Detecting and Correcting User Activity Switches: Algorithms and Interfaces

Jianqiang Shen, Jed Irvine, Xinlong Bao, Michael Goodman, Stephen Kolibaba, Anh Tran, Fredric Carl, Brenton Kirschner, Simone Stumpf, Thomas G. Dietterich

1148 Kelley Engineering Center, School of EECS, Oregon State University
Corvallis, OR 97331, U.S.A.
shenj@eecs.oregonstate.edu

ABSTRACT

The TaskTracer system allows knowledge workers to define a set of activities that characterize their desktop work. It then associates with each user-defined activity the set of resources that the user accesses when performing that activity. In order to correctly associate resources with activities and provide useful activity-related services to the user, the system needs to know the current activity of the user at all times. It is often convenient for the user to explicitly declare which activity he/she is working on. But frequently the user forgets to do this. TaskTracer applies machine learning methods to detect undeclared activity switches and predict the correct activity of the user. This paper presents *TaskPredictor2*, a complete redesign of the activity predictor in TaskTracer and its notification user interface. TaskPredictor2 applies a novel online learning algorithm that is able to incorporate a richer set of features than our previous predictors. We prove an error bound for the algorithm and present experimental results that show improved accuracy and a 180-fold speedup on real user data. The user interface supports negotiated interruption and makes it easy for the user to correct both the predicted time of the task switch and the predicted activity.

ACM Classification Keywords

I.2.1 Artificial Intelligence: Applications and Expert Systems—*Office automation*; H.5.2 Information Interfaces and Presentation: User Interfaces—*Graphical user interfaces*

Author Keywords

Activity recognition, online learning, resource management

General Terms

Algorithms, Design, Human Factors.

INTRODUCTION

The TaskTracer system [6, 13] is an intelligent activity management system [9, 10, 2, 7] that helps knowledge workers manage their work based on two assumptions: (a) the user's work can be organized as a set of ongoing activities such as "Write TaskTracer Paper" or "CS534 Class", (b) each activity is associated with a set of *resources*. "Resource" is an umbrella term for documents, folders, email messages, email contacts, web pages and so on. TaskTracer collects events generated by the user's computer-visible behavior, including events from MS Office, Internet Explorer, Windows Explorer, and the Windows XP operating system. The user can declare a "current activity" (via a "Task Selector" UI), and TaskTracer records all resources as they are accessed and associates them with the current declared activity. TaskTracer then configures the desktop in several ways to support the user:

- **Task Explorer**: This presents a unified view of all of the resources associated with the current activity and makes it easy to launch those resources in the appropriate application. This supports interruption recovery by showing the user the most recent resources that they were working on for each activity (as opposed to the global Recent Documents facility of Windows).
- **Folder Predictor**: This modifies the Open/Save dialogs so that they are activity-aware. Folder Predictor predicts for all folders f associated with the current activity, the three folders that jointly minimize the expected number of clicks to reach f. The Open/Save dialog is initialized in the most likely of these folders, and shortcuts to all three folders are made available in the dialogue box [1].
- **Time Reporting**: This allows the user to produce a report showing how much time was spent on each activity during the most recent day, week, month, year, etc.
- **Email Tagging**: Incoming and outgoing email messages are automatically tagged (via another machine learning component) with the activity (or activities) to which they are associated.

All of these services (except incoming email tagging) require TaskTracer to know the current activity of the user. TaskTracer provides a taskbar component, the Task Selector, that makes it easy for the user to explicitly declare the current activity. However, when the user forgets to do this, resources become associated with incorrect activities. This will cause errors in the resources displayed in Task Explorer, incorrect folder predictions, and errors in time reporting.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IUI'09, February 8-11, 2009, Sanibel Island, Florida, USA. Copyright 2009 ACM 978-1-60558-331-0/09/02...$5.00.

To address this problem, we have previously developed two machine learning methods for detecting activity switches and predicting the current activity of the user [13, 12]. The first of these was a "single window classifier" that based its predictions only on the file path (or URL) of the current resource and the title of the active window. An advantage of this was that the predictions could be computed immediately and rapidly each time the user switched focus from one window to another. The disadvantage was that with such impoverished features, the accuracy of the predictions was not very high.

The second learning method was based on analyzing the *sequence* of recently-visited resources. It first applied the single-window classifier to each of these resource visits. Then it detected activity switches by applying a Viterbi algorithm with a fixed switch cost. This led to somewhat improved accuracy but had the side effect of introducing a potentially significant delay between the time of the switch and the time the switch was detected. This led in turn to major usability problems. The UI that we developed for notifying the user consisted of a pop-up window. By the time the switch was detected, the user was typically deeply engaged in the new activity, so that the pop-up window created an expensive interruption. Furthermore, the pop-up window UI only allowed the user to accept or correct the prediction. However, if the true switch occurred at time t and the prediction was accepted at time t', then all resources visited between t and t' would be incorrectly associated with the old activity. There was no way—short of going to Task Explorer and dragging the incorrectly-associated resources to the right activity—to fix these errors.

An additional drawback of both of these learning methods is that they employed a batch SVM algorithm [8, 3]. This must store all of the training examples and reprocess all of them when retraining is required. SVM training time is roughly quadratic in the number of examples, so the longer TaskTracer is used, the slower the training becomes. Furthermore, activity prediction is a multi-class prediction problem over a potentially large number of classes. For example, our busiest user worked on 299 different activities over a four-month period. To perform multi-class learning with SVMs, we employed the one-versus-rest approach. If there are K classes, then this requires learning K classifiers. If there are N examples per class, then the total running time is roughly $O(N^2K)$. This is not practical in an interactive, desktop setting.

The goal of the redesign of the Task Predictor (and of this paper) was to address these four problems:

- Accuracy: How can we improve the accuracy of the activity switch predictor?
- Memory/CPU Cost: Can we switch to a more efficient learning algorithm that requires constant memory and CPU time to train and make predictions?
- Prediction Delay/Interruption Cost: Can we develop a user interface that is able to manage the delay between the time a switch occurs and the time it is predicted? How can this UI minimize interruption costs?
- Retroactive Association Changes: When the user changes the time of an activity switch, how can that change be reflected in the associations between resources and activities?

