IBHYS: A New Approach to Learn Users Habits

Jean David Ruvini and Christophe Fagot
{ruvini, fagot}@lirmm.fr
LIRMM UMR 5506 CNRS-UMII
161, Rue Ada
34392 Montpellier Cedex 5, France
Tel.: +33 4 67 41 86 13 Fax: +33 4 67 41 85 00

Abstract

*Learning interface agents search regularities in the user behavior and use them to predict user's actions. We propose a new inductive concept learning approach, called **IBHYS**, to learn such regularities. This approach limits the hypothesis search to a small portion of the hypothesis space by letting each training example build a local approximation of the global target function. It allows to simultaneously search several hypothesis spaces and to simultaneously handle hypotheses described in different languages. This approach is particularly suited for learning interface agents because it provides an incremental algorithm with low training time and decision time, which does not require, from the designer of the interface agent, to describe in advance and quite carefully the repetitive patterns searched. We illustrate our approach with two autonomous software agents, the Apprentice and the Assistant, devoted to assist users of interactive programming environments and implemented in Objectworks-Smalltalk-80. The Apprentice learns user's work habits using an **IBHYS** algorithm and the Assistant, based on what has been learnt, proposes to the programmer sequences of actions the user might want to redo. We show, with experimental results on real data, that **IBHYS** outperforms **ID3** both in computing time and predictive accuracy.*

1 Introduction

The discovery of repetitive patterns in data is a challenging problem, which has a lot of applications in various domains: fast text searching where patterns are strings of symbols, data mining where patterns are association rules and, particularly, learning interface agents where patterns are sequences of users actions.

Learning interface agents [13] are software agents that assist users of interactive environments by learning their habits and preferences from experience and predicting what they are going to do next. The problem of learning users habits can be modeled as the task of searching repeated patterns in the sequence of the actions the user has performed during work sessions. These repetitive patterns can be noisy (if the user does not perform each time exactly the same sequence of actions), unordered (if the user perform the same actions but not each time exactly in the same order) or both noisy and unordered. Learning interface agents would highly benefit from an incremental and low computing time algorithm to learn such general regularities, able to predict user's action in real time.

Although efficient algorithms have been proposed to find exact or noisy repetitive patterns in strings [9, 24], no algorithm, to our knowledge, solve the problem of finding both noisy and unordered repetitive patterns. On the other hand, machine-learning approaches like inductive concept learning should theoretically be able to learn such general regularities. However, most of the algorithms ([15, 20, 7, 6]) search hypothesis spaces to acquire the definition of a target concept, and large and complex spaces critically slow down the learning process. Therefore, this approach fails to provide an efficient algorithm for learning interface agents because user's actions can be described with a lot of attributes with large set of possible values and training examples generally belong to large hypothesis spaces. Conversely, paradigms like instance-based learning [4] build a local approximation of the target function, and limit the hypotheses search to a small portion of the hypothesis space, but defer the processing of training examples until a new instance must be classified, and require a lot of computing time to predict the target value for a new instance. These paradigms are unable to provide an algorithm to predict user's actions in real time.

We provide here an alternative approach for learning interface agents called instance-based hypothesis search (**IBHYS**). As learning with radial basis functions [19] which approximates the global target function by a combination of local approximations, our approach lets each training example build a set of hypotheses that locally approximate the global target function, limiting the hypothesis search to a small portion of the hypothesis space. However, as a generalization of learning with radial ba-

sis functions , it does not restrict the approximation to a combination of Gaussian functions and allows to handle simultaneously hypotheses described in different languages. This approach is particularly suited for learning interface agents because it provides an incremental algorithm with low training time and decision time, which does not require, from the designer of the interface agent, to describe in advance and quite carefully the repetitive patterns searched. By specifying several hypothesis spaces, he gives the algorithm the potential to find various repetitive patterns.

We illustrate our approach with two interface agents, the Apprentice and the Assistant, that actively assist users of the *Smalltalk*[1] interactive programming environment. This work finds its roots in the idea of programmer apprentices [23, 21] which were ambitious attempts to automatically assist programmers in the task of code production. This produced remarkable results but the task was certainly too complex in whole generality and such apprentices are not really integrated in todays standard programming environments. We propose to apply the techniques of autonomous software agents [10, 14], to merge, eventually extend, the above ideas. The Apprentice and the Assistant aim at letting programmers focus on the essential part of programming (design and write code) by automating the achievement of repetitive tasks.

