Prediction and Discovery of Users' Desktop Behavior

Omid Madani and **Hung Bui** and **Eric Yeh**
Artificial Intelligence Center, SRI International
333 Ravenswood Ave., Menlo Park, CA 94025

Abstract

We investigate prediction and discovery of user desktop activities. The techniques we explore are unsupervised. In the first part of the paper, we show that efficient many-class learning can perform well for action prediction in the Unix domain, significantly improving over previously published results. This finding is promising for various human-computer interaction scenarios where rich predictive features of different types may be available and where there can be substantial nonstationarity. In the second part, we briefly explore techniques for extracting salient activity patterns or *motifs*. Such motifs are useful in obtaining insights into user behavior, automated discovery of (often interleaved) high-level tasks, and activity tracking and prediction.

1 Introduction

An exciting and promising domain for machine learning continues to be the area of action monitoring and personalization. Adaptive systems find applications in task completion, for instance in aiding users' desktop activity, or in reminding users of actions they may have forgotten (assisted living) or proposing alternative possibilities. In this paper, we investigate user action prediction as well as discovery of salient activity. Our approaches are unsupervised, in that the user does not explicitly try to teach the system about her activity. The system simply observes and learns to predict actions or discovers salient patterns (motifs). As in (Davison and Hirsh 1998; Korvemaker and Greiner 2000), our experiments will be primarily in the Unix domain (Greenberg 1988), as much data on a variety of users is available, and we can compare prediction accuracy. We also report on logs of our own desktop activity using the TaskTracer system (Dragunov et al. 2005). The paper is divided into two parts.

1.1 Action Prediction. Here, we focus on the problem of predicting the entire command line that the user would want to type next. Depending on the attributes of the context, such as time of day, current working directory, and recently performed actions, the system may predict that the next command will be "make", or "latex paper.tex", or "cd courses", and so on. The user interface can depend on the particularities of the task. For instance, as explained in (Korvemaker and Greiner 2000), the top five predictions of the system can be tied to the function keys F1 through F5. If the user observes the correct command suggested (*e.g.*, on the top bar of the window), a simple key press executes the action. This can nicely complement other Unix facilities that aid in typing commands. We note that while our experiments are in the Unix domain, similar problems arise in other desktop interaction contexts. For instance, in the Windows domain, the problem can be predicting the next directory to which or from which the user will choose to save an email attachment or load a file (Bao, Herlocker, and Dietterich 2006). Other domains include devices with a limited interface, such as cell phones.

1.1.1 Challenges. The prediction task entails a number of challenges, including (1) high dimensionality and in particular many classes, (2) space and time efficiency, and (3) nonstationarity. We seek algorithms that can capture context well. This means the effective aggregation of the predictions of a rich set of features (predictors). A core aspect that distinguishes our approach is viewing the task as a *many-class* learning problem (multiclass learning with many classes: 100s, 1000s, $\cdots$). The different number of items to predict, entire commands or parameters and so on, ranges in hundreds in our experiments. We employ recently developed efficient *indexing* algorithms (Madani and Connor 2008; Madani and Huang 2008). Efficiency is paramount in this domain: the system must quickly respond and remain adaptive. The algorithms we describe are efficient both in space consumption and in time. As we will see, the prediction problem is significantly nonstationary. In this paper, we evaluate the many-class algorithms in this setting for the first time (as opposed to the more common batch setting). Due to nonstationarity, an important question is whether a learner has sufficient time (learning period) to be able to learn good aggregation of the many features. Indeed (Korvemaker and Greiner 2000), after trying a number of context attributes to improve prediction over using a basic method (as we explain), and failing to improve accuracy, conclude that all the prediction signal may have been gleaned, yet we show that via using improved learning techniques we can gain substantially (an average of 4% to 5% in absolute accuracy improvement, or about 10% relative, over 168 users). We also find that the one-versus-rest linear SVM has very poor accuracy in this domain.

1.2 Motif Discovery. In the second part of the paper, we briefly present our efforts on extracting *salient* repeated patterns, in the form of directed subgraphs, from the action logs (both Unix and desktop). Such patterns provide insights into user behavior, can be used as features for prediction, and can be a basis for learning of higher-level user tasks.