This paper presents *TaskPredictor2*, a new switch detection system that fixes all of these problems.

SYSTEM ARCHITECTURE

TaskPredictor2 captures a set of richer contextual information and provides a better user experience. It consists of the following components (see Figure 1):

- The **Association Engine** manages the associations between resources and activities. Each time the user visits a resource, an event is triggered. In response, the Association Engine starts a 20-second timer. If the resource is still in focus after 20 seconds and if it has not been previously associated with the current declared activity, then it is automatically associated with the current declared activity. In addition to recording the association in a data base table, the Association Engine also places an entry in an Association History database table that keeps track of the time the association was established, the id of the resource, and the id of the activity. When the user accepts or corrects an activity switch, the UI sends an "AssociationHistoryRevisions" event specifying the time interval that is affected by the change and the id of the activity that now should be associated with this interval. The Association Engine then revises the associations for this time interval.
- The **State Estimator** monitors desktop events. Once each minute, it computes an information vector that summarizes the state of the desktop and sends this to the Switch Detector.
- The **Switch Detector** analyzes the information vectors provided by the State Estimator and converts them into a collection of feature vectors as described below. These are processed by the learned classifier to predict whether there has been an undeclared activity switch and (if so) the time of the switch, the id of the new activity, and the prediction confidence. This is passed to the Notification Controller and the UI.
- The **Notification Controller** chooses when and how to notify the user with the goal of minimizing both the interruption cost and association errors. When the system becomes confident that the user forgot to declare an activity switch, the user might be busy with something. Notifying the user right then would have a high interruption cost.
- The **UI** presents the switch notification to the user and provides the user a variety of ways to respond to (and correct) the predicted switch. It provides feedback to the Switch Detector so that it can retrain its predictors.

The current version of the Notification Controller employs a simple heuristic to determine when to issue the switch alert: If the system is confident that an activity switch has occurred

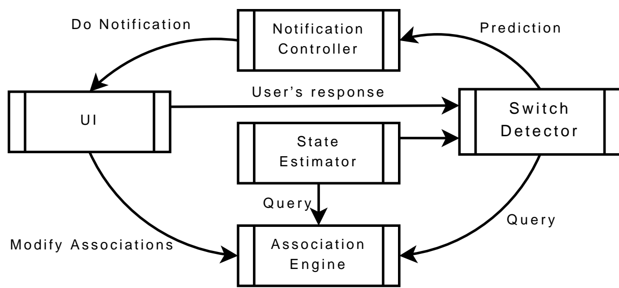
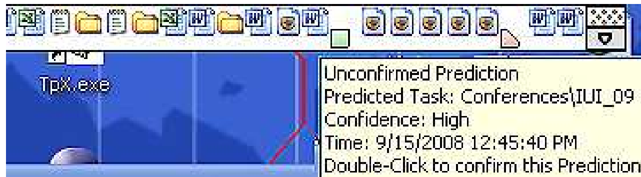


Figure 1. The system architecture of the activity recognition system. Arrows indicate information flows.



a) Notification Controller issues a switch alert (the red triangle) + tool tip.

b) Hovering the mouse over the resource icon gives more information.

c) If a switch is changed, UI will ask for confirmation (the Save/Cancel box).

Figure 2. Interactions between TaskPredictor2 and the user.

and the user is at an operation boundary and likely has low interruption cost (e.g., opening a file dialog box or switching focus to another window), then an alert should be given. The alert takes three forms depending on the confidence of the predictor: (a) displaying a red switch icon (see Figure 2), (b) shaking the timeline UI for one second, or (c) opening a pop-up window. The rationale is that highly confident predictions are more likely to be correct, so ignoring them is more likely to lead to association errors.

Figure 2 shows an example of the interactions between UI and the user. The UI consists of two rows of icons. The upper row displays one icon for each time that a resource was visited by the user (in the order that these visits occurred). Multiple visits result in multiple icons—however, very brief visits are ignored. Information about each resource (path name/URL, associated activity) is shown as a tool tip if the user hovers the mouse over the resource icon (see Figure 2b). The user can also double-click on an icon to cause that resource to be opened and brought into focus.

The lower row displays one icon for each activity switch. The shape and color of the icon indicates the type of switch. User-declared switches are displayed as green boxes. Unconfirmed switch predictions are displayed as red triangles. Confirmed switch predictions are displayed as a green "house". A tool tip shows the activity name, the type of switch, and the time of the switch. The user can drag and drop a switch icon to change the time of the switch. However, we do not permit the drag operation to go past an other existing switch time.

In the examples shown in Figure 2, the user first worked on activity *IUI-09*. Then he explicitly switched to *Reading News* and browsed some webpages. The green box indicates this switch. After reading the news, he resumed working on activity *IUI-09*. Unfortunately he forgot to explicitly indicate that switch. TaskPredictor2 detected this undeclared switch and decided that it was the right time to notify the user. So it drew a red triangle to indicate this switch and shook UI to attract the user's attention. The user then hovered the mouse over the triangle to see the predicted task (Figure 2a). To confirm that the prediction is correct, the user can double-click the switch icon. The icon then turns into a "house" (see Figure 2c). If the prediction is wrong (i.e., there was no switch at all), the user can right click and select "Remove" from a pop-up menu. If the predictor has correctly identified an activity switch but predicted the wrong activity, the user can right click and choose "Change the Task". This brings up an Activity Chooser UI. If the switch is predicted at the wrong time point, the user can drag the switch icon to the correct point in time. As the icon moves, the UI provides tool tips to show the full name of the resource, the time it was accessed, and the activity to which it is currently assigned so that the user can find the right switch point. In all cases, the UI requests confirmation ("Save") or allows the user to cancel out of the changes ("Cancel"). If the user ignores a notification, TaskPredictor2 does nothing. In most cases, if the user ignores a notification long enough, it will disappear because the user will continue to accumulate time on an open resource, and this resource will become associated with the current declared activity. This will, in turn, cause TaskPredictor2 to stop predicting that an activity switch has occurred, because it now has evidence that the open resource is associated with the declared activity.