This paper summarizes the Apprentice-Assistant architecture and focuses more particularly on the issues related to the learning task. Section 2 presents the Apprentice and the Assistant, shows how users actions are recorded and monitored, and defines the learning task. Section 3 describes our **IBHYS** approach, proposes a formalism, gives a general procedure and shows how the Apprentice makes use of this procedure to learn users habits. Section 4 gives experimental results and shows that our **IBHYS** algorithm outperforms the well known **ID3** algorithm [20].

Related works

The idea of employing machine-learning in user-modeling appeared with learning interface agents [22, 12, 14, 1] and begins to be studied in the user-modeling community [16, 18]. However, no incremental algorithm, with low computing time have been proposed to solve the problem of learning repetitive patterns in whole generality. Most of the existing learning interface agents [11, 1, 2, 17] reduce the learning task to the prediction of a few attributes with small set of possible values, and not at all try to predict complex actions like those performed in a programming environment.

[14] studies the question of employing ID3 [20], a decision tree algorithm, in a learning assistant for meeting calendar management. However, by allowing their assistant to spend several hours learning each night, the authors do not propose a low computing time solution to learn users habits.

The closest work to our agents are *OpenSesame!*. OpenSesame! runs in background on Macintosh system 7, and learns repetitive tasks in opening and closing files or applications, emptying trash, rebuilding desktop. Its first weakness is that it is disruptive and frequently solicits the user. Conversely, our Assistant only makes suggestions the user is free to ignore and never request the user. OpenSesame! limits the learning task to a dozen of actions and only learns noisy habits. Unfortunately, it appeared unable to learn simple repetitive opening of folder when we have tested it and the paper describing this system does not bring any other information on this point.

Eager [3] is an interesting software that assists users of the HyperCard environment by anticipating actions. *Eager*, as Holte's assistant for browsing in information libraries [8], is unable to learn noisy or unordered repetitive patterns, does not build a base of habits and forgets habits after it has performed them.

Note that our multiple description languages approach, allows to integrate the graph-based induction technique used in [25], with benefit of low computing time and incrementallity.

2 The Apprentice and the Assistant

We illustrate the **IBHYS** approach with two interface agents, the Apprentice and the Assistant, devoted to assist users of interactive programming environments. Our Apprentice learns user's habits i.e. the tasks the user performs repetitively for which he has not the opportunity or the will to write scripts or macros. The Assistant's task is to accelerate and facilitate the programmers tasks by automating the achievement of repetitive tasks. Based on what the Apprentice has learnt, the Assistant proposes, in a non-obtrusive window the user is free to ignore, sequences of actions the user might want to redo. Both of them operate without explicit intervention of the user. The Apprentice and the Assistant have been developed in Smalltalk 4.0[2]. Figure 1 shows a snapshot of a Smalltalk screen including an assistant window in which the Assistant makes a suggestion triggered by the opening of an exception window.

2.1 Monitoring User's Actions

We define an action to be *any interaction between the user and the interface that affects an interface tool*. By interface tool we mean a software component of the Smalltalk environment (browsers, debuggers, inspectors and editors). We

[1] `ObjectWorks Smalltalk`, copyright `Parc-Place systems`.

[2] We are currently working to adapt our agents to the newest version of Smalltalk

