LiveAction: Automating Web Task Model Generation

SALEEMA AMERSHI, University of Washington
JALAL MAHMUD and JEFFREY NICHOLS, IBM Research Almaden
TESSA LAU, Willow Garage
GERMAN ATTANASIO RUIZ, IBM Argentina

Task automation systems promise to increase human productivity by assisting us with our mundane and difficult tasks. These systems often rely on people to (1) *identify the tasks* they want automated and (2) *specify the procedural steps* necessary to accomplish those tasks (i.e., to create task models). However, our interviews with users of a Web task automation system reveal that people find it difficult to identify tasks to automate and most do not even believe they perform repetitive tasks worthy of automation. Furthermore, even when automatable tasks are identified, the well-recognized difficulties of specifying task steps often prevent people from taking advantage of these automation systems.

In this research, we analyze real Web usage data and find that people do in fact repeat behaviors on the Web and that automating these behaviors, regardless of their complexity, would reduce the overall number of actions people need to perform when completing their tasks, potentially saving time. Motivated by these findings, we developed LiveAction, a fully-automated approach to generating task models from Web usage data. LiveAction models can be used to populate the task model repositories required by many automation systems, helping us take advantage of automation in our everyday lives.

Categories and Subject Descriptors: H5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia—*User issues*

General Terms: Algorithms, Design, Experimentation, Human Factors

Additional Key Words and Phrases: Task modeling and automation, machine learning, Web usage mining, Web repetition

ACM Reference Format:
Amershi, S., Mahmud, J., Nichols, J., Lau, T., and Ruiz, G. A. 2013. LiveAction: Automating Web task model generation. *ACM Trans. Interact. Intell. Syst.* 3, 3, Article 14 (October 2013), 23 pages.
DOI: http://dx.doi.org/10.1145/2533670.2533672

1. INTRODUCTION

Task automation systems promise to increase human productivity by assisting us with our mundane and difficult tasks or autonomously completing them on our behalf (e.g., [Anupam et al. 2000; Dragunov et al. 2005; Lau et al. 2004, 2010; Leshed et al. 2008; Spaulding et al. 2009; Sun et al. 2006]). Yet, aside from in some targeted domains and specific types of tasks (e.g., email filters, spreadsheet macros, scripts for online multiplayer games), few people take advantage of task automation in their everyday lives.

This research was done when S. Amershi was at IBM Research – Almaden. She is now at Microsoft Research.
This research was done when T. Lau was at IBM Research – Almaden.
Author's address: S. Amershi; email: samershi@cs.washington.edu.

DOI: http://dx.doi.org/10.1145/2533670.2533672

A well-known barrier to widespread adoption of task automation systems is the need to create the *task models* that make automation possible [Allen et al. 2007; Gil et al. 2011; Leshed et al. 2008; Oliver et al. 2006; Shen et al. 2009; Spaulding et al. 2009]. A task model is some executable representation of "how-to" knowledge or the procedural steps necessary to accomplish a well-defined goal (e.g., a rule specifying that messages from Bob should go into a "Work" folder, a macro computing the running total of purchased items, a script outlining how to cast multiple spells on a hostile warlock). Recognizing this barrier has led to an emergence of authoring systems that ease some of the burden of task model creation (e.g., [Allen et al. 2007; Bergman et al. 2005; Garland et al. 2001; Gil et al. 2011; Lau et al. 2004; Leshed et al. 2008; Shen et al. 2006; Spaulding et al. 2009; Sun et al. 2006]). Yet, task automation remains uncommon.

We argue that a more fundamental problem lies in a basic assumption made by current task automation systems: that people can and will recognize repetitive tasks worthy of automating. Our interviews with users of a demonstration-based Web task automation system revealed that most people only thought to automate tasks if they noticed themselves performing it numerous times in a row manually. Others only automated complex but infrequently performed tasks, believing that straightforward everyday tasks do not warrant automation, even if they are repetitive. This latter finding could be partially attributable to usability issues and the difficulty of creating task models. However, most of our interviewees also stated that they do not even believe they do repetitive tasks. Not surprisingly, then, automation systems that rely on human-driven task model creation risk being underused because people have difficulty identifying tasks they can or should automate.

We propose that *any* repetitious behavior should be a candidate for automation because automating things we have done before frees up time for us to do new things. We analyzed real Web usage logs from seven people amounting to over 19 months worth of data, including over 62,000 actions, and found that people do in fact repeat behaviors on the Web (i.e., sequences of actions within Web pages or sites). Moreover, fully automating these behaviors, regardless of their simplicity or complexity, would result in overall action savings of 14.3% per person on average (4.7% SD) and close to 20% for some people.

Drawing on the premise that people find it difficult to identify automatable Web-based tasks and our initial evidence that automating repetitive Web action sequences can reduce the amount of work we must perform to complete those tasks (e.g., an automated script may be run instead of manually executing a repetitive task), we explored a machine learning-based approach to Web task model construction. Our LiveAction approach automatically detects repetition in Web usage logs (obtained from CoScripter Reusable History, previously ActionShot [Li et al. 2010], recordings), generalizes from the detected patterns, and constructs executable task models from those patterns as finite-state-automata. Our machine learning-driven approach to task model generation is intended to (1) help people identify automatable tasks by automatically detecting repetitious behavior and (2) facilitate task automation by modeling that repetitious behavior. Our evaluations show that LiveAction task models can automate repetitious next-steps with an overall accuracy of 78.5% (84.2% if people are allowed to guide the automation at each step by selecting from potential alternatives presented by LiveAction). Therefore, our LiveAction models could potentially populate the task model repositories required by many automation systems (e.g., CoScripter [Leshed et al. 2008]) and help us take advantage of task automation in our everyday lives.

Specifically, our contributions are:

— A semiformal interview study with users of a Web task automation system [Leshed et al. 2008] investigating when people are prompted to create task models with

the purpose of automating their Web-based tasks and identifying issues preventing them from doing so. Our interviews reveal that people find it difficult to identify tasks to automate and many do not believe they perform repetitive tasks for which automation would help.
— An analysis of real, element-level Web usage logs (e.g., logs of button and link clicks or entering text) to test our hypothesis that the benefit of task automation lies in automating repetitive behaviors, regardless of complexity. Our findings show that people do in fact repeat their Web behaviors and that fully automating these would result in action savings of 14.3% on average and up to almost 20% for some people. Our results complement research in Web usage mining typically performed at the page level, as permitted by server logs.
— The presentation and evaluation of LiveAction, a fully-automated, machine learning-based approach to task model generation based on repetition detected in Web usage logs. Our evaluations indicate that LiveAction models can automate repetitious next-steps with an accuracy of 78.5% (84.2% with human feedback).

