User Initiated Learning for Adaptive Interfaces

Kshitij Judah
Thomas Dietterich
Alan Fern
Jed Irvine
Michael Slater
Prasad Tadepalli
Oregon State University

Melinda Gervasio
Christopher Ellwood
Bill Jarrold
SRI International

Oliver Brdiczka
Palo Alto Research Center

Jim Blythe
University of Southern California

Abstract

Intelligent user interfaces increasingly employ machine-learning components in order to learn about individual users and optimize the interface accordingly. In all of these cases, the learning tasks must be defined before deployment and a machine-learning expert must be involved in the development process. Unfortunately, this methodology significantly limits the potential utility of machine-learning in intelligent user interfaces since there is no way for a user to create new learning components that serve specific needs as they arise. Our work addresses this shortcoming by developing a framework for user-initiated learning (UIL), where the end user is able to define new learning tasks in a natural way, after which the system automatically generates a learning component, without the intervention of a machine-learning expert. In particular, we consider a UIL scenario where the user is able to define learning problems for predicting when certain activities are forgotten by the user (e.g. encrypting an email message), upon which the system can generate warnings. We describe the knowledge representation and reasoning mechanisms required to replace the machine-learning expert, so as to automatically generate training examples, label the examples, select legal and relevant features, and finally learn the required concept. We present an implementation of this approach in the context of a popular email client and give initial experimental results.

1 Introduction

There is an increasing push toward making user interfaces more customizable and adaptive to the specific needs and tendencies of a user. A common way to achieve this is to integrate one or more machine learning components into the interface that are tuned to learn about particular aspects of a user's behavior and environment. The learned knowledge can then be used for customization. As an example, the Computer Assistant that Learns and Organizes (CALO) project, which our work is part of, has developed many machine learning components for tasks such as predicting the email recipient list, the importance of a message, the folder a user wishes to navigate to, and the project a file is associated with.

The current development paradigm for such adaptive interfaces is to define all learning tasks before deployment and to employ machine learning experts to develop the associated learning components. Unfortunately, this paradigm limits its use of machine learning to a relatively small number of learning tasks that are believed to be the most generally useful across a user base. This precludes the user from exploiting the potential benefits of machine learning for their specific needs as they arise after deployment. Furthermore, even if the user could plug in their own machine learning components after deployment, the development would require a machine-learning expert.

The goal of this research is to empower the end-user to define new learning tasks as the need arises and to solve those tasks without the intervention of a machine-learning expert. We call this new functionality user-initiated learning (UIL), and in this work we focus on a class of UIL problems where the user is able to define learning tasks for predicting when specified activities have been forgotten by the user. For example, the user might be interested in having the system learn to predict when they are likely to attach a file to an email and to warn them before an email is finally sent if the system believes the user forgot an attachment.

At a high-level the development of a UIL system requires solving two key problems. First, we must give the end-user a natural interface for defining new learning tasks, and also for providing the system with any hints that they might have about solving the task more effectively. Second, we must automate the reasoning processes performed by the machine-learning expert in deciding what constitutes legitimate training data for the learning component. In this paper, we describe specific solutions to these problems that we implemented in a prototype UIL system for a popular email client.

In what follows, in Section 2 we define the UIL problem followed by an overview of our system and the UIL process in Section 3. Sections 4-9 describe the key components of the UIL system, followed by experimental results using email data from a real user in Section 10. Finally in Section 11 describes the path toward deployment of our current prototype.

2 Problem Description

To help motivate our problem, consider a scenario where a scientist is collaborating on a classified project where sensitive emails are often exchanged with collaborators. The protocol for indicating the sensitivity of an email is to set the sensitivity flag to confidential before sending it out. However, most emails are not sensitive, even for this project, and as a result the scientist often forgets to set the sensitivity flag when warranted. In this scenario, it would be desirable for the scientist to be able to instruct the system to detect when such a mistake is about to be made and to interrupt the send process with a reminder in such cases. Unfortunately, there is currently no natural way for an end user to extend the user interface to support such a functionality.

In the simplest of cases, a sophisticated user might be able to write a macro to solve the problem. However, the situation just describe can not necessarily be addressed using simple macros since the desired functionality requires that the system be able to predict when an outgoing email should be set to confidential. This can be a non-trivial prediction problem, which requires reasoning about a combination of factors such as the email text, subject, recipient list, etc. While sophisticated machine learning software might be able to learn such a predictor from observations of the user, the end user has no natural way of employing this technology. Currently all machine learning mechanisms in software applications, e.g., spam filters, are developed by machine-learning experts before deployment, and hence are limited to only those mechanisms that are believed to be the most generally useful.