Paper Organization. The next section describes the problem domain, choice of features, algorithms, and evaluation methods. Section 3 presents a variety of experiments and comparisons, Section 4 presents our motif discovery work, Section 5 discusses related work, and Section 6 concludes with future work. An expanded version of this paper with further experiments is in preparation (Madani, Bui, and Yeh 2009).

2 Preliminaries

Our setting is standard multiclass supervised learning, but with many classes and nonstationarity. A learning problem consists of a sequence of instances, each training instance specified by a vector of feature values, x, and the class that the instance belongs to y_x (the *positive class*). We use x to refer to the instance itself as well. Given an instance, a *negative* class is any class $c \neq y_x$. x_f denotes the value of feature f in the vector x. We enforce that $x_f \geq 0$. If $x_f > 0$, we say feature f is *active* in instance x, and denote this aspect by $f \in x$. The number of active features in x is denoted by $|x|$.

2.1 Data sets and Tasks

The bulk of our experiments is performed on a data set collected by Greenberg on Unix usage (Greenberg 1988). This data set is fairly large, collected on 168 users over 2 to 6 months There are four user types: 52 computer scientists, 36 expert programmers, 55 novice programmers, and 25 non programmers. This data set also allows us to compare to previous published results.

We use the terminology of (Korvemaker and Greiner 2000): a (full) command is the entirety of what is entered, this includes the "stub", meaning the "executable" (or action) part, and possibly options and parameters (file and directory names). Thus, in "ls -l home", "ls" is the stub part, "home" is the parameter, and the command is "ls -l home". We will focus on the task of learning to predict the (full) commands, as in (Korvemaker and Greiner 2000). For this task, the number of unique classes (commands) on average per user is roughly 470. As one may expect, this average is highest for computer scientists as a group (around 700 on average), next is experienced programmers (500), and novice programmers and non programmers have about the same (300). It was found that computer scientists were the hardest to predict as a group (Korvemaker and Greiner 2000). As in (Korvemaker and Greiner 2000), over all the users, we obtain 303,628 episodes (commands entered).

2.2 The Choice of Features and Representation

We experimented with the following feature types, which we break into two broad categories. *Action features* reflect what the user has done recently. The ones we used are the commands typed at times $t-1$ and $t-2$,[1] as well as only the stub and only parameter portions of the command, at time $t-1$. We also found the *start-session* feature to be useful (each user's log is broken into many sessions in which the user begins the session, and after some interaction, exits the session). The start-session feature was treated like other action features. Thus, after starting, the start session feature would be the "command" taken at time $t-1$ and after one command entered, it would be the command taken at time $t-2$.

The other type of features may be called *State Features*, *i.e.*, those that reflect the "state" the user or the system is in. State features do not change as quickly as action features. We used the current working directory as one state feature. Importantly, we also used a "default" or an *always-active-feature*, a feature with value of 1 in every episode and that would be updated in every update (but not necessarily in every episode). This feature has an effect similar to the LIFO strategy (see Section 2.4). Episodes have less than 10 features active on average. The very first episode of a user has three features active: the always-active feature, start session feature, and a feature indicating that the last command had no parameter (the "NULL" parameter).[2]

We did not attempt to predict the start of session nor the exiting action. The recorded logs also indicated whether an error occurred. A significant portion of the commands led to errors (*e.g.*, mistyped commands) (about 5% macro averaged over users). We did not treat them differently. These decisions allowed to us to compare to the results of (Korvemaker and Greiner 2000).

2.3 Online Evaluation

All the algorithms we evaluate output a ranking of their predictions. As in (Korvemaker and Greiner 2000; Davison and Hirsh 1998), unless specified otherwise, we report on the *cumulative online* accuracy (ranking) performance of R_1 (standard accuracy, or one minus zero-one error) and R_5 (accuracy in top five predictions), computed for each user, then averaged over all the 168 users (macro averaged). For this evaluation, for each user, the sequence of episodes (instances or commands) is ordered naturally in the order they were typed. Formally, let k_{x_i} be the rank of the positive class y_{x_i} for the i-th instance x_i in the sequence. Let $\mathbb{I}\{k_{x_i} \leq k\} = 1$ iff $k_{x_i} \leq k$, and 0 otherwise (Iverson bracket). Then R_k (R_1 or R_5) for a given user with M instances is

$$R_k = \frac{1}{M} \sum_{1 \leq i \leq M} \mathbb{I}\{k_{x_i} \leq k\} \qquad (1)$$

On each instance, first the system is evaluated (predicts using the features of the instance), then the system trains on that instance (the true class is revealed). The algorithms we present in Section 2.4 perform a simple efficient prediction

[1] As separate features, *i.e.*, even if the same command was typed again, it has a different feature id for times $t-1$ and $t-2$.

