Machine Learning Techniques to Make Computers Easier to Use

Hiroshi Motoda
The Inst, of Scientific and Industrial
Research, Osaka University
Mihogaoka, Ibaraki, Osaka 567, Japan

Kenichi Yoshida
Advanced Research Laboratory,
Hitachi, Ltd.
Hatoyama, Saitama 350, Japan

Abstract

Identifying user-dependent information that can be automatically collected helps build a user model by which to predict what the user wants to do next and to do relevant preprocessing. Such information is often relational and is best represented by a set of directed graphs. A machine learning technique called graph-based induction *{GBI)* efficiently extracts regularities from such data, based on which a user-adaptive interface is built that can predict next command, generate scripts and prefetch files in a multi task environment. The heart of *GBI* is pairwise chunking. The paper shows how this simple mechanism applies to the top down induction of decision trees for nested attribute representation as well as finding frequently occurring patterns in a graph. The results clearly shows that the dependency analysis of computational processes activated by the user commands which is made possible by *GBI* is indeed useful to build a behavior model and increase prediction accuracy.

1 Introduction

Computers are still not easy to use. The main reason is their ignorance about the user. The user information that is available to an interactive computer system is limited, and thus, the user model acquisition is a difficult problem. Classical acquisition methods like user interviews, application-specific heuristics, and stereotypical inferences are often not appropriate, and a better automated method is being sought.

Finding regularities in data is a basis of knowledge acquisition, and extracting behavioral patterns from the user information is one such problem. Each user has a different way of doing the same thing and identifying the information that can characterize the user and be automatically collected is crucial. Once such information is found and if an appropriate machine learning technique can induce regularities in each user's behavior to carry out his/her intended task, we can use them to guide the daily work and to do some preprocessing, which may facilitate easiness of usage and increase efficiency.

We discuss three learning tasks, command prediction, script generation and file prefetching in multi task environments. The scope of user behavior is limited to a sequence of task execution *(e.g.,* editing, formatting, viewing, etc.) using plural application programs.

Most studies that attempted to develop a user-adaptive interface system only analyzed the sequence of user behaviors, from which to automate the repetitions (See 7). In this setting, the data can easily be represented by at tribute-value pairs, each attribute denoting the sequence order and its value, the command. Since the command sequence does not necessarily typify the user's behavior, the user model constructed from only the sequence information may not adequately capture the user's behavior. We focused on the process I/O information that is also automatically collected along with the command sequence. Since this is dependency information and its relationship cannot be fixed in advance, it is not straightforward to represent this by attribute-value pairs.

We show that graph-based induction [Yoshida and Motoda, 1995] can nicely be applied to the three learning tasks. In this paper, we revisit *GBI,* show how it can extract typical patterns from a set of directed graphs and how it can induce classification rules using a similar technique in the Top Down Decision Tree Induction (TDDT) algorithm. The first and the second learning tasks are implemented as *ClipBoard* which is a window like UNIX shell [Yoshida and Motoda, 1996], and the third task is implemented as *Prefetch daemon* that is hidden from the user. The results clearly show that the dependency analysis of computational processes activated by the user's commands, which is made possible by *GBI,* is indeed useful. *ClipBoard* is in daily use and its prediction accuracy and response time are satisfactory. *Prefetch daemon* works as expected only for I/O intensive task due to an implementation problem, and thus needs further improvement.

The following section introduces the three learning tasks. Subsequent sections describe the learning method *GBI* and summarize the results of learning experiments performed to date. The last two sections consider lessons learned from this study and directions for future research.

2 Learning Tasks

Command prediction is a real time task that takes a user's operational history and predicts the next command. Figure 1 shows, in a simplified form, an example of operational history when a user is making a document using a *latex* document formatter. The bold arrows show the command sequence. The history includes, in addition to this, I/O relationships between commands, and thus, takes the form of a directed graph. Each link has a label that corresponds to a file extension. For example, the link connecting *latex* to *emacs* has a label *tex.* However, one link is reserved for sequence information. *ClipBoard* keeps recording and updating the history, and at any point of operation, predicts the next command. The learning task is to induce classification rules from the past history. For each command in the past, a directed graph of a certain depth (number of sequentially connected links) and width (number of sibling links) is taken out[1]. Each directed graph forms a training example. Its root is a class and the rests are considered to be nested attributes.

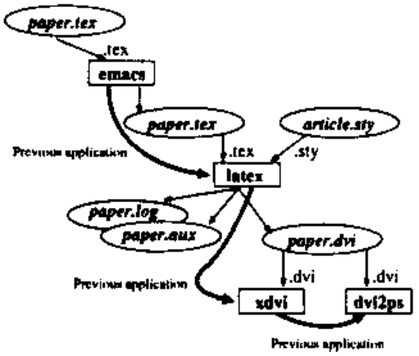
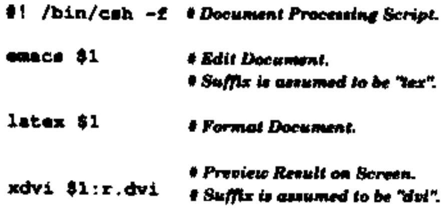


Figure 1: I/O relationships between commands (applications)

Script generation is a batch task that extracts frequently occurring patterns from a large graph representing a history of order of days, generalizes the arguments and generates shell scripts to execute a sequence of operations by a single command. Figure 2 shows an example of the generated scripts when a user repeatedly calls up *emacs, latex* and *xdvi*

File prefetching is a real time task that predicts files to be used in the immediate future and prefetches them into the cache. Unlike the command prediction, prefetching must predict a few steps ahead and thus more than one file. The learning task is done in a batch mode using a large directed graph. The task is to extract frequently occurring patterns first like script generation, from each of which a prefetch rule is generated and then to merge them into a single trie structure (example shown in Fig. 10). The prefetching is made in real time based on this trie. Since prefetching is automatic, this task is invisible.



```
#! /bin/csh -f      # Document Processing Script.

emacs $1            # Edit Document.
                    # Suffix is assumed to be "tex".

latex $1            # Format Document.

                    # Preview Result on Screen.
xdvi $1:r.dvi       # Suffix is assumed to be "dvi".
```