As in our earlier method, there can be a substantial delay between the time the user switches activities and the time TaskPredictor2 detects this. A key design goal of this UI was to support this by providing asynchronous (negotiated) interruption and retroactive correction of incorrect associations. After the user confirms a change, the information is sent to the Association Engine to update the resource-activity associations.

AN ONLINE LEARNING ALGORITHM

To address the accuracy and CPU cost of our previous predictors, we adopted an efficient online learning algorithm and provided it with a richer set of features. Online learning algorithms have been shown efficient in many large-scale problems. The algorithm is based on the Perceptron family of algorithms [11] and, particularly, on the Passive-Aggressive algorithm [4].

Feature Design and Scoring Function

The State Estimator monitors various desktop events (Open, Close, Save, SaveAs, Change Window Focus, etc.). Every s seconds (default s=60) or when the user declares an activity switch, it computes an information vector $\mathbf{X}_t$ describing the time interval t since the last information vector was computed. This information vector is then mapped into feature vectors by two functions: $\mathbf{F}_A : (\mathbf{X}_t, y_j) \rightarrow R^k$ and $\mathbf{F}_S : (\mathbf{X}_t) \rightarrow R^m$. The first function $\mathbf{F}_A$ computes *activity-specific* features for a specified activity y_j; the second function $\mathbf{F}_S$ computes *switch-specific features*. The activity-specific features include

- Strength of association of the active resource with activity y_j: if the user has explicitly declared that the active resource belongs to y_j (e.g., by drag-and-drop in TaskExplorer), the current activity is likely to be y_j. If the active resource was implicitly associated with y_j for some duration (which happens when y_j is the declared activity and then the resource is visited), this is a weaker indication that the current activity is y_j.
- Percentage of open resources associated with activity y_j: if most open resources are associated with y_j, it is likely that y_j is the current activity.
- Importance of window title word x to activity y_j. Given the bag of words Ω, we compute a variant of TF-IDF [4] for each word x and activity y_j: $\text{TF}(x,\Omega) \cdot \log \frac{|\overline{S}|}{\text{DF}(x,\overline{S})}$. Here, $\overline{S}$ is the set of all feature vectors not labeled as y_j, TF(x,Ω) is the number of times x appears in Ω and DF$(x,\overline{S})$ is the number of feature vectors containing x that are not labeled y_j.

These activity-specific features are intended to predict whether y_j is the current activity. The switch-specific features predict the likelihood of a switch. They include

- Number of resources closed in the last s seconds: if the user is switching activities, many open resources will often be closed.
- Percentage of open resources that have been accessed in the last s seconds: if the user is still actively accessing open resources, it is unlikely there is an activity switch.
- The time since the user's last explicit activity switch: immediately after an explicit switch, it is unlikely the user will switch again. But as time passes, the likelihood of an undeclared switch increases.

To detect an activity switch, we adopt a sliding window approach: at time t, we use two information vectors ($\mathbf{X}_{t-1}$ and $\mathbf{X}_t$) to score every pair of activities for time intervals $t-1$ and t. Given an activity pair $\langle y_{t-1}, y_t\rangle$, the scoring function g is defined as

$$g(\langle y_{t-1}, y_t\rangle) = \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_{t-1}, y_{t-1}) + \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_t, y_t) + \phi(y_{t-1} \neq y_t)\left(\Lambda_2 \cdot \mathbf{F}_S(\mathbf{X}_{t-1}) + \Lambda_3 \cdot \mathbf{F}_S(\mathbf{X}_t)\right),$$

where $\Lambda = \langle \Lambda_1, \Lambda_2, \Lambda_3\rangle \in \mathbb{R}^n$ is a set of weights to be learned by the system, $\phi(p) = -1$ if p is true and 0 otherwise, and the dot $(\cdot)$ means inner product. The first two terms of g measure the likelihood that y_{t-1} and y_t are the activities at time $t-1$ and t (respectively). The third term measures the likelihood that there is no activity switch from time $t-1$ to t. Thus, the third component of g serves as a "switch penalty" when $y_{t-1} \neq y_t$.

We search for the $\langle \hat{y}_1, \hat{y}_2\rangle$ that maximizes the score function g. If $\hat{y}_2$ is different from the current declared activity and the score is larger than a specified threshold, then a switch prediction and the score are sent to the Notification Controller. At first glance, this search over all pairs of activities would appear to require time quadratic in the number of activities. However, the following algorithm computes the best score in linear time:

$$\begin{aligned}
y^*_{t-1} &:= \arg\max_y \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_{t-1}, y) \\
A(y^*_{t-1}) &:= \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_{t-1}, y^*_{t-1}) \\
y^*_t &= \arg\max_y \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_t, y) \\
A(y^*_t) &= \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_t, y^*_t) \\
S &= \Lambda_2 \cdot \mathbf{F}_S(X_{t-1}) + \Lambda_3 \cdot \mathbf{F}_S(\mathbf{X}_t) \\
y^* &= \arg\max_y \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_{t-1}, y) + \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_t, y) \\
AA(y^*) &= \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_{t-1}, y^*) + \Lambda_1 \cdot \mathbf{F}_A(\mathbf{X}_t, y^*)
\end{aligned}$$

Each pair of lines can be computed in time linear in the number of activities. We assume $y^*_{t-1} \neq y^*_t$. To compute the best score $g(\langle \hat{y}_1, \hat{y}_2\rangle)$, we compare two cases: $g(y^*, y^*) = AA(y^*)$ is the best score for the case where there is no change in the activity from $t-1$ to t, and $g(y^*_{t-1}, y^*_t) = A(y^*_{t-1}) + A(y^*_t) + S$ if there is a switch. When $y^*_{t-1} = y^*_t$, we can compute the best score for the "no switch" case by tracking the top two scored activities at time $t-1$ and t.

