Efficient Online Learning and Prediction of Users' Desktop Actions

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

Abstract

We investigate prediction of users' desktop activities in the Unix domain. The learning techniques we explore do not require explicit user teaching. We show that simple efficient *many-class* learning can perform well for action prediction, significantly improving over previously published results and baselines. This finding is promising for various human-computer interaction scenarios where a rich set of potentially predictive features is available, where there can be many different actions to predict, and where there can be considerable nonstationarity.

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. In our approach, the user does not explicitly try to teach or inform the system about her activity. The system simply observes and learns to predict her actions. As in [Davison and Hirsh, 1998; Korvemaker and Greiner, 2000], our experiments will be in the Unix domain [Greenberg, 1988], as much data on a variety of users is available, and we can compare prediction accuracy to previous methods.

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 notices the correct command suggested (*e.g.*, on the top bar of the window), a single key press executes the whole action. An alternative mode could be autocompletion. These possibilities can nicely complement other Unix facilities that aid in typing commands. We note that while our experiments are in the Unix domain, the approach is general, and 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 *et al.*, 2006]. Other domains include devices with a limited interface, such as cell phones.

Challenges. The prediction problem entails several challenges, including (1) high dimensionality and in particular many classes, *i.e.*, many possible candidates to predict, in addition to many features, (2) space and time efficiency, and (3) nonstationarity, as a user's task can change several times a day, and projects can change and evolve over longer periods. We seek algorithms that can capture context well. This means the effective aggregation of the predictions of a rich set of features (predictors). We apply recently developed *indexing* algorithms to this task [Madani and Connor, 2008; Madani and Huang, 2008]. A core aspect that distinguishes our approach is viewing the task as a *many-class* learning problem (multiclass learning with numerous classes: 100s, 1000s, $\cdots$). The different number of items to predict, entire commands or the parameter portion and so on, ranges in hundreds in our experiments. Furthermore, the set of classes is not known a priori and grows over time. Efficiency is paramount in this domain: the system must quickly respond while remaining 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 a nonstationary setting for the first time, as opposed to the more common stationary or batch setting. Due to nonstationarity, an important question is whether a learner has sufficient time, *i.e.*, adaptation period, to be able to learn effective aggregation of the many features. If a prediction task exhibits too much nonstationarity, aka concept drift, the learner may not have enough time to learn the effective associations between the features and classes. 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 experiment with

other learners and find that the one-versus-rest linear SVM has very poor accuracy in this task.

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 discusses related work, and Section 5 concludes with future work. An expanded version of this work including further experiments is in preparation.

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. Our experiments are performed on a data set collected by Greenberg on Unix usage [Greenberg, 1988]. This data set is fairly large, collected on 168 users with four types of backgrounds (52 computer scientists, 36 expert programmers, 55 novice programmers, and 25 non programmers) over 2 to 6 months. 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 was 469. As one may expect, this average was highest for computer scientists as a group (around 700 on average), next was experienced programmers (500), and novice programmers and non programmers had 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 used the following feature types, which we break into two broad categories. *Action features* reflect what the user has done recently. We used 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 $t-1$. Further history did not change overall accuracy significantly. We also found the *start-session* feature 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, immediately after starting a session, 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 as we explain). 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 such as 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 methods 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) performances R_1 and R_5, computed for each user, then averaged over all the 168 users (macro averaged). R_1 performance on a given user is simply standard accuracy or one minus zero-one error, and R_5 is accuracy in top five predictions. For this evaluation, for each user, the sequence of episodes (instances or commands) is given in the order they were typed. Formally, given a ranking, let k_{x_i} be the rank of the positive class y_{x_i} for the i-th instance x_i in the sequence (k_{x_i} is infinite if y_{x_i} is not in the ranking). 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 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 also compare to MaxEnt and linear one-versus-rest SVMs in a more traditional stationary or "batch" setting, on a subset of the users, as explained in Sec. 3.5.

[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.