Figure 2: Example of a generated script

3 Graph-based Induction

3.1 Finding Regularities in a Directed Graph

GBI was originally intended to find interesting concepts from inference patterns by extracting frequently appearing patterns in the inference trace. In [Yoshida and Mo toda, 1995], it is shown that *GBI* was able to discover the notion of NOT and NOR from the simulation traces of an electric circuit. In this application, the original inputs are causal relations of voltage and current between various nodes of the circuit; there is no notion of logical operation. However, by finding regularities in the input traces, it was able to lift up the abstraction level and find more abstract concepts. Later, we showed that the same idea can be applied to other types of learning (speed up learning and classification rule learning) [Yoshida *et a*/., 1994]. "

The original *GBI* was so formulated to minimize the graph size by replacing each found pattern with one node that it repeatedly contracted the graph. The graph size definition reflected the sizes of extracted patterns as well as the size of contracted graph. This prevented the algorithm from continually contracting, which meant the graph never became a single node. Because finding a subgraph is known to be NP-hard, the ordering of links is constrained to be identical if the two subgraphs are to match, and an opportunistic beam search similar to genetic algorithm was used to arrive at suboptimal solutions. In this algorithm, the primitive operation at each step in the search was to find a good set of linked pair nodes to chunk (pairwise chunking).

Because the search is local and stepwise, we can adopt an indirect index rather than a direct estimate of the graph size to find the promising pairs. On the basis of this notion, we generalize the original *GBI,* and further extend it to cope with the classification problem. The idea of pairwise chunking is given in Fig. 3, and the general algorithm in Fig. 4.

The selection criterion of the pair nodes should be such that its use can find interesting patterns *(e.g.,* patterns occurring more frequently than others or patterns more

[1] In the experiments described in 4.1, the depth was set 5 and the width 128 (this is maximum and automatically adjusted).

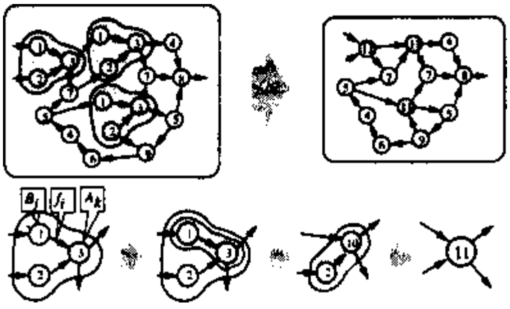


Figure 3: The idea of graph contraction by pairwise chunking

```
GBI(G)
  Selection of pair nodes $(A_k, f_i, B_j)$
  Chunk the pair nodes into one node: $c$
  $C := C \cup \{c\}$
  $G_c :=$ contracted graph of $G$
  While termination condition not reached
      $C := C \cup GBI(G_c)$
  end_while
  Output $C$
```

Figure 4: Generalized algorithm of *GBI*

easily identifiable than others). Proper termination condition must be used in accordance with the selection criterion *(e.g.,* iteration number, chunk size, change rate of selection indexes, etc.). Examples of such indexes are information gain [Quinlan, 1986], information gain ratio [Quinlan, 1993] and gini index [Breiman *et* a/., 1984].

We use information gain as an index here, but the other indexes can be used in the same way. Unlike decision tree building where the index is used for selecting an attribute, here we have to select linked pair nodes. Each node has a value (color) and each link has a label. We can interpret the triplet (A_k, f_i, B_j) as saying that the value of the i-th attribute f_i of the parent A_k is B_j or when the i-th attribute f_i takes the value B_j, its immediate result is *Ak.* The problem is which (i, j, *k)* to select to chunk. A natural way is to focus on one of the three elements, and select the best remaining two to identify the chosen element. Three alternatives exist: a) focus on k, b) focus on *i* and c) focus on *j.* Case a) tries to find the attribute and its value pair that best characterizes the chosen immediate result. Likewise, case b) tries to find the result and the attribute value pair that best characterizes the chosen attribute, and case c) tries to find the attribute and its result pair that best characterizes the chosen attribute value. Which one to adopt depends on what the directed graph represents in terms of the original problem description. The default is to choose a).

In what follows, only case a) is described. The other two are obtained by permutating the subscripts. Let the underline in the subscript mean its complement *(e.g., i* means the attributes other than the i-th.), the overline in the variable mean a vector for the unspecified subscript(s) (*e.g.*, $\overline{n_k}$ means $\{n_{i,j}\}$ for a k), and the superscript *yes* and *no* mean the result of the division by test. The amount of information that is required to identify k before selecting the triplet is

$$I(\overline{n_k}) = -\sum_{i,j} \frac{n_{k,i,j}}{N_k} \log_2 \frac{n_{k,i,j}}{N_k},$$

where N_k is the number of nodes that have value A_k and $n_{k,i,j}$ is the number of the triplets (A_k, f_i, B_j) (*i.e.*, the number of nodes that have value A_k and their i-th attributes have value B_j.).

