Predicting UNIX Command Lines:
Adjusting to User Patterns

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

Abstract

As every user has his own idiosyncrasies and preferences, an interface that is honed for one user may be problematic for another. To accommodate 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 describes such an adaptive system — here a UNIX-shell that can predict the user's next command, and then use this prediction to simplify the user's future interactions. We present a relatively simple model here, then explore a variety of techniques to improve its accuracy, including a "mixture of experts" model. In a series of experiments, on real-world data, we demonstrate (1) that the simple system can correctly predict the user's next command almost 50% of the time, and can do so robustly — across a range of different users; and (2) that it is extremely difficult to further improve this result.

Keywords: human computer interaction, machine learning, automated modeling

1 Introduction

Today there are a wide variety of interactive computer applications, ranging from web-browsers and searchers, through spreadsheets and database 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 an asterisk ($*$), WORD will change that character to a bullet ($\bullet$) and go into its `List` environment.

Unfortunately, different users have different preferences, therefore the tricks that are appropriate for one user may be problematic for another (*e.g.*, not all users like the fact that WORD automatically formats anything starting with "http://" as a web link). 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* — this typically means 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, pursues 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 use this information to make the interaction simpler for the user (perhaps by automatically resetting some system parameters, or defining appropriate new macros).

This paper investigates a specific manifestation of this task: We design and implement a UNIX command shell that predicts the user's behavior from his previous commands, and then uses 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. Section 2 provides the background for this work, focusing on the Davison/Hirsh system (DH98) which serves as a precursor to our system. Section 3 first scales their system to address our "learn complete commands" task, and observes that the resulting system can correctly predict the user's next command almost 50% of the time, over a wide range of different users. It then presents a body of well-motivated techniques that should improve on that system — *e.g.*, provide the adaptation system with other types of information, including longer history, etc. Our empirical studies, however, demonstrate that these ideas often produced inferior results. Section 4 explains these disappointing (read "what not to do") results, then presents a slightly different approach, of combining the predictions made by a set of relatively-simple experts, each using its own class of information. It then demonstrates that this ap-

```
[1] vi cw.c
[2] make cw
cc -g cw.c -o cw -lm
[3] cw puzzle01
bus error (core dumped)
[4] vi cw.c
[5] make cw
cc -g cw.c -o cw -lm
[6] cw puzzle01
Solution found in 15584 attempts.
[7] cw puzzle02
Solution found in 349295 attempts.
[8] cw puzzle03
segmentation violation (core dumped)
[9] ddd cw core
[10] vi cw.c
[11] make cw
cc -g cw.c -o cw -lm
Undefined symbol: print first refer-
enced in file cw.o
ld: fatal: Symbol referencing errors. No out-
put written to cw
*** Error code 1
make: Fatal error: Command failed for target `cw'
[12] vi cw.c
[13] make
cc -g cw.c -o cw -lm
[14] cw puzzle03
Solution found in 13 attempts.
[15] elm -s"it works" fred
```

Figure 1: Wilma's Command Sequences

proach does work slightly more effectively than the base system. We conclude by arguing that the relatively simple system appears to have "gleaned" essentially all of the accuracy possible from the data collected; if true, this explains why no number of tricks can produce a system that will be significantly more accurate.

1.1 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-cw` 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 a script to perform the `latex-dvips-gv` steps for him, he would not have make the mistake of running `dvips` on a LaTeX source file.

2 Background

UNIX command line prediction appears at first glance to be a trivial task: after all, how many commands can one

```
[1] vi cw.tex
[2] latex cw.tex
[3] dvips cw.dvi
[4] gv cw.ps
[5] vi cw.tex
[6] latex cw.tex
[7] dvips cw.dvi
[8] gv cw.ps
[9] vi cw.tex
[10] latex cw.tex
[11] dvips cw.tex
dvips: ! Bad DVI file: first byte not preamble
[12] dvips cw.dvi
[13] gv cw.ps
[14] wall
Can anyone help me with latex???
^D
```

