Chapter 3

Incremental Probabilistic Action Prediction

3.1 Introduction

How predictable are people? Each of us displays patterns of actions throughout whatever we do. Most occur without conscious thought. Some patterns are widespread among large communities, and are taught, as rules, such as reading from left to right, or driving on the correct side of the road. Other patterns are a function of our lifestyle, such as picking up pizza on the way home from work every Friday, or programming the VCR to record our favorite comedy each week. Many are a result of the way interfaces are designed, like the pattern of movement of your finger on a phone dialing a number you call often, or how you might log into your computer, check mail, read news, and visit your favorite website for the latest sports scores.

As computers pervade more and more aspects of our lives, the need for a system to be able to adapt to the user, perhaps in ways not programmed explicitly by the system’s designer, become ever more apparent. A car that can offer advice on driving routes is useful; one that can also guess your destination (such as a pizza parlor because it is Friday and you are leaving work) is likely to be found even more useful, particularly if you didn’t have to program it explicitly with that knowledge. The ability to predict the user’s next action allows the system to anticipate the user’s needs (perhaps through speculative execution or intelligent defaults) and to adapt to and improve upon the user’s work habits (such as automating repetitive tasks). Additionally, adaptive interfaces have also been shown to help those with disabilities [GDMW95, DM92].

This chapter will provide an overview of prediction models, and discuss some aspects of the approaches possible as well as enumerate characteristics that an idealized prediction approach might have. However, we will leave experimental examination of
most of these methods until Chapter 4, and spend the rest of this chapter providing an introduction to just one approach — the use of a simple mechanism for the prediction of user actions. The domain of UNIX command prediction will serve as a testbed for the development of IPAM, an algorithm based on first-order Markov-models with an emphasis on recency for prediction of the next action, which we describe below in Section 3.4.

3.2 Background

The problem of modeling a user’s behavior has received much attention from researchers in recent years. Given an accurate model, the user’s interface can be customized — perhaps to improve ease of comprehension or navigation by providing hints or suggestions, or to increase the likelihood of generating a sale on an e-commerce site, or to invisibly work to improve the user’s perceptions of performance by anticipating the needs of the user in advance.

Regardless of the ultimate application, the task is the same — to build a model that can predict future user actions. Building such a model can take various forms, including user interviews and explicit feedback, or by passively watching the user [MMS85, Cha99]. This chapter focuses on perhaps the most common source of information about the user — the user’s past actions.

However, we are also concerned with building and using those predictions within an interactive system — typically one that uses each action as a chance to learn and an opportunity to make predictions to assist the user. This means the learning algorithm needs to be usable in an online setting, unlike batch learning approaches that are typically too slow to be retrained using a large dataset whenever they need to be updated.

“Machine learning is concerned with the question of how to construct computer programs that automatically improve with experience.” [Mit97]. However, while learning to predict a user’s action is certainly within the domain of machine learning, the problem domain is specific enough that the approaches commonly employed are not
necessarily typical machine learning algorithms.

The approaches taken for action prediction vary considerably, from subsequence matching (as in [DH97b, SKS98, PP99b]) in which past sequences of actions can provide a possible next action for the current sequence, to Markov and Markov-like models including both single and multi-step (e.g., [Bes95, PM96, JG99, DK01, PP99a, SYLZ00]) in which the possible next actions are encoded as states with probabilities from the current state (labeled with the most recent action). Other approaches are possible, such as something in between like Prediction by Partial Match (e.g., [KL96, Pal98, PM99, FJCL99]) and TDAG [Lai92, LS94] which combine Markov models of varying order, to inducing a grammar (as in [NM96]). Many of these methods have connections to data compression (in particular, TDAG, PPM, and grammar induction), and often have complex implementations. We will discuss a number of them in Chapter 4. In many cases, however, simple methods can do well, as we show in this chapter by introducing a simple Markov-like algorithm for action prediction and test it in the domain of UNIX commands.