2. RELATED WORK

Our research contributes knowledge about how people behave on the Web. Specifically, our analysis of repetition in Web behavior contributes to research in Web usage mining. The goal of Web usage mining is to detect and analyze patterns of user behavior on the Web [Kosala and Blockeel 2000]. Mining is typically performed offline on Web server logs, permitting pattern discovery at the page-level. Page-level analysis has established that navigational patterns on the Web are dominated by revisitation of previous Web pages [Adar et al. 2009; Cockburn and McKenzie 2000]. Our repetition analysis complements this body of work by revealing that people also demonstrate repetitive behavior *within* Web pages. This analysis is made possible by our element-level mining of CoScripter Reusable History logs [Li et al. 2010] which records Web page actions (e.g., button and link clicks, entering text). Understanding that not only do people often revisit Web pages, they also routinely perform the same sequences of actions within those Web pages, we can further improve our user experience on the Web.

Web usage mining has also been used to personalize ads and e-commerce recommendations [Kohavi et al. 2004], direct our navigation [Jin et al. 2005; Pitkow and Pirolli 1999], prefetch and cache pages we are likely to visit [Mabroukeh and Ezeife 2009], and identify user interests to improve Web site structure and functionality [Heer and Chi 2002]. While these and most previous work in Web usage mining have focused on page-level support [Pierrakos et al. 2003], our approach operates at the element-level for the purpose of Web *task automation*, a service unachievable through page-level modeling. For example, page level modeling (where states are Web pages) cannot capture interactions of Web page elements inside a single Web page (e.g., filling a Web form). Detecting and modeling user behaviors at the element-level also introduces several new challenges. For example, to provide individualized automation, we consider Web usage data from a single user rather than multiple users, limiting the amount of data from which we can discover patterns. As another example, our element-level data is inherently noisier than page-level server side logs because it inherits all of the noise present at the server level (e.g., erroneously navigating to a page and then back tracking) while containing additional element-level noise (e.g., clicking on page elements extraneous to the intended task). In addition, server-side logs mainly consist of path-structured URLs facilitating pattern detection on similar pages (i.e., detecting similar URLs), whereas our element-level data contains unstructured text (e.g., 'click the "add to cart" link', 'enter "your name" in the "username" textbox') and therefore requires additional processing before pattern detection is possible. We address these and other challenges in our algorithmic solution to element-level modeling.

Prior research on task model creation has advocated demonstration-based authoring. CoScripter [Leshed et al. 2008] is an automation system that enables people to record Web-based tasks and then replay them to automate those tasks in the future. Others have proposed authoring via natural language instruction [Gil et al. 2011] or a combination of natural language and programming by demonstration [Allen et al. 2007]. All of these require forethought in deciding which tasks to automate, whereas the CoScripter Reusable History (previously ActionShot) [Li et al. 2010] and Smart Bookmarks [Hupp and Miller 2007] systems enable retroactive authoring by recording user actions and then allowing them to manually search for, extract, edit and rerun previous actions. As with LiveAction, all of these systems relieve a person from dealing with the underlying task model representation. However, they also require human initiative and guidance in contrast to automatically learning models from previous actions. In addition, because these approaches infer task models from a single demonstration (and possibly some additional interaction to resolve errors and ambiguity), the models learned tend to be specific rather than generalized.

Other researchers have proposed learning more generalized task-models from multiple demonstrations of a task. For example, Sheepdog [Law et al. 2004] learns a task model from several expert demonstrations of a task. These models can then be used to automate technical support solutions. Similarly, the Collagen [Garland et al. 2001], ITL [Spaulding et al. 2009], and DocWizards [Bergman et al. 2005] systems use machine learning to construct task models from user demonstrations and subsequent annotations or edits. Again, with these approaches a person is charged with the task of demonstrating a procedure for the purpose of teaching a system to learn a model. LiveAction differs by acknowledging that a person's primary objective is to accomplish the intended tasks, not to train models. As such, LiveAction learns models from previously observed actions as a person naturally performs their tasks. Furthermore, our approach assumes noisy data inherent in raw user actions rather than clean expert demonstrations.

Research on automatically learning task models from natural user actions has typically employed supervised techniques in which a person provides manually labeled task examples. For example, TaskPredictor [Shen et al. 2006] uses manually labeled task examples to learn a classifier-based model for predicting those tasks in the future. Here the model represents task resources (e.g., documents and applications) rather than the procedural steps of the task. As with our approach, the Guide-O system [Sun et al. 2006] learns a procedural, finite-state-automata-based task model from sequences of real Web transactions. The task models are then used to guide visually impaired users in completing online tasks. However, again the sequences must be manually identified and hand labeled before Guide-O can construct and use task models.

Closely related to our work is that involving unsupervised machine learning for generating task models from natural data. These approaches remove the burden of task model generation from already overloaded end-users. The SWISH system [Oliver et al. 2006] uses unsupervised clustering techniques to automatically model and detect tasks from raw user interaction data on the desktop. Like with the TaskPredictor system [Shen et al. 2006] however, task models in SWISH represent sets of related resources rather than procedural steps.

Work that shares our motivation of automatically detecting and automating repetitive tasks is Cypher's seminal programming by demonstration work on the Eager system [Cypher 1991]. Eager detects looping behavior on HyperCard by searching through a person's interaction history for events similar to the current action. LiveAction extends this idea by automatically detecting and modeling more complex behaviors (e.g., branching) and is more tolerant to noisy data. Most similar to our work is that of Mahmud et al. [2009] and Shen et al. [2009]. In a follow up to their

Guide-O work [Sun et al. 2006], Mahmud et al. [2009] generate finite-state-automaton models from unlabelled or partially labeled click-streams. LiveAction differs from this in that it learns executable task models (where each state is a Web action such as clicking a button or entering text in a textbox) rather than conceptual models (where each state represent semantic concept such as "search result" or "item taxonomy") for guiding visually impaired users through their tasks.