Figure 2: Fred's Command Sequences

person possibly use regularly? Although novice users[1] are typically easy to predict (as their command set is typically very small), novice users become advanced users with experience, and these advanced users often use hundreds of commands, and moreover often augment these repertoire of commands over time. Over the last decade, many people in the Human-Computer Interaction (HCI) and Machine Learning (ML) have addressed the task of predicting UNIX command lines. Greenberg collected a large amount of data (Gre88), providing the community with the usage patterns (287,000 command lines) of 168 users of varying skill levels (non-programmers, novice programmers, experienced programmers, and computer scientists), over a period of two to six months. There are approximately 5,100 distinct command stubs[2], ranging from 7 to 359 per user (average = 89) in approximately 62,000 distinct command lines, ranging from 35 to 3,160 per user (average = 467). Although the data is ten years old, it is still quite usable, and provides a reasonable benchmark to compare our work against others.

Davison and Hirsh (DH97; DH98) more recently provided a hand-crafted algorithm that adapts to user activity over time, generating a simple probabilistic model to predict command stubs. They keep track of which sequences of commands are typed, and use this to estimate the probability that any command C_{t+1} will immediately follow each possible C_t. However, predicting a user's next command line is a moving target — *e.g.*, external unobserved events (such as mail arriving or deadlines approaching) can occur, and the user goals can change. As this means the underlying command distribution is not stationary, standard ML techniques are not applicable. Therefore, instead of building a conditional probability table (CPT) based on stub *frequencies*, they instead estimate probabilities in a different fashion: Each time they see one command following another —

[1] Novice users (as classified by Greenberg (Gre88)) are typically characterized by a lack of knowledge about existing commands, as well as a lack of knowledge about how to effectively use (or abuse) system resources.

[2] A **stub** is the "executable" of the command, rather than the complete command line — *i.e.*, if the command line is "`latex foo.tex`", the stub is "`latex`".

	vi	latex	dvips	gv	wall
vi	0	1	0	0	0
latex	0	0	1	0	0
dvips	0	0	0.16	0.84	0
gv	0.8	0	0	0	0.2

Table 1: Probability of Stub_{t+1} following Stub_t

say $C_t = \texttt{ls}$ and $C_{t+1} = \texttt{make}$ — they first update all of the current $P\,(\,C_{next} = \chi \,|\, C_{current} = \texttt{ls}\,)$ entries by multiplying each by some fixed $\alpha \in (0, 1)$; they then add $1 - \alpha$ to the particular $P\,(\,C_{next} = \texttt{make} \,|\, C_{current} = \texttt{ls}\,)$ entry. (Note they explicitly maintain only the observed pairs, and implicitly assign each unseen pair the probability of 0.) For example, applying this scheme (which we call "AUR") to the sequence shown in Figure 2, produces the distribution shown in Table 1. (Here we use $\alpha = 0.8$.)

Their system is "on-line": at each point, after observing the sequence of stubs $\vec{C} = (stub_1, \ldots, stub_t)$, it then predicts the five stubs with the largest probability values: $(stub^{*1}_{t+1}, \ldots, stub^{*5}_{t+1})$. It is then told the correct $stub_{t+1}$ that the user actually typed, which it uses to update its CPT, (which in turn is then used to predict $stub^{*}_{t+2}$, etc.). To define *accuracy*, we observe how frequently the predicted command completely matches the user's actual command. This Davison and Hirsh method obtains nearly 75% accuracy, in that the user's actual command stub is one of the 5 predicted stubs, three-fourths of the time.

Our task is extremely similar to theirs, except we will focus on learning *entire commands*, rather than just stubs; and we have built a real-time system for this task, rather than just performing a retrospective analysis. Moreover, many of our individual experts will use something very similar to AUR.

Yoshida and Motoda (YM96) examine another aspect of the issue: by exploring file I/O relationships (*i.e.*, which programs use which files) they can suggest default commands to run on files. For example, suppose a user runs "`emacs foo.tex`", "`latex foo.tex`", "`dvips foo.dvi`" and "`gv foo.ps`". Later on, when the user creates another file "`bar.ps`", their system will suggest running "`gv`", because the file ends in "`.ps`". After a number of interactions have been observed, complete shell scripts can be generated for a sequence of events (*e.g.*, for compiling or writing LaTeX documents). Additionally, they explore file/web cache pre-fetching to reduce load delays (MY98).

