Inferring What a User is Not Interested In

Robert C. Holte John Ng Yuen Yan
Computer Science Department, University of Ottawa,
Ottawa, Ontario, Canada K1N 6N5.

Abstract

This paper describes a system we have developed 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.

A Learning Apprentice for Browsing

"Browsing" is the searching of a computer library for an individual library item. The human doing the search (the "user") aims to find an item (the "target") that best meets his/her requirements. The user's mental model of the target is called the "search goal". Our testbed browsing application is software reuse. The library is a collection of object-oriented software. An item in the library is a "class" containing locally defined "instance variables" and "methods". A class also inherits the variables and methods of its superclass in the inheritance hierarchy. A class's functionality is determined by its methods. The aim of browsing is to find the class whose functionality is closest to the required functionality.

In our browsing system the user is initially presented 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 from any available class list and then specifies the operator to be applied. An example of a class-based operator is "Defined Methods"; when applied to class C this creates a list of the methods C defines locally. 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 "mark" one or more methods in this window and then specify the operator. For example the operator "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.
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	

To increase the speed and success rate of browsing we have developed a "learning apprentice" (Mitchell et al. 1985) for browsing. 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. It then uses the inferred goal to evaluate the "relevance" of every individual item in the library. The items are sorted according to their relevance and displayed in a special window (the "suggestion box").

The degree of match between two names (of classes or methods) is a number between 1.0 (two identical names) and 0.0 (two names having no subterms (words) in common). The matching process also takes into account methods that are inherited or used by a class. The use of subterms, not whole names, during matching produces some subtle effects. For example, if "Wait" is a subterm of two names in the analogue its effective contribution to the overall score is greater than if it had only occurred in one.

Our initial learning apprentice was very successful, inferring the target before it was found by the user about 40% of the time (Drummond et al. 1995). Its inference rules are all "positive", i.e. all draw conclusions of the form "the user is interested in items with feature X". Table 1 shows a typical sequence of browsing actions. After the first 9 actions, it would have inferred that the user is interested, in different degrees, in the class name "Prompter" and the method names "PaintBackground", "WaitFor", and "WaitForUser".

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. 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. In our system 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 "the user is not interested in W" is added permanently to the analogue. This assertion is categorical; it overrides any previous or subsequent positive inference about W. In the above example, negative inference would produce an assertion of this form for "Wait" and for "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.

Experimental Evaluation

The re-implemented version of the original learning apprentice is called V1 below. V2 is V1 with negative inference added. As in (Drummond et al. 1995) the experiments were run with an automated user, i.e. a computer program (called Rover) that played the role of the user, rather than human users. This enables large-scale experiments to be carried out quickly and also guarantees that experiments are repeatable and perfectly controlled. A similar experimental method is used in (Haines & Croft 1993) to compare relevance feedback systems.

The library used in the experiment is the Smalltalk code library. Each of the 389 classes (except for the four that have no defined methods) 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 (V1 and V2) 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.

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 V1 shows that

TABLE 2. Wins-Losses-Draws			
	Wins	Losses	Draws
V1	70 (23.4%)	207 (69.2%)	22 (7.4%)
V2	161 (52.6%)	110 (36.0%)	35 (11.4%)

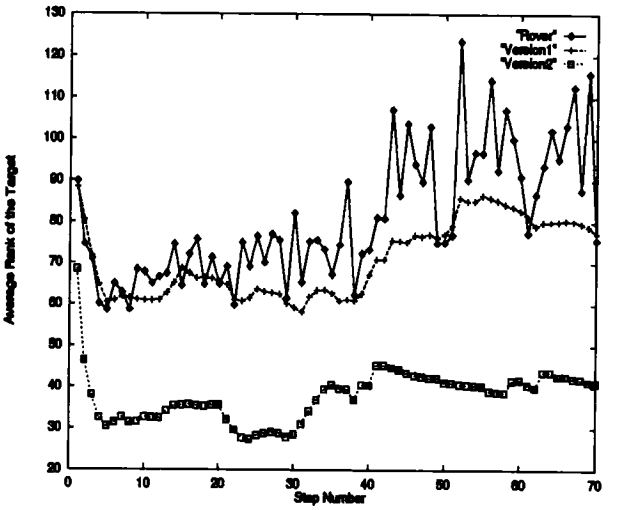
TABLE 3. Search Length when V1 and V2 differ.				
	Average Search Length			Number
	V1	V2	|V1-V2|	of Targets
V1 beat V2	9.9	11.5	1.6	11
V2 beat V1	23.6	12.6	11.0	120

23.4% of the runs were wins for V1. This figure is considerably lower than the 40% win rate of the original learning apprentice. The difference may in part be due to differences in the re-implementation of the learning apprentice or Rover, but is 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 originally. V2 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.

Table 3 is a direct comparison of the learning apprentices to each other. The first row summarizes the runs in which V1 identified the target before V2 did. This happened only 11 times, and on these targets V1 was only 1.6 steps ahead of V2. The second row summarizes the runs in which V2 identified the target before V1. This happened 120 times 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 that it frequently almost doubles the speed with which the learning apprentice identifies the target.

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 higher than the user's own rank for the target. Figure 1 plots the average rank of the target on each step. The average for step N is computed using only the targets that are still active at step N. The best rank is 1; the higher the average rank, the worse the system. From this perspective, V1 is much better than Rover. Except for the first 5-10 steps of a search, the target is 10-20 positions higher in V1's suggestion box than it is in Rover's most recent class list. V2 dramatically outperforms V1. After just one step it ranks the target 20 positions higher than

FIGURE 1. Average Rank of the Target at each step



Rover and V1; 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.

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 these produce a dramatic improvement in performance. The new system is more than twice as effective at identifying the user's search goal and it ranks the target much more accurately at all stages of search.

Acknowledgements

This research was supported in part by a 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

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.

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.

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