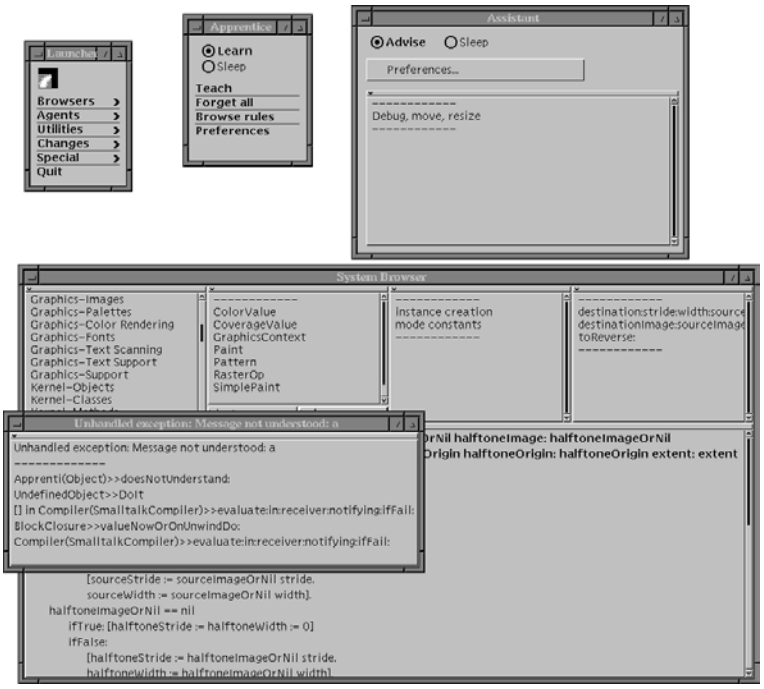

Figure 1: The user has executed a program that has raised an exception. The Assistant window displays a suggestion. It offers to open, move and resize a debugger. If the user accepts the suggestion by clicking on it, the Assistant will automatically open a debugger, move and resize it as the user uses to do

naturally represent actions with Smalltalk objects. For example, class `ActionMenu` models selection of an item in a menu , class `ActionList` models selection of an item in a list, class `ActionSelect` models text highlighting, class `ActionError` models the opening of an error notification window and class `ActionButton` models mouse click on a button. Each class of actions defines instance variables to store parameters for the action. In the following, we will note actions `Class(Tool,Parameter)`. Let us call *trace* the ordered collection of all the actions the user has performed during a work session. The figure 2

```
...
ActionSelect(aStringHolder,'anObject cass')
ActionMenu(aStringHolder,doIt)
ActionError(nil,doesNotUnderstand)
ActionMenu(aDebugger,debug)
ActionMenu(aDebugger,move)
ActionMenu(aDebugger,resize)
ActionList(aDebugger,learn)
...
```

Figure 2: A sample of the trace

shows an example of a trace where the user opens a debugger to correct an error.

2.2 The learning task

Our Assistant should be able to automatically select and propose sequences of actions that the user might want to redo. It has to detect situations in which these repetitive tasks are not fulfilled and it has to avoid them by offering to automate them. These repetitive tasks are all the - exact, noisy or unordered, both noisy and unordered - repetitive sequences of actions of the trace. The task of the Apprentice is to build knowledge that precisely characterize the situations in which these repetitive sequences should be proposed to the user.

Let us call *situation*, for such a repetitive sequence, the last n actions[3] of the trace that immediately precede it. For a given repetitive sequence, there are as many situations as occurrences of this sequence. To characterize these situations, we make the hypothesis that they may be different occurrences of a few *situation patterns* which characterize them. Therefore, in the machine-learning framework, the task of the Apprentice can be seen as a concept learning problem where each exact repetitive sequence of the trace is a concept c which training examples are all the pairs of

[3]The value of n clearly depends on the application field, and is called *description length*.

the form $\prec s, c \succ$, where s is a situation associated to c. The Apprentice has to induce the general definitions of situation patterns given a set of training examples.

Let R be a repetitive sequence of actions, and let A_1, A_2, A_3 and lowercase letters from a to f denote actions. We can distinguish 3 kinds of interesting situation patterns:

1. *Noisy*: For instance, the 2 training examples $\prec A_1aA_2bcA_3, R \succ$ and $\prec A_1dA_2efA_3, R \succ$ of the trace "$...A_1aA_2bcA_3R...A_1dA_2efA_3R...$" can be characterized by the situation pattern $A_1 * A_2 * *A_3$ where the stars denote differences called *errors* or *noise* on the whole actions or on the values of the attributes of the actions.

2. *Unordered*: the training examples $\prec A_1A_3A_2, R \succ$, $\prec A_2A_3A_1, R \succ$ and $\prec A_3A_2A_1, R \succ$ can be characterized by the pattern $\{A_1A_2A_3\}$.

3. *Noisy and unordered*: $\prec A_1aA_2, R \succ$ and $\prec A_2bA_1, R \succ$ can be characterized by the pattern $\{A_1 * A_2\}$.

Figure 6 shows such examples of situation patterns. The 3 kinds of patterns detailed above can be seen as 3 different kinds of hypotheses from 3 different hypothesis spaces which suppose 3 different description languages of the training example and the hypotheses. Therefore, the task of the Apprentice is to find in this different hypothesis spaces, the hypotheses that best explain the membership of each training example (situation) to the related concept (repetitive sequence).