We seek to build the capability of *user-initiated learning (UIL)*, which gives the end user the power to extend the user interface in ways that require specialized machine learning mechanisms, but does so naturally without requiring machine-learning expertise. This paper will focus on a particular class of UIL problems, where the user is able to request that the system learn to predict when they have forgotten a particular activity (e.g., setting the sensitivity flag) and to post a reminder when that happens. In our system, the user will communicate such a request by demonstrating a procedure of interest (e.g., email composition) and indicating which steps of the procedure he may forget to execute. Note that often the indicated actions will be conditional in the sense that they are only executed during some instances of the procedure (e.g., only for confidential emails), which is a primary reason that the user may forget them. This gives rise to a prediction problem where the system will attempt to learn the conditions under which the conditional actions are typically executed, or in other words, predict when the user intended to perform those actions. The learned predictor can then be used by the UIL system to issue reminders when appropriate.

3 UIL System Overview

We implemented our UIL system as part of the Computer Assistant that Learns and Organizes (CALO) desktop, which provides a variety of infrastructure support that is used by our system as well as other components useful in building intelligent user interfaces. The key aspects of this infrastructure relevant to the UIL are described further in Section 4. Figure 1 depicts the basic architecture and process flow of the UIL system which is composed of five main components: the *task demonstration component*, the *feature guidance component*, the *training instance reasoner*, the *machine learning component*, and the *reminder component*. Below we overview the basic steps of the UIL process and the role that each component plays. Later sections of the paper will describe each component in greater detail. A video of the UIL process is available which depicts the process from the user's perspective.[1]

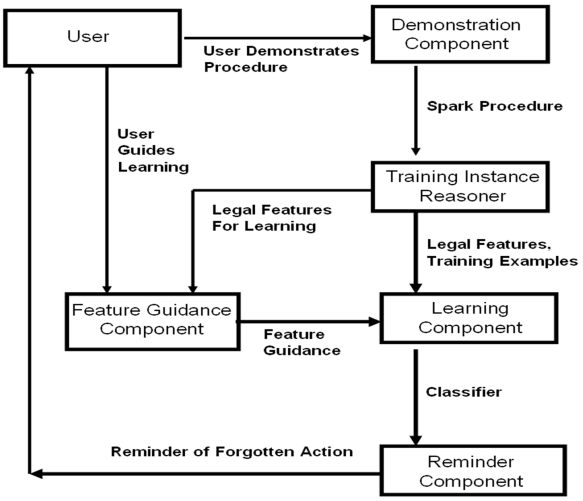


Figure 1: UIL architecture

Step 1: Demonstrating the Learning Task (Section 5). The UIL process begins with the user demonstrating a procedure, or sequence of UI events (e.g., sending a confidential message in Microsoft Outlook). All of the events of the demonstration are captured by the task demonstration component, which displays the captured steps to the user in an easy to read text format. The demonstration component then allows the user to highlight a subsequence of the demonstrated steps (e.g., the step of setting the sensitivity flag) and marks it as a conditional action sequence that the system should learn to predict. Finally the user is allowed to select a *reminder point* in the procedure where the system should issue a reminder to the user if they forget the conditional steps (e.g., when the send button is pressed). The demonstration component then gives the training instance reasoner a program in the SPARK procedural reasoning language [Morley and Myers, 2004] that represents the newly created conditional procedure.

Step 2: Feature Guidance (Section 6). In addition to allowing a user to initiate a learning task via demonstration, our system also allows the user to provide useful hints about how to solve the associated prediction problem. In particular, through the feature guidance component the user is able to navigate through a graphical display of the ontology related to the learning task and to highlight the pieces of information that they believe will be most useful in making predictions. For example, in many prediction tasks involving email, the

[1] **http://web.engr.oregonstate.edu/ irvine/uil.wmv**

user would likely indicate that the words in the email body are important and even provide a number of specific keywords. This information, if provided by the user, is given to the machine learning component to serve as a learning bias and ideally reduce the number of training examples required to achieve a particular accuracy.

Step 3: Training Instance Generation (Section 7). The job of the training instance reasoner is to create labeled training examples corresponding to the demonstrated SPARK procedure. The training instance reasoner extracts instances from the CALO knowledge base, which stores all past and newly arriving desktop information such as emails, documents, projects, contacts, etc. For example, if the learning task involves email, then every email is a potential training instance. For each potential instance, this reasoner uses SAT-based inference to perform two tasks. First, it must reason about the training instance and SPARK procedure to determine the correct target label, which will be treated as the desired output of the learned predictor on that training instance. Second, the reasoner must determine the set of system information (e.g., words in email body) that can be used as possible features by a predictor. For each instance these two reasoning steps combine to produce a training instance composed of a set of features and a target label, which is then forwarded to the machine learning component.

Step 4: Learning a Predictor (Section 8). The job of the machine learning component is to produce a predictor based on the sequence of training examples, while taking into account any available feature guidance provided by the user. Our UIL system utilizes a logistic regression model for prediction, which has the advantage of providing probabilistic predictions and allows for the specification of priors on the feature weights that can take the user's feature guidance into account. However, with this flexibility comes the problem of choosing the precise values for the prediction threshold and prior parameters, which can dramatically impact performance. Thus, a key aspect of our learning component is the automatic selection of these parameters via cross-validation techniques. The resulting predictor along with estimates of its accuracy are passed on to the UIL reminder component.