[2] We could also have added the home directory.

and possible update on each instance. We note that the system always fails on the first instance, and more generally on any instance for which the true class has not been seen before. As in (Korvemaker and Greiner 2000), such instances are included in the evaluation. On average per user, about 17% of commands are not seen before. This number goes to over 25% for parameter portion of commands, and down to 6% for stubs. We will also report on some variations, such as when the instances are permuted, to obtain insights on algorithms' performances in the more common "stationary" evaluation setting.

2.4 Algorithms

The main learning algorithm that we propose for the prediction task, shown Figure 1, employs exponential moving average updaring, and we refer to it as EMA ("Emma"). On every instance (episode), the algorithm first predicts the class (in our case, the user's next command line) and, if margin threshold is not met, updates each active feature using exponential moving average updating. The connection (prediction) weight from feature f to class c is denoted by $w_{f,c}$. Updates are kept efficient as each active feature resets weights (connections) that fall below a threshold w_{min}. In our experiments, w_{min} is set to 0.01; thus, the maximum out-degree of a feature, denoted d, is 100. The connections of a feature are implemented via a dynamic sorted linked list. The first time a feature is seen (in some episode), it is not connected to any class (all its connection weights are implicitly 0). We have termed the learned representation an *index*, *i.e.*, a mapping that connects each feature to a relatively small subset of the classes (the features "index" the classes). Both prediction and updating on an instance x take time $O(d|x|\log(d|x|))$. Several properties of EMA were explored in (Madani and Huang 2008). It was shown that EMA updating is equivalent to a quadratic loss minimization for each feature, and a formulation for numeric feature values, as given in Figure 1, was derived. An update is performed only if a margin threshold δ_m is not met (a kind of mistake-driven updating). This leads to down weighing the votes of redundant features, more effective aggregation, and ultimately better generalization (Madani and Connor 2008).

We compare EMA against the method used in (Korvemaker and Greiner 2000) as well as linear SVMs, and another indexing variant OOZ (Madani and Huang 2008). In the approach of (Korvemaker and Greiner 2000), a restricted version of EMA updating was deployed, which was referred to as the alpha updating rule at the time, or AUR. AUR can also be viewed as a nonstationary version of the bigram method in statistical language modeling. A similar strategy was used in (Davison and Hirsh 1998), but for the task of stub prediction. In AUR, only the last command is used as a predictor, with one exception: if that command does not give at least five predictions, a default predictor is used to fill in the remaining of the five slots. Whenever a command appears as a feature, it is updated, and the default predictor is updated in every episode (using exponential moving average). Thus, the differences with our presentation of EMA are that we use multiple features and aggregate their votes (their method did not sum, only merge the predictions if need be), we update in a mistake-driven manner (in particular we use a margin threshold), we drop weak edges, and we l_2 normalize the feature vectors. We also compare against a last-in-first-out strategy, or LIFO. LIFO keeps track of the last five *unique* commands and reports them in that (reverse chronological) order, so that the command typed last (time $t-1$) is reported first.[3]

EMA($x, y_x, \beta, \delta_m, w_{min}$)
1. $\forall c, s_c \leftarrow \sum_{f \in x} x_f w_{f,c}$ **/* Score classes for ranking/prediction */**
2. $\delta_x \leftarrow s_{y_x} - s_{c'}$, **where** $s_{c'} \leftarrow \max_{c \neq y_x} s_c$ **/* compute margin */**
3. if $(\delta_x < \delta_m)$, **then** $\forall f \in x$ **do: /* If margin not met, update */**
/* All active features' connections are decayed, then the connections to true class is boosted */
3.1 $\forall c$, $w_{f,c} \leftarrow (1 - x_f^2 \beta) w_{f,c}$ **/*** $0 < x_f \leq 1$ ***/**
3.2 $w_{f,y_x} \leftarrow w_{f,y_x} + x_f \beta$ **/* Boost connection to true class */**
3.3 If $w_{f,c} < w_{min}$, **then /* drop small weights */**
$w_{f,c} \leftarrow 0$