3.3 UNIX Command Prediction

3.3.1 Motivation

We consider here user actions within a command line shell. We have concentrated initially on UNIX command prediction\(^1\) because of its continued widespread use; the UNIX shell provides an excellent testbed for experimentation and automatic data collection. However, our interest is in more general action prediction. This chapter, therefore, reflects our focus on the underlying technology for action prediction, rather than on how prediction can be effectively used within an interface.

We wish to address the task of predicting the next element in a sequence, where the sequence is made up of nominal (unordered as well as non-numeric) elements. This type of problem (prediction of the next step in a series) is not studied often by machine

\(^1\)In this work we ignore command arguments and switches, but others [KG00] have replicated our approach on full command lines and similarly found it to be quite successful.
learning researchers; concept recognition (i.e., a boolean classification task such as sequence recognition) is more common, as is the use of independent samples from a distribution of examples. UNIX commands, and user actions in general, however, are not independent, and being nominal, don’t fall into the domain of traditional statistical time-series analysis techniques.

Below we use the user histories from two studies to suggest that relatively naive methods can predict a particular user’s next command surprisingly well. Finally, we will present and analyze a novel algorithm that satisfies these characteristics and additionally improves on the performance of our previous work [DH97a, DH97b].

3.3.1.1 Evaluation Criteria

In most machine learning experiments that have a single dataset of independent examples, cross-validation [Sto74, Sto78, BFOS84] is the standard method of evaluating the performance of an algorithm. When cross-validation is inappropriate, partitioning the data into separate training and test sets is common. For sequential datasets, then, the obvious split would have the training set contain the first portion of the sequence, and the test set contain the latter portion (so that the algorithm is not trained on data occurring after the test data). However, since we are proposing an adaptive method, we will be evaluating performance online — each algorithm is tested on the current command using the preceding commands for training. This maximizes the number of evaluations of the algorithms on unseen data and reflects the expected application of such an algorithm.

When considering performance across multiple users with differing amounts of data, we use two methods to compute averages. Macroaveraged results compute statistics separately for each user (e.g., the mean accuracy), and then average these statistics over all users. Alternatively, microaveraged results compute an average over all data, determining the number of correct predictions made across all users divided by the total number of commands for all users combined. The former provides equal weight to all users, since it averages across the average performance of each user; the latter gives equal weight to all predictions, emphasizing users with large amounts of data.
Figure 3.1: A portion of one user’s history, showing the timestamp of the start of the session and the command typed. (The token BLANK marks the start of a new session.)

3.3.1.2 People Tend To Repeat Themselves

In order to determine how much repetition and other recognizable regularities were present in the average user’s command line work habits, we examined two datasets of many users' activities in a UNIX shell. In the first (Rutgers) dataset, we collected command histories of 77 users, totaling over 168,000 commands executed during a period of 2-6 months [DH97a, DH97b] (see Figure 3.1 for an example of the data that was collected). The bulk of these users (70) were undergraduate computer science students in an Internet programming course and the rest were graduate students or faculty. All users had the option to disable logging and had access to systems on which logging was not being performed.

The average user had over 2000 command instances in his or her history, using 77 distinct commands during that time. On average over all users (macroaverage), 8.4% of the commands were new and had not been logged previously. The microaverage of new commands, however, was only 3.6%, reflecting the fact that smaller samples had larger numbers of unique commands. Approximately one out of five commands were the same
as the previous command executed (that is, the user repeated the last command 20% of the time).

The second (Greenberg) dataset is a larger collection of user activities with the csh UNIX shell [Gre88, Gre93]. It contains more than twice as many users and approximately twice as many commands overall. This dataset was acquired after completing the tests on the first, but we will present the results of both datasets when appropriate.