Furthermore, our approach assumes unlabelled, unsegmented action sequences and learns multiple task-specific models rather than one model characterizing all transactions. Shen et al. [2009] automatically generate desktop workflows from interaction histories. A workflow is a model of the resources (e.g., emails and documents) and actions taken on those resources (e.g., attaching files to emails, copying and pasting documents) during the completion of a business related process. As with our approach, workflow models here are automatically generated based on repetition frequency. These workflow models can be used for tracking task state. However, in contrast to our LiveAction models, they typically do not contain the detailed actions necessary to fully automate those tasks. In other words, the workflow model actions are represented at a higher level (e.g., attaching a file or sending an email). However, in our representation, actions are represented as element level Web actions (e.g., clicking a button or selecting an element from a combo box).

Our work is also related to task model representation in human computer interaction (HCI), such as GOMS (Goals, Operator, Methods, Selection rules) [Card et al. 1983; Foley et al. 1991], CTT (ConcurTaskTrees) [Paterno et al. 1997] and HTA (Hierarchical Task Analysis) [Annett and Duncan 1967]. In a GOMS model [Stuart et al. 1983], goals are what users want to accomplish, operators are actions to reach the goal, methods are sequences of operators to reach the goal and selection rules are used to select a certain method from a set of methods. A method to generate a GOMS model from user interaction is also described in Hudson et al. [1999]. Our automata-based task model is conceptually similarity to the GOMS representation as a task model specifies a task to be accomplished (a goal), contains a sequence of executable actions (operators) and often contains branching to select an action from multiple possible actions (similar to selection rules). HTA [Annett and Duncan 1967] describes tasks in terms of hierarchy of operations which people use to reach a goal. Currently, our task model does not explicitly contain such a hierarchy of operations, or tasks defined in terms of multiple subtasks organized in a hierarchy. ConcurTaskTrees (CTT) is another notation used for task model specification and has been used for model-based user interface design [Paterno et al. 1997]. It also uses hierarchical structure of tasks which is different from our current representation using a finite state automaton. An algorithm to generate task models represented in CTT notation from input Web pages is described in [Paganelli and Paterno 2003]. This algorithm does both page level analysis and element level analysis to create task models in CTT notation for a Web page and an entire Web site. Task models created using this approach can represent an entire set of activities in a Web site and is useful for usability evaluation, model analysis, and user interface design. In contrast to this design-centric approach, we propose the creation of task models from user-generated logs, which can capture specific user activities including repetitious activities. This is useful for personalization in a Web site. Furthermore, the CTT approach requires the availability of server-side Web site codes in order to correctly represent a task model. Our approach to creating task models requires client-side Web access logs, which are easier to obtain then server side codes.

3. FORMATIVE TASK MODEL CREATION INTERVIEW STUDY

To better understand when people are prompted to create task models with the intention of automating their tasks and to identify what prevents them from doing so,

we conducted semiformal interviews with eight users (three female) of the CoScripter Web task automation system [Leshed et al. 2008] who were all members of our research lab. CoScripter task models are linear, pseudo-natural language scripts that can be executed to automate Web-based processes. To create a CoScripter task model, a person demonstrates a task while CoScripter records their browser actions. CoScripter presents its recorded model as a sequence of human-understandable and directly editable actions (e.g., 'enter "your name" in the "username" textbox', 'click the "search" button'). To automate that task in the future, a person simply steps through the actions or plays them back within the CoScripter environment. People can also share and reuse models via the online CoScripter repository.

We asked each of our interview participants to identify two to three CoScripter models that they had previously authored and recall their creation (e.g., "Explain why you decided to create this model," "When did you decide to create this model?"). We then asked general questions about the types of tasks they tended to create or not create models for and why (e.g., "Do you always create models for tasks you know can be automated? Why or why not?").

3.1. Results

Our participants cited two main instigators of task model creation in CoScripter:

— *Realizing a task is repetitive after several manual executions.* Seven out of eight of our participants said they created some of their task models only after noticing themselves manually repeating a task. These participants recalled repeating their task many times in a row or over a short period of time (even up to eight times in a row) before realizing it and then deciding to create a task model for it. Some of these task models were used for transient tasks (e.g., "repeatedly searching for a specific piece of furniture on Craigslist") while others were used for regularly occurring tasks (e.g., "logging into weekly e-meetings").
— *Recognizing a complex task in advance.* Only three of our participants said that they created task models prior to executing those tasks. In these cases, participants recognized the benefit of creating a task model because the task was perceived as complex (e.g., "creating a new teleconference account"). Some of our participants characterized such tasks as "long and complicated" or "obnoxious" even though they were preformed infrequently (e.g., "every few months" or "once a year").

When questioned about their reasons for not creating task models to automate their tasks, our participants revealed that they:

— *Do not believe they have many repetitive tasks.* Surprisingly, five of our eight participants stated that the main reason they did not create task models was because they did not feel they performed many repetitive tasks.
— *Feel the cost of creating task models is too high.* As in previous research, our participants also cited high overhead as a reason for not creating task models (five out of eight). One of our participants commented that "only complicated tasks are worth creating [models] for, not everyday tasks." Another stated that "most of the time the steps in my task are relatively straightforward" (e.g., "going to a site to read the news" or "checking a bank statement") and therefore not worth the effort of creating a task model.

While these findings provide initial evidence that people have difficulty recognizing tasks they can or should automate, further research is necessary to validate these results with a larger and more diverse population. However, given that our interview participants would be considered computer experts, it is conceivable that non-experts might have more difficulty recognizing or creating task models. This suggests that

Table I. Web Usage Log Statistics Collected for Our Repetition Analysis and Task Model Generation Evaluations

PID	Days	Actions	Domains	Sequences	Avg. Seq. Length
1	271	27,538	109	1,557	11.4
2	141	19,371	128	1,377	10.4
3	61	4,472	69	367	10.8
4	42	6,393	66	435	9.0
5	24	490	5	38	16.0
6	21	2,958	9	63	31.7
7	21	1,371	25	105	10.5
Total	*581*	*62,830*	*411*	*3,940*	*NA*
Avg.	*83*	*8,975.7*	*58.7*	*562.9*	*14.3*
SD	*93.1*	*10,444.1*	*48.3*	*638.1*	*8.0*

relying on people to supply automation systems with manually created task models could impede their everyday use of automation.