Regularized Passive-Aggressive Algorithm

In TaskPredictor2 we decided to adopt an error-driven online approach [11] to train Λ. Online learning algorithms only need to process the most recent observation to update the classifier. They have very limited requirements for CPU time and memory. Their efficiency has made them popular for many large-scale learning problems. An advantage of perceptron-style algorithms is that they are "conservative"—that is, they only retrain when an error is made. This further reduces the CPU cost of training, and it may also contribute to prediction accuracy by avoiding overfitting on data points that are already correctly classified.

We further chose to adopt a modification of the *Passive-Aggressive* (*PA*) algorithm [4]. The basic idea of this algorithm is that when an error is committed and feedback is received, the algorithm updates the weights as little as possible while ensuring that the error would not be committed again. More specifically, it chooses a step size that is exactly large enough to ensure that the classifier will correctly classify the erroneous case with a margin of 1. This is highly desirable in an intelligent user interface, because it avoids the problem where the user provides feedback but the algorithm only takes a small step in the right direction, so the user must repeatedly provide feedback until enough steps

have been taken to fix the error, which has been reported to be extremely annoying.

The standard Passive-Aggressive algorithm works as follows: Let the real activities be y_1 at time $t-1$ and y_2 at time t, $\langle \hat{y}_1, \hat{y}_2 \rangle$ be the highest scoring incorrect activity pair. When the system makes an error, it would update Λ based on the following constrained optimization problem:

$$\Lambda_{t+1} = \arg\min_{\Lambda \in \mathbb{R}^n} \frac{1}{2} \|\Lambda - \Lambda_t\|_2^2 + C\xi^2$$
$$\text{subject to } \; g(\langle y_1, y_2 \rangle) - g(\langle \hat{y}_1, \hat{y}_2 \rangle) \geq 1 - \xi. \qquad (1)$$

The first term of the objective function, $\frac{1}{2} \|\Lambda - \Lambda_t\|_2^2$, says that Λ should change as little as possible (in Euclidean distance) from its current value Λ_t. The constraint, $g(\langle y_1, y_2 \rangle) - g(\langle \hat{y}_1, \hat{y}_2 \rangle) \geq 1 - \xi$, says that the score of the correct activity pair should be larger than the score of the incorrect activity pair by at least $1 - \xi$. Ideally, $\xi = 0$, so that this enforces the condition that the margin (between correct and incorrect scores) should be 1.

The purpose of ξ is to introduce some robustness to noise. We know that inevitably, the user will occasionally make a mistake in providing feedback. This could happen because of a slip in the UI or because the user is actually inconsistent about how resources are associated with activities. In any case, the second term in the objective function, $C\xi^2$, serves to encourage ξ to be small. The constant parameter C controls the tradeoff between taking small steps (the first term) and fitting the data (driving ξ to zero). Crammer, et al. [4] show that this optimization problem has a closed-form solution, so it can be computed in time linear in the number of features and the number of classes.

The Passive-Aggressive algorithm is very attractive. However, one risk is that Λ can still become large if the algorithm runs for a long time, and this could lead to overfitting. Hence, we modified the algorithm to include an additional regularization penalty on the size of Λ. The modified optimization problem is the following:

$$\Lambda_{t+1} = \arg\min_{\Lambda \in \mathbb{R}^n} \frac{1}{2} \|\Lambda - \Lambda_t\|_2^2 + C\xi^2 + \frac{\alpha}{2} \|\Lambda\|_2^2$$
$$\text{subject to } \; g(\langle y_1, y_2 \rangle) - g(\langle \hat{y}_1, \hat{y}_2 \rangle) \geq 1 - \xi. \qquad (2)$$

The third term in the objective function, $\frac{\alpha}{2} \|\Lambda\|_2^2$, encourages Λ to remain small. The amount of the penalty is controlled by another constant parameter, α.

We now derive a closed-form update rule for this modified Passive-Aggressive algorithm. First, let us expand the g terms. Define $\mathbf{Z}_t = \langle \mathbf{Z}_t^1, \mathbf{Z}_t^2, \mathbf{Z}_t^3 \rangle$ where

$$\begin{aligned}
\mathbf{Z}_t^1 =& \mathbf{F}_A(\mathbf{X}_{t-1}, y_1) + \mathbf{F}_A(\mathbf{X}_t, y_2) \\
& - \mathbf{F}_A(\mathbf{X}_{t-1}, \hat{y}_1) - \mathbf{F}_A(\mathbf{X}_t, \hat{y}_2) \\
\mathbf{Z}_t^2 =& (\phi(y_1 \neq y_2) - \phi(\hat{y}_1 \neq \hat{y}_2))\, \mathbf{F}_S(\mathbf{X}_{t-1}) \\
\mathbf{Z}_t^3 =& (\phi(y_1 \neq y_2) - \phi(\hat{y}_1 \neq \hat{y}_2))\, \mathbf{F}_S(\mathbf{X}_t).
\end{aligned}$$

Plugging this into the above optimization problem allows us to rewrite the inequality constraint as the following simple form:

$$\Lambda \cdot \mathbf{Z}_t \geq 1 - \xi.$$

We can then derive the following result:

LEMMA 1. *The optimization problem (2) has the closed-form solution*

$$\Lambda_{t+1} := \frac{1}{1+\alpha}(\Lambda_t + \tau_t \mathbf{Z}_t), \qquad (3)$$

where

$$\tau_t = \frac{1 - \Lambda_t \cdot \mathbf{Z}_t + \alpha}{\|\mathbf{Z}_t\|_2^2 + \frac{1+\alpha}{2C}}. \qquad (4)$$

A detailed proof is presented in the Appendix. The update rule (3) can be viewed as shrinking the current weight vector Λ_T and then adding in the incorrectly-classified training example with a step size of $\tau_t/(1+\alpha)$. The step size is determined (in the numerator of (4)) by the size of the error $(1 - \Lambda_t \cdot \mathbf{Z}_t$ and (in the denominator) by the squared length of the feature vector and a correction term involving α and C.

