The Trials and Tribulations of Building an Adaptive User Interface

Benjamin Korvemaker & Russell Greiner
{benjamin, greiner}@cs.ualberta.ca
Department of Computer Science
University of Alberta
Edmonton, Canada

Abstract

As every user has his own ideosyncracies and preferences, an interface that is honed for one user may be problematic for another. To accomodate a diverse range of users, many computer applications therefore include an interface that can be *customized* — *e.g.*, by adjusting parameters, or defining macros. This allows each user to have his "own" version of the interface, honed to his specific preferences. However, most such interfaces require the user to perform this customization by hand — a tedious process that requires the user to be aware of his personal preferences. We are therefore exploring *adaptive* interfaces, that can autonomously determine the user's preference, and adjust the interface appropriately.

This paper reports a series of experiments towards building such an adaptive interface — here a UNIX-shell that can predict the user's next command based on his previous interactions, and use this to simplify the user's future interactions. After summarizing the Davison/Hirsh (1998) work (for learning "command stubs"), we then explore several ways of extending and improving this system; *e.g.*, to predict entire command lines, to use various other types of information, etc.

Keywords: online learning, learning and adaptation, learning user preferences

1 Introduction

There are today a wide variety of interactive computer applications, ranging from web-browser and searchers, through spreadsheets and data-base management systems, to editors, as well as games. As these systems become more complicated — as required to be able to accomplish more tasks, better — their interfaces necessarily also become more complex. Many of these systems have begun including tricks to help the users; *e.g.*, if the user begins an empty file with "Dear John", WORD will suggest a "Letter" template; similarly if the user begins a line with "*", WORD will change that character to a bullet "•" and go into its `List` environment.

Unfortunately, different users have different preferences, which means the tricks that are appropriate for one user may be problematic for another. (*E.g.*, there apparently are people who like the Microsoft "Office Assistant"...) Moreover, different users want to do different things with the system, as they have very different abilities, background knowledge, styles, etc. This realization — that "one size does NOT fit all" — argues for *customizable* interfaces, that can provide different interfaces for different users, and hence allow each user to have an interface that is honed to his individual preferences.

Of course, many of today's application programs can be customized; *e.g.*, most editors and shells include macro- or scripting- facilities. However, this customization process must typically be done *by the user* — which means, typically, that it is *not* done by the user, as this customization process (1) requires that the user *knows how* to make this modification (*e.g.*, knows both the names of the relevant parameters, and how to modify them); (2) requires the user to be *aware* of his specific preferences, and (3) is usually quite *tedious*.

This research project, therefore, pushes on a different approach: Build application systems that can *autonomously* adapt themselves to the individual users. In particular, we focus on techniques for detecting patterns in the user's interactions, and then using this information to make the interaction simpler for the user, perhaps by automatically re-setting some system parameters, or defining appropriate new macros.

This paper investigates a specific manisfestation of this task: We have built a UNIX command shell that can predict the user's behavior from his previous commands, and then use these predictions to simplify his future interactions with the shell. The rest of this introductory section presents two illustrative examples to help describe our task more precisely. After Section 2 provides the background for this work, Section 3 then sketches our algorithm, based on the earlier Davison/Hirsh system (DH98). Finally, Section 4 presents our empirical results, over a large existing dataset of 186 users. It discusses in particular the range of studies we ran, to better understand this challenge. All told, this paper presents what has been tried before, summa-

```
...
% vi crossword.c
% make crossword
cc -g crossword.c -o crossword -lm
% crossword puzzle01
bus error (core dumped)
% vi crossword.c
% make crossword
cc -g crossword.c -o crossword -lm
% crossword puzzle01
Solution found in 15584 attempts.
% crossword puzzle02
segmentation violation (core dumped)
% ddd crossword core
% vi crossword.c
% make crossword
cc -g crossword.c -o crossword -lm
Undefined symbol:  print
first referenced in file crossword.o
ld: fatal: Symbol referencing errors.
No output written to crossword
*** Error code 1
make: Fatal error: Command failed for target
       'crossword'
% vi crossword.c
% make
cc -g crossword.c -o crossword -lm
% crossword puzzle03
Solution found in 13 attempts.
% elm -s"it works" fred
...
```

Figure 1: Wilma's Command Sequences

rizes what we have learned, and suggests some of the problems.

Examples: Figure 1 shows Wilma's interactions with a shell, as she works on her crossword-solving program. It is easy to see that there is a consistent pattern within Wilma's activities; *e.g.*, the `vi-make-crossword` sequence is repeated several times. Although Wilma has taken advantage of traditional UNIX facilities, she still has to type a number of characters.

Now examine Fred's command sequences; Figure 2. He has already completed the assignment and is trying to write up his results. Unfortunately, Fred is not very familiar with LaTeX and is having trouble formatting an equation.

Here, an even more obvious pattern is visible. Note that, if Fred had had a script to perform the `latex-dvips-ghostview` steps for him, he wouldn't have made the mistake of running `dvips` on a LaTeX source file.