Step 5: Reminding the User (Section 9). The job of the reminder component is to monitor the UI activity and to issue a reminder to the user whenever it is detected that the user might have forgotten an action sequence. The reminder component uses the prediction of the learned predictor and the SPARK program to drive a SAT-based reasoning process that attempts to infer whether the user forgot a learned action sequence or not. If it is determined that the user did forget, then the reminder component sends an appropriate signal to the UI and prompts the user with a message, which interrupts the normal UI flow. This provides the user with an opportunity to carry out missing actions if they were actually forgotten. Otherwise, the user simply dismisses the reminder.

4 CALO Infrastructure

CALO is an adaptive, personalized assistant designed to assist users in office-based electronic desktop environments [Cheyer *et al.*, 2005; Myers *et al.*, 2007]. CALO is intended to provide intelligent assistant capabilities across standard applications on the Windows platform, supported by various learning and reasoning modules to support management and prioritization of information and tasks. Before detailing UIL system components, we provide an overview of CALO infrastructure relevant to our UIL system, including the CALO ontology and the Microsoft Outlook instrumentation.

4.1 CALO Ontology.

In order for CALO components to interoperate they need a shared source of information about system events, the user, his/her colleagues, meetings, e-mails, projects, documents etc. For example, our UIL system needs to be able to capture events from applications such as Microsoft Outlook and also access information related to email messages. In support of these needs the CALO ontology [Chaudhri *et al.*, 2006a] serves as the representational foundation for a centralized knowledge base (KB). The KB provides background knowledge and serves as a target for the information generated by a collection of engineered harvesters and learned extractors and classifiers that interoperate with the CALO framework.

The ontology, implemented in OWL [Chaudhri *et al.*, 2006b], represents a variety of classes and relationships. Some of the classes correspond to actual entities found in traditional operating system such as files, folders, snf -mail messages. Other classes such as `Person` or `Project` correspond to entities that are abstract from the point of view of an operating system but quite salient to a human user. Instances of these classes are inter-related via a large number of semantic properties. The ability to easily access all of these properties is important to the UIL system, since all predictions will ultimately be based on this information. Of particular importance to UIL is the ontology of possible user actions and system events. Examples of classes representing generic or abstract actions include `Modify` or `Open`

Importantly, in order for CALO to observe user actions, operating system applications must be instrumented. The ontology serves as an interlingua supporting communication about instrumented actions. Such messaging is implemented in the form of a publish / subscribe framework. This component is known as the task interface registry. Agents may subscribe to messages about certain classes of actions and in so doing, take advantage of the ontology's subsumption hierarchy. For example, our UIL system, which is currently focused on the Microsoft Outlook email application, can be alerted to any email related event by subscribing to all messages that are about instances of the class `EmailTask`.

5 Initiating Learning via Demonstration

We employ the Integrated Task Learning (ITL) component of CALO as a means for capturing user demonstrations of target learning tasks. The ITL component is a general mechanism that integrates a number of independently developed components for learning user workflows [Spaulding *et al.*, 2009] and supports a number of capabilities for acquiring procedures, including learning from demonstration and procedure editing. Although originally designed to support the user-driven acquisition of automated procedures, it was straightforward

```
{defprocedure do_rememberSensitivity
....
[do: (openComposeEmailWindow $newEmail)]
[do: (changeEmailField $newEmail "to")]
[do: (changeEmailField $newEmail "subject")]
[do: (changeEmailField $newEmail "body")]
[if: (learnBranchPoint $newEmail)
  [do: (changeEmailField $newEmail "sensitivity")]]
[do: (sendEmailInitial $newEmail)]
....
}
```

Figure 2: An example of a SPARK procedure produced by ITL based on a user demonstration.

to extend the ITL component to serve as UIL's demonstration capture component.

ITL supports UIL's demonstration needs via two sub-components. First, the LAPDOG sub-component [Gervasio *et al.*, 2008] transforms an observed sequence of instrumentation events, into a SPARK procedure [Morley and Myers, 2004] that captures and generalizes the dataflow between the actions. Given a captured procedure, ITL then allows for procedure editing capabilities through the Tailor sub-component [Blythe, 2005b; 2005a]. For UIL, Tailor was extended to provide the ability to add a condition to one or more steps in a procedure—where, in this case, the condition corresponding to the new learning task. The resulting annotated SPARK procedure can then be utilized by later components to create training instances and identify situations where reminders are required.

Figure 2 shows an example of a SPARK procedure produced by the demonstration process for a task where the user wishes to teach the system to learn to predict when the sensitivity field should be changed. The original procedure captured by LAPDOG did not include the **if:** conditional. Rather, this conditional was added by the user via the Tailor interface, which directs the learner to learn a classifier that can predict whether the branch is taken or not.