3 IBHYS: the Instance-based Hypothesis Search approach

3.1 A formal framework

The general framework of our work is called inductive concept learning. Let $\mathcal{C} = \{c_1, c_2, ..., c_p\}$ denote a set of concepts and $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2 \cup ... \cup \mathcal{E}_p$ a set of training examples. A training example is a pair of the form $\prec x, c \succ$ where x is the description of the example and $c \in \mathcal{C}$ the concept to which the example belongs. Our approach aims at acquiring the general definition of each concept $c_i \in \mathcal{C}$ from the set $\mathcal{E}_i$ of positive examples and the set $\mathcal{E} - \mathcal{E}_i$ of negative examples. Precisely, it builds approximations, called hypotheses, of each concept c_i. Let $\mathcal{H}$ be the result of the learning process, i.e. the set of hypotheses that actually approximate $c_1, c_2, ..., c_p$. A hypothesis can be seen as a set of constraints on the descriptions of the training examples. A hypothesis h is said to *match* a training example $\prec x, c \succ$, if x satisfies all the constraints of h (conversely, $\prec x, c \succ$ is said to satisfy h). Besides, h *explains the membership* of a positive example $\prec x, c \succ$ if h matches $\prec x, c \succ$ and h aims at approximating c. Our approach has two important advantages.

First, it does not explore a hypotheses space but builds local approximations[4] of the concepts of $\mathcal{C}$ by letting each training example $\prec x, c \succ$ choose the most relevant hypotheses that correctly explain its membership to c. To do so, some hypotheses are successively submitted to $\prec x, c \succ$. Using an evaluation criterion ([5]), $\prec x, c \succ$ is able to compute the relevance of a hypothesis h for the concept c. As a consequence, a training example $\prec x, c \succ$ has to keep $x_{\mathcal{H}}$ the set of the most relevant hypotheses that correctly explain its membership to c.

Second, hypotheses of $\mathcal{H}$ can be expressed in different description languages. The way the hypotheses are treated in the algorithm, described in section 3.2, is independent from their description; and the evaluation of the relevance of the hypotheses is only based on the number of training examples they match. A hypothesis h keeps two important values: 1) n_c^h, the numbers of training examples that h matches in each $c \in \mathcal{C}$ (these values are used by the training examples to measure the relevance of h). 2) $r_{\mathcal{E}}^h$, the number of elements of $\mathcal{E}$ that judge h relevant (this attribute allows to remove from $\mathcal{H}$ a hypothesis that no training example has chosen).

Finally, let us describe the three following operators:

1. MATCH: $\mathcal{H} \times \mathcal{E} \rightarrow$ IB
   Data: a hypothesis h, a training example $\prec x, c \succ$
   Result: true if h matches $\prec x, c \succ$, and false if h does not

   This is the classical subsumption operator. As stated above, our approach allows to handle several hypotheses description languages, so the MATCH operator is strongly linked to these description languages. In fact, MATCH can be seen as a filter among several matching operators, one per description language:

   $\text{MATCH}(h, \prec x, c \succ) = Match_i(h, \prec x, c \succ)$

   where h is expressed in the i^{th} description language, and $Match_i$ is the matching operator of this language.

2. SUBMIT: $\mathcal{H} \times \mathcal{E} \times$ IR $\rightarrow Void$
   Data: a hypothesis h, a training example $\prec x, c \succ$, a threshold k
   Result: updates the set of $x_{\mathcal{H}}$ regarding the hypothesis h

   The training example $\prec x, c \succ$ computes the relevance of the hypothesis h (using [5]) for the concept c and updates its set $x_{\mathcal{H}}$. The hypothesis h may be: 1) relevant for $\prec x, c \succ$ (h matches $\prec x, c \succ$), and is added to $x_{\mathcal{H}}$; 2) irrelevant for $\prec x, c \succ$ and $\prec x, c \succ$

[4] $\mathcal{H}$ is a global approximation of the concepts of $\mathcal{C}$, and each hypothesis of $\mathcal{H}$ is a local approximation of a concept of $\mathcal{C}$

rejects it. $\prec x, c \succ$ may possibly remove the less relevant hypothesis in $x_{\mathcal{H}}$ if it must keep the size of $x_{\mathcal{H}}$ constant. Besides, $r^h_{\mathcal{E}}$ is updated.

Regarding to the application field, the threshold k can be used either to bound the size of $x_{\mathcal{H}}$, or to set the minimum relevance accepted to add any hypothesis in $x_{\mathcal{H}}$.