The amount of the same information after the selection is

$$E(A_k, f_i, B_j) = \frac{N_{k,i,j}}{N_k} I(n^{yes}_{k,i,j}) + \frac{N_{k,\underline{i,j}}}{N_k} I(\overline{n^{no}_{k,i,j}}),$$

where

$$I(n^{yes}_{k,i,j}) = -\frac{n_{k,i,j}}{N_{k,i,j}} \log_2 \frac{n_{k,i,j}}{N_{k,i,j}} = \frac{N_{k,i,j}}{N_{k,i,j}} \log_2 \frac{N_{k,i,j}}{N_{k,i,j}} = 0$$ [2]

$$I(\overline{n^{no}_{k,i,j}}) = -\sum_{i',j'(\neq i,j)} \frac{n_{k,i',j'}}{N_{k,\underline{i,j}}} \log_2 \frac{n_{k,i',j'}}{N_{k,\underline{i,j}}}.$$

$$Info\_gain(k,i,j) = I(\overline{n_k}) - E(A_k, f_i, B_j) =$$
$$\frac{1}{N_k}\{ \sum_{i',j'(\neq i,j)} n_{k,i',j'} \log_2 \frac{n_{k,i',j'}}{N_{k,\underline{i,j}}} - \sum_{i,j} n_{k,i,j} \log_2 \frac{n_{k,i,j}}{N_k} \}$$

The best attribute and its value for each k to select is

$$Argmax_{(i,j)}\{Info\_gain(k,i,j)\} = (k, i_{0k}, j_{0k}).$$

Thus, the best triplet is determined to be

$$Argmax_k\{N_k Info\_gain(k, i_{0k}, j_{0k})\} = (k_0, i_0, j_0).$$

This is recursively repeated until a termination condition is satisfied.

3.2 Inducing Classification Rules

In case of the classification problem, we interpret the root node as a class node and the links attached to it as the primary attributes. The node at the other end of each link is the value of the attribute, which has secondary attributes. Thus, each attribute can have its own attributes recursively, and the graph (*i.e.*, each instance of the data) becomes a directed tree. In this case, the pairwise chunking must start at the root node and go backwards following the links. Here, we have to select the attribute and its value pair that best characterizes the class. So the selection index is slightly different from the normal *GBI* described above.

The amount of information before the selection is

$$I(\overline{n}) = \sum_k \frac{n_k}{N} \log_2 \frac{n_k}{N},$$

where n_k is the number of nodes that have class value $A_k (k = 1, K)$, and $N = \sum_k n_k$. The amount of information after the test by the attribute f_i whose value is B_j is

[2] Note that the triplets that go into the *yes* branch are all identical, implying $I(n^{yes}_{k,i,j}) = 0$.

$$E(f_i, B_j) = \frac{N_{ij}^{Yes}}{N} I(\overline{n_{ij}^{Yes}}) + \frac{N_{ij}^{No}}{N} I(\overline{n_{ij}^{No}}).$$

Thus, the best attribute f_i and its value B_j to select for testing is

$$Argmax_{(i,j)}\{I(\overline{n}) - E(f_i, B_j)\} = (i_0, j_0).$$

This is recursively repeated until each subgroup, after testing, contains a single class value or some stopping condition is satisfied.

4 ClipBoard Interface

Figure 5 shows the system configuration for *ClipBoard Interface* and *Prefetch Daemon.* The process I/O recorder is a part of the operating system and records all the I/O operations of each command issued. This information is represented together with the command sequence by a directed graph as operation history. *GBI* program runs on this graph and generates prediction (classification) rules and typical patterns. The mouse-based command controller uses these to 1) select the next command, and to 2) create UNIX shell scripts. The prefetch daemon uses the typical patterns to generate prefetch rules and merges them into a trie structure to 3) prefetch files.

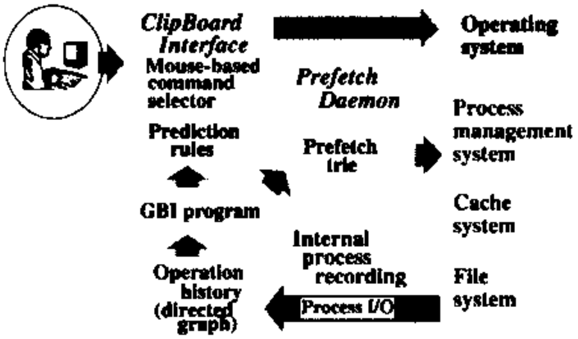
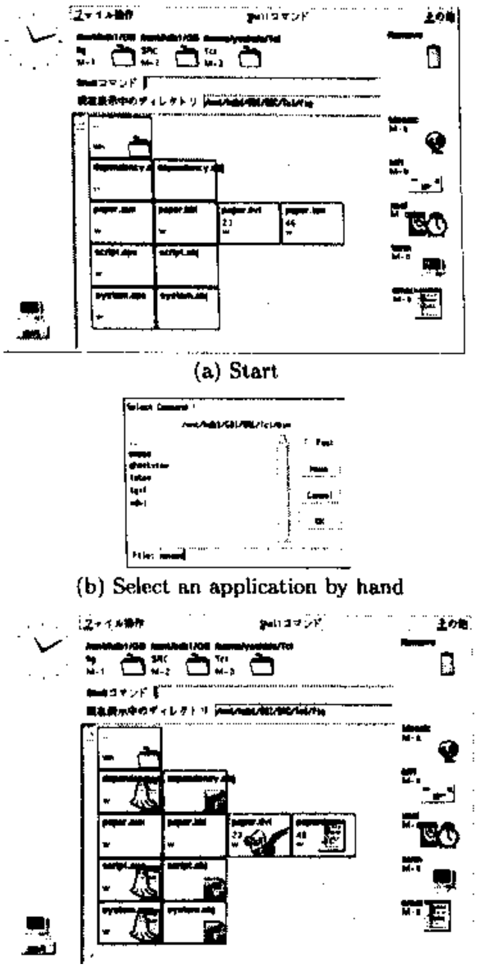


Figure 5: ClipBoard and prefetch system configuration

Figure 6 displays the screen images of *ClipBoard* during a simple document processing task. When *ClipBoard* starts without any information, the screen lists only file names (Fig. 6 (a)). At this stage, after selecting a file to be processed, the dialogue box appears so that the user can specify the command (Fig. 6 (b)). If the user specifies *emacs,* it treats *emacs* as the default for the file with the extension *tex. ClipBoard* tries to learn the appropriate command for each file extension, and recommends the command by icons (Fig. 6 (c)). *ClipBoard* never asks the user for information. The user can always override *Clip Boards* recommendation, which triggers the learning task. Icons for the same files change over time reflecting context changes.