6 Feature Guidance Interface

In order to speed up learning, the system allows the user to provide additional knowledge in the form of feature guidance, although the user can easily skip this part of the UIL process if desired. The interface presents to the user a view of the portion of the ontology that is relevant to the current learning task. For example, in our email-related tasks this includes the class of EmailMessage along with other related objects like Project, Sender, ToRecipient, CCRecipient etc. The user is allowed to navigate through the ontology in order to select attributes that are believed to be useful for solving the task. For example, the user might indicate as important the set of recipient email addresses, the subject text, or the body text. Furthermore, in addition to being able to indicate that certain fields are important, the user can answer questions specific to each field that further specialize the advice. For example, when the user navigates to the body text attribute they will have the option of entering any number of keywords that they believe will be useful as features. Figure 3 shows an example of this, where the user has highlighted in the left panel a number of attributes, and in the right panel entered more specific information about the attributes.

The result of the user guidance process is a set of user

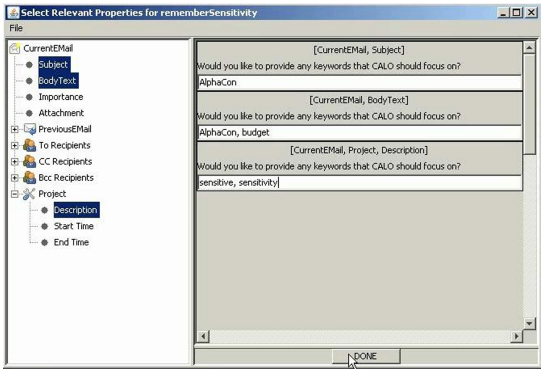


Figure 3: Feature guidance interface.

selected features that are forwarded to the machine learning component. It is important to note that the fact that the user selects certain features does not mean that other features will not be considered by the system. Rather, our system uses the user selections only to bias the machine learning system in favor of the selected features. Other features are also considered but with more caution. The details of how this bias is implemented are in Section 8.

7 Training Instance Generation

The UIL system employs the training instance reasoner to autonomously generate labeled training instances for consumption by the machine learning component. First, the reasoner must determine which objects of interest represent valid training examples and for those objects assign a label to them, which will serve as the target output for the predictor. In our case, the objects of interest will always be emails, but in general the types of the objects to be considered can be inferred from the Spark program. Second, the reasoner must determine which properties of the training instances are valid for use as features during learning.

An important aspect of our system is that it is able to automatically extract training instances from relevant objects in the CALO knowledge base. This allows for prior user data to be leveraged for new learning tasks when appropriate, rather than only using newly arriving examples. However, this poses poses some challenges since objects in the CALO ontology are not necessarily annotated with the user actions used to create them. Thus, it becomes necessary to make inferences about those actions in order to relate those objects to the Spark procedures which define the learning tasks. To accomplish this in a general way, below we describe a SAT-based reasoning process that employs a simple domain model of the UI actions and the ways they effect the system attributes.

7.1 Domain Model.

Our domain model needs to capture the interactions among email related actions and properties. We utilize a propositional logic for this, were we define a proposition for each email action that can appear in a Spark program and one proposition for each email property. Some example action propositions include: ComposeNewMail, ForwardMail, ReplyToMail, ModifyToField, ModifyCC, ModifySubject, and

ModifyBody. Action propositions are defined to be true relative to the current email under consideration iff their corresponding UI action was taken during the creation of the email. Some example property propositions include: NewComposition, ForwardedComposition, HasCCField, and HasBody. These propositions are defined to be true relative to an email being considered iff the email satisfies the corresponding property. For example, the HasBody is true if the email has a non-empty body. Note that it is straightforward to compute the truth values of property propositions given an email, but is less direct for action propositions since the knowledge based does not store the actions that were used to create an email.

To provide a link between actions and email properties we specifying a domain theory, which includes a single axiom for each property proposition that defines the possible ways that the proposition can be made true. Some example axioms include:

$$\begin{aligned}
\text{NewComposition} &\iff \text{ComposeNewMail} \\
\text{ReplyComposition} &\iff \text{ReplyToMail} \\
\text{HasAttachment} &\iff (\text{AttachFile} \vee \text{ForwardMail}) \\
\text{HasSubject} &\iff (\text{ModifySubject} \vee \text{ReplyToMail} \vee \text{ForwardMail}) \\
&\ldots.
\end{aligned}$$

This domain theory is only a crude approximation to reality but is sufficient for our purposes.

7.2 Inferring Class Labels.

Given an email from the CALO knowledge base and a demonstrated Spark program we now wish to assign a label to the email. We do this by first constructing a formula called the *Label Analysis Formula (LAF)* that captures key constraints arising from the Spark program and domain axioms. The LAF involves all of the domain propositions plus three new propositions: Label, which is represents the truth value of the branch condition in the Spark procedure, or equivalently whether the user intended to select the conditional actions; Forget, which indicates whether the user intended to execute the conditional steps but forgot to do so; and ProcInstance, which indicates that the current email corresponds to an instance of the Spark procedure. Here we say that an email is an *instance of* the Spark procedure whenever the action sequence that generated the email includes all of the unconditional actions in the procedure, possibly including other actions. Any such email can be used as a possible training instance.

