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
William Jarrold
SRI International

Oliver Brdiczka
Palo Alto Research Center

Jim Blythe
University of Southern California

Abstract

Intelligent user interfaces employ machine-learning to learn and adapt according to user peculiarities. In all these cases, the learning tasks are predefined and a machine-learning expert is involved in the development process. This significantly limits the potential utility of machine-learning since there is no way for a user to create new learning tasks for specific needs as they arise. We address this shortcoming by developing a framework for user-initiated learning (UIL), where the end user can define new learning tasks, after which the system automatically generates a learning component, without the intervention of an expert. We describe the knowledge representation and reasoning required to replace the expert, so as to automatically generate labeled training examples, select features, and learn the required concept. We present an implementation of this approach for a popular email client and give initial experimental results.

1 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. 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 described cannot 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.

2 UIL System Overview

We implemented our UIL system as part of the Cognitive Assistant that Learns and Organizes (CALO) desktop. 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. CALO provides a variety of infrastructure support that is used by our UIL system including the CALO ontology, knowledge base (KB), and Microsoft Outlook instrumentation. The CALO ontology [Chaudhri *et al.*, 2006a], implemented in OWL [Chaudhri *et*

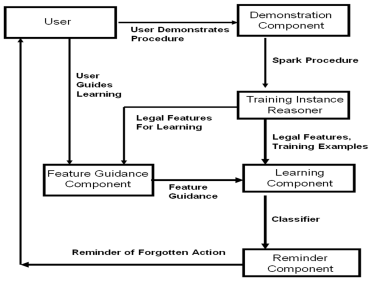


Figure 1: UIL architecture

al., 2006b], serves as the representational foundation for a centralized KB that 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. For example, historical information about emails, people, projects, files and their relationships are all stored in the KB.

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 *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]

Step 1: Demonstrating the Learning Task (Section 3). 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 4). 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 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 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 5). 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 learning component.

Step 4: Learning a Predictor (Section 6). The job of the 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 7). 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.