Finally, the Reactive Keyboard (DW92) uses length-k modeling to predict the next keystrokes based on the previous keystrokes typed. This is similar to our work, except we are predicting the next complete command line based on the previous command lines typed, rather than a keystroke at a time. This leads to a fundamental difference: while they deal with only a small number of possible "tokens", our system must deal with an unbounded number of possible commands.

Command_t	Command_{t+1}	Prob
vi cw.tex	latex cw.tex	1.0
latex cw.tex	dvips cw.dvi dvips cw.tex	0.8 0.2
dvips cw.dvi	gv cw.ps	1.0
dvips cw.tex	dvips cw.dvi	1.0
gv cw.ps	vi cw.tex wall	0.8 0.2

Table 2: Probability of Command_{t+1} following Command_t

3 Basic Approach

As noted above, our goal is to extend the Davison and Hirsh system, in several ways. First, we implemented an interactive system. As described below (Section 4.1), we first estimate a distribution over the next command — computing the probability that this $t + 1^{st}$ command will be "`ls`", versus "`cd ..`" versus "`latex foo.tex`", etc. We then map the top five to the F-keys — the best prediction to F1, the second best to F2, ..., the fifth best to F5 — and lists these mappings (read "predictions") in the title bar of the xterm; see Figure 3. Throughout our many designs, we insisted that the adaptive system remain "real-time"; *i.e.*, the total process — updating the various data structures, computing the most likely commands, resetting the display, etc. — could not take over 500 milliseconds. Note, also, that this approach complements the existing history and file/command completion and correction facilities already available with the shell we used, ZSH.

The second extension is to predict *complete commands*, rather than just stubs. Our first approach was simply to extend the Davison/Hirsh system to predict these complete commands; producing Table 2 from the Figure 2 dialog. The resulting system was 47.4% accurate over the Greenberg data.[3] Hoping to improve of this score, we then implemented several modifications.

First, we built a system that used both the previous command and the error code it returned. For example, after running `latex foo.tex`, we figured the user would only go on to `dvips foo.dvi` if the `latex` command was successful (read "return error code of 0"), and would otherwise go perhaps to `emacs foo.tex`. Unfortunately, our empirical data shows this degraded the performance; see Table 4. We had similar negative results when we tried incorporating day-of-week, or time-of-day.

We also considered *parsing* the actual commands, to enable some simple types of generalization — *e.g.*, after observing that "`dvips foo.dvi`" followed "`latex foo.tex`" and "`dvips bar.dvi`" followed "`latex bar.tex`", our system should anticipate that "`dvips blop.dvi`" may follow "`latex blop.tex`", even though it has never seen anything related to "`blop`" before. To do this, we used an parser to produce an "abstract" pattern of commands; *e.g.*, to re-represent the commands from Figure 2 as:

[3] Our system matched the Davison and Hirsh results: we could get almost 75% accuracy when predicting *stubs*.

Figure 3: Implementation

```
[1] vi cw.tex
[2] latex ⟨T-1 ARG1⟩
[3] dvips ⟨T-1 ARG1 STEM⟩.dvi
[4] gv ⟨T-1 ARG1 STEM⟩.ps
[5] vi ⟨T-1 ARG1 STEM⟩.tex
...
```

Our system can then attempted to match the "current pattern" against these abstracted command lines (rather than the actual commands). This too degraded performance.

Finally, we considered conditioning on the previous *two* commands, $\langle C_{t-1}, C_{t-2} \rangle$, rather than just C_{t-1}. (That is, we used AUR to compute $P\,(\, C_t = \chi \,|\, C_{t-1} = a, C_{t-2} = b \,)$ based on the observed triples b, a, χ.) So from the Figure 1 dialog, we would claim an 84% chance that that the user will type `cw`, after seeing her type the the sequence $\langle\, \texttt{make}, \texttt{vi} \,\rangle$ — *i.e.*, $P\,(\, C_t = \texttt{cw} \,|\, C_{t-1} = \texttt{make}, C_{t-2} = \texttt{vi} \,) \;=\; 0.84$. This too failed — producing a system whose accuracy dropped to only 36.9%.