Given these new propositions the LAF is constructed by including all domain axioms in addition to two new *Spark axioms* related to the Spark procedure. In particular, the new axioms place constraints on the ProcInstance, Forget, and Label propositions. To do this, let $U_1, ..., U_n$ be the set of propositions corresponding to the unconditional actions in the SPARK procedure and $C_1, ..., C_m$ be the propositions corresponding to conditional actions. The Spark axioms are then given by:

$$\begin{aligned}
\text{ProcInstance} &\iff (U_1 \wedge U_2 \wedge ... \wedge U_n) \\
(\neg \text{ Forget} \wedge \text{Label}) &\iff (C_1 \wedge C_2 \wedge ... \wedge C_m)
\end{aligned}$$

The first constraint allows one infer that an email is an instance of the procedure iff it can be proven that all of the unconditional actions were taken. The second constraint indicates that the conditional actions are taken by the user iff they intended to perform the conditional actions and did not forget to do so.

Given an email we can use the LAF to label it as follows. First, for each property proposition P we compute its truth value by inspecting the email and then add the clause P to the LAF if it is true and add $\neg P$ otherwise. Second we add the unit clause $\neg$ Forget to the LAF resulting in a formula E, which encodes all of the information we have about the email domain, the Spark program, the current email, and encodes the assumption that the user was not forgetful. To produce a label for the email, we first attempt to prove that the query ProcInstance $\wedge$ Label is entailed by E. If it is then we have proven that, under the assumption that the user did not forget any intended steps, the email is an instance of the procedure and is a positive example of the learning task. Otherwise we attempt to prove the query ProcInstance $\wedge \neg$ Label and if we are successful the email is a negative instance of the learning task. Otherwise, either the email was not an instance of the procedure and/or there was not enough information to conclusively infer the label of the instance. In this later case we ignore the email and do not create a training instance. In our current UIL system we use the YICES SAT-solver [**?**] as our reasoning engine, which is able to solve our relatively small problems almost instantaneously. It can be proven that any training instance generated by this reasoning process is guaranteed to have the correct labels under the assumption that the user was not forgetful for the instance. Thus, the rate of label noise produced by our reasoning engine is related to the level of forgetfulness of the user, which will typically low enough for machine learning mechanisms to overcome.

7.3 Inferring Feature Legality.

When generating training examples, one must carefully consider which features the learning algorithm to allowed to use for making predictions to avoid generating useless learning problems. As a simple illustration of this consider the task of learning to predict whether the user intended to attach a file to an email. The reasoning process described above will label emails according to whether or not they have an attachment. If, however, we provide the machine learning algorithm with a feature that indicates whether a email includes an attachment, then the learned predictor will be able to achieve 100% accuracy by simply returning the value of that feature. However, the predictor would be useless in the context where we wish to remind the user that they have forgotten an email, since in all cases where the user forgets there would be no attachment, causing the predictor to predict that nothing was forgotten. The general problem encountered here is that one must avoid the use of features that are deterministically related to the branch condition via the domain theory and hence not valid for use by the predictor. It is possible to perform a reasoning process involving he LAF to determine the set of email properties that are legal to use, however, space precludes the details.

8 Machine Learning Component

This section presents the details of the machine learning component employed in UIL. The objective is for this component

to fully automate the creation of a predictor given the training examples produced by our system and the feature guidance, if any, provided by the user.

8.1 Learning Algorithm.

We use logistic regression as our basic learning algorithm [**?**]. This algorithm learns a linear discriminant function over a feature vector x that represents the probability that the label y is positive given x as follows, $P(y = 1|x, w) = \frac{1}{1+exp(-w \cdot x)}$ where w is the weight vector to be learned, which weights the features against one another. Logistic regression algorithms, typically learn a weight vector w by optimizing the log-likelihood of the training data, which is a convex optimization problem that can typically be solved quite effectively via gradient methods. However, just optimizing the log-likelihood can often lead to overfitting of the training data, particularly when there are a large number of features. For this reason, weight regularization is often incorporated into the learning process by assuming a prior distribution on the weights that assigns higher probability mass to weight vectors with small magnitudes. At typical prior, and the one we use, is to assume that each weight is distributed according to a zero mean Gaussian distribution with a specified variance σ^2, where smaller variances correspond to more extreme regularization. The problem of optimizing the weights given such a prior is still convex and can be easily solved via gradient methods.