3. HYPGEN: $\mathcal{E} \times 2^{\mathcal{O}} \rightarrow 2^{Hyp}$
   Data: a training example $\prec x, c \succ$, a set of objects O
   Result: a set of hypotheses

   HYPGEN is the hypotheses generation operator. Hyp denotes the space of all the hypotheses that can be generated. HYPGEN is strongly linked to the application field, and allows hypotheses to be formed by comparison between a training example $\prec x, c \succ$ and a set of objects $\mathcal{O}$. Hypotheses can be generated by comparison with other training examples, hypotheses, or any other objects useful for the hypotheses generation (in the procedure described after, $\mathcal{O} = \mathcal{E} \cup \mathcal{H}$). The main advantage of our approach is that it makes possible to handle simultaneously several different description languages of the hypotheses. HYPGEN can be seen as the combination of several hypothesis generators, one per description language:

   HYPGEN$(\prec x, c \succ, O) = \bigcup_{i=1}^{n} HypGen_i(\prec x, c \succ, O)$

   where $HypGen_i$ is the hypothesis generator for the i^{th} description language. Suppose $O = \mathcal{E}$. Suppose the training examples are described both by directed graphs and conjunctions of attribute-value pairs. A simple example of hypothesis generator outputs the maximal tree included in $\prec x, c \succ$ and in all the description of the element of O. Another hypothesis generator compares $\prec x, c \succ$ to all the elements of O. For each pair ($\prec x, c \succ$,$\prec y, d \succ$) it returns a hypothesis which description is constituted of the attribute-value pairs that x and y shares in common.

   One of the interest of using multiple hypothesis generators can be evaluated easily. Suppose that training examples are described with n boolean features. This leads potentially to an hypothesis space of 2^n elements. Suppose now, this set of n features can be split in 3 disjoint sets of $n/3$ features. These 3 sets lead to 3 hypothesis spaces of $2^{n/3}$, and $3 \star 2^{n/3} \ll 2^n$.

3.2 A General Algorithm

We now give an illustration of the **IBHYS** approach through a general procedure, called **NewExample**. The main steps of this procedure are explained below. In the following, $\mathcal{E}$ is the set of training examples currently available, $\mathcal{H}$ the set of the currently relevant hypotheses, and the threshold k is used in the SUBMIT operator. Given a new training example $\prec x, c \succ$, the procedure updates the sets $\mathcal{E}$ and $\mathcal{H}$.

The most important steps of the **IBHYS** procedure are:

Step 1 Each hypothesis of $\mathcal{H}$ have to know the number of training examples it matches in each class. These values allow the training examples to evaluate the relevance of the hypotheses of $\mathcal{H}$.

Step 2 The numbers of training examples matched by each hypothesis have been modified (step 1). Some hypotheses that were relevant for a training example $y \in \mathcal{E}$ may not be relevant any more. This may happen if the hypotheses matching y are all almost as relevant as each other.

Step 3 Using $\prec x, c \succ$, $\mathcal{E}$ and $\mathcal{H}$, the operator HYPGEN generates a set of hypotheses that will be individually studied only if they are not yet in $\mathcal{H}$. Whatever the description languages of the hypotheses are, all the generated hypotheses will be dealt with in the same way (described in the steps 4 and 5).

Step 4 Each hypothesis generated by HYPGEN have to know the amount of training examples it matches in each class, allowing the training examples to evaluate its relevance.

Step 5 The currently studied hypothesis h must be evaluated by the training examples of $\mathcal{E}$ to measure its relevance. For each training example $\prec y, c \succ \in \mathcal{E}$, this step updates the set $y_{\mathcal{H}}$ of its relevant hypotheses (and the value $r^h_{\mathcal{E}}$).

Step 6 This step aims at removing the irrelevant hypotheses of $\mathcal{H}$. A hypothesis h is said to be irrelevant if none of the training examples of $\mathcal{E}$ has chosen it (*i.e.* $r^h_{\mathcal{E}} = 0$).

3.3 The Apprentice algorithm

The Apprentice learns user's habits every 100[5] actions of the user. It first searches the training examples that appear in the last 100 actions of the trace, and invokes the procedure **NewExample** for each new training example it has found.

We defined 3 operators (taking 2 situations in input) to allow the Apprentice to learn the 3 kinds of situation patterns defined above: `noisy` which builds a pattern which has the commune characteristics, regarding the position, of

[5] Again, this value depends on the application field.