Task: Our immediate goal is a UNIX shell that can anticipate the user's next command, and use this information to simplify his interactions. One way to use that information would be to fill the user's current buffer with this predicted command — *e.g.*, after "`vi crossword.c`", this shell could load "`make crossword`" into Wilma's buffer. Wilma could then execute this command by simply typing a carriage return. Alternatively, she could delete this buffer, or a portion thereof, and retype whatever else she wishes. Similarly, after Fred's "`latex crossword.tex`", the shell could suggest "`dvips crossword.dvi`"; assuming Fred accepted this suggestion, this could save Fred the hassle of figuring out why "`dvips crossword.tex`" did not work.

```
...
% vi crossword.tex
% latex crossword.tex
% dvips crossword.dvi
% ghostview crossword.ps
% vi crossword.tex
% latex crossword.tex
% dvips crossword.tex
% ghostview crossword.ps
% vi crossword.tex
% latex crossword.tex
% dvips crossword.tex
dvips: ! Bad DVI file: first byte not preamble
% dvips crossword.dvi
% ghostview crossword.ps
% wall
Can anyone help me with latex???
^D
...
```

Figure 2: Fred's Command Sequences

However, as this single most-likely command is still very unlikely, we decided to use the slightly different approach of suggesting the 5 most likely commands. As shown in Figure 3, our shell will display (in the top of the screen) these 5 most likely commands. It also binds them to the $F1$ through $F5$ keys; the user can execute any of these commans by simply pressing the associated function key. Notice either option means the user can achieve his results with less typing, and so do his work in less time, while making fewer mistakes.

In a nutshell, our shell will use this record of previous commands to predict what command the user is likely to use next. These examples illustrate why our approach has a chance of succeeding: People tend to follow patterns. For example, after a successful `latex` command, many users then type `dvips` to produce a postscript file from the `dvi` file that `latex` generated; and if that succeeds, use `ghostview` to display the generated paper. [1]

Later versions of this type of shell could take more sophisticated actions based on this information. For example, they could define (and inform the user of)

[1] Note that even commands that seem random, such as "`df`", "`uptime`" or "`readmail`", are probably (probabilistically) predictable based on some external event. Of course, this may require elaborate instrumentation.

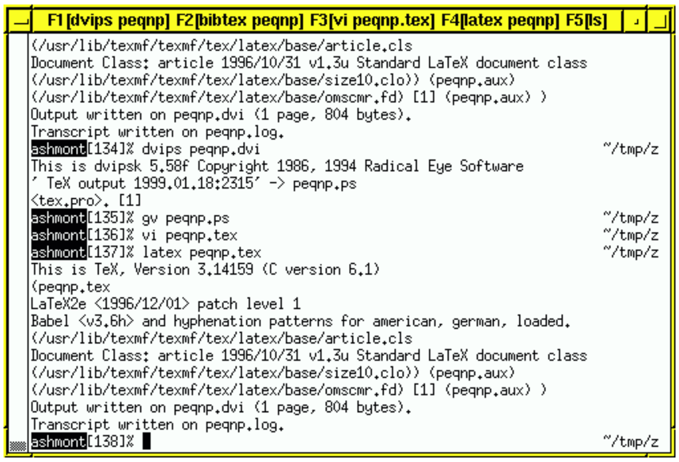

Figure 3: Display of the Implementation

new macros, corresponding to (variabilized forms of the) sequences observed most often. Or the adaptive shell could pre-fetch information for the next anticipated command(s) — *e.g.*, get the fonts for the possible "`dvips`" command, swap out programs to preemptively make room for `netscape`, swap out `netscape` before starting some color-intensive task, or format the upcoming `man` page. They might also be able to detect patterns that correspond to problems — *e.g.*, that the user is thrashing — and then provide useful assistance.

2 Background

The extended version of this paper provides a comprehensive survey of adaptive user interfaces (*e.g.*, (Lan97; SH93; HBH$^+$98) and others) and other work related to our task. Here, for space reasons, we can only discuss the two most relevant previous results.

Our project is a direct extension of the seminal work by Davison and Hirsh, (DH97; DH98) which predicts user command *stubs* (commands without options and parameters, *e.g.*, the "`latex`" of the command "`latex foo.tex`") from the user's previous command stubs. Their hand-crafted algorithm generates and uses a table whose $\langle i, j\rangle$ entry is the probability of command stub s_i occurring immediately after the stub s_j; *i.e.*, $P(\,\mathrm{Stub}_{t+1} = s_i \,|\, \mathrm{Stub}_t = s_j\,)$ where the random variable Stub_i denotes the stub typed as the user's i^{th} command. After seeing the current stub "stub_t", their performance system could then predict the most likely subsequent stub

$$\mathrm{stub}^*_{t+1} \;=\; \operatorname*{argmax}_s \{P(\,\mathrm{Stub}_{t+1} = s \,|\, \mathrm{Stub}_t = \mathrm{stub}_t\,)\} \tag{1}$$

The main challenge, then, is how to fill this $N \times N$ table of numbers (where there are N possible stubs). The "obvious" approach, of using empirical frequency over the samples observed, is arguably appropriate only if the data is iid (independent and identically distributed), which means in particular that the frequency of seeing some command does not change over time. That assumption is clearly false in our situation.