8.2 Incorporating Feature Guidance.

Feature guidance from the user is incorporated by setting a significantly larger variance for the priors on the user selected features compared to the unselected features features. By specifying a large variance for user selected features, we are essentially telling the learner that there is a high prior probability that the corresponding weight values are not near zero and thus should contribute significantly toward predictions. This allows the learner to be more aggressive about assigning non-negligible values to those weights, requiring less statistical evidence for doing so than if the variances were smaller.

8.3 Autonomous Parameter Tuning.

When making predictions with the logistic regression classifier it is typical to select a probability threshold τ such that a prediction of 1 is returned if $P(Y = 1|x, w) \geq \tau$, and otherwise a prediction of 0 is returned. The selection of τ can dramatically impact the usefulness of the predictions. In order to fully automated the learning process, it is important that both the variance parameters and τ be tuned automatically to optimize performance. To do this, we implemented a search over the parameter space, using leave-one-out cross-validation to estimate the performance under each parameter setting. Our current system uses the κ criterion, a standard statistical metric, as a measure of prediction performance, which tends to be a more meaningful metric than accuracy when the label distribution of the training data is skewed, which we find is often the case in UIL scenarios (e.g. most emails are not marked as sensitive). For the variance parameter of unselected features, we search over a range of 0.01 to 100 in step sizes of 0.1, while keeping the variance for selected features equal to 1000. For the threshold parameter we consider a range from 0 to 1 in linear steps. Logistic regression algorithms are quite fast, which made this search tractable, however, for slower algorithms or larger data sets there are many more sophisticated search strategies that could be used rather performing an exhaustive search.

9 Reminding the User

The reminder component is responsible for monitoring user's activities and alerting him if he forgets to execute some conditional actions from previous learning tasks. To do this whenever the user reaches a reminder point, as specified in the demonstrated SPARK procedure, the reminder unit attempts to infer whether or not the user has forgotten the conditional steps. This is straightforward when instrumentation is available that allows for constant monitoring of the user actions. However, our current instrumentation support does not allow us to easily do this for all actions and thus we again resort to the use of automated reasoning to help infer the actions that are not directly observable.

The reasoning process starts with the LAF formula from Section 7, which encodes constraints about the SPARK program and domain model. We add to this formula a set of unit clauses that represent the observed properties of the current email under consideration, again as described in Section 7. We then use the learned predictor to make a prediction, which is assumed to be a correct prediction of the user's intention to perform the conditional actions. Recalling that the proposition Label in the LAF corresponds to the user's intention we then add the unit literal Label to the formula if the prediction is positive and otherwise add the unit literal $\neg$Label. Given the resulting formula we then ask if the query ProcInstance $\wedge$ Forget is entailed, and if it is the assistant issues a warning to the user that they might have forgotten the conditional steps. It can be shown that this process will only issue warnings when the user actually has forgotten the steps under the assumption of a perfect predictor. Thus, the quality of the assistance provided by the reminder component is primarily related to the quality of the predictor.

10 Empirical Evaluation

We evaluated our system on two email related learning tasks. First, we consider the *attachment prediction* task, which involves learning to predict when the user intends to attach a file to an email. Second, we consider the *importance prediction* task, which involves learning to predict when the user intends to set the importance of an email to either high or low as opposed to not setting importance. Both of these are easily specified via our demonstration interface. For both of these learning tasks we used a knowledge base that contained 340 real emails authored by a single desktop user. The user provided 18 features as guidance to our learner for each task, which here were all key words in the body text.

We conducted experiments by first dividing the dataset into a training set of 256 instances and a test set of 84 instances. To simulate the effect of a growing email knowledge base, we further divide the training set to create multiple training sets of increasing sizes: 64 non-overlapping training sets of size

4, 32 sets of size 8, 16 sets of size 16 and so forth. For each training set size, we train on individual training set and use the learned classifier on the test set to compute the Kappa coefficient, which is a common evaluation metric in cases when the labels have a skewed distribution, which is the case for us (positive class percentages are 26% and 10% for our data). Finally, we compute the mean Kappa coefficient for each training set size, which allows us to plot learning curves. In order to evaluate the relative impact of the user-provided features and our automated parameter tuning, we generated learning curves for 4 different configurations of our system: A) No Feature Guidance + No Parameter Tuning, B) Feature Guidance + No Parameter Tuning, C) No Feature Guidance + Parameter Tuning, and D) Feature Guidance + Parameter Tuning.

10.1 Basic Learning Curves.

For the attachment prediction problem, Figure 4(a) shows the learning curves for each of our 4 configurations. We see that except configuration A, which did not include user guidance or tuning, the other three configurations exhibit positive learning curves of similar quality. This indicates that in large part the feature guidance can compensate for lack of parameter tuning and vice versa. We do see that for larger training set sizes that including both feature guidance and tuning results in the best performance. We can also observe a slight edge for configurations that include feature guidance for small data sets compared to just using parameter tuning. A likely reason for this is that the variance of cross-validation, our tuning method, is higher for small data sets, making it less effective. We obtained similar trends for the importance prediction task as shown in Figure 5(a). For this task, however, there appears to be a much more significant benefit for using both tuning and feature guidance for the larger data sets.