2.4 Algorithms. The main learning algorithm that we propose for the prediction task, shown Figure 1, employs exponential moving average updating, and we refer to it as EMA ("Emma"). On every instance, the algorithm first predicts the class (in our task, the user's next command line) and, if a margin threshold is not met, updates the prediction connections of the active features. EMA is an online learning algorithm which learns a nonnegative weight matrix. Assume the features correspond to the rows, the classes correspond to the columns, and a matrix entry $w_{f,c}$ (entry in row f, column c), corresponds to the prediction weight from feature f to class c. The property that makes learning and classification efficient in the face of many classes, distinguishing the approach from other learning methods, is that the matrix is kept row-sparse (most entries in every row are kept at zero and are not explicitly represented). Next, we briefly explain the algorithm and our implementation further.

Each feature keeps track of its list of connections to a relatively small subset of the classes (a row in the weight matrix). The first time a feature is seen, in some episode, it is not connected to any class (its connections list is empty, and all its connection weights are implicitly 0). The prediction connections of each feature is implemented via a dynamic linked list. Updates are kept efficient as each active feature resets weights that fall below a threshold w_{min} to zero, which results in removing the corresponding connection entry from the list. In our experiments, w_{min} is set to 0.01; thus, in case of a single class per instance and Boolean feature values, the maximum out-degree of a feature (the list size), denoted d, would be 100. Both prediction and updating on an instance x take time $O(d|x|\log(d|x|))$. We call 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). There are other possibilities for index learning [Madani and Huang, 2008; Madani and Connor, 2008], although EMA may be the best for nonstationary situations. Several properties of EMA are 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 (predictions) of redundant features thus more effective aggregation of features' predictions, and ultimately better generalization [Madani and Connor, 2008]. In Section 3 we explore the effects of the various parameters such as the learning rate (β), the margin threshold (δ_m), and the choice of features. EMA was not evaluated for nonstationary tasks.

We compare EMA against the method used in [Korvemaker and Greiner, 2000]. In that work, effectively a very restricted version of EMA updating was deployed, which was referred to as the alpha updating rule at the time, or AUR. AUR could also be viewed as a "nonstationary bigram" technique (stationary n-gram techniques are used in statistical language modeling, *e.g.*, [Rosenfeld, 2000]). 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 for efficiency, 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 (most recent) five *unique* commands and reports them in that order, *i.e.*, in reverse chronological, so that the command at time t-1 is top ranked.[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 the connected classes**
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 first decayed, then
// the connections to the 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 tiny weights**
$w_{f,c} \leftarrow 0$ **// remove from list of connections**

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, $0 < \beta \leq 1$, x_f is the "activity" level of active feature f, $0 < x_f \leq 1$, and $w_{f,c}$ denotes the connection weight from feature f to class c. 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 from feature's connections list).

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

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

3 Experiments

In this section, we first report on accuracies and comparisons to AUR and LIFO. We then present experiments on effects of parameter choices on EMA, feature utilities, context (directory) change, nonstationarity and the need for continued learning, and comparisons to other methods.

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 lap-

[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.

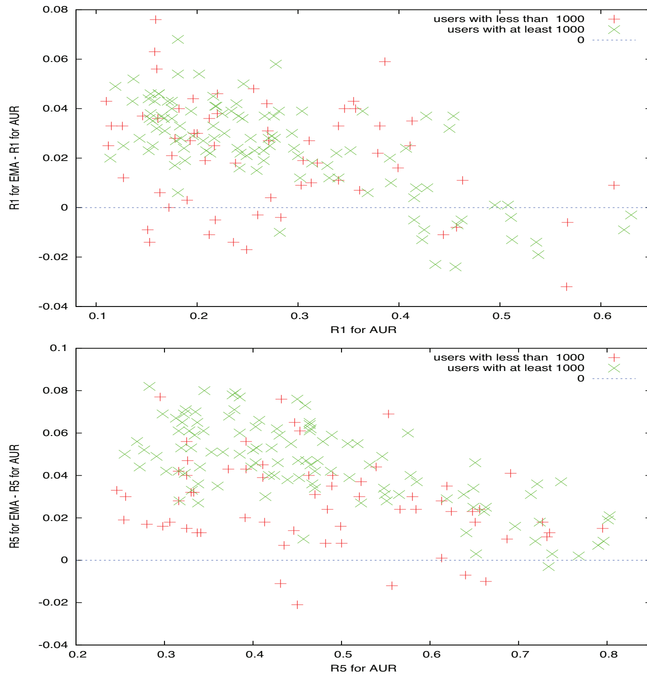

Figure 2: The spread of performance over users. For each user, the x-axis is the performance of AUR (R1 or R5), the y-axis is the difference (subtraction) from EMA's (above 0 means higher performance for EMA).

top. 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), over those users with fewer than 1000. We note that Korvemaker 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 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).

