Learning Email Procedures for the Desktop

Melinda Gervasio[1] Thomas J. Lee[1] Steven Eker[2]

{[1]Artificial Intelligence Center, [2]Computer Science Laboratory}, SRI International
333 Ravenswood Ave.
Menlo Park, California 94025
melinda.gervasio@sri.com tomlee@ai.sri.com eker@csl.sri.com

Abstract

In this electronic age, we are all knowledge workers, tackling on a daily basis the information that flows through our email, file systems, web browsers, calendars, and various other desktop applications. Email has come to be the center of desktop activity for many of us: we set up meetings, exchange documents, manage projects, forward interesting links, track requisitions, and perform a whole host of tasks through email. Many of these activities involve information flowing from one action to the next, across applications. They are also often repetitive, structured procedures, involving an ordered set of steps, many but not all of which are amenable to automation. In this paper, we present our approach to learning generalized dataflow procedures on the desktop from demonstrations by the user. Our system LAPDOG runs on the CALO desktop assistant and is capable of acquiring fully or partially automated procedures, spanning multiple applications, from demonstrations that may include unobservable actions.

Introduction

Every day, we use a variety of applications on our electronic desktops to create and manage various information products, including email, files, folders, and web pages. The tasks we perform are often repetitive but not necessarily identical—for example, we may send documents of a particular type to certain people, but the exact documents or exact recipients may differ each time. Some tasks are simple, involving a single action such as replying to an email message, while others are more complex, for example, performing a web search and downloading files, followed by sending email with the files attached. Because email has become such a central part of the electronic workplace, often serving purposes well beyond exchanging messages (Bellotti et al., 2003), many of these repetitive tasks involve email but they also often involve other information artifacts.

One approach to assisting users is to automate these repetitive tasks through task learning—in particular, *learning from demonstration*, also often referred to as programming by demonstration or programming by example (Cypher, 1993; Lieberman, 2001). In this paradigm, the user executes the task to be learned one or more times to provide training examples from which the learner can induce a general procedure to achieve the task. Learning-from-demonstration systems typically employ knowledge-intensive approaches, leveraging expert advice, user annotations, strong background knowledge, and other auxiliary information to learn from very few examples. This makes such systems particularly attractive in situations where rapid learning is desirable or many examples are difficult to come by. While there has been much work in this field over the years, most has been in the context of a single application, which has the advantages of actions being naturally amenable to uniform treatment and states being feasibly tracked.

Another approach is to focus on *atomic* or single-step email tasks—for example, email foldering (Segal & Kephart, 2000), speech act identification (Cohen et al., 2004), reply prediction (Dredze et al., 2005), or cc prediction (Pal & McCallum, 2006). The systems help users by offering suggestions to streamline tedious actions such as filing or replying to email, or by alerting them to potential mistakes such as missing cc recipients.

Alternatively, one can instead focus on the workflows that have become prevalent but are only implicitly represented in email. Approaches include the development of interfaces to facilitate the capture and management of these activities (Bellotti et al., 2007) as well as the application of machine learning techniques to help discover and recognize workflows (e.g., Kushmerick & Lau, 2005; Kushmerick et al., 2006).

The work we describe in this paper makes two main contributions to email assistance. First, we approach email assistance from the larger context of assistance for the electronic desktop, presenting an approach that can learn multistep tasks across disparate applications, including but not limited to email. Second, we present an approach to procedure learning that combines learning and reasoning to enable an adaptive assistant to learn from demonstrations that may involve unobservable actions.

We begin by describing our framework for desktop assistance, including our specific application domain and its underlying instrumentation and automation infrastructure. We then present an overview of our

approach to learning procedures from demonstration. We follow this with a presentation of a variety of use cases to highlight the different kinds of procedures we can learn and the various learning techniques used. We conclude with a discussion of related work and future directions.

Email Assistance within the Desktop Context

Each of us has tasks we perform repetitively on our computers. A journal editor might shepherd submissions through a standard reviewing process involving inviting qualified reviewers, issuing reminders, and sending reviews to authors. In preparing quarterly reports, a project lead might send out requests for material to team members, combine all the responses into a provided template, and send back the report. An administrative assistant scheduling a meeting might book a room, arrange for catering, and send out the agenda and other related information. And in filing a bug report a programmer might log in to an issue tracker, start a ticket, and copy error messages from a console into the report.