Currently, *ClipBoard* interface is written by Tcl/Tk. The GBI program has both C and Lisp versions. The prefetch daemon is written by Java.



(a) Start

(b) Select an application by hand

(c) *ClipBoard* suggests *emacs* for *paper.tex*

Figure 6: Screen image of *ClipBoard*

4.1 Command Prediction

I/O Information Analysis

Consider an operation history in Table 1. As shown in steps (A), (B), and (C), the file *paper.dvi* is processed by three different commands: *xtex, xdvi* and *dvi2ps.* Figure 7 shows the corresponding directed graphs that are inputs to *GBI.* The algorithm described in 3.2 first chooses the *dvi* attribute (f_i) and its value *latex* (B_j) for testing, and chunks the triplets *(xdvi, dvi, latex)* in (B) and *(dvi2ps, dvi, latex)* in (C). The *No* branch contains only one instance, (A), and the *Yes* branch contains two instances, (B) and (C). Next, the algorithm chooses the *sequential* attribute and its value *xdvi* for testing and chunks the triplet *((dvi2ps, dvi, latex), seq., xdvi)* This separates (C) from (B) and the induction stops[3]. The

[3] In reality, there are many occasions in history where *dvi* files are used by the same command that has different de-

Table 1: Operation history

Step	Application	Input File
(A)	xtex	paper.dvi
	emacs	paper.tex
	latex	paper.tex
(B)	xdvi	paper.dvi
(C)	dvi2ps	paper.dvi

shaded parts in Fig. 7 are the typical patterns. Figure 8 is the interpretation of the extracted patterns.

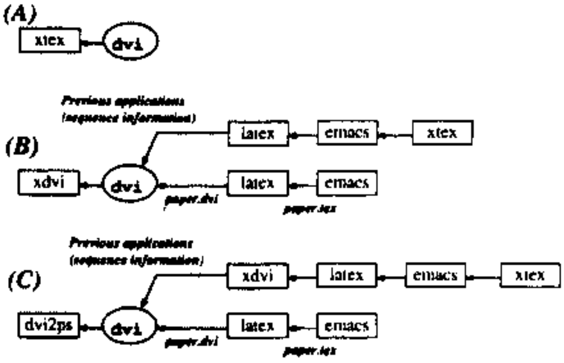


Figure 7: Typical patterns in the operation history

```
(C) If   dvi file was made from latex,
    and  the previous command was xdvi,
    Then Next Application is dvi2ps.
(B) If   dvi file was made from latex,
    and  the previous command was latex,
    Then Next Application is xdvi.
(A) If   None of the above,
    Then Next Application is xtex.
```

Figure 8: Interpretation as prediction rules

GBI assumes the existence of a strong correlation between the linked attributes. As described in 3.2, the algorithm follows the standard TDDT induction, but the attributes to be selected are dynamically modified in the process. Note that it is impractical to represent the graph structure by a single table of attribute-value pairs.

Evaluation

The above algorithm for the classification problem was implemented and tested for the command prediction problem using both artificially generated and real operation data.

Artificial data were generated approximating user's behavior by a Markov model that comprises five different tasks. The model used is shown in Fig. 9. About 2000 different sequences were generated, in which commands that were not in the model *(e.g., ls , df,* etc.) were added as noise. Three fold cross validation was used to evaluate the prediction accuracy. The results are shown in Table 2. This table includes the results obtained by other methods for comparison.

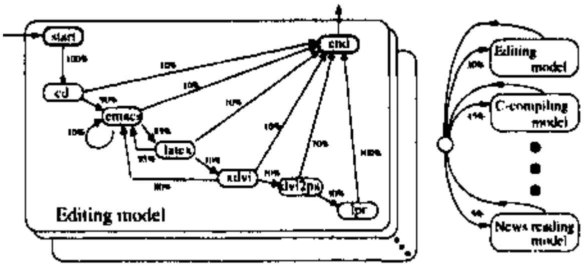


Figure 9: Markov model used to generate artificial data

Table 2: Prediction accuracy for artificial data

Noise	Induction Methods				
	Def.	*LD*	*CART*	GBI_1	GBI_2
15%	35.5	35.5	48.6	51.5	73.6
20%	33.8	33.8	45.2	52.1	73.7
25%	33.0	32.2	41.6	47.2	72.1

Def.: Default value for most frequently used command
LD: Linear Discrimination Method
GBI\: Without dependency info, for the root node (command to prediCt)
GBI_2. With dependency info, for the root node

There are two cases for *GBI.* GBI_1 is the case where dependency information is used only for the commands (nodes) preceding the root node. In other words, no dependency information is used for the root node. This reflects the fact that the argument is not known in advance to predict the next command. GBI_2 is the case where the dependency information for the root node (command to predict) is also used. This corresponds to a case where the file to process is specified, and this is exactly what the current *ClipBoard Interface* does[4]. In [Yoshida and Motoda, 1996] the former is called command prediction and the latter, application selection.

Default is the simplest way of prediction that always assumes the most frequently used command to be the next command. *LD* is a linear discrimination method [James, 1984], which gave the same answer as the default and did not improve the accuracy. The best result by the conventional method was achieved by the decision-tree learning method *CART* [Breiman *et* al., 1984]. *LD* and *CART* use only sequential information because these methods cannot deal with information

pendency, in which case the chunking process becomes more complicated.

[4] This is not a strong restriction because files associated with a given task are generally known and the prediction of the command for each of these files can be made with this method.

having a graph structure. From these results, it is clear that the I/O dependency information (in particular, the one immediately before the command to predict) plays an important role in increasing the accuracy of prediction.