So, while we thought each of these approaches *had* to work, our data proved otherwise: essentially all of these winning ideas caused the performance to degrade. Undaunted, we tried another approach…

4 "Mixture of Experts" Approach

Each of the previous ideas attempted to exploit some other type of information. Of course, we would like to include them *all*. One major problem, of course, is dimensionality: Even assuming there are only 500 commands, we first need to consider 500^2 possible command-pairs. For each of these, we need to also consider time of day (say in 4 buckets), error code (say in 2 buckets), and day of week (say in 7 buckets) which means we will need to estimate at least $14,000,000$ parameters. Unfortunately, this is not just cumbersome, but unlearnable: people will change their patterns long before observing enough commands to produce good estimates here.

We therefore need to reduce the dimensionality of this space. Here we used a standard trick of "factoring" the range of options. We word this using the "mixture of experts" model (JJNH91): Building several relatively simple experts $\{E_i\}$, each predicting the next command from some different set of available information; see Figure 4. After Section 4.1 motivates and provides the combination rule we used, the remaining 4 subsections describe the four actual experts we used: E_{AC} uses the previous k <u>A</u>ctual <u>C</u>ommands (for various k); E_{PC} uses the previous k <u>P</u>arsed <u>C</u>ommands; E_{L100} focuses on the <u>L</u>ast <u>100</u> commands; and E_{F1} uses the <u>F</u>irst <u>1</u> command in a session. The concluding Subsection 4.6 then shows provides empirical data to illustrate that these experts, when combined, did produce a slight improvement.

Source of Prediction	Accuracy $\pm\sigma$
Previous 1 Cmd	47.4% ± 0.0009
Previous 1 Cmd, w/parsing	44.0% ± 0.0009
Previous 2 Cmds	36.9% ± 0.0009
Previous 1 Cmd + Error Code	46.9% ± 0.0009
Previous 1 Cmd + Day of Week	46.7% ± 0.0009
Previous 1 Cmd + Time of Day	46.6% ± 0.0009

Table 3: Early Results

4.1 Combining the Voices

Here, we assume each expert E_i has used some aspects of the current body of information Ω (previous commands, error code, time and date, etc), to produce its prediction for what command the user will type next; $P\,(\, E_i = y \,|\, \Omega \,)$ ($E_i = y$ indicates that expert E_i predicts y will be the next command, and $C = \chi$ is the composite prediction. The specific predictors are described below.) Our goal is to combine these, to produce an improved prediction — $P\,(\, C = \chi \,|\, \Omega \,)$.

To simplify the derivation, assume there are only two experts. We can then write

$$P\,(\, C = \chi \,|\, \Omega \,) \;=\; \sum_{y,z} P\,(\, C = \chi, E_1 = y, E_2 = z \,|\, \Omega \,) \;=$$

$$\sum_{y,z} P\left(C = \chi \,\middle|\, \begin{array}{l} E_1 = y \\ E_2 = z \\ \Omega \end{array} \right) P\left(E_1 = y \,\middle|\, \begin{array}{l} E_2 = z \\ \Omega \end{array} \right) P\,(E_2 = z|\,\Omega\,)$$

$$= \sum_{y,z} P\left(C = \chi \,\middle|\, \begin{array}{l} E_1 = y \\ E_2 = z \end{array} \right) P\,(\, E_1 = y \,|\, \Omega \,)\, P\,(\, E_2 = z \,|\, \Omega \,)$$