3.3.1.3 Earlier Results

In previous work [DH97a, DH97b], we considered a number of simple and well-studied algorithms. In each of these, the learning problem was to examine the commands executed previously, and to predict the command to be executed next. We found that, without explicit domain knowledge, a naive method based on C4.5 [Qui93] was able to predict each command with a macroaverage accuracy of 38.5% (microaverage was 37.2%). Given a sequence of commands, C4.5 was trained to use the preceding two commands to predict the next. Thus for each prediction, C4.5 was trained on the series of examples of the form (Command$^{i-2}$, Command$^{i-1}$) ⇒ Command$^i$; for 1 ≤ $i$ ≤ $k$, where $k$ is the number of examples seen so far. Command$^0$ and Command$^1$ are both defined to have the special value BLANK to allow prediction of the first and second commands where otherwise there would be no previous two commands.

While the prediction method was a relatively straightforward application of a standard machine learning algorithm, it has a number of drawbacks, including that it returned only the single most likely command. C4.5 also has significant computational overhead. It can only generate new decision-trees; it does not incrementally update or improve the decision tree upon receiving new information. (While there are other decision-tree systems that can perform incremental updates [Utg89], they have not achieved the same levels of performance as C4.5.) Therefore, C4.5 decision tree generation must be performed outside of the command prediction loop.

Additionally, since C4.5 (like many other machine learning algorithms) is not incremental, it must revisit each past command situation, causing the decision-tree generation to require more time and computational resources as the number of commands
in the history grows. Finally, it treats each command instance equally; commands at
the beginning of the history are just as important as commands that were recently exe-
cuted. Note that C4.5 was selected as a common, well-studied decision-tree learner with
excellent performance over a variety of problems, but not with any claim of superiority
over other algorithms applicable to this domain.

These initial experiments dealt with some of these issues by only allowing the learn-
ing algorithm to consider the command history within some fixed window. This pre-
vented the model generation time from growing without bound and from exceeding all
available system memory. This workaround, however, caused the learning algorithms
to forget relatively rare but consistently predictable situations (such as typographical
errors) and restricted consideration only to recent commands.

3.3.2 An Ideal Online Learning Algorithm (IOLA)

By focusing on the application of adaptive interfaces, we can propose characteristics of
an “ideal” learning algorithm for such problems. An Ideal Online Learning Algorithm
(IOLA) and its realization would:

1. have predictive accuracy as good as the best known resource-unlimited methods;

2. operate incrementally (modifying an existing model rather than building a new
   one as new data are obtained);

3. be affected by all events (remembering uncommon, but useful, events regardless
   of how much time has passed);

4. not need to retain a copy of the user’s full history of actions;

5. output a list of predictions, sorted by confidence;

6. adapt to changes to the target concept;

7. be fast enough for interactive use;

8. learn by passively watching the user; and
Figure 3.2: Standard (Bayesian) incremental update: one row of a 5x5 state transition table, before and after updating the table as a result of seeing action A3 follow A1.

9. apply even in the absence of domain knowledge.

Such an algorithm would be ideally suited for incorporation into many systems that interact with users.

3.4 Incremental Probabilistic Action Modeling (IPAM)

3.4.1 The algorithm

In our previous work, we implicitly assumed that the patterns of use would form multi-command chains of actions, and accordingly built algorithms to recognize such patterns. If, however, we make the simpler Markov assumption that each command depends only on the previous command (i.e., patterns of length two, so that the previous command is the state), we can use the history data collected to count the number of times each command followed each other command and thus calculate the probability of a future command. This could be implemented by the simple structure of an \( n \times n \) table showing the likelihood of going from one command to the next, as shown in Figure 3.2a.

For the anticipated use of action prediction in an adaptive interface, however, an incremental method is desirable. Instead of just using a static table of probabilities calculated on some training data, the table can be updated after every new instance (as shown by the portions of a state transition table before and after an update in Figure 3.2). From the recorded counts, the fractions can be interpreted as probabilities for the various state transitions. However, as mentioned earlier, we believe it is useful to weigh recent events more highly when building an adaptive model. This can be accomplished.
in this probabilistic model by the use of an update function with an exponential decay (in which the most recent occurrence has full impact; older occurrences have ever-decreasing contributions). Given the previous table of probabilities and another table containing probabilities from new data points, a combined new table may be computed by the weighted average of the two, where the weights sum to 1. So, for example, if the weights were both .5, the new probabilities would have equal contributions from the old table and from the new. Assuming that the table updates were performed periodically, the data points making up the first table would be contributing only \( \frac{1}{n+1} \) percent of the final weights (where \( n \) is the number of table updates so far).