4. REPETITION ANALYSIS

Participants in our interview study did not believe they perform many repetitive tasks. They also felt that only complex (and typically infrequent) tasks were worth automating. In contrast, we hypothesized that the benefit of task automation systems lies in automating repetitive tasks, regardless of complexity or frequency.

4.1. Analysis

To test our hypothesis, we collected actual Web usage data from seven people in our research lab (two female, five from our interview study) and analyzed the amount of repetition exhibited per person. Web usage logs were obtained via CoScripter Reusable History [Li et al. 2010] (CRH), a browser plug-in that records element-level Web actions. Our participants had installed CRH in at least one of the Web browsers they used regularly for at least two and up to 13 months (amounting to a total of 581 days worth of usage data). From this set, we obtained 62,830 Web actions from 411 unique domains (e.g., "amazon.com," "acm.org"). Table I shows the usage data we obtained per participant.

Note that the prevalence of tabbed browsing complicates the problem of measuring repetition as it allows people to concurrently progress through multiple tasks [Dubroy and Balakrishnan 2010]. Therefore, to avoid the added complexity of distilling distinct tasks from actions collected over multiple tabs, we restrict our analysis of repetition to behaviors within a single Web domain.

Before we could measure repetition, we first had to preprocess (or transform) each participant's CRH data into a format suitable for pattern discovery. After preprocessing, we cast the problem of estimating repetition as a problem of measuring overlap within sequences. Measuring sequence overlap is a well-studied problem with important applications in text processing, data compression and bioinformatics (e.g., DNA sequencing) [Gusfield 1997].

Segmenting Logs into Action Sequences. To measure task repetition, we first had to extract *sequences of actions* over which we could detect repetition (or overlap). To obtain action sequences, we segmented each person's log data by estimating task boundaries as follows. For each domain, we first segmented CRH logs per day (assuming tasks did not span multiple days). Then we segmented logs within each day using a time-based heuristic as follows. We computed the mean time between consecutive domain actions (excluding those spanning day boundaries), and then segmented the logs

when the time between consecutive actions exceeded one standard deviation of the mean. Intuitively, this heuristic assumes that the time between consecutive actions within a task is less than the time between actions across task boundaries. We remove sequences of two actions or less from our resulting set of sequences as these likely do not contain repetitious behavior and add noise. Table I shows the number of sequences we obtained using these heuristics per participant and the average length of each sequence (averaged across domains). Note that the average length of a sequence is the average number of actions in a sequence, which is 14.3 for our Web usage logs with a standard deviation of 8.0.

Mapping Low-Level Actions to Action Classes. Because element-level actions contain unstructured text, conceptually equivalent actions are often represented in a variety of different ways (e.g., "Click the 'login' button" versus "Click the 'Log-in' button"). Therefore, to treat equivalent actions the same during repetition discovery, we mapped individual CRH actions into related *action classes*, which are classes of similar actions in Web pages such that members of an action class perform similar functions (e.g., entering a password into a textbox in a login form). This concept is similar with our notion of *instruction-class* for test scripts [Mahmud and Lau 2010]. We used a conservative heuristic for mapping actions to classes which only maps highly similar strings to the same class (and ensures that dissimilar strings such as "Enter username" and "Enter password" are never mapped to the same class).

First, we interpreted each CRH log action as a Web page command as in [Lau et al. 2009; Li et al. 2010; Mahmud and Lau 2010]. Each command contains three parts: the ActionType indicating the element-level action taken (e.g., click, enter, select), the ObjectType representing the type of page element on which the action was taken (e.g., button, link, radiobutton, checkbox, textbox), and the ObjectLabel identifying the target element (e.g. caption of a button, link text, label of a textbox). The system in Li et al. [2010] which generates CRH logs relies on heuristics to extract a human-readable label for the target of each action. For example, if an action is on a Web page element that contains a caption or label field, then that becomes the ObjectLabel. If no such caption or label field is present, then it uses heuristics, such as using the accessibility text (for image elements) or nearby text (e.g., typically appear in the left side of a Web page element in a Web form) to assign the ObjectLabel.

For example, the "Click the 'login' button" action would be interpreted as the following command: <'click','button','login'>. As another example, "Enter '12345678' into the 'Account Number' textbox" would be interpreted as <'enter','textbox','Account Number'>. Note that for mapping an action to an action class, we do not consider the ObjectValue, which is '12345678' in this example. As yet another example, "click the 'my account' link" would be interpreted as <'click','link','my account'>.

Next, we map commands to action classes sequentially as they were observed. That is, an incoming command is mapped to an existing action class (possibly containing multiple commands) if it met the following criteria:

— The ActionType and ObjectType of the incoming command is the same as that of the action class. For example, the ActionType and ObjectType of the actions "Click the 'international' link" and "Click the 'US' link" are similar. However, the following two actions have a dissimilar ObjectType but similar ActionType: "Click the 'international' link" and "Click the 'international' tab."
— The difference between the ObjectLabel of the incoming command and any command in the class is less than some threshold, where the difference is measured as the Levenshtein string edit distance between labels. Our experiments showed that an edit distance threshold of three was sufficient for achieving our goal of conservatively mapping similar actions together. For example, consider the following two actions:

"Select 'California' from the 'State Name' listbox" and "Select 'New York' from the 'State-name' listbox". Their ObjectLabel are "State Name" and "State-name". We compute the edit distance between these strings and consider them as equivalent since the distance is less than three.

If no such class exists, we create a new action class for the incoming command. Let us consider the following sequence of actions:

go to "http://www.cheaptickets.com"
enter "new york" into the first "To City name or airport" textbox
enter "08/10/10" into the first "Leave" textbox
enter "08/15/10" into the first "Return" textbox
enter "san jose" into the first "From City name or airport" textbox
enter "LGA" into the "To City name or airport" textbox
turn on the first "incl. nearby airports" checkbox
click the "Search Flights" button