Instead, their algorithm used a different way to estimate these probability values, one designed to emphasize recent new commands. On seeing say the command "`cd ...`" followed by "`latex ...`" it would first decay the probabilities on the "`cd` line" of the table by an (empirically determined) factor $\alpha < 1$ — *i.e.*, it would first reset, for each stub `s`,

$$\hat{P}(\,Stub_{t+1} = \mathtt{s} \,|\, Stub_t = \mathtt{cd}\,) \quad *= \quad \alpha \tag{2}$$

Note here that

$$\sum_s \hat{P}(\,Stub_{t+1} = \mathtt{s} \,|\, Stub_t = \mathtt{cd}\,) \;=\; \alpha$$

as this summation had been 1 before this reduction. They then add the remaining $1-\alpha$ quantity to the $\langle \mathrm{latex}, \mathrm{cd}\rangle$ entry:

$$\hat{P}(\,Stub_{t+1} = \mathtt{latex} \,|\, Stub_t = \mathtt{cd}\,) \quad += \quad 1-\alpha \tag{3}$$

Hence, this row continues to sum to 1, which is why we can view these values as probabilities.) This rule tremendously boosts the value of $\hat{P}(\,Stub_{t+1} = \mathtt{latex} \,|\, Stub_t = \mathtt{cd}\,)$, even if its previous value was 0. This effect is extremely useful; *e.g.*, Davison/Hirsh note that 20% of commands are simply the repeating the immediately prior command!) We will later refer to this algorithm as the "Alpha-Updating Rule" or "AUR".

Davison and Hirsh could then predict the single most-likely command, stub^*_{t+1} (from Equation 1); however this was correct only 39.9% of the time. They therefore switched to predicting the *five* most likely commands, and found that the actual stub (*i.e.*, the one the user actually entered) was in this top-5 list almost 3/4 of the time.[2]

Evaluation Criterion: Our basic algorithm is a direct extension of theirs; see Section 3. We also adopted their evaluation criterion: Our systems are **on-line**: after observing the sequence of stubs, $\langle \mathrm{stub}_1, \ldots, \mathrm{stub}_t\rangle$, each then predicts stub^*_{t+1} (Equation 1), or perhaps the top five: $\{\mathrm{stub}^{*1}_{t+1}, \ldots \mathrm{stub}^{*5}_{t+1}\}$. It is then told the correct stub_{t+1} that the user actually typed, which it can use first to update its "classifier", and then to predict stub^*_{t+2}, (or perhaps $\{\mathrm{stub}^{*1}_{t+2}, \ldots \mathrm{stub}^{*5}_{t+2}\}$), etc. We define "*accuracy*" as how frequently the predicted command completely matches the user's actual command. That is, after K instances, we compare the correct sequence $\langle \mathrm{stub}_1, \ldots, \mathrm{stub}_K\rangle$ to our prediction $\langle \mathrm{stub}^*_1, \ldots, \mathrm{stub}^*_K\rangle$, and report as accuracy the number of times "$\mathrm{stub}^*_i \;=\; \mathrm{stub}_i$", divided by K. In

[2] They limited n to 5 as more than five commands typically causes the user to focus on the predictions rather than the task at hand (or ignoring the predictions altogether).

the "top 5" case, the system gets a "point" whenver $\text{stub}_i \in \{\text{stub}_i^{*1}, \ldots \text{stub}_i^{*5}\}$. We use the same criterion, *mutatis mutandis*, for our studies.

Acknowledging that the prediction system may take a while to "lock onto" the particular person, we sometimes ignore the mistakes made during the first, say, 100 commands. Here, the accuracy would be the number of times "$\text{stub}_i^* = \text{stub}_i$" *over time i=101..K*, divided by $K - 100$. (We will later refer to this as "accuracy−100".)

Other people have done work in the area as well. In particular, Greenberg (Gre88) collected a large dataset for 168 users, and used this to classify users into novice users, experienced users, scientists and non-users (see Table 2 below). Although these datasets were collected 10 years ago, they still are the largest and most-complete source of user command histories that are publicly available. We use this dataset (albeit for different purposes) in the studies reported here.[3]

We finally note that, while the concept of command prediction has been explored for at least ten years, nothing has yet been deployed into the mainstream market. We believe this can be partially explained by observing:

- Computing resources may not have been sufficient to adequately predict user commands, until recently.
- It is difficult to collect sample data. UNIX users are a territorial bunch, with their favorite editors, OS variants, and shells.[4]
- It is typically very difficult to craft data collection mechanisms that are transparent to the user. While the ZSH source code is relatively organized and legible (especially when compared to TCSH source code), it still took 30 hours to find and change the appropriate 5 lines of code.