We can extend this model further, to an algorithm that starts with an empty table and updates after every command. An empty table is one in which all commands are equally likely (initially a uniform probability distribution). After seeing a command, \( c_i \) for the first time, a new row is added for that command, and has a uniform distribution. When the second command, \( c_{i+1} \), is seen, it too gets a new row with a uniform distribution, but we update the first row (since we saw \( c_i \) followed by \( c_{i+1} \)) by multiplying all elements of that row by a constant \( 0 \leq \alpha \leq 1 \), and the probability of seeing \( c_{i+1} \) is increased by adding \( (1 - \alpha) \). In this way, we emphasize more recent commands, at the expense of older actions, but the sum of the probabilities in each row is always 1. An example can be found in Figure 3.3.

Note that an \( \alpha \) of 0 equates to a learner that always predicts what it most recently saw for that command, and an \( \alpha \) of 1 corresponds to an algorithm that never changes its probabilities (in this case, preserving a uniform distribution).
Figure 3.4: Selected rows of a 5x5 state transition table, before and after updating the table as a result of seeing action A4 for the first time.

Update(PreviousCommand, CurrentCmd):
- Call UpdateRow(DefaultRow, CurrentCmd)
- Call UpdateRow(PreviousCommand’s row, CurrentCmd)

UpdateRow(ThisRow, CurrentCmd):
- If initial update for ThisRow, copy distribution from default row
- Multiply probability in each column by alpha and add (1-alpha) to column that corresponds to CurrentCmd

(a) The update function.

Predict(NumCmds,PreviousCmd):
- Call SelectTopN(NumCmds, PreviousCmd’s row, {})
- Let P be the number of commands returned
- If P < NumCmds, call SelectTopN again, but ask for the top
  (P - NumCmds) commands from the default row and to exclude those commands already returned
- Return the first set followed by the default set of predicted commands

SelectTopN(NumCmds,Row,ExcludeCmds):
- Sort the probabilities in Row
- Eliminate commands in ExcludeCmds
- Return the top NumCmds from sorted list

(b) The predict function.

Figure 3.5: Pseudo-code for IPAM.

For prediction, the command with highest probability for the given state would be output as the most likely next command. Instead of making no prediction for a command with an empty row, we can track probabilities in an additional default row, which would use the same mechanism for updating but would apply to all commands.
Figure 3.6: For a range of $\alpha$ values, the predictive accuracy of the Incremental Probabilistic Action Modeling algorithm on the Rutgers data is shown.

seen so far (without consideration of the preceding command). Finally, since we are keeping track of overall likelihoods in this default row, we can use it to initialize rows for new commands (making the assumption that these default statistics are better than a uniform distribution). Figure 3.4 illustrates the process of adding a new row when an action is first seen.

See Figure 3.5 for pseudo-code for the $\text{Update}$ and $\text{Predict}$ functions that implement this Incremental Probabilistic Action Modeling (IPAM) algorithm.

### 3.4.2 Determining $\alpha$

We empirically determined the best average $\alpha$ by computing the performance for this algorithm on the Rutgers dataset with each of seven values of $\alpha$ (from .65 to .95 in increments of .05). While the best value of $\alpha$ varied, depending on how performance was calculated over the dataset, our subjective choice for the best overall was .80. (See Figure 3.6 for a graph of the parameter study of $\alpha$ showing the average user’s performance as well as the average performance over all commands.) We will use this value of $\alpha$ for the rest of the experiments in this chapter. Since $\alpha$ controls the amount of influence recent commands have over earlier commands, we expect that this value will vary by problem domain. To ensure fair evaluation of our approach with this parameter, we do not recalculate a best $\alpha$ for the Greenberg data, but instead
Figure 3.7: Macroaverage (per user) predictive accuracy for a variety of algorithms on the Rutgers dataset.