Algorithm 1: NewExample: the main procedure of IB-HYS

Data: $\mathcal{E}$ a set of training examples
$\mathcal{H}$ a set of hypotheses
k the threshold to keep the hypotheses
$\prec x, c \succ$ a new training example

Result: Updating of $\mathcal{E}$ and $\mathcal{H}$ considering x

begin

1 \\ *Update the number of examples*
\\ *matched by each hypothesis.*
foreach $h \in \mathcal{H}$ **do**
  if MATCH*(h,*$\prec x, c \succ$*)* **then**
    $n^h_{\mathcal{C}} \leftarrow n^h_{\mathcal{C}} + 1$;

\\ *Insert* $\prec x, c \succ$ *in the set of examples.*
$\mathcal{E} \leftarrow \mathcal{E} \cup \{\prec x, c \succ\}$;

2 \\ *Update the set of relevant hypotheses*
\\ *of each example in* $\mathcal{E}$.
foreach $h \in \mathcal{H}$ **do**
  foreach $\prec y, d \succ \in \mathcal{E}$ **do**
    SUBMIT(h,$\prec y, d \succ$,k);

3 \\ *Treatment of hypotheses created thanks to* x.
foreach $h \in$ HYPGEN*(*$\prec x, c \succ$*,* $\mathcal{E} \cup \mathcal{H}$*)* **do**
  if $h \notin \mathcal{H}$ **then**
4 \\ *Number of example matched*
    \\ *by* h *in each class.*
    foreach $\prec y, d \succ \in \mathcal{E}$ **do**
    if MATCH*(h,*$\prec y, d \succ$*)* **then**
      $n^h_{yc} \leftarrow n^h_{yc} + 1$;

5 \\ *Submit the hypothesis to the examples.*
    foreach $\prec y, d \succ \in \mathcal{E}$ **do**
    SUBMIT(h,$\prec y, d \succ$,k);

    \\ *Insert* h *in the set of hypotheses.*
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{h\}$;

6 \\ *Delete of* $\mathcal{H}$ *the hypotheses*
\\ *witch are not relevant.*
foreach $h \in \mathcal{H}$ **do**
  if $r^h_{\mathcal{E}} = 0$ **then** $\mathcal{H} \leftarrow \mathcal{H} - h$;

end

the two situations and stars (denoting noise) for their differences, `unordered`: which returns the set of the actions of the first situation, if and only if, these actions all appear in the second situation, with no constraint on their positions, and `noisyUnordered`. We also defined 3 classes of hypotheses that define their own method MATCH to test coverage of training examples, and 3 hypothesis generators, based on the 3 operators defined above.

Let us call *knowledge base* the set of hypotheses produced by our **IBHYS** algorithm. After each action of the user, the Assistant inspects the knowledge base and selects all habits which hypothesis cover the last actions of the user. The Apprentice then displays suggestions corresponding to the related concepts in an non-obtrusive window. The user is free to take it into account or not. A simple mouse-click on one of these suggestions automatically performs the related actions.

4 Experimental results

The Apprentice and the Assistant are currently used by the first author. Experiments reported here where conducted during the development of an "ASCII to HTML" translator, on real data. We compare our **IBHYS** algorithm to **ID3** which is the decision tree algorithm used in [14] to "explore the potential of machine-learning methods to automatically create and maintain ... customized knowledge for personal software assistants".

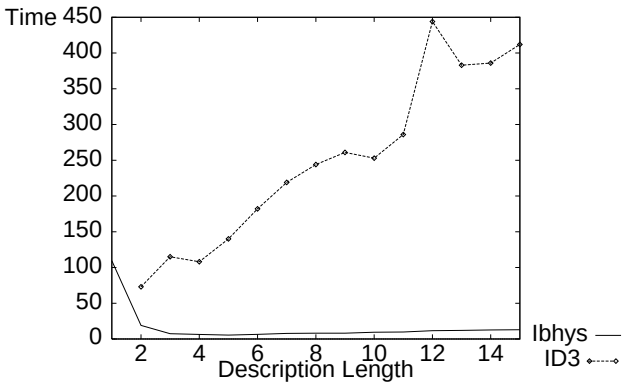

Figure 3: Computing time in minutes versus description length

Figure 3 and 4 show that **IBHYS** outperforms **ID3** regarding the computing time. The figure 3 plots the computing time versus the description length (cf. 2.2) on a trace of 1000 actions, and the figure 4, the computing time versus the size of the trace, for a description length of 10 actions. Time is given in minutes.