Task learning—in particular, learning procedures from demonstration—provides a way of automating these complex, repeated tasks. Our system, LAPDOG (*Learning Assistant Procedures from Demonstration, Observation, and Generalization*), addresses four specific characteristics of these tasks: dataflow, loops, unobservable (implicit) actions, and cross-application procedures.

Tasks on the electronic desktop typically involve a *dataflow*: information flows from one action to the next and actions are naturally characterized as having inputs and outputs. For example, we might characterize the action of assembling a quarterly report as requiring the team members' individual contributions as *inputs* and generating the report as its *output*. The document then serves as an input to the subsequent action of sending email.

These tasks also often involve *loops*—one or more actions repeated over a set of objects. For example, the journal editor will typically repeat for each reviewer the process of inviting the reviewer, sending the review, and issuing reminders.

Third, these tasks may involve actions that have *no observable* manifestation. For example, the user might email a meeting summary to all the participants of a meeting but there is no specific, observable action linking the meeting to the recipients of the email message.

Last but not least, all these tasks involve desktop applications or artifacts beyond email—electronic documents, document preparation software, calendar events, projects and teams, managerial relationships, etc.

The Instrumented CALO Desktop

CALO is a cognitive, personalized assistant for the electronic desktop (CALO, 2006). It provides numerous assistant capabilities across standard applications on the Windows platform backed by a variety of machine learning modules. A single, centralized knowledge base serves as both a source of background knowledge and a target for information generated by a collection of engineered harvesters and learned extractors and classifiers.

CALO's instrumentation framework was designed to capture *actions* corresponding to application-level user operations rather than state changes. To date, the Microsoft Office applications, Firefox, Thunderbird, and Windows Explorer have been fairly well instrumented and work continues on providing the corresponding automation hooks as well. The goal of instrumenting at the application level rather than the system level is to capture actions as users tend to think of them when they use an application. For example, users typically think of opening and closing documents rather than pulling down menus and clicking on menu items. The tradeoff is that while we can observe actions across a wide variety of applications, we cannot observe finer-grained actions within individual applications. For example, we can observe an email composition window being opened and attachments being added to it but we cannot observe the individual keystrokes as the user writes the message.

A consequence of this instrumentation framework is that the mapping from observations to actions is straightforward. Unlike in other programming by demonstration systems where instrumentation captures state and the learning task is to induce the actions that will effect the observed state changes, our primary learning task is to generalize the dataflow and the procedural structure seen in a demonstration.

CALO's Task Manager, built on the SPARK agent framework (Morley & Myers, 2004), is responsible for executing CALO procedures, both engineered and learned. Aside from the aforementioned applications, CALO also instruments various aspects of its own operations, including the execution of procedures. Thus, the user can invoke SPARK procedures as part of a demonstration, and LAPDOG can learn procedures involving them.

LAPDOG

LAPDOG takes as input a demonstration consisting of a sequence of actions, and outputs a generalization of this sequence in the form of an executable procedure. Space constraints preclude a detailed presentation of our learning algorithm so we limit discussion to its key features. We begin by defining terminology before presenting an overview of the learning algorithm and the different kinds of generalizations performed by LAPDOG.

Terminology

An action consists of a unique name and a set of typed input and output arguments. Notationally, inputs and outputs are prefixed with + and – respectively. For example, in Figure 1, the second step depicts an action that opens the specified email attachment and outputs the path to its temporary location on the drive.

```
procedure_inputs(-C:\\My Documents)
openEmailAttachment(+ATTACHMENT35 +EMAIL42
                    -C:\\tmp\file.doc)
  splitPathname(+C:\\tmp\file.doc
                -[C:\\tmp -file.doc])
  joinPathname(+C:\\My Documents +file.doc
               -C:\\My Documents\file.doc)
saveAsMSOfficeDocument(+C:\\tmp\file.doc
                       +C:\\My Documents\file.doc)
```

Figure 1. Sample action sequence illustrating dataflow and dataflow completion through information-producing actions (italicized).