10.2 Robustness to Bad Guidance.

In order to evaluate the potential impact of bad feature guidance to our system, we ran some additional experiments. To generate bad feature guidance, we restricted our attention to features corresponding to key words in the email body text. We then use SVM based feature selection in Weka to produce a ranking of the user selected features/words in terms of their predictive utility. Finally, we replaced the top 3 words in the ranking with randomly selected words with the resulting set representing "bad" feature guidance/advice.

The learning curves in Figure 4(b) shows learning curves for good and bad advice both with and without parameter tuning. First we observe that without parameter tuning, the inclusion of the bad advice results in a dismal learning curve. By incorporating parameter tuning, however, we see that even with the bad advice the learning curve is quite good. This shows that the use of parameter tuning can be critical when there is a possibility of obtaining bad advice. For importance prediction the corresponding experiment is shown in Figure 5(b). Here we see that learning is more robust to bad advice for the smaller training sets but degrades performance significantly later on. Again for the larger training sets we see that parameter tuning is critical to overcoming bad advice, but for this task, even with parameter tuning the bad advice results in significantly worse performance than with good advice.

10.3 Estimating Utility of the Predictors.

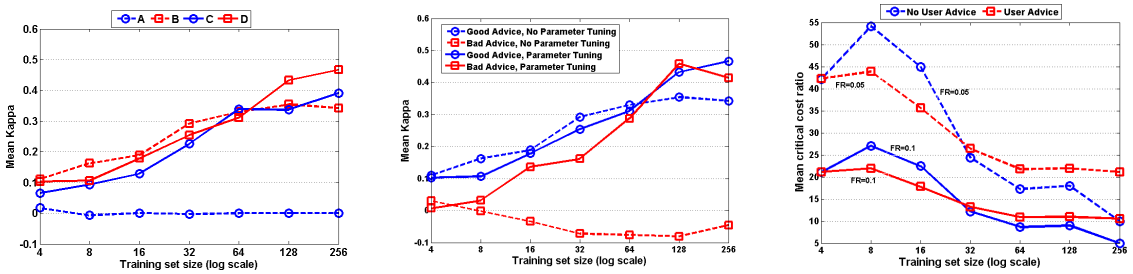
Here we investigate whether the reminder assistant using our learned predictors might be able to decrease the overall UI cost to the user. There are two types of user costs: 1) *the cost of forgetting*, which for example, in the attachment scenario involves potential delays for recipients and the need to resend an email, 2) *the cost of interruption* by the system with a reminder in cases when the user did not really forget anything. If we knew these costs for the user, we could easily compute the expected cost using our reminder assistance versus not using it. However, we do not know the values of these costs and cost elicitation is beyond the scope of this paper.Rather, we assess the utility of our predictor by measuring a new metric that we call the critical cost ratio (CCR).

To understand CCR, consider the ratio of the forgetting cost to the interruption cost, which will typically be greater than one. Given a fixed predictor, it is possible derive an expression for the minimum value of this ratio such that the cost of using the reminder assistant is equal to the cost without it. We define the CCR for the predictor to be this minimum ratio. Thus, if the CCR for a predictor is 10 then the cost of forgetting must be more than 10 times the cost of interruption for the reminder assistant to provide a net benefit. The expression for the CCR is given by $CCR = \frac{(1-CR)\times FPR}{PR\times FR\times TPR}$ where FPR and TPR are the false positive and true positive rates of the predictor, FR is the frequency that the user forgets the conditional actions when they intend to take them, and CR is the frequency that the user intends to take the conditional actions.

Figures 4(c) and 5(c) give the learning curves plotted in terms of CCR for our prediction tasks. We have graphs for two forgetting rates (FR=0.1 and 0.05) and for each we give results both with and without feature guidance with parameter tuning always on. The first observation is that for the largest training set sizes the values of CCR are quite reasonable for natural cost models. In particular, for the attachment scenario the CCR drops to about 2 when advice is used, which means that a net benefit would be apparent when the cost of forgetting is just a factor of 2 larger than the cost of interruption. For the importance task the CCR drops to about 10 when advice is used, and surprisingly even lower without advice. For smaller training sets the CCRs grow to be quite large, but are less than 100. Whether these CCR ratios would be adequate depends on the particular user and scenario. In many cases such high values would indicate that the predictor should not be used for that amount of training data.

