1995), "A Learning Agent that Assists the Browsing of Software Libraries", technical report TR-95-12, Computer Science Dept., University of Ottawa.

Finin, T.W. (1983), "Providing Help and Advice in Task Oriented Systems", *IJCAI'83*, pp. 176-178.

Fischer, G., A. C. Lemke, T. Mastaglio, and A. I. Morch (1990), "Using Critics to Empower users", *Proceedings of CHI-90 ("Empowering People")*, pp. 337-347.

Goodman, B.A. and Diane J. Litman (1990), "Plan Recognition for Intelligent Interfaces", *CAIA'90*, pp. 297-303.

Haines, D., and W.B. Croft (1993), "Relevance Feedback and Inference Networks", *Proceedings of the 16th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2-11.

Harman, D. (1992), "Relevance Feedback Revisited", *Proc. 15th International Conference on Research and Development in Information Retrieval (SIGIR'92)*, pp. 1-10.

Hermens, L.A., and J.C. Schlimmer (1993), "A machine-learning apprentice for the completion of repetitive forms", *Proc. 9th Conference on Artificial Intelligence for Applications*, pp. 164-170.

Hook, K., J. Karlgren, and A. Woern (1993), "Inferring Complex Plans", *Intelligent User Interfaces*, pp. 231-234.

Jennings, A., and H. Higuchi (1993), "A User Model Neural Network for a Personal News Service", *User Modeling and User-Adapted Interaction*, vol. 3, pp. 1-25.

Lang, Ken (1995), "NewsWeeder: Learning to Filter Netnews", *Proceedings of the 12th International Conference on Machine Learning,* Morgan Kaufmann, pp. 331-339.

Maes, P., and R. Kozierok (1993), "Learning Interface Agents", *AAAI'93*, pp. 459-465.

McCalla, G., J. Greer, and R. Coulman (1992), "Enhancing the Robustness of Model-Based Recognition", *Proceedings of 3rd International Workshop on User Modelling*, pp. 240-248.

Mitchell, T.M., S. Mahadevan and L. Steinberg (1985), "LEAP: A Learning Apprentice for VLSI Design", *IJCAI'85*, pp. 573-580.

Schlimmer, J.C. and L.A. Hermens, (1993), "Software Agents: Completing Patterns and Constructing User Interfaces", *Journal of Artificial Intelligence Research*, vol. 1, pp. 61-89.

Sheth, B. and P. Maes (1993), "Evolving Agents For Personalized Information Filtering", *CAIA'93*, pp. 345-352.

Silverman, B.G. and T.M. Mazher (1992), "Expert Critics in Engineering Design: Lessons Learned and Research Needs", *AI Magazine*, vol. 13, no. 1, pp. 45-62.

Witten, I.H. and Dan Mo (1993), "TELS: Learning Text Editing Tasks from Examples", *Watch What I Do*, Allen Cypher (ed.), MIT Press, pp. 183-204.