The same algorithm was tested against the real data that had been taken from the log of daily usage over three months of a single user. The data include about 2000 kinds of commands. Two-thirds of them was used as a training data set and the rest as a test data set. The result is shown in Table 3. It is clear that *GDI* outperforms the other methods. Interestingly GBI_1 is much better than *CART* in real data. This is probably because the number of commands actually used is much larger than the artificial data case and the noise level is also higher. Unfortunately the value for GBI_2 is not-available for the same data set. It is instead estimated by the daily usage when the performance approached the steady state. Once again, the role of I/O dependency is clear.

Table 3: Prediction accuracy for real data

Methods	*Def*	*LD*	*CART*	GBI_1	GBI_2
Accuracy %	22.6	22.6	34.6	57.8	~80.0

The non-essential commands such as *ls* and *df* can be naturally ignored by a mouse-based interface system. If we ignore these effects and focus on the important commands, we obtain the results shown in Table 4, which is by far better. While evaluation of *ClipBoard* is still ongoing, most of the important commands predicted by *ClipBoard* is quite adequate, and the user does not feel any burden in using it.

Table 4: Prediction accuracy of selected commands (GBI_1)

Command	*emacs*	*make*	*latex*	*backup*	*xdvi*
Accuracy %	69	85	92	86	100

4.2 Script Generation

I/O Information Analysis

To be precise, the I/O recorder keeps track of 1) all process creations in the operating system, and 2) all I/O operations (*open* system calls). Thus, even in a multi-window and/or a multi-task environments, it is possible to extract relationships between commands that may have been issued across the different shells. We use the whole graph to extract patterns. The extracted patterns are frequently appearing ones in the history, and we convert them to shell scripts. The input file name is changed to the argument of the script with extensions retained (See Fig. 2).

Evaluation

Table 5 lists the scripts generated from the sample history, which involves about 10,000 process creations and about 130,000 I/O operations. The number of processes includes system programs that were not invoked by the user (*e.g.,* telnet daemon, line printer spooler daemon, etc.), some user commands (*e.g.,* shell scripts), and created child processes. The number of the actual commands invoked by the user was approximately 2000.

Table 5: Generated scripts with more than three commands

Scripts	Scripts	Scripts

1. `emacs $1`
`diff $1`
`$1.bak`
`cp $1 $1.bak`
	2. `emacs $1`
	`diff $1`
	`$1.bak`
	`cp $1 $1.bak`
	`make`
		3. `cp $1 $1.bak`
		`chmod 500 $1`
		`rm $1.bak`

4. `emacs $1.c`
`cc $1.c`
`a.out`
	5. `emacs $1.tex`
	`latex $1.tex`
	`xdvi $1.dvi`
		6. `emacs $1.c`
		`cc $1.c`
		`strip a.out`

Since the algorithm only considers the frequency or its equivalent as measured by the index, evaluation of the usefulness or importance of the generated scripts must be rendered to the user. Unlike the case for command prediction, there is no direct feedback from the user. The scripts in Table 5 have clear meanings except script 3. Without having knowledge about the C compiler, *ClipBoard* could generate scripts 4 and 6. *ClipBoard* did not use any pre-specified knowledge about *latex* and related commands in generating script 5. Script 1 is a unique script for this particular user. Without *ClipBoard* the user has to write this by him or herself.

5 Prefetch Daemon

I/O Information Analysis

In a multi-task environment different users work on the same machine for different tasks (*e.g.,* editing and programming). Even though the I/O operation sequence of each task has regularity, the overall I/O sequence is affected by the subtle timing of each task progress. The graph structure can encode the correct information even in a multi-task environment. Just like in the case of script generation, GBI analyzes the process data and represents them by a set of directed graphs, from which it extracts typical patterns. Each of the patterns represents an aspect of the user (we call it user model for convenience). Figure 10 shows how these patterns are used to prefetch files. First, each of the patterns is converted into a prefetch rule. Unlike the command predictions, the point here is not to predict the root node from the rest, but to predict from the bottom (first) node in the sequence how certain files are going to be used along the subsequent command execution. Each rule consists of a sequence of events, *i.e.,* command executions and I/O operations, with a list of files to be prefetched. Next, each rule is merged into a single trie structure. The system checks the common event sequences in the rules and merges the same parts into a single structure. For example, in Fig. 10, the first nodes of the two user models, A and B, are the same and are thus merged. In order to improve the prefetch accuracy, the frequency information in the log is used to prune files[5]. The generation of trie structure is performed as a batch process.

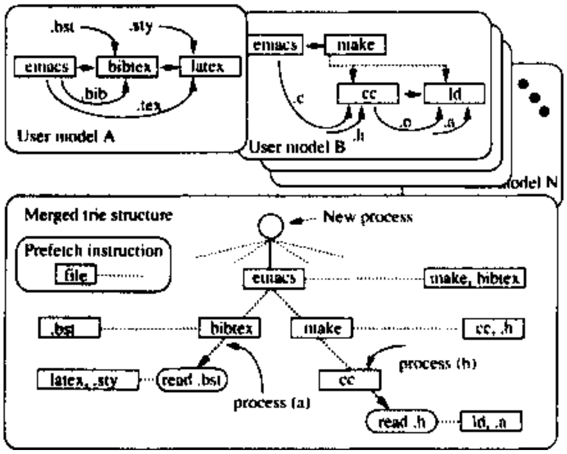


Figure 10: User models and a merged trie structure for prefetching

Evaluation

After the batch process constructs the trie structure, the prefetch daemon uses this trie structure to prefetch files. The daemon maintains the status information for each process. If a new process is activated, the prefetch *daemon* creates a new pointer which points the root node of the trie structure. If the process executes command *emacs (i.e.,* the program memorized in the succeeding trie node), the daemon prefetches program files *make* and *bibtex* and updates the pointer. In Fig. 10 *process (a)* shows the position of the pointer after it executed *emacs* and then *bibtex.* Each time it updates the pointer, it also looks for the same command from the root *(i.e.,* the command just below the root node) as if a new process with this command was initiated. When it finds the command, it also prefetches the associated files. This is recursive. If the actual events of the process exhibit a different sequence from the trie, all the pointers for this process are removed and the prefetch daemon ignores the process until a new process is initiated.