where the last line uses the assumptions that the prediction will depend only on the values that the experts say (*i.e.*, $P\,(\, C = \chi \,|\, E_1 = y, E_2 = z, \Omega \,) \;=\; P\,(\, C = \chi \,|\, E_1 = y, E_2 = z \,)$); and E_2's prediction is independent of E_1's given the background Ω — $P\,(\, E_1 = y \,|\, E_2 = z, \Omega \,) \;=\; P\,(\, E_1 = y \,|\, \Omega \,)$. We further simplify the equations by making the assumption that $P\,(\, C = \chi \,|\, E_1 = y, E_2 = z \,) \;=\; 0$ when $\chi \notin \{y, z\}$, and moreover, that we can lump together various $y \neq z$ cases. We also ignore the actual commands — *i.e.*, assume $P\,(\, C = \text{ls} \,|\, E_1 = \text{ls}, E_2 = \text{ls} \,) \;=\; P\,(\, C = \text{make} \,|\, E_1 = \text{make}, E_2 = \text{make} \,)$, and so forth. This means we need only estimate 3 additional numbers:

$$\begin{array}{lcl} P_{=,=} & = & P\,(\, C = \chi \,|\, E_1 = \chi, E_2 = \chi \,) \\ P_{=,\neq} & = & P\,(\, C = \chi \,|\, E_1 = \chi, E_2 \neq \chi \,) \\ P_{\neq,=} & = & P\,(\, C = \chi \,|\, E_1 \neq \chi, E_2 = \chi \,) \end{array}$$

(Recall

$$P_{\neq,\neq} \;=\; P\,(\, C = \chi \,|\, E_1 \neq \chi, E_2 \neq \chi \,) = 0\,)$$

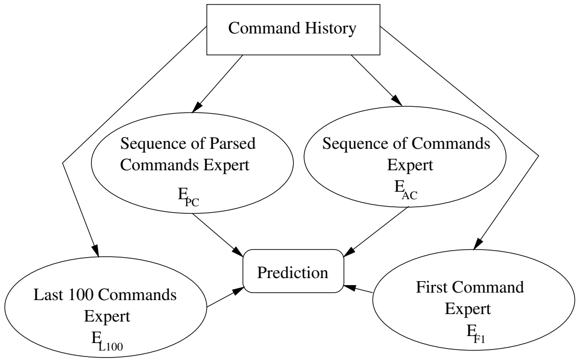

Figure 4: Combination of multiple Experts

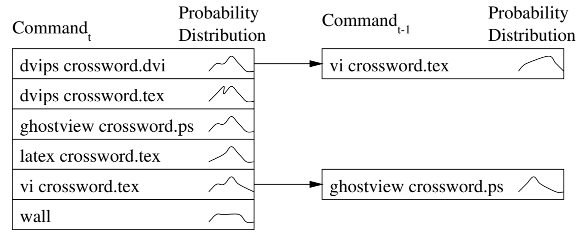

Figure 5: History Expert

This means

$$
\begin{aligned}
P\,(\,C=\chi\,|\,\Omega\,) \;&=\; P_{=,=} \quad P_1\,(\,\chi\,|\,\Omega\,)\;P_2\,(\,\chi\,|\,\Omega\,) \\
&+\; P_{=,\neq} \quad P_1\,(\,\chi\,|\,\Omega\,)\;(1-P_2\,(\,\chi\,|\,\Omega\,)) \\
&+\; P_{\neq,=} \quad (1-P_1\,(\,\chi\,|\,\Omega\,))\,P_2\,(\,\chi\,|\,\Omega\,)
\end{aligned}
$$

where $P_i\,(\,\chi\,|\,\Omega\,)$ abbreviates $P\,(\,E_i=\chi\,|\,\Omega\,)$. To further simplify the computation, we only consider a subset of the possible commands χ — only those commands that either E_1 or E_2 rank in their (respective) top 10.

Of course, this all scales up to larger sets of experts: In general, if we are considering k experts, then (in addition to having each expert learn its own distribution $P\,(\,E_i=\chi\,|\,\Omega\,)$), we will also need to estimate 2^k-1 probability values.

4.2 Using m Previous Actual Commands

As noted above (when outlining AUR; Section 2), the distribution over the user's next command is not stationary, which means standard techniques (such as using simple frequencies) will not work. The E_{AC} expert therefore adopts a version of the AUR approach, in using the observed previous commands $C_{t-1}, C_{t-2}, \ldots$ in predicting the next (currently unobserved) C_t command. Of course, we are dealing with the entire command, rather than just stubs. See Table 2.