3 Prediction Algorithm

As stated above, our adaptive shell monitors the commands typed by the user (along with other associated information, see below) then uses this information to predict, after each command, which command the user will type next. To reduce the invasiveness of the data collection, we modified the UNIX shell, ZSH,[5] both to collect the relevant information, and then to reset the display and the function keys, as shown in Figure 3.

The real challenge, of course, is the actual prediction algorithm, which determines, for each possible command *cmd*, the conditional probability that *cmd* will be the next command, based on the earlier commands, etc.

In particular, we need to compute the probability that the next command will be cmd_{t+1}, based on the available information — *i.e.*, $P(\,\text{Cmd}_{t+1} = cmd_{t+1} \,|\, \texttt{cmd}_t, \ldots)$. Moreover, we need to produce a good estimate to this distribution after very few samples, which is further complicated by the realization that the "local distribution" (*i.e.*, the probability that one command will immediately follow another) is not stationary. As we want a prediction system that can track the user's distribution of commands, we therefore lift the Davison and Hirsh idea of emphasizing the immediately prior command, then decaying the probabilities over time; see the AUR (Section 2, Equations 2 and 3).

However, we extended their work in two significant ways. First, we want to predict entire *command lines*, rather than just command stubs. Although predicting the next command stub might save a few keystrokes, UNIX stubs are notoriously short — in fact, over our dataset, the average stub length is only 4.2 characters. Assuming the predictions are mapped to a function key and that the ENTER key must be pressed afterwards, reducing the number of keystrokes from four to two is not a significant savings. Further, when one considers that the function keys are usually not particularly close to the "home keys", the effort spent repositioning the hand may well negate the time saved. Note, however, the average length of the entire command line is 9.7 characters; and reducing 9.7 to 2 *would* be useful. We therefore focus on this task, even though it is, of course, much harder, especially for commands that happen frequently, but with highly varying arguments — *e.g.*, `cd`, `ls`, `vi` and `finger`.

The second extension was to help us solve this harder task: We want to allow our predictor to use other information in producing these predictions — such as the entire previous command line, the error code of that previous command, the time of day and day of week, etc.

DataStructure to Encode Probabilities: We could imagine using a huge multi-dimensional table for this, whose $\langle i, j, k, \ell, m\rangle$ entry is the conditional probability that $P(\,\text{Cmd}_{t+1} = i \,|\, \text{Cmd}_t = j,\ \text{Error} = k,\ \text{time} = \ell,\ \text{day} = m\,)$, where i and j vary over the possible command lines, k over the range of possible error codes, ℓ over the times in a day and m over the 7 days in the week. Unfortunately, our dataset included over 60,000 *different* commands lines. Even if we make this specific to the individual users, note there was some individual

[3] To simplify this presentation, we are not included the more recent, but less complete, data obtained from our implementation.

[4] One potential subject explained why he turned us down: "... switching shells is like cutting off my fingers and sawing (sic) them on backwards and I have to relearn how to use my thumb as my little finger ..."

[5] We chose ZSH as it claims to be able to emulate the more popular shells (TCSH, CSH, BASH, KSH), and it contains most of the mechanisms required to collect the parameters we want.

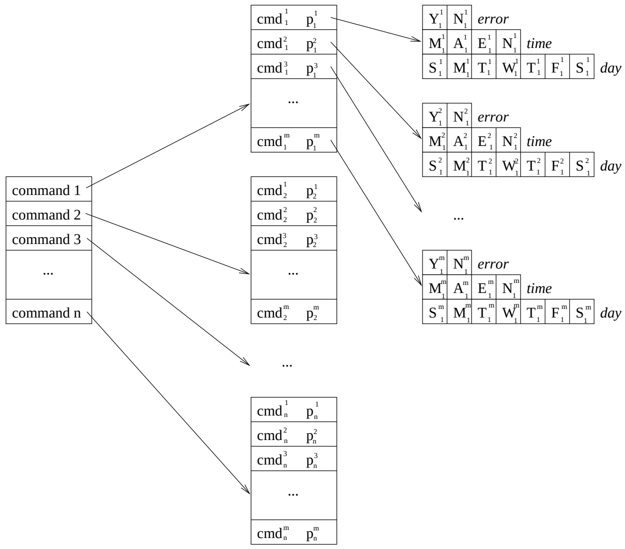

Figure 4: Structure Encoding Probabilities

users that used over 3000 different commands![6] Even if we quantize the error codes to only two values (0 or non-0), and the time to only hours, this table would need to have over $3000 \times 3000 \times 2 \times 24 \times 7 = 3,024,000,000$ entries — which would be difficult even to store, much less to estimate from a paucity of data (BD77). Recall that we want our system to be able to help users after observing only a small handful of interactions.