The time to compute this update is linear in the number of features. Furthermore, the cost does not increase with the number of classes, because the update involves comparing only the predicted and correct classes.

It is worth asking how much accuracy is lost in order to obtain such an efficient online algorithm compared to a batch algorithm (or, more generally, to the best possible algorithm). By extending existing results from computational learning theory, we can provide a partial answer to this question. Let us assume that the length of each vector $\mathbf{Z}_t$ is no more than some fixed value R:

$$\|\mathbf{Z}_t\|_2 \leq R.$$

This is easily satisfied if every feature has a fixed range of values. Suppose there is some other algorithm that computes a better weight vector $\mathbf{u} \in \mathbb{R}^n$. Let $\ell_t = \max\{0, 1 - \Lambda_t \cdot \mathbf{Z}_t\}$ be the hinge loss of Λ_t at time t. The hinge loss is the amount by which the constraint in Problem (2) fails to be satisfied. The hinge loss will be at least 1 for misclassified examples. It is between 0 and 1 for examples that are correctly classified by only a small margin less than 1. Similarly, let $\ell_t^* = \max\{0, 1 - \mathbf{u} \cdot \mathbf{Z}_t\}$ denote the hinge loss of $\mathbf{u}$ at iteration t. With these definitions, we can obtain a result that compares the accuracy of the online algorithm after T examples to the total hinge loss of the weight vector $\mathbf{u}$ after T examples:

THEOREM 1. *Assume that there exists a vector $\mathbf{u} \in \mathbb{R}^n$ which satisfies $h = \frac{1-\alpha^2}{R^2 + \frac{1+\alpha}{2C}} - \frac{\alpha}{2+\alpha} \|\mathbf{u}\|_2^2 > 0$. Given $\alpha < 1$, the number of prediction mistakes made by our algorithm is bounded by $m \leq \frac{1}{h} \|\mathbf{u}\|_2^2 + \frac{2C}{h(1+\alpha)} \sum_{t=0}^{T} (\ell_t^*)^2$.*

A detailed proof is presented in the Appendix. This theorem tells us that the number of mistakes made by our online al-

gorithm is bounded by the sum of (a) the squared hinge loss of the ideal weight vector $\mathbf{u}$ multiplied by a constant term and (b) the squared Euclidean norm of $\mathbf{u}$ divided by h. This bound is not particularly tight, but it does suggest that as long as $\mathbf{u}$ is not too large, the online algorithm will only make a constant factor more errors than the ideal (batch) weight vector.

Theoretical analysis can provide one more insight into the behavior of the online algorithm. Because of the penalty $\frac{\alpha}{2}\|\Lambda\|^2$ on the weight vector, it turns out that this algorithm automatically decreases the influence of older training instances. Let I_t be the set of instances $\mathbf{Z}_i$ such that $i < t$ and $\mathbf{Z}_i$ caused an update. From Lemma 1, the learned score function at iteration t can be rewritten as

$$g_t(\mathbf{Z}) = \text{sign}\left(\sum_{i \in I_t} \frac{\tau_i}{(1+\alpha)^{|I_t \setminus I_i|}} \mathbf{Z}_i \cdot \mathbf{Z}\right) \quad (5)$$

where $|I_t \setminus I_i|$ is the number of elements in the set difference of I_t and I_i. This indicates that the algorithm shrinks the influence of the old observations and puts more weight on the recent ones.

Pragmatic Issues in Selecting Training Examples

If the user were behaving correctly all the time, then every time the user declared an activity switch, we would obtain a training example of a correct switch, and whenever the user did not declare an activity switch, we would get an example of a non-switch. But of course the whole reason to include an intelligent switch predictor is because the user makes mistakes. When can we reliably train the classifier? We answer this question as follows. We create positive examples of activity switches whenever (a) the user declares an activity switch or (b) the user confirms a predicted switch (possibly after modifying the predicted activity and/or the predicted switch point). We create negative examples of activity switches for (a) the first $d = 10$ minutes after a declared or accepted activity switch and (b) the first $d = 10$ minutes after the user Removes a predicted switch. The rationale for the 10-minute interval is that the declared activity is usually correct immediately after the user switches to the activity. But as the amount of time since the confirmed switch increases, it becomes more and more likely that the user has switched without making an explicit declaration. Using a fixed 10-minute window can be viewed as a crude Bayesian prior over the likelihood of the user forgetting to declare a switch. An interesting direction for future research would be to learn such a model based on the actual behavior of the user.

There is an additional pragmatic issue that results from the combination of affordances provided by the UI. In particular, consider the "dragging" affordance. Consider Figure 2a, where a prediction has been made immediately after five Firefox web pages and just prior to two Word documents. Suppose that this prediction was incorrect in that there was no activity switch. Then the user would right-click and select "Remove" from the pop-up menu. Our current system treats this as a negative training example, and this sets up the

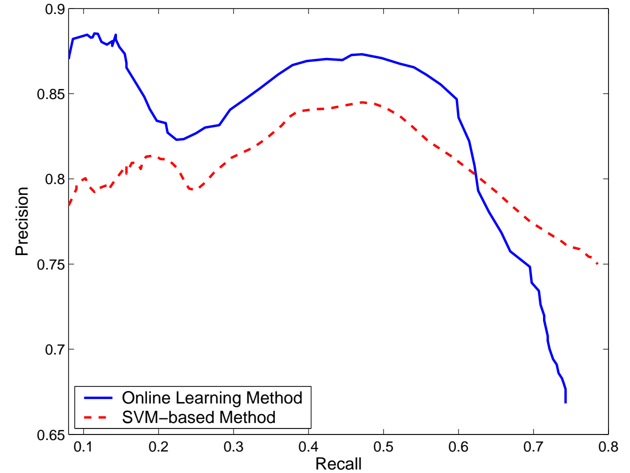


Figure 3. User 1: Precision of different learning methods as a function of the recall, created by varying the threshold.