The first action is mapped to action-class A: < go to, ,"cheaptickets"> (note that for loading a Web page into a browser, the ObjectType field is empty and ObjectLabel is the URL of the Web page). The next action is mapped to the action-class B: <enter, textbox, "To City name or airport">. The following three actions are mapped to action classes C:<enter, textbox, "Leave">, D:<enter, textbox, "Return"> and E:<enter, textbox, "To City name or airport">. The 6th action is mapped to the action-class B and the 7th action is mapped to the action-class F:<turn on, checkbox, "incl. nearby airports">. The final action is mapped to the action-class G:<click, button, "Search Flight">. Thus the sequence becomes ABCDEBFG, when expressed as a sequence of action-classes. Observe that the action-class B is repeated in this sequence. As another example, consider the following sequence of actions:

go to "www.amazon.com"
enter "Garmin" into the "Search for" textbox
append "Garmin fr60" to the "Search for" textbox
click the "Go" button
click the "Garmin FR60 Men's Red Fitness Watch (Includes Heart Rate Monitor and USB ANT Stick)" link
select "R60 Men's Black Fitness Watch Bundle (In..." from the "Select" listbox
select "R60 Men's Red Fitness Watch (Includes He..." from the "Select" listbox
click the first "Add to Shopping Cart" button
click the first "Close" link
enter "garmin premium heart rate monitor" into the "Search for" textbox
click the first "Go" button
enter "garmin premium strap" into the "Search for" textbox
click the first "Go" button
click the "Garmin Premium heart rate monitor (soft strap)" link
click the first "Add to Shopping Cart" button
click the "Proceed to Checkout" link
enter your password into the "My Password is" textbox
click the "Continue" button
click the first "One-Day Shipping" button
click the first "Place Your Order" button

This sequence is mapped as the following sequence of action-classes: ABCDEFFGHBDBDIGJKLMN, where, A: <go to, , "amazon">, B: <enter, textbox, "Search for">, C: <append, textbox, "Search for">, D: <click, button, "Go">, E: <click, link, "Garmin FR60 Men's Red Fitness Watch (Includes Heart Rate Monitor and USB ANT Stick)", F: <select, listbox, "R60 Men's Black Fitness Watch Bundle (In...">, G: <click, button, "Add to Shopping Cart">, H: <click, link, "Close">, I: <click, link, "Garmin Premium heart rate monitor (soft strap)">, J: <click, link, "Proceed to Checkout">, K:

<enter, textbox, “My Password is”>, L: <click, button, “Continue”>, M: <click, button, “One-Day Shipping”>, and N: <click, button, “Place Your Order”>.

Detecting Repetition. After segmenting each participant’s CRH logs and mapping individual actions onto action classes, we are left with one set of action class sequences per domain over which we can estimate repetition (i.e., measure sequence overlap). However, because natural Web usage data is noisy and may contain spurious actions, this requires detection of non-contiguous overlap (e.g., sequence ‘a-b-c’ and ‘a-c-d’ share the common, non-contiguous, subsequence ‘a-c’). Therefore, we use an accepted metric for computing non-contiguous sequence overlap from string processing applications known as longest common subsequence (LCS) [Bergroth et al. 2000].

The LCS metric computes the longest common subsequence between a pair of sequences (we normalize by the average length of the two sequences). For example, sequence ‘a-b-c’ and ‘a-c-d’ has an LCS value of 2/3. We interpret this to mean that these two sequences have ∼67% overlap. Note that sequence order matters in LCS (e.g., while ‘a-b-c’ and ‘c-b-a’ share the same actions, these sequences would only produce an LCS value of 1/3).

Because we are interested in the amount of overlap in a set of sequences (i.e., per domain), we measure the LCS between each pair of sequences and then compute the average and maximum of these pair-wise computations to get a sense of the amount of repetition in that domain. Intuitively, average pair-wise LCS conveys the overall amount of repetition observed in a domain, whereas maximum pair-wise LCS relates the magnitude of the most repetitive domain behavior.

We also estimated the number of actions that could be saved by automating detected repetition. For each participant we calculated the actions saved, t_p by

$$t_p = \sum_{i}^{domains} \left(\frac{s_i}{S} * lcs\_avg_i \right),$$

where lcs_avg_i is the average pair-wise LCS value computed for domain i, s_i represents the number of action sequences in domain i and S represents the total number of sequences computed for participant p. Intuitively, this means that the more we interact within a domain and the more repetition observed, the more actions we could save.

4.2. Results

Figure 1(a) illustrates the amount of repetition detected in our participant’s CRH logs as relative frequency histograms of LCS-based repetition (in percentages). For example, the second green bar in the histogram means that on average, 23% of domains visited by a person show between 10 and 20% overlap or repetition (according to our average pair-wise LCS method).

According to LCS, we see that people do indeed show evidence of repetition in their Web actions and the amount of repetition varies across different domains. In some domains, people show a very high amount of repetition. For example, in 4.9% of people’s domains, 70% or more of their behaviors within that domain are repetitious according to our average pair-wise LCS metric (20.8% of domains according to max pair-wise LCS). Examples of domains with highly repetitious behaviors include a financial services Web site that is only accessed to pay off the same monthly bill and the Web site for a favorite takeout restaurant where the same food is almost always ordered.

Our results also show that in 32.5% of domains, people show at least 20% and up to 50% repetition according to average LCS (19.3% of domains according to max LCS). An example domain with moderate repetition is a person’s main airline Web site that is sometimes accessed for booking regular flights home and sometimes accessed

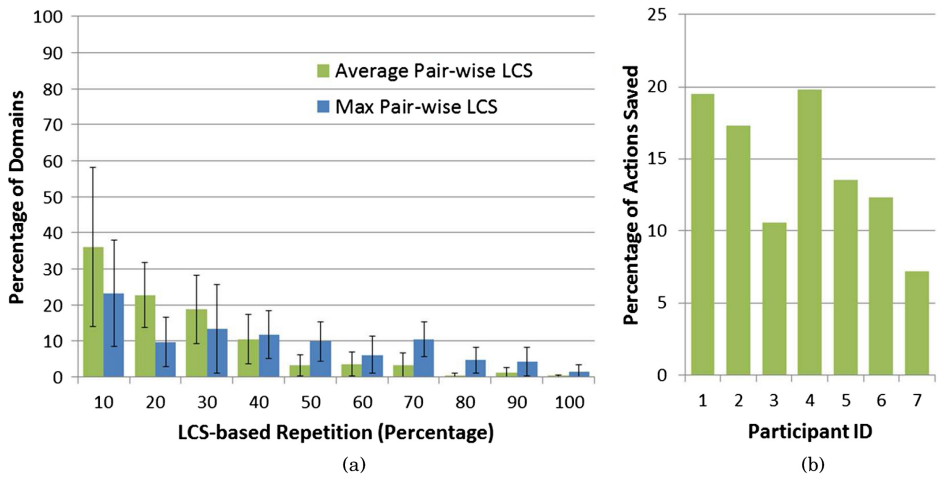

Fig. 1. (a) Relative frequency histograms of LCS-based repetition and (b) percentage of actions saved by automation per participant.

to check the status of trips. An example of a domain with low repetition (less than 10%) is the ACM Web site, which is accessed for a wide variety of reasons including searching for different papers, managing distribution lists and reading the news. However, even these Web sites show some repetition, such as periodically renewing an ACM membership.