To reduce the number of parameters, we instead used a different structure, which imposes a bound on the number of possible current commands (set to n), and on the number of possible predicted commands (set to m). We also let the set of m predicted commands vary, depending on the current command. Here, we need only maintain the "active" subset of the command-pairs $\langle \mathrm{cmd}_i, \mathrm{cmd}_j \rangle$, based on the largest $P(\,\mathrm{Cmd}_{t+1} = \mathrm{cmd}_i \mid \mathrm{Cmd}_t = \mathrm{cmd}_j\,)$ values.

We also further quantized the time values, to only the 4 values: morning, afternoon, evening and night. The resulting structure is shown in Figure 4. To explain the notation: the value of $\mathrm{command}_7$ might be "`latex crossword`"; it points to an associated cmd_7^j table. There, perhaps cmd_7^1 is "`dvips crossword.dvi`" with $P_7^1 = 0.13$; and cmd_7^2 = "`vi crossword.tex`" with $P_7^2 = 0.09$; etc.

The value stored in

$$Y_7^1 \;=\; P(\,\mathrm{Cmd}_{t+1} = \texttt{dvips crossword.dvi} \mid \mathrm{Cmd}_t = \text{“}\texttt{latex crossword.tex}\text{”},\ \mathrm{ErrorCode} > 0\,)$$

[6] All told, the Greenberg data involved over 61,774 *distinct* commands, of the total of 303,628 commands typed by the 168 users. (Table 2 provides more details.) Some users used as few as 35 different commands, while one user used 3153. The corresponding values for stubs alone ranged from 7 to 358; there were a total of 6391 different stubs used.

is the probability that the next command is "`dvips crossword.dvi`", given that the previous command was $\mathrm{command}_7$ = "`latex crossword.tex`", and this execution produced an error.

In general, of course $Y_i^j = P(\,\mathrm{Cmd}_{t+1} = \mathrm{cmd}_i^j \mid \mathrm{Cmd}_t = \mathrm{command}_i,\ \mathrm{ErrorCode} > 0\,)$; note that $\sum_{j=1}^{m} Y_7^j = 1$. Similarly, $N_i^j = P(\,\mathrm{Cmd}_{t+1} = \mathrm{cmd}_i^j \mid \mathrm{Cmd}_t = \mathrm{command}_i,\ \mathrm{ErrorCode} = 0\,)$; $A_i^j = P(\,\mathrm{Cmd}_{t+1} = \mathrm{cmd}_i^j \mid \mathrm{Cmd}_t = \mathrm{command}_i,\ \mathrm{Time} = \mathrm{Afternoon}\,)$; $W_i^j = P(\,\mathrm{Cmd}_{t+1} = \mathrm{cmd}_i^j \mid \mathrm{Cmd}_t = \mathrm{command}_i,\ \mathrm{Day} = \mathrm{Wednesday}\,)$; etc.

In our experiments, we set $n = 2000$ and $m = 10$; this means our encoding involved at most $n \times m \times 2 \times 4 \times 7 = 1,120,000$ values, but less for users who used under 2000 commands. We show below that this is a robust prediction mechanism that is (relatively) resource friendly. The rest of this section discusses how we estimated these values; the next section, how we used these quantities to predict the most likely command(s).

Estimating the Probabilities: To produce this "table", we need first to define which commands to include as "current commands" (*i.e.*, the far left table in Figure 4), and as "predicted commands" (middle column), and second; to compute the actual conditional probabilities used to fill the values.

As explained above, as the distribution is not stationary, we should not simply use frequency estimates — *e.g.*, we should not estimate P_i^j as the number of times we observed $\mathrm{Cmd}_t = c_i$ followed by $\mathrm{Cmd}_{t+1} = c_j$ (over all ts), divided by the total number of times that $\mathrm{Cmd}_t = c_i$. We instead used the AUR, in several places: First, as we can keep only 2000 "current commands", we need to know which commands are still active. We therefore maintain probabilities for the various commands, and update them using this technique. We then keep only the 2000 command lines with highest values. This means we maintain commands used recently, and let the commands that are unused for an extended time, fall away.

It also means new commands have a chance of being included, which would not happen if we, instead set the probability to the empirical frequency. For example, imagine that the first time `latex` was used was in the 1000^{th} command. In the "probability $\approx$ empirical frequency" approach, the probability of `latex` would be only 1/1000. As this is probably the smallest value, it is likely the one that would be flushed! Note that this is problematic, as this means that the system will never consider *any* new commands — even if the next 100 commands are all `latex`!