A *demonstration* is a sequence of actions depicting user activity. In general, a demonstration expresses a dataflow, with the outputs of an action serving as inputs to succeeding actions. In Figure 1, the *saveAsMsOfficeDocument* action uses as one of its inputs the temporary path output by the previous *openEmailAttachment* action. This is termed a *support* of the input argument by the previous output argument.

All arguments in a demonstration are literals (constant terms). Literals may be scalar or structured; in particular they may be lists or sets of scalars. In the learned procedure, an argument may be an *expression* that is either a literal, a variable, or a function invocation that in turn has input arguments.

The learned procedure, like an action, has inputs and outputs that specify its *parameter signature*. The signature is part of the overall dataflow. Inputs provide values that may support arguments within the procedure. Outputs provide results of the procedure to the invoker, which is either the user or another procedure.

The learned procedure consists of its signature and a sequence of *statements* that are either generalized actions or loops. A loop replaces a repeated sequence of actions with a *loop body* that generalizes the sequences. Loop formulation is a form of *structure generalization*.

Typically, if a literal occurs multiple times in a demonstration, all occurrences are replaced by a single variable in the learned procedure. By *variablizing* the demonstration, LAPDOG generalizes the demonstration, rather than simply learning a verbatim repetition of it. Variablization is a form of *parameter generalization*.

Algorithm Overview

LAPDOG takes as input a demonstration, and outputs a generalization of it in the form of an executable procedure. First, dataflow completion is performed on the demonstration. Parameter and structure generalization are then performed, producing a hypothesis space of alternative procedures consistent with the demonstration. Finally, the hypothesis space is reified, producing a single procedure that is registered with the CALO Task Manager.

Dataflow Completion. In order for the learned procedure to be executable, each input argument of each action must be supported. LAPDOG performs *dataflow completion* to ensure that this condition is met. Because demonstrations may include unobservable actions (e.g., mental operations by the user), LAPDOG performs depth-bounded forward search over a library of *information-producing actions* to find a sequence that will generate required inputs from known outputs. Examples of information-producing actions are knowledge base queries, information extractors, and string manipulation operations. Information-producing actions have no side effects and may thus be executed to verify the expected results without changing the state of the world. If the search fails, the unsupported value may either be left as a constant or made an input parameter to the learned procedure; the choice is left to the user.

In Figure 1, CALO will only observe the user opening the email attachment and then saving out the document. Thus, the dataflow will be incomplete as the second input of *saveAsMSOfficeDocument* will be unsupported. However, through the inclusion of the path and file manipulation actions, the save path is computed and the dataflow completed.

Generalization. In general, dataflow completion produces a set of viable completions. The generalization step considers each in turn, halting when a valid generalization is discovered. Currently, dataflow completion outputs only the shortest completions but alternative control strategies could consider all completions or order them according to some other cost metric.

Sometimes there are alternative parameter generalizations that are valid in a situation. For example, if an argument has two different supports, either may be used to support it. Similarly, there may be multiple valid structure generalizations.

Generalization produces an intermediate representation that retains all alternative generalizations. Statement arguments are represented as sets of expressions, rather than single expressions. To accommodate alternative structure and dataflow generalizations, the procedure body is represented as a set of statement sequences rather than a single sequence.

Supporting and supported arguments are replaced with a common variable, enforcing codesignation in the learned procedure. An argument that is a function of other arguments (e.g., the first element of a list) is replaced with the corresponding function invocation. An argument with multiple supports is replaced with a set of expressions, one per support.

The use of values supported as elements of a list or set triggers a search for matching sequences of actions that can be unified into the body of a loop that iterates over the list or set in question. Nested loops are not considered; however, sequential loops and loops that iterate over multiple lists simultaneously are supported. The repetition need not be verbatim; rather, the repeated sequences must have a consistent generalization. For example, to generalize to an iteration over the values in a list, the first repetition must refer to the first element in the list, the second repetition to the second element, etc.

If a loop body contains actions with output arguments, the loop also *generates* a list or set of values. Each generated list or set is supported in subsequent actions; in

particular, loops may be discovered over the generated list or set.

All parameter and structure generalizations consistent with a given demonstration are retained, forming the space of all valid procedures. Additional demonstrations can then be used to further refine the hypothesis space.