Discussion of Accuracies. Though we have observed substantial improvements, the overall performance on R_5 may seem still somewhat low, raising the question of the utility of the methods. In general, prediction accuracy in similar tasks such as statistical language modeling, where there are many classes, is low, and researchers use smoother measure

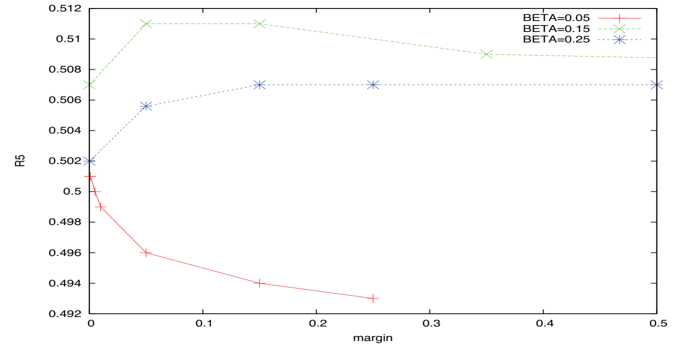

Figure 3: Plots of R5 performance (macro average over users) as a function of choices for margin threshold and the learning rate.

such as perplexity to measure improvements (*e.g.*, [Rosenfeld, 2000]). We also note that our accuracy is an all-or-nothing measure, and to replicate and compare to previous results, we included the instances that corresponded to user errors. Furthermore, for around 20% of instances, the class was not observed before (Sections 2.2 and 2.3). We note also that R5 performance on stub prediction or parameter prediction alone was higher, exceeding 70%. Prediction for auto completion or a piecemeal prediction strategy could prove more useful (*e.g.*, first predict the stub, and after the intended stub is obtained, predict the parameter(s)).

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 good results here. In previous studies in text categorization [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 $R5$ 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 or "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 ($\beta = 0.15, \delta_m = 0.15$) are used and all features

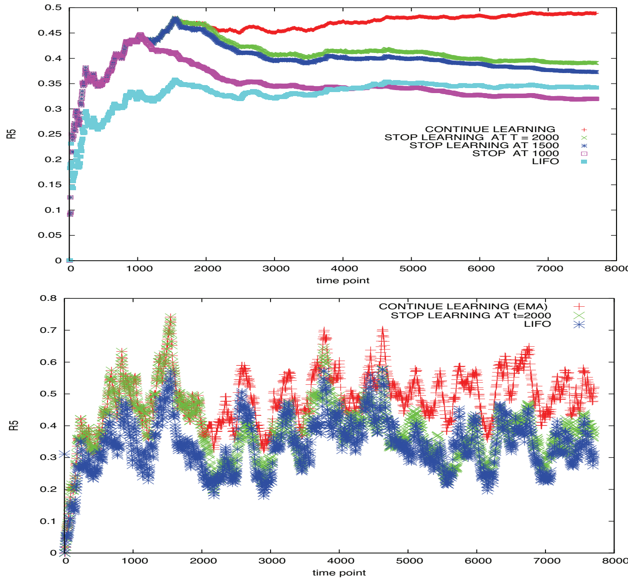

Figure 4: The need for sustained learning: Performance plots (R5 for scientist 52) EMA and LIFO, as well as a few cases in which EMA learning is stopped after a certain episode (1000, 1500, and 2000) (top: cumulative performance, bottom: moving averages of R5 with β of 0.01). Continued learning is required to sustain prediction performance.

	R1cd	R5cd	R1no	R5no
LIFO	0.002	0.37	0.037	0.26
EMA	0.43	0.56	0.19	0.38

Table 2: Example of performance after a possible context change: R1 and R5 accuracies on instances immediately after directory changes (R1cd and R5cd), and otherwise (no directory change, R1no and R5no) for scientist 36. Out of just over 12k episodes, this user had 1285 many "directory-changed" episodes. Performance of EMA increases substantially under "directory-changed".

are available (Sect. 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).