Figure 1(b) shows the percentage of actions that our participants could have saved by fully automating their repetitive behaviors (i.e., automating behaviors in an idealistic scenario not requiring user interaction or guidance). This shows that all of our participants could have reduced their overall number of Web actions by at least 7.3% and up to 19.8%. According to our analysis, the average actions that could be saved per person by full automation is 14.3% (4.7% SD).

4.3. Discussion

To our knowledge, this is the first element-level analysis of Web behavior patterns. Our results indicate that people do repeat behaviors on the Web and the amount of repetition varies across domains, including some domains that consistently show only the same behaviors. Moreover, we show that fully automating these behaviors could save an average of 14.3% of people's actions and up to almost 20% for some. It should be noted however that full automation is difficult to achieve without some user involvement (e.g., via actions to initiate the automation or verification of the automated steps). Therefore, the number of actions saved in practice will be a function of the automation system used. However, it is fair to say that reducing the amount of work required to complete mundane tasks by automating those tasks could provide people time for more interesting endeavors.

While our findings suggest the benefit of automation, a detailed analysis of the detected patterns is necessary. For example, some of the repetition we detected included repeatedly clicking the "next" or "go back" links in a Web page (e.g., when viewing pictures in Facebook or Flickr or when wanting to return back to a search results page). This could suggest the need for automation (e.g., when successively viewing images) or for structural improvement of a site. Or it could simply be noise that should not be automated. One possible way of confirming our repetition findings is a supervised analysis, where a person inspects their own data to determine when automation is

necessary. However, as our interviews revealed, people find it difficult to identify behaviors to automate.

Our analysis also makes some simplifying assumptions (e.g., including that tasks do not span multiple days or domains), and uses heuristics that could be improved (e.g., segmenting sequences by the time between actions or mapping actions to action classes using string-based metrics). However, we believe a more interesting improvement to our analysis is in designing a metric better suited for measuring task repetition in sets of sequences (as opposed to measuring repetition between pairs of sequences).

5. AUTOMATIC TASK MODEL GENERATION

Our interviews found that users of the CoScripter Web task automation system had difficulty identifying tasks to automate and believed they did not perform many repetitive tasks. However, our analysis of Web repetition showed that people do in fact repeat their actions and automating these would reduce the overall number of actions required to complete a task. Motivated by these findings, we developed LiveAction, a fully-automated machine learning-based approach to task model generation.

5.1. LiveAction

Figure 2 illustrates the steps of our LiveAction approach to building task models. Given CRH logs from a domain, we first preprocess the logs by segmenting them into action sequences and then mapping actions to action classes as described in our Repetition Analysis section. We also store the mapping from actions to action classes along with frequency counts of individual actions for online mapping and prediction. Segmenting and mapping are independent of each other and can be done in parallel. Next, we cluster similar action class sequences together and then build a finite state automaton out of each cluster.

Clustering Sequences of Action Classes. To create models of repetitious behavior, we first have to identify behaviors that are similar. For this, we adapt our method of measuring repetition in sequences in order to actually group similar sequences together. Specifically, we employ unsupervised clustering to group similar action class sequences together using a similarity metric based on LCS.

Our clustering algorithm takes a set of sequences as input and constructs a separate cluster for each of them initially. Then it iteratively computes similarity between pairs of clusters, merges the most similar together, eliminates low quality clusters and then returns the set of clusters with the highest quality [Strehl 2002]. Since clusters may contain more than one sequence, cluster similarity is defined as the average similarity between the two corresponding sets of sequences, where sequence similarity is measured using normalized LCS as in our repetition analysis. After clustering, we eliminate noisy clusters which contain either a single sequence or sequences with low intra-cluster similarity [Strehl 2002] using a similarity threshold of 0.1, determined empirically.

Building Finite State Automata. After clustering similar action sequences together, we construct an automaton for each cluster using a state-of-the-art automata construction method [Hopcroft et al. 2006] and a number of generalization heuristics specific to our needs. Automaton construction begins with a cluster of action class sequences and initially builds an automaton containing a linear path for each input sequence (a path represents a sequence of states and each state corresponds to one action class from the sequence). To generalize this automaton, we merge states using three heuristics:

— Two states are merged if they are adjacent and contain the same action class. This is motivated by the assumption that if a user repeats an action once, they may repeat