Reification. Reification selects from the hypotheses computed by generalization to form an executable procedure. First, an element from the set of structurally different procedures is selected. Then each argument in this procedure with a nonsingleton set of alternatives is replaced with a single alternative. The resulting procedure is registered with the CALO Task Manager, making it available for execution and for use in subsequent demonstrations.

Learned Desktop Procedures

We now present examples of the desktop procedures LAPDOG can learn from demonstration. A SPARK (Morley & Myers, 2004) procedure specifies a task signature (the task name and its inputs and output parameters) and a body consisting of a sequence of statements such as actions or loops. For brevity and readability, we have edited the variable names and knowledge base queries in the examples presented. Users do not see or directly edit the learned procedures as shown here. Instead, the actions and parameters of the procedure are presented in a special-purpose interface together with a small set of modification options through which users can make constrained edits to the procedures.

Consider the following scenario: administrative assistant Wilma is organizing a meeting for Karen, one of the managers she works for. This meeting is for the PExA project that Karen leads and involves both local and out-of-town participants. Wilma has just sent out email finalizing the date and time for the meeting.

Automatic Meeting Scheduling

Martha receives the meeting email and now wishes to add the meeting to her calendar. This is something she often does with meeting confirmations and seminar announcements and so she teaches CALO a reusable procedure for doing so (Figure 2).

CALO only observes the message being selected and the meeting being added, so LAPDOG needs to deduce the relation between the message and the meeting specifics. In this case, a call to a meeting information extractor completes the dataflow.

This example shows how LAPDOG can utilize other components in CALO—here, a learned meeting information extractor and a scheduling assistant. Because all the CALO components share a knowledge representation, a richer, semantically typed dataflow analysis is supported. The meeting information extractor outputs meaningful date/time objects that can then be fed directly into the *addMeeting* task of scheduling assistant.

```
{defprocedure do_addMeetingFromEmail
  cue: [do: (addMeetingFromEmail +$email
+$meetingName -$meeting)
  precondition: (True)
  body:
  [seq:
    [do: (selectEmail +$email)]
    [do: (extractMeetingInformation +$email
-$datetime -$location)]
    [do: (addMeeting +$meetingName +$datetime
+$location -$meeting)]]}
```

Figure 2. Learned procedure for adding a meeting from email, with the information-producing action inserted by LAPDOG in italics.

Meeting Information Packet

It is a week before the meeting. The hotel block has been booked, the attendee list confirmed, the agenda finalized, and various other preparations completed. Wilma now has to send a meeting information packet to all the attendees. As the admin for several busy project leaders, this is a frequent task for her and so she teaches CALO a procedure to accomplish the task (Figure 3).

```
{defprocedure do_sendInfoPack
  cue: [do: (sendInfoPack +$dir +$hotelSt
+$hotelZip +$meeting)]
  precondition: (True)
  body:
  [seq:
    [do: (queryForEmailAddrOfMeetingParticipants
+$meeting -$partEmail)]
    [do: (queryForEmailAddrOfOrganizer +$meeting
-$orgEmail)]
    [do: (getDrivingDirections +$hotelSt
+$hotelZip +"333 Ravenswood Ave." +"94025"
-$url)]
    [do: (openUrl +$url)]
    [do: (openFolder +$dir)]
    [do: (getFilesInFolder +$dir -$filelist)]
    [do: (createFromTemplate +"Meeting materials
for $_VAR3" +[... $meeting ...] -$subject)]
    [do: (createFromTemplate +"Hello everyone.
Attached is the information packet for the
$_VAR3. Here also are the driving directions to
SRI: $_VAR5. Let me know if you have any
questions. - Wilma" [... $meeting ... $url ... ]
-$body)]
    [do: (openComposeEmailWindow +$partEmail
+$orgEmail +[] +$subject +$body +$filelist]
    [do: (waitForSendEmailByUser)]]}
```

Figure 3. Learned procedure for sending information packet and driving directions for the meeting, with information-producing actions in italics and previously learned task in bold.

Knowledge Base Queries. To complete the dataflow between the meeting and the email recipients in this example, LAPDOG inserts knowledge base queries to generate the addresses of the meeting participants and the meeting organizers from the meeting. Similarly, to support

the email attachments, LAPDOG inserts a knowledge base query to obtain the files in the specified folder.