3 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 compo-

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

```
{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.

nents for learning user workflows [Spaulding *et al.*, 2009] and supports a number of capabilities for acquiring procedures, including learning from demonstration and procedure editing. ITL supports UIL's demonstration needs via two subcomponents. 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.

4 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.

5 Training Instance Generation

The UIL system employs the training instance reasoner to autonomously generate labeled training instances for consumption by the learning component. First, the reasoner must determine which objects of interest represent valid training examples and assign a label to those objects, which will serve as the target output for the predictor. Second, the reasoner must determine which properties of the training instances are valid for use as features during learning. For space reasons, here we only detail the first of these reasoning processes.

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 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.

Domain Model. Our domain model needs to capture the interactions among email related actions and properties. We utilize propositional logic for this, where we define a proposition for each email action that can appear in a SPARK procedure 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.

Inferring Class Labels. Given an email from the CALO knowledge base and a demonstrated SPARK procedure 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 procedure and domain axioms. The LAF involves all of the domain propositions plus three new propositions: Label, which 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:

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

The first constraint allows one to 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 procedure, 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 we are successful 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 YICES [Dutertre and Moura, 2006] 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 be low enough for machine learning mechanisms to overcome.

6 Learning Component

The role of the learning component is 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. We currently 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.

In addition, to help avoid overfitting we incorporate a zero mean Gaussian prior distribution over the weights as a weight regularization approach. Setting the prior variance σ^2 to smaller values corresponds to more extreme regularization. In order to incorporate feature guidance from the user we set a significantly larger variance for the priors on the user selected features compared to the unselected features. This tells the learner that there is a high prior probability that the weight values for the selected features are not near zero and thus should contribute significantly toward predictions.

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 automate 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 linear search over values of τ and the variance of the unselected features. 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.

7 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. Space precludes details, however, it can be shown that our 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.

8 Empirical Evaluation

We evaluated our system on two email related learning tasks: 1) *attachment prediction*, which involves learning to predict when the user intends to attach a file to an email, and 2) *importance prediction*, which involves learning to predict when the user intends to set the importance of an email to either high or low. We used a knowledge base that contained 340 real emails authored by a single desktop user. The user provided 18 features as guidance for each task, which were all keywords in the body text.

We divided 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 (κ) [2]. Finally, we compute the mean κ 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.

Learning Curves. For the attachment prediction problem, Figure 3(a) shows the learning curves for each of our 4 configurations. Except for configuration A, which did not include user guidance or tuning, the other three configurations exhibit positive learning curves of similar quality. This indicates that the feature guidance can compensate for lack of parameter tuning and vice versa. We do see that for larger training set sizes, 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 4(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.

Robustness to Bad Guidance. We now consider the impact of bad feature guidance. To generate bad feature guidance, we restricted our attention to features corresponding to keywords in the email body text. We then used 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 3(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 4(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.

Estimating Utility of the Predictors. Here we investigate whether the reminder assistant 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 of using our reminder assistance versus not using it. Rather, since we don't know these costs, we assess the utility of our predictor by measuring a new metric called the critical cost ratio (CCR).

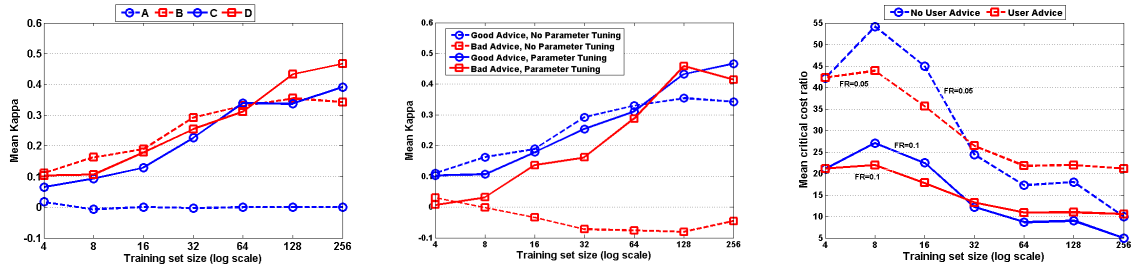
To understand CCR, consider the ratio of the forgetting cost to the interruption cost, which will typically be greater than one. Given a fixed predictor, it is possible derive an expression for the minimum value of this ratio such that the cost of using the reminder assistant is equal to the cost without 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 3(c) and 4(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.

9 Towards Deployment

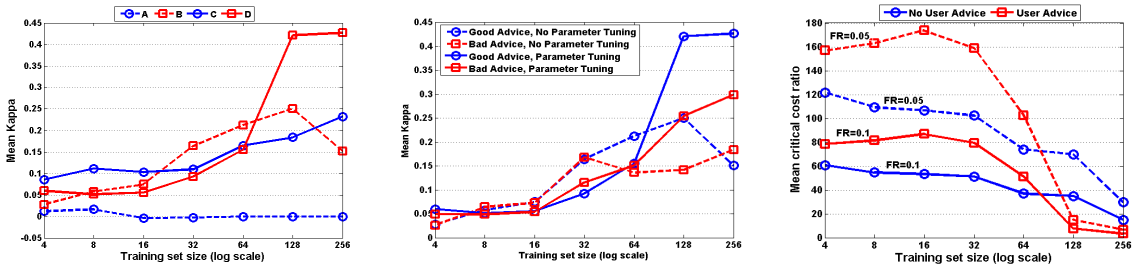
During the course of the UIL development, we encountered many challenges in developing user-extensible learning sys-

[2] κ is a common evaluation metric in cases when the labels have a skewed distribution



(a) (b) (c)

Figure 3: Learning curves for attachment prediction.



(a) (b) (c)

Figure 4: Learning curves for importance prediction.

tems, which include building instrumented end-user applications, translating accurate models of user behavior into an ontology, and designing self-tuning learning components. Our UIL prototype faced these challenges successfully and provided a first approximation of a solution. However, although the prototype UIL system we developed is fully functional, we would need to improve the system's reliability and performance before wide scale deployment in the field. In addition, the usability of the system needs further consideration. The current UIL prototype has been primarily tested by AI researchers and it is important to translate the current UI terminology into terms that the typical end user can understand. For example, in the ITL application, we would want to replace the "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.

10 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 author(s) and do not necessarily reflect the views of the 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.

[Dutertre and Moura, 2006] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, Computer Science Laboratory, SRI International, 2006.

[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.