	1	2 Accuracy		3 Excess		4 Hypotheses	
Trace	$\mid \mathcal{E} \mid$	ID3	Ibhys	ID3	Ibhys	ID3	Ibhys
100	31	4,48	10,76	51,00	50,00	29	25
200	81	16,73	34,60	47,50	48,00	64	69
300	128	60,15	49,25	46,67	47,33	92	130
400	174	39,77	36,29	47,00	50,25	106	169
500	230	73,49	75,81	41,20	42,00	121	217
600	279	69,95	76,17	38,17	38,17	125	228
700	329	59,04	68,67	38,14	38,14	160	272
800	374	45,45	69,09	37,50	37,75	171	278
Mean		46,13	52,58	43,40	43,96		

Figure 5: Accuracy

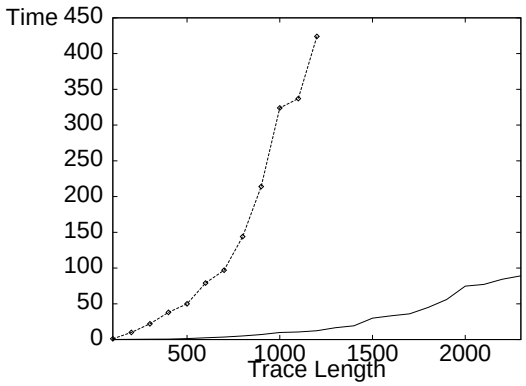

Figure 4: Computing time in minutes versus trace length

Table in figure 5 is a direct comparison of the respective accuracies of **IBHYS** and **ID3**. These tests were performed on a trace of 1000 actions, with $\mid x_{\mathcal{H}} \mid = 2$ (see SUBMIT in section 3). These 1000 actions were split in a training set and a test set. The leftmost column lists the size of the training sets used, and column 1 lists the number of training examples (repetitive sequences) the algorithms have found in the training sets. Column 2 shows the predictive accuracy on new examples. It shows that **IBHYS** had correctly predicted a repetitive sequence in 52.58% of the case, versus 46.13% for **ID3**. Column 3 lists the "excess rate" that is, the number of time the algorithms have predicted erroneous repetitive sequences whereas no prediction were expected. This excess rate value is very important. Hight values means that the agent constantly bothers the user with useless suggestions. **IBHYS** and **ID3** have almost the same excess rate. Finally, column 4 lists the number of hypotheses the algorithms have learnt. Note that both **IBHYS** and **ID3** have an average decision time of 10 milliseconds.

Due to the fact that the Apprentice and the Assistant have been implemented in Smalltalk 4.0, and that few programmers still use this environment, we could not find programmers to intensively test our agents. Of course, we are working to adapt our agents to the newest version of the Smalltalk environment. However, we can give examples of the habits learnt during the experiments reported here (figure 6). Habit 1 means that the user systematically moves and resizes a debugger he has opened after an error; habit 2 shows that the user systematically removes system comments of new methods.

5 Conclusion

We have proposed a new approach, called **IBHYS**, and an incremental algorithm with low computing time, for inductive concept learning, particularly suited for learning interface agents. This approach lets each training example build a set of hypotheses that locally approximate the global target function, limiting the hypothesis search to a small portion of the hypothesis space. Because training examples can choose among several description languages to form an hypothesis, and different description languages to form different hypotheses, it allows to handle simultaneously hypotheses described in different languages. We presented an application of this approach to learn user's habits of interactive programming environments and propose an original assistance to programmers based on two software agents, the Apprentice and the Assistant. We showed, with experimental results on real data, that **IBHYS** outperforms **ID3** both in computing time and predictive accuracy.

IBHYS seems a promising approach for data-mining. Further studies will be conducted to evaluate our **IBHYS** approach with respect to standard (Irvine collection) machine-learning datasets. In the context of the Apprentice and the Assistant, an important limitation of **IBHYS** is that it bounds in advance the length of the description and, therefore, the length of the situation patterns searched. We are investigating to bypass this limitation.

We are currently working to adapt the Apprentice and

1.

Situation pattern	*Repetitive sequence*
`ActionErreur(nil,*)`	`ActionMenu(aDebugger,move)`
`ActionMenu(aDebugger,debug)`	`ActionMenu(aDebugger,resize)`

2.