Driving Directions. The procedure involves the use of a previously learned *getDrivingDirections* procedure to provide directions from the hotel to the meeting site (SRI). Note that the *getDrivingDirections* step includes constants for the destination address. Effectively, it has been specialized in this procedure to provide directions specifically to SRI

Email Template. The procedure in Figure 3 uses string templates to form the email subject and body, where terms of the form *$VAR_i* in the template string are to be replaced by the i^{th} element in the subsequent argument list. These templates are generated during LAPDOG's dataflow completion step when LAPDOG recognizes that the action *createMessageFromTemplate* could potentially generate the email body with the right template. It generates this template through the application of an inverse action that creates a template given a message and known strings.

Partial Automation. A final thing to note about the learned procedure is that it is only partially automated. Certain actions (in this case, sending email) marked as nonautomated are converted to actions involving waiting for the user to perform them. Thus, rather than automatically sending out the email message, CALO opens up an email composition window, prepopulated according to the dataflow it has generalized, and then waits for the user to actually send the message. This lets the user verify the message to be sent and also allows minor changes to be made during execution.

Saving the Information Packet

```
{defprocedure do_saveAttach
 cue: [do: (saveAttach +$email +$newfdr)]
 precondition: (True)
 body:
 [seq:
  [do: (selectEmail +$email)]
  [do: (queryForEmailAttachments +$email
-$attList)]
   [forin: -$att +$attList
    [seq:
     [do: (openEmailAttachment +$att +$email
-$oldpath)]
     [do: (splitPathName +$oldpath -$oldfdr
-$file)]
     [do: (joinPathName +$newfdr +$file
-$newpath)]
     [do: (saveAsMSOfficeDocument +$oldpath
+$newpath)]]]]}
```

Figure 4. Learned procedure for saving information packet (induced loop in bold, inserted information-producing actions in italics).

Martha has now received the email containing the various attachments and, as usual, she wishes to open all the documents and also save them out to a single folder on her computer. She demonstrates this process, which involves opening up each attachment and then saving it out to a specified folder. LAPDOG generalizes this into a loop over all the attachments of any email message (Figure 4). Note that the loop body includes information-producing actions to complete the dataflow as discussed in an earlier section.

Summary and Related Work

The examples presented illustrate the kinds of desktop procedures involving email that LAPDOG can learn from demonstration. By searching over information-producing actions and knowledge base relations, it can insert dataflow-completing steps so that it can learn procedures even from demonstrations involving unobservable actions. Through parameter generalization, LAPDOG converts arguments into variables and more general expressions. Through structure generalization, it acquires looping constructs over sets and lists, further extending the applicability of its learned procedures. Finally, through the recognition of nonautomated actions, LAPDOG can learn procedures that are only partially automated, leaving the user to perform those steps that CALO cannot execute.

Much of the work on using machine learning to assist in email-related activities has focused on classification tasks—for example, predicting which folders to file email (Segal & Kephart, 2000), whether the user will reply to a message (Dredze et al., 2005), the recipients who should be cc'd (Pal & McCallum, 2006), or the intent of a message (Cohen et al., 2004). We can utilize these learned components in dataflow completion—for example, using a cc predictor instead of knowledge base queries to provide email recipients. We could also use the predictors as procedure triggers. For example, if CALO observes that the user always invokes *addMeetingFromEmail* after reading an email meeting request, it can learn to suggest this procedure in these situations, Since the learners' output is typically uncertain, we would need to extend our dataflow criteria to accommodate partial but likely supports.

Previous systems that learn from demonstration have typically addressed learning within a single, usually proprietary or reimplemented, application, due to instrumentation and automation requirements rarely being met by existing applications and platforms. On the Macintosh, platform-based learning systems have been able to rely on AppleScript to support communication between applications (e.g., Paynter & Witten, 2001; Lieberman, 1998). CALO serves a similar function for us, providing a shared infrastructure between the Windows OS, selected Windows applications, and the CALO components, including LAPDOG.

Our work has focused on automating highly structured processes, which calls for learning general procedures. However, there are also many desktop activities that are only semi-structured—for example, going through our email, writing a paper, or developing code. An alternative assistance paradigm is to instead continually predict the user's next action, based on larger task or activity models.