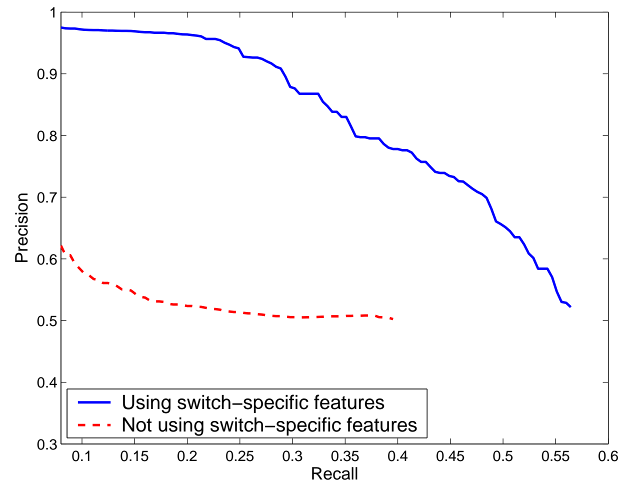


Figure 4. User 2: Precision of different learning methods as a function of the recall, created by varying the threshold.

start of a 10-minute segment in which all user behavior will be trusted and converted to training examples.

However, we can infer more from the user's Remove action. Suppose that there had been an activity switch at an earlier point (e.g., immediately after the first two Firefox web pages). In that case, the UI would have permitted the user to drag the red triangle to that point and then confirm. Therefore, under the assumption that the user knows about this affordance, we can infer from the fact that the user did not do this, that there was *not* an activity switch anywhere in the time interval between the first Firefox page and the last Word file. Our implementation does not currently do this, but it raises an interesting question for future research. We only discovered this issue after the UI was deployed. It would be interesting to develop a design tool that could discover these kinds of counter-factual inferences earlier in the design process.

EXPERIMENTAL EVALUATION

We deployed TaskTracer on Windows machines in our research group and collected data from two regular users, both of whom were fairly careful about declaring switches. In addition, an SVM-based version of TaskPredictor2 was running throughout this time, and the users tried to provide feedback to the system throughout.

The first user (User 1) is a "power user", and this dataset records 4 months of daily work, which involved 299 distinct activities, 65,049 instances (i.e., information vectors), and 3,657 activity switches. The second user (User 2) ran the system for 6 days, which involved 5 activities, 3,641 instances, and 359 activity switches.

To evaluate the online learning algorithm, we make the assumption that these activity switches are all correct, and we perform the following simulation. Suppose the user forgets to declare every fourth switch. We feed the information vectors to the online algorithm and ask it to make predictions. A switch prediction is treated as correct if the predicted activity is correct and if the predicted time of the switch is within 5 minutes of the real switch point. When a prediction is made, our simulation provides the correct time and activity as feedback.

The algorithm parameters were set based on experiments with non-TaskTracer benchmark sets. We set $C = 10$ (which is a value widely-used in the literature) and $\alpha = 0.001$ (which gave good results on the benchmark data sets).

Performance is measured by *precision* and *recall*. Precision is the number of switches correctly predicted divided by the total number of switch predictions, and Recall is the number of switches correctly predicted divided by the total number of undeclared switches. We obtain different precision and recall values by varying the score confidence threshold required to make a prediction.

The results comparing our online learning approach with our previous approach [12] are plotted in Figures 3 and 4. Our previous approach only uses the bag of words from the window titles and pathname/URL to do inference. The new approach incorporates much richer contextual information to detect activity switches. This makes our new approach more accurate. Compared with the previous approach which must train multiple binary SVMs, the online learning approach is also much more efficient. On an ordinary PC, it only took 4 minutes to make predictions for User 1's 65049 instances while the previous approach needed more than 12 hours!

Qualitative Evaluation and the User Experience

Within our group, we are currently using the SVM-based version of the system. It is exactly the same as the system described in this paper, except that it employs the LibSVM implementation [3] to train the classifier. Table 1 shows the actual results of the four-month usage period for User 1. A total of 138 predictions were made. Of these, 42 (30.4%) were ignored (probably because the user didn't notice them when the UI was buried under another window). Of the 96 non-ignored predictions, the user removed 36 (37.5%) of them. The user confirmed 37 (38.6%) without change, 3 more (3.1%) after changing the activity, 19 (19.8%) after changing the time, and 1 more (1.0%) after changing both the time and the activity.

Qualitatively, User 1 reported that TaskPredictor2 was very accurate. When it makes a prediction, the prediction is usually either correct or sensible, by which we mean that the predicted activity is one that is related to the correct activity. For example, a challenging case for activity prediction arises when the user is working on activity A2 and accesses a resource that was previously only known to be associated with activity A1. Two common scenarios where this arises are (a) using a document as a template for a new activity (e.g., opening up an old syllabus, editing it to change the course name, meeting time, etc., and then saving it) and (b) assembling summary documents (project reports, curriculum vitae, annual performance reviews) by copying information from multiple source documents. In such cases, TaskPredictor2 will predict that the user has switched to activity A1. This is a sensible prediction, and users are not surprised by these kinds of predictions. However, from the user's point of view, he/she is working on activity A2, so the user typically Removes or ignores such predictions. This causes the source documents to become associated with A2. It isn't clear whether this result is good or bad. A good aspect of this is that at some later time, when the user returns to working on activity A2, TaskTracer will include the source documents in the list of resources associated with A2. This can provide an answer to the question "Where did I get those slides from?". A bad aspect of this is that when the user accesses the source documents at some later time, TaskPredictor2 may predict that the user is switching to activity A2 when the user is really switching to activity A1. We have attempted to deal with this case by including the "strength of association" feature in the information vector. Ideally, if a resource is more strongly associated with A1 than with A2, then this will cause the predictor to predict a switch to A1 in this case.