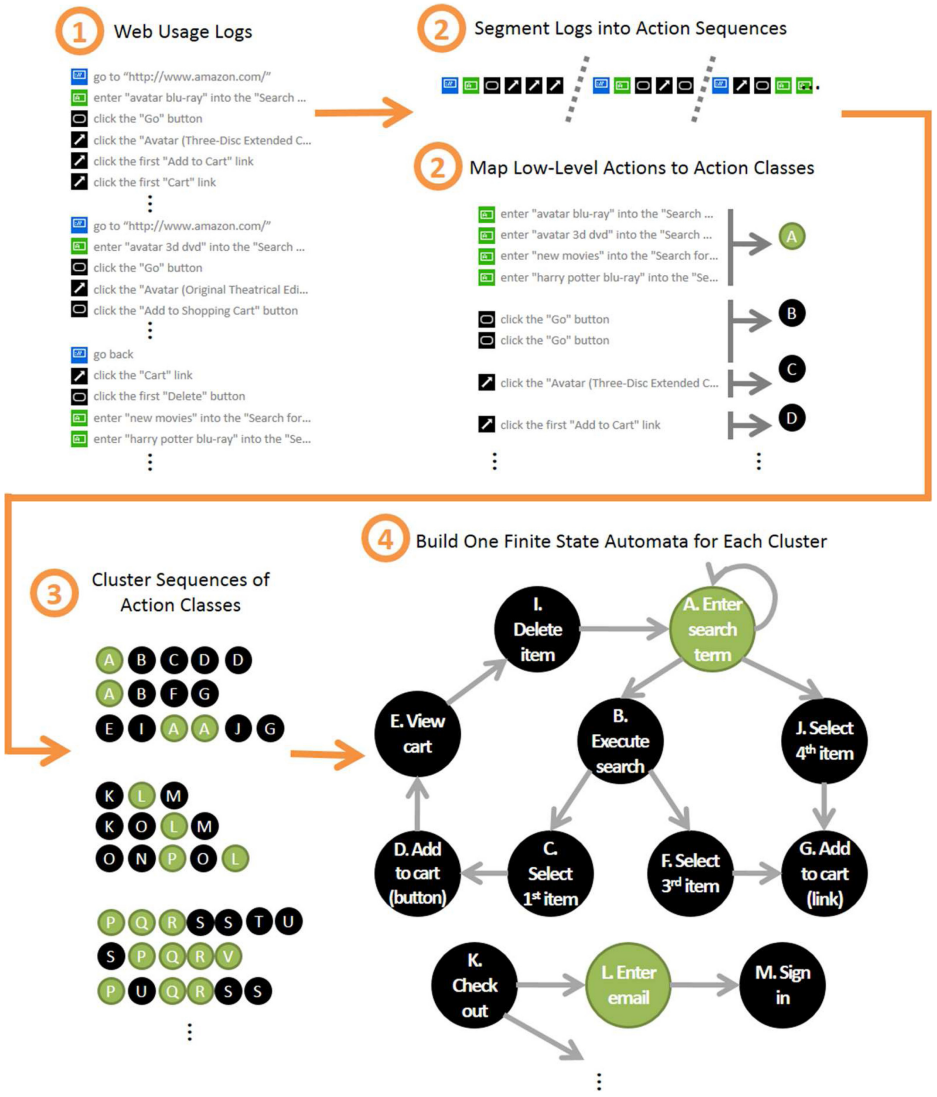

Fig. 2. LiveAction automatically generates task models given Web usage logs (1). First, we preprocess logs by segmenting them into action sequences over which we can detect repetition (2) and mapping low-level actions to conceptually equivalent action classes (2). Next we cluster action class sequences together based on their similarity (3) and then build a finite state automaton for each cluster using a state-of-the-art automata learning algorithm (4). LiveAction task models are intended for use by task automation systems that rely on task repositories to operate.

it again. After this merge, a self-loop is created on the merged state. For example, the self-loop on state A ("Enter search term") in Figure 2 was created as a result of state merging using this heuristic.

— Two states are merged if they contain the same action class and transitions from them result in the same set of states. For example, transitions from state F and J in Figure 2 result in the same state G (however, they are not merged in this case since their action classes are different). This heuristic captures the behavior of performing the same action after related actions. For example, in an e-commerce Web site, a

user may add an item to a shopping cart after either searching for an item and then selecting a result from the search results list or choosing an item category and then selecting an item from the item list. Hence, states in the automaton which represent selecting an item followed by adding that item to a shopping cart should be equivalent.

— Two states are merged if they have the same action class and transitions are made to them from same set of states. This heuristic merges similar branches in an automaton. For example, imagine a user is shopping for laptops on an electronics Web site. First, they select a link specifying a particular laptop brand (e.g., "dell") and then refine their search by selecting a link specifying a price range of "$400–$500." Later on in their search, they go back and again select the "dell" laptop brand link, but this time they specify a price range of "$500–600" by selecting the corresponding link. In this case, the states corresponding to 'selecting the $400–$500 link' and 'selecting the $500–$600 link' are merged because they have the same action class and transitions to them are made from the same state (e.g., 'selecting the dell link'). For another example, the only transitions to states B and J in Figure 2 are made from state A (however, B and J are not merged since their action classes are different).

6. EVALUATION

In this section we discuss our method and results of evaluating how well our automatically generated task models capture repetition in Web usage data.

6.1. Method

We conducted two experiments: a standard *leave-one-out-cross-validation (LOOCV) evaluation* measuring performance given a corpus of accumulated usage data and an *incremental evaluation* monitoring how our models improve with more data. To measure quality in capturing repetition, we tested the performance of our models in predicting the next action in a sequence given previous actions observed up to that point. Accuracy in predicting next steps impacts the capacity of our models to automate repetitious behavior.

Prediction. Next action prediction requires identifying the correct model out of possibly multiple models and then estimating the most likely next step given previous actions. First, each action in the sequence of actions seen up to and including the current action is mapped to an action class based on the previously computed mapping from action to action class for the given domain. Then the sequence of action classes is matched against the automata built for that domain out of previously observed sequences.

Matching a sequence, s, against a set of automata is carried out as follows. Given an automaton, A, with states $\{S_1, S_2,...S_n\}$, we compute a match score for each state, $m(s, A, S_i)$, by trying to *accept* all possible subsequences of the sequence starting from that state. Traditionally, automaton acceptance starts from a designated *start state* and checks for a path from that state that exactly matches a given sequence [Hopcroft et al. 2006]. We relax this requirement to accommodate our need to handle noisy sequences originating from actual Web usage logs. In our modified automaton acceptance test, we check all possible subsequences of the input sequence for a match from a state. For example, the sequence A→B→N→D will be accepted by the first automaton in Figure 2 since there is a path for the subsequence A→B→D from state A. Here N corresponds to selecting the 5th item from the search results, an action not used to build the automaton. The longest matching subsequence of s starting from S_i is considered the best match for S_i and $m(s, A, S_i)$ is then computed as the ratio of the length of this

subsequence to the length of the s. In our example, the subsequence A→B→D is the longest matching subsequence and therefore the match score of the input sequence is 3/4 or 0.75 starting from state A.

Using this method, we compute a match score for all states of an automaton. The state with the highest match score is identified as the *starting state* for the input sequence, and the match score of the automaton, $m(s, A)$, is the score computed for the starting state. Continuing with the same example, state A is deemed as the starting state for the input sequence A→B→N→D and the match score of the automaton for that sequence becomes 0.75. An automaton is deemed as a *matching automaton* for s if the score computed for the automaton is above some acceptance threshold, chosen empirically as 0.6. The matching automaton with the highest match score is then returned as the correct model. For the first automaton, the match score (0.75) is above the threshold and hence it is returned as the correct model for the input sequence.