For email workflows, Kushmerick et al. (2006) present a promising start involving the automated discovery of structured activities and email activity classification, while TaskTracer (Stumpf et al., 2005) presents a similar approach to task prediction for the desktop

Deployments and Future Work

LAPDOG's most recent deployment was tested during the CALO Test in September 2007. The CALO Test is an annual evaluation of the CALO system, and task learning is just one of many technologies evaluated. LAPDOG, in particular, was used to acquire two procedures—one for finding contact information on the web and another for sending email travel notification to appropriate parties. While by no means a thorough evaluation of the system, users were able to successfully teach CALO these procedures through demonstration. Anecdotal evidence indicates that users found the paradigm of learning from demonstration the most natural and easy to use.

In late 2006, we embarked on an effort to transition LAPDOG into a prototype version of a military application for collaborative visualization. LAPDOG was used to automate a variety of tasks, including briefing content creation and collection, operations, and intelligence analysis. The resulting system was used in a number of field studies in 2007, involving both expert and novice users, and the feedback has been predominantly positive. Users see the technology as having promise not only as an automation tool, but also as a training tool. A large-scale evaluation of the system is planned for late 2008.

We are continuing to extend LAPDOG's learning capabilities in various ways. We are developing techniques for acquiring loops over implicit lists/sets, to address the current requirement that the sets/lists for looping already be explicitly supported. We are also designing a more efficient, interleaved control flow between dataflow completion and generalization. To help disambiguate between multiple hypotheses, we are investigating both interactive learning, where the key will be to balance the need for information with the burden on the user, and the use of multiple demonstrations, where the challenge is to be able to incorporate them over time. Finally, LAPDOG is part of an integrated task learning effort in CALO and we are exploring the combination of more user-intensive learning techniques such as learning from instruction and procedure composition with learning from demonstration.

Acknowledgments. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185.

References

Bellotti, V., Ducheneaut, N., Howard, M. A., & Smith, I. E. (2003). Taking email to task: the design and evaluation of a task management centered email tool. *Proceedings of the 2003 ACM Conference on Human Factors in Computing Systems*.

Bellotti, V., Thornton, J. D., Chin, A., Schiano, D. J., & Good, N. (2007). TV-ACTA: embedding an activity-centered interface for task management in email. *Proceedings of the 4th Conference on Email and Anti-Spam*.

CALO (2006). *Cognitive Agent that Learns and Organizes*. http://caloproject.sri.com/

Cohen, W. W., Carvalho, V. R., & Mitchell, T. M. (2004). Learning to classify email into "speech acts". *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*.

Cypher, A. (Ed.) (1993). *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press.

Dredze, M., Blitzer, J., & Pereira, F. (2005). Reply expectation prediction for email management. *Proceedings of the 2nd Conference on Email and Anti-Spam*.

Kushmerick, N. & Lau, T. (2005). Automated email activity management: an unsupervised learning approach. *Proceedings of the 2005 International Conference on Intelligent User Interfaces*.

Kushmerick, N., Lau, T., Dredze, M., & Khoussainov, R. (2006). Activity-centric email: a machine learning approach. *Proceedings of the 21st National Conference on Artificial Intelligence*.

Lieberman, H. (Ed.) (1998). Integrating user interface agents with conventional applications. *Proceedings of the 1998 International Conference on Intelligent User Interfaces*.

Lieberman, H. (2001). *Your Wish is My Command: Programming by Example*. San Francisco, CA: Morgan Kaufmann.

Morley, D. & Myers, K. (2004). The SPARK agent framework. *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems*.

Pal, C. & McCallum, A. (2006). CC prediction with graphical models. *Proceedings of the 3rd Conference on Email and Anti-Spam.*

Paynter, G. & Witten, A. (2001). Domain-independent programming by demonstration in existing applications. In *(Lieberman, 2001).*

Segal, R. & Kephart, J. (2000). Incremental learning in SwiftFile. *Proceedings of the 17th International Conference on Machine Learning*.

Stumpf, S., Bao, X., Dragunov, A., Dietterich, T. G., Herlocker, J., Johnsrude, K., Li, L., & Shen, J. (2005). Predicting User Tasks: I know what you're doing. *Proceedings of the AAAI Workshop on Human Comprehensible Machine Learning.*