CONCLUSION

This paper has described TaskPredictor2, a novel activity-switch predictor for the desktop environment. Its user interface displays activity switches [14, 5] on a time-line that shows accessed resources. This gives the user contextual hints and makes it easy for the user to adjust the time at which an activity switch (declared or predicted) actually occurred. Instead of simply reminding the user about the switch, it allows the user to edit the activity switch history and thereby adjust resource-activity associations. The learning component exploits rich contextual information, including both activity-specific features and switch-specific features, to predict activity switches. The predictor parameters are trained via a novel online learning algorithm. This algorithm is more accurate, more efficient in memory, and runs 180 times faster than our previous SVM-based method.

Table 1. Usage Statistics for User 1; SVM Version

Switch Type	Ignored	Removed	Confirmed: No Change	Confirmed: Change Activity	Confirmed: Change Time	Confirmed: Change Both
Low Confidence	1	1	1	1	2	0
Medium Confidence	10	2	4	0	3	0
High Confidence	31	33	32	2	14	1
User-Declared	N/A	5	N/A	2	21	1

Acknowledgments

This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185/0004. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA, or the Air Force Research Laboratory (AFRL).

APPENDIX

Proof of Lemma 1

PROOF. The Lagrangian of the optimization problem in (2) is

$$L(\Lambda,\xi,\tau) = \frac{1}{2}\|\Lambda-\Lambda_t\|_2^2 + C\xi^2 + \frac{\alpha}{2}\|\Lambda\|_2^2 + \tau(1-\Lambda\cdot\mathbf{Z}_t-\xi), \quad (6)$$

where $\tau \geq 0$ is the Lagrange multiplier. Differentiating this Lagrangian with respect to the elements of Λ and setting the partial derivative to zero gives

$$\Lambda = \frac{1}{1+\alpha}\Lambda_t + \frac{\tau}{1+\alpha}\mathbf{Z}_t. \quad (7)$$

Differentiating the Lagrangian with respect to ξ and setting the partial derivative to zero gives

$$\xi = \frac{\tau}{2C}. \quad (8)$$

Expressing ξ as above and replacing Λ in Eq 6 with Eq 7, the Lagrangian becomes

$$\begin{aligned} L(\tau) =& \frac{1}{2}\left\|\frac{\tau}{1+\alpha}\mathbf{Z}_t - \frac{\alpha}{1+\alpha}\Lambda_t\right\|_2^2 + \frac{\tau^2}{4C} \\ &+ \frac{\alpha}{2}\left\|\frac{\tau}{1+\alpha}\mathbf{Z}_t + \frac{1}{1+\alpha}\Lambda_t\right\|_2^2 \\ &+ \tau\left(1-\frac{\Lambda_t\cdot\mathbf{Z}_t}{1+\alpha} - \frac{\tau\|\mathbf{Z}_t\|_2^2}{1+\alpha} - \frac{\tau}{2C}\right). \end{aligned}$$

By setting the derivative of the above to zero, we get

$$1 - \frac{\tau}{2C} - \frac{\tau}{1+\alpha}\|\mathbf{Z}_t\|_2^2 - \frac{(\Lambda_t\cdot\mathbf{Z}_t)}{1+\alpha} = 0 \quad (9)$$

$$\Rightarrow \quad \tau = \frac{1-(\Lambda_t\cdot\mathbf{Z}_t)+\alpha}{\|\mathbf{Z}_t\|_2^2 + \frac{1+\alpha}{2C}}. \quad (10)$$

Combining Eq 7 and Eq 10, completes the proof. □

Proof of Theorem 1

PROOF. Let $\Delta_t = \|\Lambda_t - \mathbf{u}\|_2^2 - \|\Lambda_{t+1} - \mathbf{u}\|_2^2$. We can prove the bound by lower and upper bounding $\sum_t \Delta_t$.

Since Λ_0 is a zero vector and the norm is always non-negative, $\sum_t \Delta_t = \|\Lambda_0 - \mathbf{u}\|_2^2 - \|\Lambda_T - \mathbf{u}\|_2^2 \leq \|\Lambda_0 - \mathbf{u}\|_2^2 = \|\mathbf{u}\|_2^2$.

Obviously, only if $t \in I_T$, $\Delta_t \neq 0$. We will only consider this case here. Let $\Lambda_t' = \Lambda_t + \tau_t\mathbf{Z}_t$, $\Lambda_{t+1} = \frac{1}{1+\alpha}\Lambda_t'$. Δ_t can be rewritten as

$$(\|\Lambda_t - \mathbf{u}\|_2^2 - \|\Lambda_t' - \mathbf{u}\|_2^2) + (\|\Lambda_t' - \mathbf{u}\|_2^2 - \|\Lambda_{t+1} - \mathbf{u}\|_2^2) = \delta_t + \epsilon_t. \quad (11)$$

We will lower bound both δ_t and ϵ_t. Let $\psi^2 = (1+\alpha)/2C$. For δ_t, we have

$$\begin{aligned} \delta_t &= \|\Lambda_t - \mathbf{u}\|_2^2 - \|\Lambda_t + \tau_t\mathbf{Z}_t - \mathbf{u}\|_2^2 && (12)\\ &= \|\Lambda_t - \mathbf{u}\|_2^2 - \|\Lambda_t - \mathbf{u}\|_2^2 \\ &\quad - 2\tau_t\mathbf{Z}_t\cdot(\Lambda_t - \mathbf{u}) - \|\tau_t\mathbf{Z}_t\|_2^2 && (13)\\ &\geq 2\tau_t\ell_t - 2\tau_t\ell_t^* - \tau_t^2\|\mathbf{Z}_t\|_2^2 && (14)\\ &\geq 2\tau_t\ell_t - 2\tau_t\ell_t^* - \tau_t^2\|\mathbf{Z}_t\|_2^2 - (\psi\tau_t - \ell_t^*/\psi)^2 && (15)\\ &= 2\tau_t\ell_t - \tau_t^2(\|\mathbf{Z}_t\|_2^2 + \psi^2) - (\ell_t^*)^2/\psi^2 && (16) \end{aligned}$$