Situation pattern	*Repetitive sequence*

`ActionSelect(aBrowser,`
`message selector and argument names`
`''comment stating purpose of message''`
`| temporary variable names |`
`statements)`
	`ActionMenu(aBrowser,cut)`

Figure 6: Example of user's habits

the Assistant to the newest version of the Smalltalk environment. We hope that they will be soon available to full time programmers for intensive tests.

Acknowledgements

We would like to thank Christophe Fiorio for his Algorithm LaTeX style and his help in the preparation of the final manuscript of this paper.

References

[1] R. Armstrong, D. Freitag, T. Joachims, and T. Mitchell. Webwatcher: A learning apprentice for the world wide web. In *AAAI Spring Symposium on Information Gathering*, 1995.

[2] A. Caglayan, M. Snorrason, J. Jacoby, J. Mazzu, and R. Jones. Lessons from open sesame!, a user interface learning agent. In *Proceedings of PAAM96*, pages 61–74, Apr. 1996.

[3] A. Cypher. EAGER: Programming repetitive tasks by example. In *Proceedings of ACM CHI'91*, Programming by Demonstration, pages 33–39, 1991.

[4] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York, 1973.

[5] O. Gascuel and G. Caraux. Distribution-free performance bounds with the resubstitution error estimate. *Pattern Recognition Letters*, 13:757–764, 1992.

[6] J. Hertz, A. Krogh, and R. G. Palmer. *An Introduction to the Theory of Neural Computation*. Lecture Notes Volume I. Addison Wesley, 1991.

[7] J. H. Holland. *Adaptation in natural artificial systems*. University of Michigan Press, Ann Arbor, 1975.

[8] R. C. Holte and C. Drummond. A learning apprentice for browsing. In O. Etzioni, editor, *Software Agents — Spring Symposium*. AAAI Press, Mar. 1994.

[9] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *4th Annual ACM Symposium on Theory of Computing*, pages 125–136, Denver, Colorado, 1–3 May 1972.

[10] Y. Lashkari, M. Metral, and P. Maes. Collaborative interface agents. In *Proceedings of AAAI'94*, pages 444–449, 1994.

[11] P. Maes. Agents that reduce work and information overload. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):31–40, July 1994.

[12] P. Maes. Social interface agents: Acquiring competence by learning from users and other agents. In O. Etzioni, editor, *Software Agents — Spring Symposium*, pages 71–78. AAAI Press, Mar. 1994.

[13] P. Maes and R. Kozierok. Learning interface agents. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 459–464, Menlo Park, CA, USA, July 1993. AAAI Press.

[14] T. Mitchell, R. Caruana, D. Freitag, J. McDermott, and D. Zabowski. Experience with a learning personal assistant. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):81–91, July 1994.

[15] T. M. Mitchell. *Version Spaces: An Approach to Concept Learning*. PhD thesis, Electrical Engineering Dept., Stanford University, 1979.

[16] J. Orwant. Heterogenous learning in the doppelgänger user modeling system. *User Modeling and User-Adapted Interaction*, 4(2):107–130, 1995.

[17] T. R. Payne and P. Edwards. Interface agents that learn: an investigation of learning issues in a mail agent interface. *Applied Artificial Intelligence*, 11:1–32, 1997.

[18] W. Pohl. Learning about the user – user modeling and machine learning. In V. M. J. Herrmann, editor, *ICML'96 Workshop Machine Learning meets Human-Computer Interaction*, pages 29–40, 1996.

[19] M. J. D. Powell. Radial basis functions for multivariable interpolation: A review. In *Algorithms for Approximation*, pages 143–167, Oxford, 1987. Clarendon Press.

[20] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[21] C. Rich and R. C. Waters. The programmer's apprentice. *Computer*, pages 11–25, Nov. 1988.

[22] J. C. Schlimmer and L. A. Hermens. Software agents: Completing patterns and constructing user interfaces. *Journ. of AI Research*, 1:61–89, Nov. 1993.

[23] R. Waters. The programmer's apprentice: Knowledge-based program editing. *IEEE Trans. Software Engineering*, SE-8(1):1–12, Jan. 1982.

[24] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, Oct. 1992.

[25] K. Yoshida and H. Motoda. Automated user modeling for intelligent interface. *Int. J. of Human Computer Interaction*, 3(8):237–258, 1996.