We initially used only the immediately previous command C_{t-1}, then tried conditioning on the previous two commands, $\langle\, C_{t-1}, C_{t-2}\,\rangle$. Unfortunately, relatively few command-pairs occurred frequently; this means we are relatively unlikely to have seen many examples of any particular pair before… which meant our system overfit the available data, and produced a system with reduced accuracy (only 36.9%).

A better approach is to condition on the previous two commands *only when we have reason to believe these predictions will be meaningful*; that is, only when this pair has occurred frequently before. Otherwise, we just condition on the single, immediately previous C_{t-1}. Of course, we can apply this recursively, to consider conditioning on the previous three $C_{t-1}, C_{t-2}, C_{t-3}$ when there is sufficient data to support any conclusions reached, or previous 4, or whatever. (This is idea underlying the IMM system (SDKW98).)

We therefore built a data structure to encode this information; such as Figure 5, which corresponds to the Figure 2 example. As shown on the left-hand-side, each single command indexes a node, which also has a distribution; *e.g.*, the top left node contains the distribution $P\,(\,C_t=\chi\,|\,C_{t-1}=\texttt{dvips cw.dvi}\,)$. As there are at least $k=2$ examples of $\langle\,C_{t-1}=\texttt{dvips cw.dvi},\ C_{t-2}=\texttt{vi cw.tex}\,\rangle$, this top-left node also has a child, shown to its right. That node also includes a distribution $P\,(\,C_t=\chi\,|\,C_{t-1}=\texttt{dvips cw.dvi},\ C_{t-2}=\texttt{vi cw.tex}\,)$. Of course, that "`dvips cw.dvi`" node could have several children, and any of these children could their own have children. *E.g.*, if ρ had the child σ which had the child τ, then the τ node would include a distribution $P\,(\,C=\chi\,|\,C_{t-1}=\rho, C_{t-2}=\sigma, C_{t-3}=\tau\,)$, and so forth. Here, we update this data structure by generating a new child whenever there are a sufficient number of instances, here $k=5$, that reach the child's location. Although the possibility of growth is unlimited, in practice the number of levels of children rarely exceeds 3.

4.3 Using m Previous Abstracted Commands

The E_{PC} expert first "abstracts" the command line, by replacing the details of the filename components (path name and extensions such as "`/usr/ralph/`" and "`.tex`") with generic, matchable terms; see Section 3. This allows it to re-use the patterns, in matching new terms. After command lines have been converted into patterns, this expert uses the same prediction method as in Section 4.2.

4.4 Short-Term Frequency Predictions

The E_{L100} expert maintains a frequency table for the last $m=100$ commands, and predicts the distribution over the $t+1^{st}$ command will be from this empirical distribution

$$P_{L100}\,(C_{t+1}=x\,|\,\Omega) \quad = \quad \frac{|x\in\{C_{t-n},\ldots,C_t\}|}{n}$$

This method performs surprisingly well by itself, and provides a robust mechanism for predicting commands — *e.g.*, after erroneous command lines (a.k.a. typos) are entered.

4.5 Predicting the Sessions First Command

The first command in a session is a special case, and problematic as there is no previous command on which to base predictions. In a method very similar to that of Section 4.4, the E_{F1} expert maintains a simple probability table for the first commands of the user's last $n=100$ sessions, and uses this to predict the user's first command. Of course, E_{F1} is only active for the first command of each session.

4.6 Results

Table 4 summarizes our results, showing that our algorithm, with all of its experts, can guess the correct command slightly more accurately than the simple AUR system. While the improvement seems minor, notice it is significant, at the $p<0.001$ level.