We also use this AUR to set and update the conditional probabilitiy values P_i^j. For each current command $\mathrm{command}_i$, we then keep only the 10 predictions with the highest P_i^j scores. We also used this AUR to

	Accuracy			
	Cmd − 0	Cmd − 100	Stub − 0	Stub − 100
previous 1 line	7.5%	******	18.6%	******
previous 5 lines	38.8%	******	60.7%	******
most freq 5 lines	34.2%	33.9%	62.2%	62.3%
from command	**46.9%**	**47.4%**	**72.7%**	**73.3%**
from command + parsing	43.8%	44.0%	******	******
from command + error	**46.9%**	46.9%	72.6%	72.6%
from command + day	46.7%	46.7%	72.4%	72.5%
from command + time	46.6%	46.6%	72.4%	72.4%
from command + day + time	46.6%	46.6%	72.4%	72.4%
using last 2 commands	36.9%	36.9%	59.2%	59.5%

Legend:
Cmd Line ≡ "Complete Command Line"
Stub ≡ "Just command itself (no args, options)"
Accuracy ≡ "top five includes next command line/stub"

Table 1: Summary of results

set the values for the Y_i^j and other values. (Note, however, that the decision of which predicted commands cmd_i^j to keep depends only on the P_i^j values, and NOT the Y_i^j quantities, nor does it depend on W_i^j, etc.)

We need an α value for each of these AURs; in our experiments, we set

$\alpha_{cmd} = 0.90$ to decide which current commands to keep
$\alpha_P = 0.95$ to update the P_i^j's
$\alpha_{Error} = 0.91$ to update ErrorCode probabilities
$\alpha_{DoW} = 0.90$ to update DayOfWeek probabilities
$\alpha_{ToD} = 0.90$ to update TimeOfDay probabilities

which we emprically found gave the best results.

4 Experiments and Results

This section presents our empirical results, based on the data from the Greenberg (Gre88) dataset.

Exp#1 first attempts to duplicate the Davison and Hirsh results, dealing only with stubs, albeit on our datasets. The rest of this section considers several extensions: Exp#2 argues for predicting *complete command lines*, rather than just the stubs, and presents our results here. There are obvious ways to parse command lines, and then re-use that information — *e.g.*, after "`latex foo.tex`", we may expect the next command to be "`dvips foo.dvi -o`"; note the foo is repeated. Exp#3 states this notion more precisely, and then presents our results. So far, everything deals only with command lines or stubs. We would naturally expect that our system could make better predictions if it were given more information about the context of the current/previous command. We therefore explored using "error codes" (Exp#4) and "current day and/or current time (Exp#5). Another source of information is the command that was typed *before* the current command; Exp#6 discusses our attempts to estimate and use $P(\text{cmd}_{t+1} \mid \text{cmd}_t, \text{cmd}_{t-1})$. Finally, Exp#7 and Exp#8 consider predicting the next command *stub*, but here using (respectively) the previous command *line*, and then the command line plus the error code. The final Exp#9 discusses the predictability of different classes of users; it also provides additional information about the Greenberg dataset we are using.

We summarize our main results in Table 1. Before explaining the details of this table (see below), note immediately the winner, across the board, is the simplest approach, of simply using the current commnd in predicting the next command — *i.e.*, $\text{argmax}\, P(\text{cmd}_{t+1} \mid \text{cmd}_t)$, using the AUR to maintain the distribution. In particular, we did not improve (and often, did not even match) this accuracy when we also used error code, day, time, or earlier commands.

In more detail: The "Cmd $-\chi$" columns in Table 1 deal with predicting the complete command line, and "Stub $-\chi$"'s, with predicting only the stubs. The $\chi = 0$ columns test the predictive accuracy over the user's ENTIRE sequence of data; the $\chi = 100$ columns test the predictive accuracy after ignoring the first 100 experiences of each user. (Recall this is considered the algorithm's learning phase, to help it adjust to the person.)[7]

To get some baseline values, we also implemented a naïve prediction method that simply predicts the last 5 commands. This system obtains 60.7% accuracy predicting command stubs and 38.8% accuracy predicting command lines. We get even worse numbers, of course, if we consider just the previous SINGLE command. We also considered a system that simply predicts the 5 most

[7] We chose 100 after finding it worked better than other training-set sizes, in the context of "learning full command lines from full command lines" (*i.e.*, the "winner"):

Number ignored:	50	100	200	300
Predictive accuracy:	47.3%	**47.4%**	46.7%	43.3%

frequent previous commands; these values were comparable. (They too appear in Table 1.) Clearly any prediction system that performs worse than these should simply be considered a failure.

The following subsections focus on the predicting the command line from various bits of evidence, in the "raw accuracy" model. The table presents the other data, dealing with the "accuracy$-$100" model, and with the stub-prediction task.

Exp#1. Predicting Command Stubs from Command Stubs. First, we simply duplicated the Davison and Hirsh algorithm, and obtained 72.7% accuracy (accuracy$-$100 score of 73.3%). This is reasonably close to their "near-75%" accuracy. (Note that we are using a different dataset.)

Exp#2. Predicting Command Lines from Command Lines. As noted earlier, predicting entire command lines would be tremendously more useful that just predicting stubs. When we use the entire command lines to predict the next command line, our accuracy is 46.9%. (47.4% when we ignore the first 100 "training-commands"). It is not surprising that this is considerably smaller than the accuracy for predicting stubs, as this is a much more difficult task. (Recall we are only satisfied with a perfect match.)