Figure 1: The EMA ("Emma") learning algorithm, which uses *exponential moving average* updating and a margin threshold. β is a learning rate or a "boost" amount, $0 < \beta \leq 1$, and x_f is the "activity" level of active feature f, $0 < x_f \leq 1$. The end effect after an update is that the weight of the connection of feature f to the positive class, w_{f,y_x}, is strengthened. Other connections are weakened and possibly zeroed (dropped).

	$R1$	$R5$	$R1_{>1k}$	$R5_{>1k}$
LIFO	0.075 $\pm$0.05	0.42 $\pm$0.15	0.07 $\pm$0.04	0.40 $\pm$0.15
AUR	0.28 $\pm$0.12	0.47 $\pm$0.14	0.28 $\pm$0.12	0.47 $\pm$0.14
EMA	0.30 $\pm$0.11	0.51 $\pm$0.13	0.30 $\pm$0.11	0.52 $\pm$0.13

Table 1: Accuracies on full command prediction. $R1_{>1k}$ and $R5_{>1k}$ are averages on users with no less than 1000 episodes (EMA: $\beta = 0.15, \delta_m = 0.15, w_{min} = 0.01$).

3 Experiments

Table 1 shows the performance comparisons between LIFO, AUR, and EMA. For all these methods, an entire evaluation on all 168 users takes less than 2 minutes on a laptop. We observe that the effective aggregation of predictors (or capturing more context) can lead to a substantial boost in accuracy, in particular in R_5. The performance on users with more than 1000 instances appears to lead to some improvement for EMA (but not for AUR)[4], over those users with fewer than 1000. We note that Kovermaker and Greiner tried a number of ways and features to improve on AUR, but their methods did not lead to a performance gain (Korvemaker and Greiner 2000). In Figure 2, the performance on each

[3] Another similar baseline is reporting the five most frequent commands seen so far. As (Korvemaker and Greiner 2000) show, that strategy performs substantially worse than LIFO (several percentage points below in accuracy), underscoring the nonstationarity aspect.

[4] For AUR, we obtained the same accuracies of (Korvemaker and Greiner 2000).

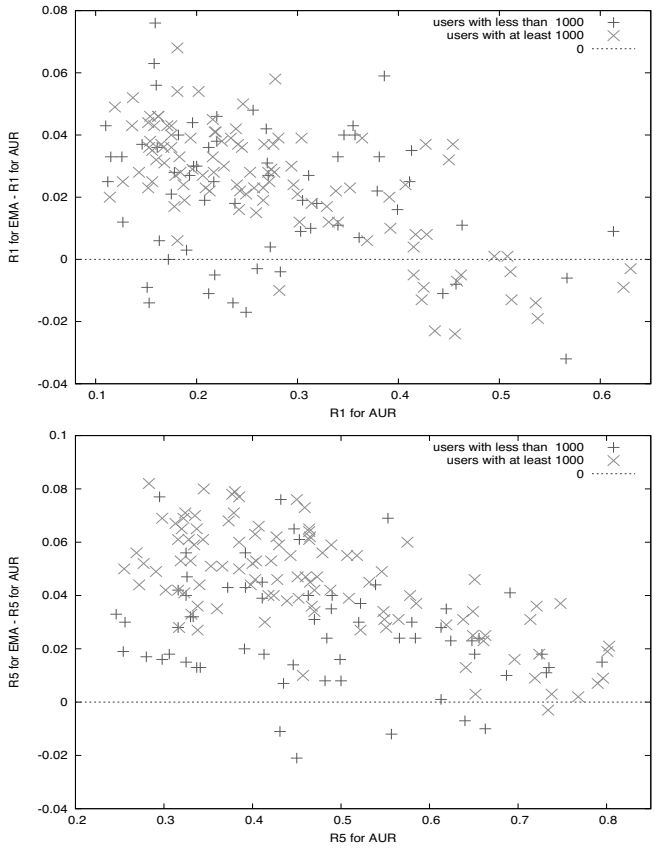

Figure 2: The spread of performance over users. For each user, the x-axis is the performance of AUR (R1 or R5), the y-axis is the difference from EMA (above 0 means higher performance for EMA).