Source of Prediction	Accuracy $\pm\sigma$
5 Most Frequent Lines	33.9% ± 0.0009
5 Most Frequent in Last 100	44.6% ± 0.0009
Simple AUR method (1 previous cmd)	47.4% ± 0.0009
Combination of Experts	47.9% ± 0.0009

Table 4: Command Line Prediction Accuracy

We explored using yet other tricks (including combinations with the "obviously appropriate" ones listed in Table 3 (KG99)), but were unable to produce significantly better results. We therefore suspect we have reached the limits of successful prediction with this dataset — the remaining error is simply a function of the underlying stochasticity of the process. Towards confirming this, we investigated how well our algorithm would work if it had complete knowledge of the future; *i.e.*, we trained it on the complete data, and then ran the trained version over the same data. We saw this system was only correct 45.5% of the time. In fact, on average, only 72% of commands are duplicates of previous commands. That is, 28% of commands are *very* difficult to predict, since they have never been seen before.

5 Conclusion

5.1 Future Work

Although the Greenberg data helped us to start our work, we worry it may miss information that is needed for better in-depth analysis of user patterns. We are therefore collecting our own data (see Figure 3). While we have collected many fewer command lines, we store with each command line substantially more information, including:

- current working directory (track directory navigation)
- command timing (allowing timing of under one second)
- computer and display identification
- shell aliases and functions

We plan to exploit this additional information in the automated construction of shell macros. (Here we will, of course, use the abstracted patterns; see Section 4.3. We also suspect that the other available information, such as error codes, will be essential here.)

Our user interface, while functional, suffers some drawbacks and would benefit greatly from auto-completion and integration with ZSH's correction facilities.

We are also investigating other applications — *i.e.*, other situations where it would be useful to predict the user's next command. In addition to detecting and exploiting patterns in other software systems (like WORD or POWERPOINT), we are also considering using this to help optimize communication; *e.g.*, when using a Palm™ organizer to `telnet` to a remote computer or to administer a NetWinder server. Bandwidth is not a large concern, but command input can be. Providing command lines on a pull-down menu could improve general `telnet` access, and accurate construction of useful aliases for long command lines would be far more valuable.

5.2 Contributions

This paper has investigated various ways of producing a system to predict the user's next command. Our explorations have produced many insights into this task: for example, we found that a fairly simple algorithm (AUR, on the single previous command) can perform adequately. We then considered many obvious "improvements", which used other types of information. We found, however, that these modifications typically *degrade*, not improve, performance. We attribute this to both the vast number of parameters that these systems needed to estimate, and to the fact that the distribution of user commands are typically non-stationary.

To achieve any improvement required using more sophisticated techniques: here, we used a "mixture of experts" approach, together with a body of "approximations" to drastically reduce the number of parameters that needed to be estimated. We are continuing to explore this framework, in incorporating other types of information (*e.g.*, about error codes and time of day). The current system, however, is quite usable, and demonstrates that it is possible to predict complete user commands correctly almost half the time, and do so in real-time.

For more information about our system, please visit `http://www.cs.ualberta.ca/~benjamin/AUI/`

Acknowledgements

We thank Saul Greenberg for the use of his UNIX usage data. Portions of this paper were adapted from work done with Thomas Jacob. Korvemaker and Greiner were supported, in part, by NSERC.

References

[DH97] Brian D. Davison and Haym Hirsh. Toward an adaptive command line interface. In *Advances in Human Factors/Ergonomics: Design of Computing Sytems: Social and Ergonomic Considarations*, 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.

[DW92] John J. Darragh and Ian H. Witten. *The Reactive Keyboard*. Cambridge Series on Human-Computer Interaction. Cambridge University Press, New York, New York, 1992.

[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.

[JJNH91] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[KG99] Benjamin Korvemaker and Russell Greiner. The trials and tribulations of building an adaptive user interface. http://www.cs.ualberta.ca/˜greiner/PAPERS/AdaptUI.ps, 1999.

[MY98] Hiroshi Motoda and Kenichi Yoshida. Machine learning techniques to make computers easier to use. *Artificial Intelligence*, 103(1–2):295–321, 1998.

[SDKW98] S. Salzberg, A. Delcher, S. Kasif, and O. White. Microbial gene identification using interpolated markov models. *Nucleic Acids Research*, 26(2):544–548, 1998.

[YM96] Kenichi Yoshida and Hiroshi Motoda. Automated user modeling for intelligent interface. *International Journal of Human-Computer Interaction*, 8(3):237–258, 1996.