The above prefetch mechanism was tested for the daily usage data (the length of the log was about 38,000). The prefetch cache size was automatically adjusted by OS. The trie had approximately 1000 nodes. Although further experiments are necessary, the preliminary experiments show that the trie structure has high prediction accuracy. For the experiment we conducted, the hit rate was almost 100%.

Unfortunately, even with the high hit rate, the current implementation slows down the CPU intensive tasks due to the CPU resources used by the prefetch daemon. We could only speed up I/O intensive tasks. It could indeed speed up the invocation of a large program such as *X-windows* and *mule* to the extent that we did not feel we had waited. The process switching overhead and the JAVA byte code interpretation are the sources of the problem. A kernel embedded file prefetcher that is coded by C and assembler would solve the problem.

6 Discussion

6.1 Learning Semantics from Syntax

Although what OBI does is simply extracting the syntactic/statistical nature of what a user has done in the past, it is still possible to extract useful semantics of the user's behavior. The user never tells the start of his/her task to *ClipBoard,* but the scripts generated by GDI does capture a piece of meaningful tasks. Most crucial is the information source. The surface form of the user's input (i.e., command sequence) was not enough. Other information that is hidden and invisible *(i.e.,* process I/O) contributed much. Standard techniques *(e.g.,* index based on information theory, cross validation, etc.) that statisticians have developed are also important factors.

6.2 Information to Capture User Behavior

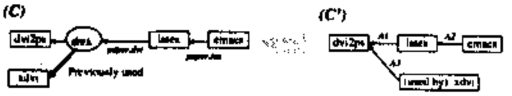
[Piernot, 1993] addresses the importance of the *context* in an interface system. File extensions we used in our analysis to capture the I/O information helped provide rich context. Other information that may help capture the user's behavior is command exit status and time of execution. For example, if the user fails to compile a program because of a simple syntactic error, the next step tends to be an editing task. If s/he succeeds, it tends to be a test run. Thus, the exit status seems to be informative. Since most users tend to check e-mail in the morning, the time of day also seems to be informative. Experiments using *ClipBoard* utilizing such information are currently under investigation.

The method of encoding information is also important. We encoded the I/O information from *how a file was* made *by application program.* The experimental results suggest the adequacy of this encoding, but this is not the only way to use the I/O information. For example, *how a file was* used *by application program* is another way of encoding. Figure 11 shows a graph format that was designed to emphasize this aspect. We confirmed the usefulness of this encoding with a version of *ClipBoard* that uses this as an alternative to the sequence information. Note that this encoding has a noise-tolerant nature. User errors, such as mistyping and wrong command selection, and unexpected interrupts, such as new mail arrival, sometimes cause noise in sequence information. The replaced I/O information is less affected by such noise.

6.3 Method of Analyzing User Behavior

If the user is always logical and consistent, the analytical methods, such as explanation-based learning, are ad-

[5] There are many patterns that partially overlap and/or are subpatterns of the others. A threshold can be set to the number of occurences of the files for them to be prefetched.



(C) is the directed graph of the history information at Step (C) in Figure 1. The shaded part indicates that *paper, dvi* was previously used by *xdvi.* (C)' is the reconstructed graph. The *dvi* node is removed for brevity and the *(used by) xdvi* node replaces the sequence information.

Figure 11: Graph encoding the knowledge of *how* a *file was* used.

equate in making the user behavior model. Unfortunately, the user is sometimes illogical and inconsistent, and capriciousness makes it difficult to apply analytical methods to the interface problem. The statistical methods, such as linear discrimination and k-nearest-neighbor [Jlames, 1984], and empirical learning methods, such as [Quinlan, 1986], seem to be more adequate. The errors, *i.e.,* mistyping and wrong command selection, are naturally ignored as noises in these methods. However, these methods are not suited to handle structural data as was the case for this study.

If we set the maximum width (number of input files) per command and the maximum depth (number of chains of I/O relationship), it is possible to design a table of attributes and values that can record all the necessary information. If we take the maximum width as 50 and the maximum depth as 5, a table with $\simeq 50^5$ attributes is created[6]. This is only for one instance. If the analysis requires 1000 cases, the table size becomes huge.

Inductive logic programming (ILP) [Quinlan, 1990; Muggleton and Feng, 1992; Pazzani and Kibler, 1992], on the other hand, is more expressive and captures the relations most naturally in first-order logic. It can also handle noise [Quinlan, 1990; Pazzani and Kibler, 1992]. To explore the potential of this approach, we tried to use FOCL, one of the most efficient ILP systems, to analyze the real data used in Section 4.1. However, FOCL took more than 4 hours to find the first test condition of the first rule; therefore we had to give up this approach[7].

GBI's expressiveness lies in between the attribute-value pairs and the first-order logic. It is a limited form of propositional calculus. Its learning potential is much weaker than that of ILP, but stronger than that of the attribute-value representations and yet as efficient. We demonstrated that command prediction we addressed in this paper is a class of the problem that GBI's framework fits well.

6.4 Other Applications

The idea of *ClipBoard* seems to be useful in designing interface systems of other kinds such as automatic chart format selection in spread sheet and data base, naive-user guidance and installation guidance-and-diagnosis systems. The last two are meant to apply the knowledge learned from expert behavior to non-expert users. During the development of *ClipBoard,* we were able to use the I/O information itself, *i.e.,* the raw history data, for debugging purposes. A good display system of this information seems to be beneficial even for an expert user.