3.3 Accuracy when Context Changes. A promise of the learning approach is to robustly continue predicting after a context change, such as a change in task/project (as long as the context has been experienced before). A rough candidate for context change is when the directory changes. On a few users, we measured the accuracies immediately after the current directory changes, and compared to when it doesn't (majority of instances). We observed the pattern given in Table

	SVM	EMA	MaxEnt	EMAonline
R1	0.01	0.210	0.236	0.284
R5	0.215	0.404	0.423	0.514

Table 3: Comparisons between EMA, one-versus-rest linear SVM, and MaxEnt, on 21 selected users who had at least 1000 episodes. The first 80% was used for training, remaining 20% for test, except for the last column, where EMA was allowed to update on every test instance after it predicted (as described in Sect. 2.3).

2. For EMA, the performance improves substantially when the directory changes, while for LIFO, R1 degrades, and R5 does not improve as much. The degradation of R1 for LIFO is understandable: the last command (change directory) is seldom repeated immediately. A majority of the immediate actions (from roughly 1300 such, 10% of total) are 'ls', but some other successful top predictions of EMA include 'fg' and 'emacs' (we observed the user mistyped 'emac' in one case, but the correct "emacs" had been predicted).

3.4 Nonstationarity and the Need for Continued Adaptation. Figure 4 shows R_5 performances as a function of time for scientist 52, for whom we have about 8000 total episodes. While the learning curve seems to reach local maxima at say around 1000 to 2000, we see the need for continued learning to *sustain* the performance: if we stop the learning, the performance curve takes a downward turn. The system's performance eventually degrades to below that of LIFO if learning is stopped.

3.5 Comparisons with other Methods. Davison and Hirsh [Davison and Hirsh, 1998] showed that their AUR rule for stub prediction (using the predictions of the previous stub) outperformed batch learning algorithms such as decision trees and naive Bayes as well as stationary variants of AUR (standard bigram method) on their task. EMA outperforms AUR on stub prediction as well (about 3% improvement in R5).

We also compared performance (on full command, as before) to one-versus-rest training of linear SVM classifiers [Chang and Lin, 2001], and MaxEnt, which is a multiclass learner [Manning and Klein, 2003]. We chose 21 users with a thousand or more instances. For each user, the first 80% of episodes are kept for training, the remaining for test (chronological splits). SVM and MaxEnt employ batch optimization techniques and are considerably slower: per user, SVM training takes close to a minute, while MaxEnt takes over 15 minutes[4], while complete EMA evaluation on all the 21 users takes under 20 seconds (we ran EMA for a single-pass). The accuracies are given in Table 3. One-versus-rest SVM trails substantially (although we used many regularization parameters, $C \in \{0.5, 1, 5, 10, \cdots\}$, and are reporting the best). MaxEnt does better than EMA, suggesting there may be room for improvement. However we note that if we let EMA continually update on remaining 20% (last column), it does best. We note that, besides the inefficiency drawbacks, MaxEnt is likely to underperform in the online evaluation scheme, as it is inherently a batch technique. We have also compared EMA to other online indexing variants (*e.g.*, OOZ [Madani

[4] There may be faster implementations available for MaxEnt and SVM. In particular, for linear SVMs substantially faster algorithms are available.

and Huang, 2008]) and other online techniques, such as multiclass perceptron [Crammer and Singer, 2003]. EMA has performed best in accuracy, possibly due to its better bias for nonstionarity. These observations, along with many other experiments, are presented in the longer technical report version of this paper (in preparation).

4 Related Work

Modeling user activities on the desktop has been an active topic of research in the AI and user modeling community. It is generally accepted that good predictive models of user activities play a central part in building an intelligent adaptive interface that can potentially help to increase user productivity. The availability of the UNIX data [Greenberg, 1988] has led to a number of efforts in building predictive models of the data as well as facilitated direct and objective comparisons among different algorithms [Davison and Hirsh, 1998; Korvemaker and Greiner, 2000]. These authors have also observed the issue of nonstationarity in the data. For the related task of automated email foldering, see also [Segal and Kephart, 2000] on the importance of incremental learning, and [Bekkerman and McCallum, 2004] on the importance of taking account of the nonstationarities in evaluation. Work in modeling user interactions with Windows can be traced to the LookOut system [Horvitz, 1999], which can observe user email and calendar activities and attempt to learn a model of calendar-related emails. More recent work in this area includes the TaskTracer system [Dragunov *et al.*, 2005] and BusyBody system [Kapoor and Horvitz, 2007]. These systems can capture a wide range of Windows and application events. The recorded events have been used for training various prediction tasks such as folder prediction [Bao *et al.*, 2006], task-switch prediction [Shen *et al.*, 2006] and user business [Kapoor and Horvitz, 2007], although some amount of explicit user feedback has been required at times (such as specifying the current user tasks).