Figure 3.8: Average per user accuracies of the top-$n$ predictions. The likelihood of including the correct command grows as the number of suggested commands increases.

3.5 Evaluation

This algorithm, when applied to the Rutgers data set, performs better than C4.5 (given an alpha of .80). It achieves a predictive accuracy of 39.9% (macroaverage) and 38.5%
Figure 3.9: Command completion accuracies. The likelihood of predicting the correct command grows as the number of initial characters given increases.

(microaverage) versus C4.5’s 38.5% and 37.2% (macroaverage and microaverage, respectively) for best guess predictions (see the bars labeled C4.5 and IPAM in Figure 3.7).

For comparison, we also show our method without the specialized update, which corresponds to naive Bayes (that is, a predictor in which the conditional probabilities to select the most likely next command are based strictly on the frequency of pairs of commands), as well as a straightforward most recent command predictor (labeled MRC) that always suggests the command that was just executed.

To be precise, over the 77 Rutgers users, IPAM beat the C4.5-based system sixty times, tied once, and lost sixteen times on the task of predicting the next command. At the 99% confidence level, the average difference between their scores was $1.42 \pm 1.08$ percentage points, showing that the improvement in predictive accuracy for IPAM over C4.5 is statistically significant, given this ideal value for $\alpha$. While in practice the improvement in accuracy might not be noticeable, but it is important to note the simplicity (as well as other IOLA characteristics) of IPAM in comparison to C4.5.

IPAM keeps a table in memory of size $O(k^2)$, where $k$ is the number of distinct commands. Predictions can be performed in constant time (when a list of next command is kept sorted by probability), with updates requiring $O(k)$ time.
Some applications of this method may be able to utilize multiple predictions for a given situation. For example, an adaptive interface might present for easy selection the top five most likely choices that a user would make. Thus, such an interface would want the five predictions that are most likely to be correct. Systems that can generate top-$n$ predictions are also valuable when the predictions are to be combined with other information with predictive value. Since IPAM generates a list of commands with associated probabilities for prediction, we can also compute the average accuracy of the top-$n$ commands for varying values of $n$ (as compared to only the single most likely command). Figures 3.8a and b show that we do get increased performance and that for $n = 5$, the correct command will be listed almost 75% of the time. This makes it possible for an interface designer to consider the trade-off of increased likelihood of listing the correct command versus the increased cognitive load of an interface showing multiple suggestions.

In UNIX command prediction, it is also helpful to be able to perform command completion (that is, taking the first $k$ characters typed and produce the most likely command that is prefixed by those characters). Such a mechanism would enable shells that perform completion when there is a unique command with that prefix (such as `tcsh`) to also be able to perform completion when there are multiple possibilities. Figures 3.9a and b measure the predictive accuracy when given 0-3 initial characters to match when applied to all of the data. (Note that command completion when given 0 initial characters is just command prediction.) As expected, accuracy improves as additional characters of the command are provided to the system. For this task, examining just the first character almost doubles predictive performance (from 40% to close to 80% when predicting the single most likely command).

### 3.6 Discussion

While not shown, the results described apply to both macroaverage performance (shown in most figures) and microaverage performance, although the former is almost always slightly higher. While the results on the first dataset (collected from users of `tcsh`) can be argued as showing the potential for this method (since the selection of `alpha`
was based on the same set), the performance on the larger, and arguably more representative, Greenberg dataset (collected almost ten years earlier from users of `csh`) demonstrates a more believable performance.