user is depicted by a point where the x-coordinate is R1 (or R5) using AUR, and the y-coordinate is that same value subtracted from R1 (or R5) obtained using EMA. We observe that R_5 values in particular are significantly improved using EMA, and the improvements tend to be higher (in absolute as well as relative value) with lower absolute value of the performance. The values for users with fewer than 1000 instances shows somewhat higher spread, as may be expected. We compared the number of wins, when results on the same user are paired, and performed a sign test. On R1, EMA wins over AUR on 141 of the users, loses on 26 users, and ties on 1 (Figure 2). On R5, winning is more robust: EMA wins in 162 cases and loses in 6. Both comparisons are significant with over 99.9% confidence (P value is < 0.001).

3.1 Ablation Experiments on EMA

Figure 3 shows (macro) average R5 performance as a function of learning rate and margin. We notice the performances are fairly close: the algorithm is not heavily dependent on the parameters. It is also interesting to note that relatively high learning rates of 0.1 and above give the best or very

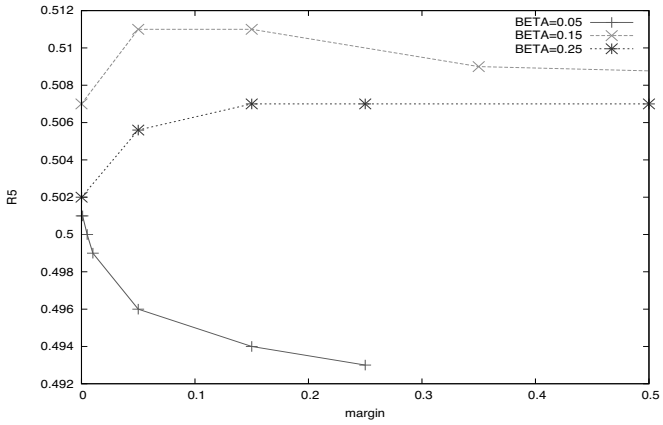

Figure 3: Plots of R5 performance (macro average over users) as a function of choices for margin threshold and the learning rate.

good results here. In previous studies in text cateogrization (Madani and Huang 2008), lower learning rates (of 0.05 or 0.01 and below) gave the best results. When we compare the best overall R5 average for learning rate of 0.15 versus 0.05, we obtain 120 wins (for learning rate of 0.15), 41 losses and 7 ties (where the average is respectively $R5$ of 0.51 for learning rate of 0.15 and 0.50 for learning rate of 0.05). As might be expected, the best results are obtained when the margin (threshold) is not at the extremes of 0 (pure mistake-driven "lazy" updating) or very high (always update). If we include more features, such as stub and parameters from time $t-2$ or features from earlier time points, tending to increase redundancy and uninformativeness, performance somewhat degrades (the average remains 0.5 or around it), and the selection of margin becomes more important. It may be possible to adjust (learn) the learning rate or margin over time as a function of user behavior for improved performance.

With the default parameters of 0.15 for both learning rate and margin, we raised the minimum weight threshold w_{min} to 0.05 and 0.1 (from default of 0.01), and respectively obtained R_5 of 0.509 (small degradation) and 0.45 (substantial degradation).

3.2 Feature Utilities

In the results given here, the default parameters are used and all features are available (as explained in Section 2.2) except for those that we explicitly say we remove. The performance is fairly robust to removal of various feature types: the remaining feature types tend to compensate. All the features tend to help the average performance somewhat. Removing the stub or the (full) command at time $t-1$ yields the largest drop in performance, leading to just over 0.50 average R_5. If we remove both, we get an $R_5 = 0.486$. The other features in order of importance are current directory, always active, start session, and parameter at time $t-1$. Removing any such type of feature results in degradation of about 0.005 (from the maximum of just over 0.51).