Of course, getting the stub right might still be useful, even if the rest of the command line is incorrect. For example, we could then give the user the option of moving the stub into the user's buffer, to be augmented, by hand, with the appropriate arguments and options. We therefore considered a different scoring measure, where we awarded the system 1 point if the correct *line* appeared in the list presented to the user; 0.5 points if the correct *stub* appeared, and 0 otherwise. Here, the average score rose to 57.8% — that is, 46.9% of the time our list included the correct line, and an additional 21.8%, it included the correct stub.

Exp#3. Predicting Command Lines from Parsed Command Lines. Most UNIX commands follow the pattern: $\langle$**stub**$\rangle$ $\langle$**switches**$\rangle$ $\langle$**arguments**$\rangle$. By parsing the command lines into a sequence of tokens, it is possible to identify common patterns between a command that occurs at time t and a command that occurs at time $t+1$. (Treating $\langle$**stub**$\rangle$ $\langle$**switches**$\rangle$ as a distinct command is reasonable, since many users alias common $\langle$stub$\rangle$ $\langle$**switches**$\rangle$ pairs (for example, aliasing "`la`" to "`ls -a`").) It then becomes straightforward to compare the arguments between two commands and identify equivalent patterns. This can be further extended by dividing each $\langle$**argument**$\rangle$ into $\langle$a-stub$\rangle$ and $\langle$a-ext$\rangle$ — *e.g.*, "`foo.tex`" becomes "`foo`" and "`tex`"). For example, a command sequence could be parsed as follows:

Original	**Parsed**
vi foo.tex	vi foo.tex
latex foo.tex	latex $\langle$**ARG** 1$\rangle$
dvips foo.dvi	dvips $\langle$**ARG – STUB** 1$\rangle$.dvi
gv foo.ps	gv $\langle$**ARG – STUB** 1$\rangle$.ps

Parsing the command lines identifies generalities that may or may not be correct.[8] For instance, after the shell had been trained on the above example, if a user then enters "`vi /etc/hosts`" the predicter may then suggest "`latex /etc/hosts`", which is **not** very likely to be correct. Further suppose that "`vi /etc/hosts`" had occurred, followed by `ping otherhost`. In the model we implement, the shell can identify relationships such as this much more readily if it does *not* parse the command lines. By parsing the command lines, we end up with a lossy data compression. The average accuracy for parsed command lines is 43.8%. (accuracy$-$100 score of 44.0%). Apparently, the benefits of knowing what to do when seeing "`vi bar.tex`" are outweighed by the loss of individual patterns.

Exp#4. Predicting Command Lines from Command Lines and Error Codes. All UNIX commands have a return code, typically an error code from within the program. Moreover, note that the next command is often be dependent on whether or not the previous command succeeded. For example, if compilation is successful, users will typically execute the new object code. But if compilation fails, we might expect the user to either run a debugger, or edit the source code. We therefore decided to include this, as part of the criteria for deciding on the proper action.

As the meaning of the error code depends on the command itself, we quantizing it down to **no error** versus **error**, (read "`0` versus `non-0`"). This also kept managable the size of the data structure, while providing us with a reasonable amount of information (5.2% of the commands returned an error). However, using the error code provides no improvement over the basic model — producing an accuracy of 46.9%.

Why? First, many commands do not (usually) return an error. Second, the total number of commands likely to follow a given command is relatively low. As we were predicting the top *five* commands, we found that the commands with a matching error code are likely to be listed, anyway (by perhaps earlier).

Exp#5. Using Day of Week and Time of Day. We had anticipated that knowing the day of the week and/or the time of day (morning, afternoon, evening, night) would be helpful, figuring that people work in different modes during the daytime versus nighttime; or between week-days and week-ends. We found, however,

[8] Note that this parsing information could also be useful if we later decide to generate scripts, as it can help distinguish the "variables" from the "constants".

Group name	Predictability	# subject	#Cmds
non-programmers	0.434	25	25,608
novice programmers	0.610	55	77,423
experienced pgms	0.420	36	74,906
computer scientists	0.369	52	125,691

Table 2: Greenberg's DataSet

that it did not help in general: as the table shows, this informatoin caused the average accuracy to go *down* slightly — by about 0.5%.

This may be because the granularity used is too coarse and so this computation ends up duplicating $P(\,\mathrm{cmd}_{t+1} \mid \mathrm{cmd}_t\,)$.

Exp#6. Predicting Command Lines from Multiple Command Lines. Identifying a long-term trend for predicting the next command might be more effective. By using the last two commands, we get 36.9% accuracy for command lines and 68.8% accuracy for command stubs. (Here, we just kept the best $n = 2000$ previous-command-pairs, and used them in the same way we had used previous-single-commands.) Note that this command line accuracy is **worse** than the naïve approach that simply predicts the previous five commands!