Although learning may be performed throughout the history of a user’s actions, the cumulative accuracy of a user does not vary much after an initial training period. Figures 3.10a and b show the performance of the IPAM and Bayes algorithms, respectively over the history of one user. The solid line depicts the current overall average predictive accuracy (from the first command to the current command), while the dashed line shows the variations in predictive accuracy when measured over a window containing only the most recent 30 commands.

We might consider initializing the table in IPAM with useful values rather than starting from scratch. For example, is the performance improved if we start with a table averaged over all other users? This lets us examine cross-user training to leverage the experience of others. Unfortunately, preliminary experiments indicate that, at least for this dataset, starting with the average of all other users’ command prediction tables does not improve predictive accuracy. This result matches with those of Greenberg [Gre93] and Lee [Lee92], who found that individual users were not as well served by systems tuned for best average performance over a group of users.

Figure 3.10: Cumulative performance (for n=3) over time for a typical user.
The goal of our work has been to discover the performance possible without domain knowledge. This can then be used as a benchmark for comparison against ‘strong methods’, or as a base upon which a system with domain-specific knowledge might be built. IPAM’s implementation, in addition, was guided by the characteristics of an IOLA, and thus has other benefits in using limited resources in addition to performance.

3.7 Related Work

The problem of learning to predict a user’s next action is related to work in a number of areas. There are many similarities in the problems studied in plan and goal recognition in the user modeling community (e.g., [Bau96, LE95, Les97]). Such work attempts to model users in terms of plans and goals specific to the given task domain. Efforts in this area typically require that the modeling system know the set of goals and plans in advance. This usually requires a significant human investment in acquiring and representing this domain knowledge. In contrast, our goal is to explore the potential for action prediction in a knowledge-sparse environment (i.e., where user plans are not known or cannot easily be developed).

A smaller number of researchers have, instead, studied methods that have similar goals of generality. Yoshida and Motoda [MY97, YM96, Yos94] apply specially developed machine learning techniques to perform command prediction. This lets them implement speculative execution, and they report fairly high predictive accuracy (albeit on a small amount of real user data). However, much of the success of their work comes from knowing a fair amount about each action a user takes, by using an extension to the operating system that lets them record the I/O accesses (e.g., reading files) that each command performs.

Greenberg [Gre93] and Lee [Lee92] have studied patterns of usage in the UNIX domain, focusing on simple patterns of repetitions. They found that the recurrence rate (the likelihood of repeating something) was high for command lines as well as for commands themselves, but that individual usage patterns varied. More recently, Tauscher and Greenberg [TG97] extended Greenberg’s recurrence analysis to URL revisitation
in World Wide Web browsers. These efforts consider only some simple methods for offering the top-$n$ possibilities for easy selection (such as the most recent $n$ commands).

Stronger methods (including a genetic algorithm-based classifier system) were attempted by Andrews in his master’s thesis [And97] to predict user commands, but he had only a small sample of users with fairly short histories ($\leq 500$) in a batch framework and thus unclear implications. In a conference paper, Debevc, Meyer, and Svecko [DMS97] report on an application called the Adaptive Short List for presenting a list of potential URLs as an addition to a Web browser. This method goes beyond recency-based selection methods, and instead tracks a priority for each URL which is updated after each visitation. The priority for a particular URL is computed essentially as the normalized averages of: a count of document usage, relative frequency of document usage, and the count of highest sequential document usage. This contrasts with our approach most significantly in that it does not consider relationships between documents and thus patterns of usage. Therefore, their Adaptive Short List computes a simplistic ‘most likely’ set of documents without regard to historical context.

A number of researchers have studied the use of machine learning in developing intelligent interfaces incorporating action prediction. For example, WebWatcher [JFM97] predicts which links on a page on World Wide Web a user will select, and Hirashima et al. [HMT98] present a method for context-sensitive filtering, but both systems rely on the precise nature of the artifacts being manipulated (namely decomposable pages of text). Predictive methods in automated form completion [SW96, HS93] are similarly tailored to the specifics on the application. Similarly, the Reactive Keyboard [DWJ90] uses simple history-matching methods for prediction, but at the detailed level of key-presses, allowing for easy word or phrase completion as a document is being typed.