We are aware of some minor things that could improve *ClipBoard's* ease of use. For example, we could improve *ClipBoard*'s selection function by highlighting the second suggestion shown in the dialog box (See Fig. G (b)) when the user wants to override *ClipBoard*"s first suggestion (which is displayed by icon).

One promising application that goes beyond those within a single machine is dynamic World Wide Web caching. The rapid growth of information gathering through WWW causes a heavy network overload, and the resulting slow response is causing a problem. Distributed caching is a promising approach. Our preliminary study [Yoshida, 1997] by *GBI* shows that it is possible to reduce the overload by extracting frequent occurring data transmission patterns from the wide area network flow and using this to allocate distribute cache storage. The simulation assumed the situation where 32,000 WWW servers are accessed simultaneously by 16 clients. Each client and router had a 32 MB cache capacity. The data were taken from the access log of our proxy server that included 2.3 million data transfers (18.7 GB in size). Figure 12 shows how the traffic changes with the time of day with and without cache, from which we observe 26% reduction of traffic between 10 am and 8 pm. The traffic reduction at the peak time amounts to 100 MB.

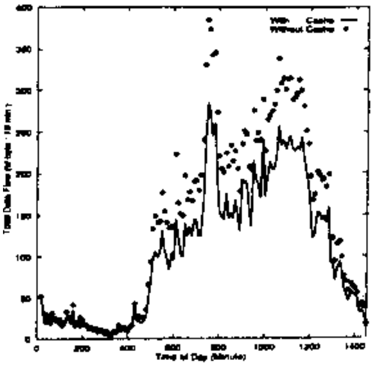


Figure 12: Network traffic distribution over the time of day with and without cache

7 Related Work

Intellectual assistance by computers has attracted many people, and various attempts have been undertaken with

[6] Note that a typical (not maximum) single run of the *latex* command receives 50 input files *(e.g.,* .tex, .aug, . sty. .bbl, .eps, .tfm, .fmt, etc).

[7] We have not taken advantage of the search strategy used in GBI.

different approaches and for different tasks. There are many terms that characterize these approaches such as learning apprentice, software agent, learning agent, interface agent, programming by example or demonstration, personal knowledge based system, etc. What is common to many of them is that they observe repetition or regularity in the user's behavior and use them for automation, prediction and customization in one way or another.

The amount of knowledge that has to be provided in advance varies among the approaches. General remarks are that making the user program everything requires too much insight, understanding and effort from the user, and having to encode a lot of domain-specific background knowledge about the task and the user also requires a huge amount of work from the knowledge engineer. Both have fixed competence, and are hard to customize to individual user differences or changes of habits. Some sort of automatic knowledge acquisition that can capture each user's habits is needed.

EAGER [Cypher, 1991] is an example of program by demonstration (PBD), which is a Hyper' Text system that keeps watching a user's actions, detects an iteration and offers to run the iterative procedure to completion by generalizing the repetitions and making macros. Myers 's demonstrational formatter [Myers, 1991] is also an example of PBD. It does not focus on the repetition, but generalizes a single example to create a template for later use, which enables the formatting of headers, itemized lists, tables, references, etc. Another example is Gold [Myers *et* a/., 1994] which is a business chart editor. It is given the knowledge of properties of the data and the typical graphics in business charts to generalize a single, or a very few examples, by interpreting them as a combination of primitives.

[Greenberg and Witten, 1988] analyzes repetitive patterns in the UNIX command histories and observes some regularities. [Masui and Nakayama, 1994] also uses the repetitive nature for a predictive user interface. When a user types a repeat key after doing repetitive operations, an editing sequence corresponding to one iteration is detected, defined as a macro, and executed at the same time. Although being simple, it covers a wide range which had to formerly be covered by keyboard macro.

All of the above approaches do not use machine learning techniques although they do guess and generalize. The Interface agent of [Maes and Kozierok, 1993] takes a machine learning approach. They address the problem of self-customizing software at a much more task independent level. The core is to learn by observing the user, i.e., by find reguralities in the user's behavior and using them for prediction. They also adapt two other learning modes: learning from user feedback and learning by being told. They used memory-based learning (k-nearest neighbor) which is good for explanation. Situations in the user are described in terms of a set of attributes which are hand-coded. The tasks that they applied are a calendar management agent and an electronic mail clerk.

The personal learning apprentice CAP [Dent *et* a/., 1992] is similar to the above. It is an interactive assistance that learns continually from the user to predict default values. Their application is a calendar management apprentice which learns preferences as a knowledgeable secretary might do. Two competing leaning methods are used: decision tree learning and backpropagation neural net. The attribute value representation suffices for this purpose. Another related system addresses the task of form-filling [Hermens and Schlimmer, 1993]. They use decision tree learning to predict default values for each field on the form by referring to values observed on other fields and the previous form copy.

[Schlimmer and Hermens, 1993]'s pen-based interactive note taking system is a self-customizing software to eliminate the need for user customization. It starts with partially-specified software and applies a machine learning technique to complete any remaining customization. The system learns a finite state machine to characterize the syntax of user's notes and learns decision tree to generate predictions. Letizia [Lieberman, 1995] is an interface agent that assists a user browsing the WWW. It tracks user behavior and attempts to anticipate items of interest by doing concurrent, autonomous exploration of links from the user's current positions. Intelligent agent for information browsing is a hot area and many systems are being pursued *(e.g.,* [Etzioni94 and Weld, 1994; Perkowits and Etzioni, 1995].

The research on prefetching is carried out by a separate community. The standard Least Recently Used (LRU) based caching offers some assistance, but ignoring any relationships that exist between file system events fails to make full use of available information. The closest work that uses the relationship would be [Kroeger and Long, 1994]. They use trie structure to memorize previous I/O sequence but no explicit learning is performed. Their results indicate that the predictive caching gains on the average 15% more cache hits than the LRU based caching. However, since they are using only sequential information, their method does not work well in a multi-task environment.