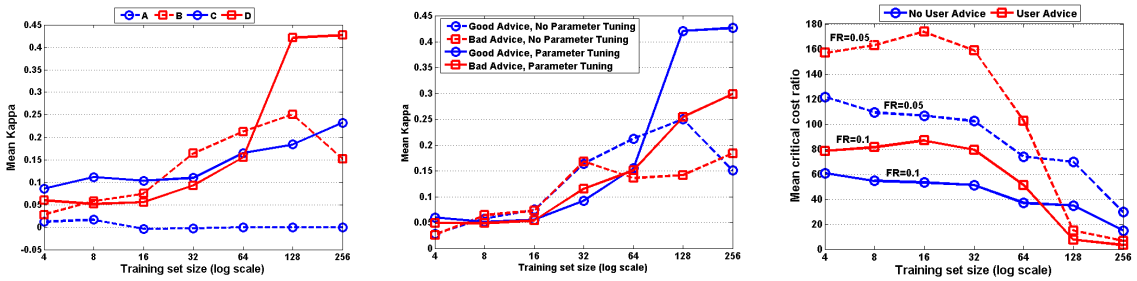
11 Towards Deployment

During the course of the UIL development, we realized that there are many challenges in developing user-extensible learning systems. These challenges included building instrumented end user applications, translating accurate models of user behavior into an ontology, and designing self-tuning learning components that were capable of handling a wide range of learning tasks. Our UIL prototype faced these challenges successfully and provided a first approximation of a



(a) (b) (c)

Figure 4: Learning curves for attachment prediction.



(a) (b) (c)

Figure 5: Learning curves for importance prediction.

solution to extensible learning systems that adapt themselves to learning tasks defined by the user. However, although the prototype UIL system we developed is fully functional, we would need to improve the system's reliability, performance, and usability before wide scale deployment in the field.

11.1 Reliability.

UIL is a set of components that runs both inside and outside of the CALO environment. In order to function properly, not only do these components need to work in harmony with each other, they must also interface with CALO in a consistent manner. While we were able to make sure all the components of the system worked coherently for the attachment and importance cases, we would need to test a wider range of supported learning tasks and fix the issues that arise. This testing would mainly focus on ensuring that all components interface with the ontology in the same way and checking that the "harvesting" of user activity data is properly encoded into ontological constructs. Once we are confident our packaged system works as desired in our development environment, we would have to test it against the proposed deployment environment.

11.2 Performance.

Beyond the UIL system working properly in the deployment environment, it must also perform adequately on a wide variety of end-user machines. The CALO system itself is a complex large footprint Java application with many components. To achieve adequate performance, we would have to engineer and test a small footprint subset of CALO that would pose a minimal performance impact on the end-user. Once CALO is optimized for a small low impact footprint, we would also optimize the UIL components to use a minimum of RAM and CPU cycles.

11.3 Usability.

In our UIL prototype, we'd want to be sure that terminology oriented towards the AI researcher is translated into terms that the typical end user can understand. For example, in the ITL application, we would want to replace "Add Learned If" button with something simpler such as "Help me remember". Also in the existing UIL prototype, not all user demonstrable actions are supported as learning tasks. We may wish to add additional user interface constructs to show the user visually what "help me remember" learning tasks are valid, i.e. supported by the task demonstration component and the rest of the UIL system.

Acknowledgements

This material is based upon work 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 authors(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or the Air Force Research Laboratory (AFRL).

References

[Blythe, 2005a] J. Blythe. An analysis of task learning by instruction. In *Proceedings of the 20th National Conference on Artificial Intelligence*, 2005.

[Blythe, 2005b] J. Blythe. Task learning by instruction in tailor. In *Proceedings of the 2005 International Conference on Intelligent User Interfaces*. ACM Press, 2005.

[Chaudhri *et al.*, 2006a] Vinay K. Chaudhri, Adam Cheyer, Richard Guili, Bill Jarrold, Karen L. Myers, and John Niekrasz. A case study in engineering a knowledge base for a personal assistant. In *Semantic Desktop and Social Semantic Collaboration Workshop at the International Semantic Web Conference*, 2006.

[Chaudhri *et al.*, 2006b] Vinay K. Chaudhri, Bill Jarrold, and John Pacheco. Exporting knowledge bases into owl. In *Proceedings of the Workshop on OWL: Experiences and Directions*, 2006.

[Cheyer *et al.*, 2005] A. Cheyer, J. Park, and R. Guili. Iris: Integrate, relate, infer, share. In *Semantic Desktop Workshop*, 2005.

[Gervasio *et al.*, 2008] M. Gervasio, T. J. Lee, and S. Eker. Learning email procedures for the desktop. In *Proceedings of the AAAI Workshop on Enhanced Messaging*. AAAI Press, 2008.

[Morley and Myers, 2004] David Morley and Karen Myers. The spark agent framework. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*. IEEE Computer Society, 2004.

[Myers *et al.*, 2007] K. Myers, P. Berry, J. Blythe, K. Conley, M. Gervasio, D. McGuinness, D. Morley, A. Pfeffer, M. Pollack, and M. Tambe. An intelligent personal assistant for task and time management. *AI Magazine*, 28(2):47–61, 2007.

[Spaulding *et al.*, 2009] A. Spaulding, J. Blythe, W. Haines, and M. Gervasio. Integrating task learning tools to support end users in real-world applications. In *Proceedings of the International Conference on Intelligent User Interfaces*. ACM Press, 2009.