Exp#7. Predicting Command Stubs from Command Lines. Above we sought ways to to improve on the prediction rate for command lines. We might be able to use some of these ideas to improve the accuracy of predicting command stubs. By using the additional information in a command line, the list of possible $\langle \textbf{command} - \textbf{stubs} \rangle$ may be reduced (hopefully reducing the chance of error). Unfortunately, we then lose the generality for commands that have not occurred before, and the command stub accuracy drops to 68.7%.

Exp#8. Predicting Command Stubs from Command Stubs and Error Codes. Although using the error code had relatively little impact on predicting command lines from command lines, it causes command stub prediction accuracy to drop slightly 72.6%. This suggests that the error code might not contribute any practical information with the current model; it but may still be useful in future work.

Exp#9. Predictability of Different Classes of Users. As mentioned earlier, the Greenberg data was originally used to learn to classify each computer user into one of the four categories shown in Table 2.

We see clearly that novice programers are by far the most predictable, and computer scientists, the least.[9] A later system may be able to exploit this information, in helping to set up a user profile, or whatever.

(The table's two columns indicate the break-down of the data: how many users were of each categories, and how many commands, total, were from each category.)

[9]It is interesting to speculate on why: novices probably know relatively few commands (which implies a smaller set of commands to predict from); and computer scientists are always trying out new things — and in particular, new command and new sequences.

5 Contributions, and Future Work

Our results, summarized in Table 4, reinforce the Davison/Hirsh claim that it is possible to predict user actions, using a fairly simple and efficient algorithms. Moreover, the accuracy is high enough to lead to a practical, usable system. While the accuracy is not as high as we would like, it has proven surprisingly difficult to improve on the accuracy score. Perhaps this shows that we have, in fact, achieved the inherent predictability of the data — *i.e.*, people truly are random (to this degree), or at least, we appear to be, given the available data.

This last theme, in turn, suggests that we should consider other sources of information, to better capture the context. For example, many users use multiple windows simultaneously (*e.g.*, one for editing and one for compiling). Moreover, today's window managers (*e.g.*, FVWM) allow users to execute applications without typing. Obtaining information about when these events occur is also important. Finally, external events have an impact on what a user will do. Identifying events such as email arrival, or high load averages, should also be considered.

Of course, our long-term goals are not simply saving keystrokes for the small population of people that use UNIX-style shells. Instead, our goal is to obtain a better understand of techniques for predicting future user interactions, in the hope of using such technologies to improve other interfaces — perhaps for common software products such as WORD or POWERPOINT. We suspect that we will be able to use these, or related, techniques not just to predict the commands, but also to detect patterns of the user interactions, and use this to provide a truly helpful assistant — much in the line as the Microsoft "Office Assistant", but one that is adaptive to the individual users. (*E.g.*, that can determine when the user doesn't want to be bothered, or that some specific user typically enjoys hearing new hints, or ...)

Simple, yet powerful, interfaces are clearly important today, given the vast number of interactive application programs. As these applications scale up, effective interfaces may become even more essential. We anticipate that adaptive interfaces, capable of producing interfaces that users will be willing to use, will be a major tool used in building effective interfaces. We hope the results presented in this paper will help future researchers better focus on the relevant aspects of this task.

Finally, any reader who wishes to be part of the subsequent studies (and then to one of the first to reap the eventual benefits of this adaptive shell) should read the material in `http://www.cs.ualberta.ca/~greiner/adapt-interface.html`.)

Acknowledgements

We would like to thank Saul Greenberg for his dataset, which recorded traces of 168 users; without that data, our main study would not have been possible. Portions of this paper were adapted from work done with Thomas Jacob. Greiner was supported, in part, by NSERC.

References

[BD77] Peter J. Bickel and Kjell A. Doksum. *Mathematical Statistics: Basic Ideas and Selected Topics.* Holden-Day, Inc., Oakland, 1977.

[DH97] Brian D. Davison and Haym Hirsh. Toward an adaptive command line interface. In *Advances in Human Factors/Ergonomics*, pages 505–508. Elsevier, 1997.

[DH98] Brian D. Davison and Haym Hirsh. Predicting sequences of user actions. In *Predicting the Future: AI Appreaches to Time-Series Analysis*, pages 5–12. AAAI Press, July 1998. WS-98-07.

[Gre88] Saul Greenberg. Using unix: Collected traces of 168 users. Research Report 88/333/45, Department of Computer Science, University of Calgary, Calgary, Alberta, 1988.

[HBH+98] Eric Horvitz, Lack Breese, David Heckerman, David Hovel, and Koos Rommelse. The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *UAI-98*, pages 256–265, July 1998.

[Lan97] Pat Langley. Machine learning for adaptive user interfaces. In *German Artificial Intelligence*, pages 53–62, 1997. Germany: Springer.

[SH93] J. C. Schlimmer and L. A. Hermens. Software agents: Completing patterns and constructing user interfaces. *JAIR*, 1:61–89, November 1993.