Programming by demonstration [Cyp93b, NM93] also has some similarities to this work. For example, Cypher’s Eager [Cyp93a] can automate explicitly marked loops in user actions in a graphical interface. They, too, are concerned with performance when integrated into a user interface. While our approach is not designed to notice arithmetic progressions in loops, we can find and use the patterns in usage that do not recur as explicit loops and do not require special training by the user. Masui [MN94] also learns
repeated user patterns, requiring the user to hit the ‘repeat’ button when the system should learn or execute a macro.

Sequence prediction is also strongly related to data compression [BCW90] since an algorithm that can predict the next item in a sequence well can also be used to compress the data stream. Indeed, many of the approaches we describe could indeed be used in this fashion, precisely because they apply when only the user’s history is available. However, we differ in that success in compression would only be an interesting phenomenon but not one that we explicitly target for our methods. Perhaps even more importantly, we target methods in which additional information sources can be easily injected. Our methods also are designed to be responsive to concept drift, since we make no assumptions about the stability of a user’s actions over time — something that tends to reduce the usefulness of dictionary or grammar-based compression schemes [NM96].

Laird and Saul [LS94] present the TDAG algorithm for discrete sequence prediction, and apply it to a number of problems, including text compression, dynamic program optimization, and predictive caching. TDAG is based on Markov-trees, but limits the growth of storage by discarding the less likely prediction contexts. It is a fast online algorithm, but it, too, does not explicitly consider the problem of concept drift — each point in a sequence has essentially the same weight as any other point.

While more distantly related, work in anomaly detection for computer systems [Kum95, Lun90] develops ways to quantify a user’s normal behavior so that unusual activity can be flagged as a potential intrusion. Most intrusion detection systems can be categorized as using either statistical-anomaly detection, or rule-based detection. While rule-based expert systems monitor for known attack patterns and thus trigger few false alarms, they are also criticized as encouraging the development of ad hoc rules [ESNP96] and require significant human engineering effort to develop. In contrast, statistical systems traditionally build profiles of normal user behavior and then search for the unusual sequences of events for consideration [DS98, FP99, Lan00]. Unlike most systems that perform anomaly detection by audit-trail processing off-line, our method works online, incrementally updating users’ profiles as additional data arrives and could be augmented to provide user recognition.
Finally, IPAM’s success has fostered work by others. Jacobs and Blockeel [JB01] use Inductive Logic Programming to capture regularities to help build shell scripts. Zukerman et al. [ZAN99] considers user models for plan recognition that predict user’s immediate activities and their eventual goals in games and the Web. Gorniak and Poole [GP00b, GP00a] consider the problem of automatically building models for action prediction without modifying the application. Ruvini and Dony [RD99] claim near IOLA status for their approach to programming by demonstration. Lastly, Korvemaker and Greiner [KG00] explicitly attempt to improve upon IPAM’s results in predicting full UNIX command lines, but find little or no improvement possible.

3.8 Summary

This chapter has introduced a method for action prediction that fulfills the requirements of an Ideal Online Learning Algorithm. Incremental Probabilistic Action Modeling has an average predictive accuracy for UNIX command prediction at least as good as that previously reported with C4.5. It operates incrementally, will remember rare events such as typos, and does not retain a copy of the user’s action history. IPAM can generate top-$n$ predictions, and by weighing recent events more heavily than older events it is able to react to ‘concept-drift’. Finally, its speed and simplicity make it a strong candidate for incorporation into the next adaptive interface.

By only using the previous command, IPAM provides a straightforward method for history-based prediction. In Chapter 4 we examine more complex prediction models and consider the application of such history-based models for the domain of interest — Web request prediction. Prediction solely from history has some limitations, however, and so in Chapters 5 and 6 we consider an alternative — the exploitation of content to assist in modeling user interests for prediction.