We get Eq 15 because $(\psi\tau_t - \ell_t^*/\psi)^2 \geq 0$. Plugging the definition of τ_t and considering $\ell_t \geq 1$, we get

$$\delta_t \geq \frac{\ell_t^2 - \alpha^2}{\|\mathbf{Z}_t\|_2^2 + \frac{1+\alpha}{2C}} - \frac{2C}{1+\alpha}(\ell_t^*)^2 \quad (17)$$

$$\geq \frac{1-\alpha^2}{R^2 + \frac{1+\alpha}{2C}} - \frac{2C}{1+\alpha}(\ell_t^*)^2 \quad (18)$$

For ϵ_t, we have

$$\epsilon_t = (1 - \frac{1}{(1+\alpha)^2})\|\Lambda_t'\|_2^2 - 2(1-\frac{1}{1+\alpha})\Lambda_t'\cdot\mathbf{u}. \quad (19)$$

Using the fact that $\|\mathbf{u}-\mathbf{v}\|_2^2 \geq 0$ which equals to $\|\mathbf{u}\|_2^2 - 2\mathbf{u}\cdot\mathbf{v} \geq -\|\mathbf{v}\|_2^2$, we get

$$(1 - \frac{1}{(1+\alpha)^2})\|\Lambda_t'\|_2^2 - 2(1-\frac{1}{1+\alpha})\Lambda_t'\cdot\mathbf{u} \quad (20)$$

$$\geq -\frac{1-\frac{1}{1+\alpha}}{1+\frac{1}{1+\alpha}}\|\mathbf{u}\|_2^2 = -\frac{\alpha}{2+\alpha}\|\mathbf{u}\|_2^2 \quad (21)$$

Using Eq 18 and 21, we get

$$\sum_{t=0}^{T} \Delta_t = \sum_{t \in I_T} \Delta_t \tag{22}$$

$$\geq \sum_{t \in I_T} \left(\left(\frac{1-\alpha^2}{R^2 + \frac{1+\alpha}{2C}} - \frac{2C}{1+\alpha} (\ell_t^*)^2 \right) - \frac{\alpha}{2+\alpha} \|\mathbf{u}\|_2^2 \right) \tag{23}$$

$$= m\left(\frac{1-\alpha^2}{R^2 + \frac{1+\alpha}{2C}} - \frac{\alpha}{2+\alpha} \|\mathbf{u}\|_2^2 \right) - \sum_{t \in I_t} \frac{2C}{1+\alpha} (\ell_t^*)^2 \tag{24}$$

$$\geq m\left(\frac{1-\alpha^2}{R^2 + \frac{1+\alpha}{2C}} - \frac{\alpha}{2+\alpha} \|\mathbf{u}\|_2^2 \right) - \frac{2C}{1+\alpha} \sum_{t=0}^{T} (\ell_t^*)^2 \tag{25}$$

Since $\sum_t \Delta_t \leq \|\mathbf{u}\|_2^2$, we have

$$m\left(\frac{1-\alpha^2}{R^2 + \frac{1+\alpha}{2C}} - \frac{\alpha}{2+\alpha} \|\mathbf{u}\|_2^2 \right) - \frac{2C}{1+\alpha} \sum_{t=0}^{T} (\ell_t^*)^2 \leq \|\mathbf{u}\|_2^2 \tag{26}$$

Since $h = \frac{1-\alpha^2}{R^2 + \frac{1+\alpha}{2C}} - \frac{\alpha}{2+\alpha} \|\mathbf{u}\|_2^2 > 0$, we get the result in the theorem. □

REFERENCES

1. X. Bao, J. Herlocker, and T. G. Dietterich. Fewer clicks and less frustration: Reducing the cost of reaching the right folder. In *Proc. of IUI-06*, pages 178–185, 2006.
2. V. Bellotti, N. Ducheneaut, M. Howard, and I. Smith. Taking email to task: the design and evaluation of a task management centered email tool. In *CHI-03*, pages 345 – 352, 2003.
3. 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.
4. K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7:551–585, 2006.
5. M. Czerwinski, E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. In *Proc. of CHI'04*, pages 175–182, 2004.
6. A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker. Tasktracer: A desktop environment to support multi-tasking knowledge workers. In *Proc. of IUI-05*, pages 75–82, 2005.
7. W. Geyer, M. Muller, M. Moore, E. Wilcox, L.-T. Cheng, B. Brownholtz, C. Hill, , and D. R. Millen. Activityexplorer: Activity-centric collaboration from research to product. *IBM Systems Journal - Special Issue on Business Collaboration*, 45(4):713–738, 2006.
8. T. Joachims. *Learning to Classify Text Using Support Vector Machines*. Kluwer Academic Publishers, 2001.
9. V. Kaptelinin. UMEA: translating interaction histories into project contexts. In *SIGCHI*, pages 353–360, 2003.
10. D. Quan and D. Karger. Haystack: Metadata-enabled information management. In *UIST-2003.*, 2003.
11. F. Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. *Neurocomputing: foundations of research*, pages 89–114, 1988.
12. J. Shen, L. Li, and T. G. Dietterich. Real-time detection of task switches of desktop users. In *Proc. of IJCAI-07*, pages 2868–2873, 2007.
13. J. Shen, L. Li, T. G. Dietterich, and J. Herlocker. A hybrid learning system for recognizing user tasks from desktop activities and email messages. In *Proc. of IUI-06*, pages 86–92, 2006.
14. G. Smith, P. Baudisch, G. Robertson, M. Czerwinski, B. Meyers, D. Robbins, and D. Andrews. Groupbar: The taskbar evolved. In *Proc. of OZCHI 2003*, pages 34–43, 2003.