EMA is an index learning method, and index learning was proposed to address the challenges of learning under many classes. There exist numerous methods for multiclass learning, such as nearest neighbors, decision trees, naive Bayes, multiclass SVMs, and MaxEnt (see *e.g.*, [Hastie *et al.*, 2001; Manning and Klein, 2003; Crammer and Singer, 2003]). The issue of space and time efficiency when facing many classes had not been the focus of past work. See [Madani and Connor, 2008; Madani and Huang, 2008] for further discussions of the relations of indexing to other learning approaches as well as empirical comparisons.

5 Summary and Future Work

We presented a simple efficient learning technique to effectively aggregate the predictions of features when there are many possible classes to predict and in a nonstationary setting. The approach is promising in other rich human computer interaction scenarios. We hope to further advance the algorithms as well as explore applications.

Acknowledgments. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185/0004, in the context of research on an adaptive cognitive system (CALO), under DARPA's PAL (Perceptive Assistant that Learns) program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or the Air Force Research Laboratory.

References

[Bao *et al.*, 2006] X. Bao, J. Herlocker, and T. Dietterich. Fewer clicks and less frustration: Reducing the cost of reaching the right folder. In *IUI*, 2006.

[Bekkerman and McCallum, 2004] R. Bekkerman and A. McCallum. Automatic categorization of email into folders: benchmark experiments on Enron and SRI corpora. Technical report, Technical Report IR-418. CIIR, UMass, 2004.

[Chang and Lin, 2001] C. C. Chang and C. J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[Crammer and Singer, 2003] K. Crammer and Y. Singer. A family of additive online algorithms for category ranking. *Jouornal of Machine Learning Research*, 3:1025–1058, 2003.

[Davison and Hirsh, 1998] B. D. Davison and H. Hirsh. Predicting sequences of user actions. In *AAAI-98/ICML'98 Workshop on Predicting the Future: AI Approaches to Time Series Analysis*, 1998.

[Dragunov *et al.*, 2005] A. Dragunov, T. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *IUI*, pages 75–82, New York, NY, USA, 2005. ACM.

[Greenberg, 1988] S. Greenberg. Using Unix: collected traces of 168 users. Technical report, University of Calgary, Alberta, 1988.

[Hastie *et al.*, 2001] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer-Verlag, 2001.

[Horvitz, 1999] Eric Horvitz. Principles of mixed-initiative user interfaces. In *ACM CHI*, 1999.

[Kapoor and Horvitz, 2007] A. Kapoor and E. Horvitz. Principles of lifelong learning for predictive user modeling. In *Proc. Eleventh Conference on User Modeling (UM 2007)*, 2007.

[Korvemaker and Greiner, 2000] B. Korvemaker and R. Greiner. Predicting UNIX command lines: Adjusting to user patterns. In *AAAI/IAAI*, 2000.

[Madani and Connor, 2008] O. Madani and M. Connor. Large-scale many-class learning. In *SIAM Conference on Data Mining*, 2008.

[Madani and Huang, 2008] O. Madani and J. Huang. On updates that constrain the number of connections of features during learning. In *ACM KDD*, 2008.

[Manning and Klein, 2003] C. Manning and D. Klein. *Optimization, Maxent Models, and Conditional Estimation without Magic*. Tutorial at HLT-NAACL and ACL, 2003.

[Rosenfeld, 2000] R. Rosenfeld. Two decades of statistical language modeling: Where do we go from here? *IEEE*, 88(8), 2000.

[Segal and Kephart, 2000] R. B. Segal and J. O. Kephart. Incremental learning in SwiftFile. In *ICML*, 2000.

[Shen *et al.*, 2006] J. Shen, L. Li, T. Dietterich, and J. Herlocker. A hybrid learning system for recognizing user tasks from desktop activities and email messages. In *IUI*, 2006.