All of the applications that use machine learning techniques do not require relational representations. The data are represented by a set of features. Analysis of sequential information is enough for the selected applications. Some require additional task specific knowledge. We showed in this paper that there are other applications that this success cannot be easily generalized, and proposed the GBI as a general induction mechanism for this type of applications.

8 Conclusion

We have modeled a user adaptive interface that can predict next command, generate scripts and prefetch files in a multi-task environment. The analysis of behavioral data indicated that the directly observable sequential records are not enough to capture the behavior, and that simultaneous use of process I/O information that is hidden from the user is beneficial. An efficient induction algorithm that can handle relational data was needed and a technique called graph-based induction was applied. It can find frequently occurring patterns from a graph representation. It also induces classification rules from structured data that have intra-relationship. Pairwise chunking, which is the heart of the algorithm, does not guarantee an optimal solution by any means, but empirical study shows that use of statistical measure results in a good solution. It is efficient and can run in real time. The command prediction module is in daily use. Shell script generation works as expected but is less used. Prefetching daemon still needs a better implementation to enjoy the real benefit .

Acknowledgments

Much of the work was conducted while the first author was at ARL, Hitachi, Ltd. Authors are grateful to Shojiro Asai and Katsumi Miyauchi of ARL for their generous support of this work. They extend their thanks to Takashi Washio, Tadashi Horiuchi and Toshihiro Kayama of Osaka University for the discussions.

References

[Breiman *et ai,* 1984] L. J. Breiman, H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth &: Brooks/Cole Advanced Books & Software, 1984.

[Cypher, 1991] A. Cypher. Eager: Programming Repetitive Tasks by Example. In *Proc. of CHI'91,* pages 33 39, 1991.

[Dent *et al,* 1992] L. Dent, J. Boticario, J. McDermott, T. Mitchell, and D. Zabowski. A Personal Learning Apprentice. In *Proc. of A AAI' 92,* pages 96-101, 1992.

[Etzioni94 and Weld, 1994] O. Etzioni94 and D. Weld. A Softbot-Based Interface to the Internet. *Commun. ACM,* 37(7):72- 76, 1994.

[Greenberg and Witten, 1988] S. Greenberg and I. H. Witten. How Users R.epeat Their Actions on Computers: Principles for Design of HISTORY Mechanisms. In *Proc. of CHI'88,* pages 171-178, 1988.

[Hermens and Schlimmer, 1993] L. A. Hermens and J.C. Schlimmer. A Machine-learning Apprentice for the Completion of Repetitive Forms. In *Proc. of the Ninth Conf. on Artificial Intelligence for Applications,* pages 164-170, 1993.

[James, 1984] M. James. *Classification Algorithms.* A Wiley-Interscience Publication, 1984.

[Kroeger and Long, 1994] T. M. Kroeger and D. D. E. Long. Predicting File System Actions from Prior Events. In *Proc. of CHI'94,* pages 319 328, 1994.

[Lieberman, 1995] H. Lieberman. Letizia: An Agent That Assists Web Browsing. In *Proc. of IJCAI'95,* pages 924-929, 1995.

[Maes and Kozierok, 1993] P. Maes and R. Kozierok. Learning Interface Agents. In *Proc. of AAAV98,* pages 459-465, 1993.

[Masui and Nakayama, 1994] T. Masui and K. Nakayama. Repeat and Predict - Two Keys to Efficient Text Editing. In *Proc. of CHI'94,* pages 118 123, 1994.

[Muggleton and Feng, 1992] S. Muggleton and C. Feng. Efficient Induction of Logic Programs. In S. Muggleton, editor, *Inductive Logic Programming,* pages 281 298. Academic Press, 1992.

[Myers *et al,* 1994] B.A. Myers, J. Goldstein, and M. A. Goldberg. Creating Charts by Demonstration. In *Proc. of CHI'94,* pages 106 111, 1994.

[Myers, 199l] B.A. Myers. Text Formatting by Demonstration. In *Proc. of CHI'91,* pages 251256, 1991.

[Pazzani and Kibler, 1992] M. Pazzani and D. Kibler. The Utility of Knowledge in Inductive Learning. *Machine Learning,* 9(l):57-94, 1992.

[Perkowits and Etzioni, 1995] M. Perkowits and O. Etzioni. Category Translation: Learning to Understand Information on the Internet. In *Proc. of IJCAI'95,* pages 930 936, 1995.

[Piernot, 1993] P. P. Piernot. The AIDE Project: An Application-Independent Demonstrational Environment. In A. Cypher, editor, *Watch What I do: Programming By Demonstration,* pages 387-405. MIT Press, 1993/

[Quinlan, 1986] J. R. Quinlan. Induction of Decision Trees. *Machine Learning,* 1:81-106, 1986.

[Quinlan, 1990] J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning,* 5(3):239-266, 1990.

[Quinlan, 1993] J. R. Quinlan. *C4-5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[Schlimmer and Hermens, 1993] J.C. Schlimmer and L. A. Hermens. Software agents: Completing Patterns and Constructing User Interfaces. *Artificial Intelligence Research,* 1:61-89, 1993.

[Yoshida and Motoda, 1995] K. Yoshida and H. Motoda. Clip: Concept Learning from Inference Pattern. *J. of Artificial Intelligence,* 75(l):63-92, 1995.

[Yoshida and Motoda, 1996] K. Yoshida and H. Motoda. Automated User Modeling for Intelligent Interface. *Int. J. of Human Computer Interaction,* 8(3):237-258, 1996.

[Yoshida *et ai,* 1994] K. Yoshida, H. Motoda, and N. Indurkhya. Graph-based Induction as a Unified Learning Framework. *J. of Applied Intelligence,* 4:297 328, 1994.

[Yoshida, 1997] K. Yoshida. WWW Cache Layout to Ease Network Overload. In *Proc. of Sixth International Workshop on Artificial Intelligence and Statistics, AISTATS97,* pages 537-548, 1997.