After matching a sequence to an automaton, a next action prediction is made as follows. First, the *current state* of the automaton is determined as the state reached after accepting the best matched subsequence of a sequence from the starting state. Using the same example, the state reached after accepting the best matched subsequence is state D ("Add to cart (button)"). Then the state with the most frequent transitions from the current state is taken as the most likely next state. Finally, we map the action class of the predicted state back to an action by selecting the most frequently occurring action in that class. For our example, state E ("View Cart") is the most likely state and hence the action *click the "view cart" link* is predicted.

Performance. To measure performance, we performed the following tests given a stream of Web usage data. For each action in the stream, we attempt to make a prediction for that action based on previous actions seen up to that point. If we can make a prediction, we record whether or not the prediction was correct (*prediction accuracy*). We also record whether or not the actual next action existed within the identified model but perhaps was not the step with the highest prediction value (*existence accuracy*). This simulates the potential for asking a person to choose the next best step out of a set of possibilities.

Note that our models cannot always make a prediction. For example, if our models have never seen similar previous actions (i.e., previous actions cannot be mapped to existing action classes), then no prediction is possible as the previous actions are not contained in any model. Similarly, if no models exist for a domain or no model matches previous actions above the acceptance threshold, then no prediction can be made. For this reason, we compute performance of our models in terms of both precision and recall. Precision in this context measures how accurate our predictions are when we can make them, whereas recall measures the number of actions that our models are able to accurately predict. Recall therefore considers no prediction for an action as incorrect. Intuitively, high precision implies that when our models are used to automate behaviors, the behaviors will likely be correct, whereas high recall means that our models will be able to automate most of our behaviors. Therefore, for each of our evaluations, we report on four measures: *prediction recall, existence recall, prediction precision* and *existence precision*.

Leave-One-Out-Cross-Validation (LOOCV). LOOCV is a standard method of evaluating performance of statistical machine learning algorithms. Given a set of presegmented sequences of actions for a domain, LOOCV entails holding one sequence out at a time, building models out of the rest of the data and then testing the prediction performance of those models on the held out sequence. We measure prediction performance on the held out sequence by predicting each step of the sequence.

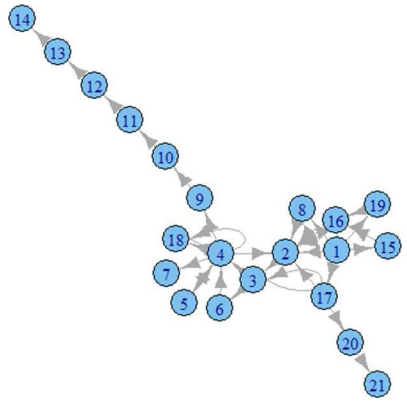

Fig. 3. A LiveAction model generated for checking application status in an immigration Web site. This model has 21 nodes and 32 edges.

Incremental Evaluation. To better examine how our automatically generated task models improve with increasingly more data, we also did the following experiment. Given a stream of usage data for a domain (i.e., a raw, unsegmented sequence of actions), we made a prediction for each incoming action and then rebuilt our models from scratch given previous actions plus the incoming action. Rebuilding involves updating the mapping from actions to action classes, recomputing sequence boundaries, reclustering of action class sequences and reconstructing automata. This scheme is inappropriate for online learning, but is sufficient for our purposes of evaluating performance and assumes offline updating (perhaps nightly or after acquiring some amount of new data). We leave defining an online updating scheme to reduce overhead as future work.

For both our LOOCV and incremental experiments we use the same CRH Web usage data collected for our repetition analysis as input and again restrict our evaluation to within a domain. We also restrict our experiments to the domains in which people exhibited the most repetition as measured by our repetition analysis. Specifically, for each person we ran our experiments on the top 10% of their most repetitive domains according to average pair-wise LCS. This amounts to 43 domains tested with an average estimated repetition (i.e., LCS value) of 63.1% (SD=16.1%). Our results therefore demonstrate the effectiveness of our LiveAction models in automating a person's most repetitive behaviors.

6.2. Results

Our automatically generated LiveAction models captured repetitive action sequences in our test data containing both cycles and branches. These models offer multiple execution paths ranging from short to long, with an average observed path length of 6.43 steps (SD = 6.45) according to the clustered action class sequences used to build models, after noisy clusters are removed. The maximum and minimum observed path lengths were 32 and 2 steps, respectively. The types of repetitive tasks detected included: checking train schedules, tracking UPS packages, recording daily exercises, checking application status (e.g., immigration), ordering drinks from a store, ordering food from a restaurant, checking mortgage balances, checking flight status, booking flights, checking traffic, buying a phone card and paying phone bills. Figures 3, 4, and 5 show examples of generated LiveAction model.

We present the performance results of our LOOCV and incremental evaluations cumulatively over actions. That is, for each prediction, we add up the number of

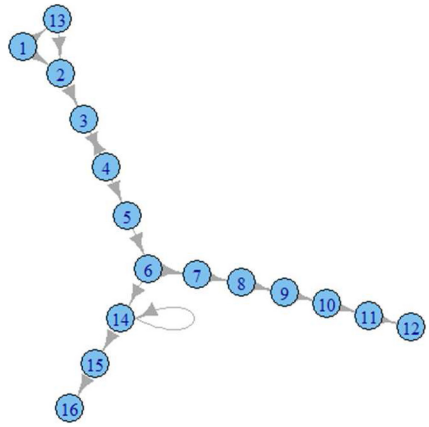

Fig. 4. A LiveAction model generated for buying a calling card from a Web site. This model has 16 nodes and 17 edges.

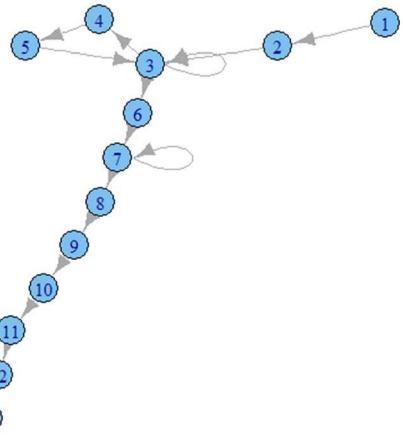

Fig. 5. A LiveAction model generated for ordering food from a restaurant Web site. This model has 13 nodes and 15 edges.

correct predictions up to and including the current prediction and divide this by the total number of predictions made so far. Figure 6 summarizes our results in terms of average and final cumulative accuracy.

Interestingly, LOOCV shows lower average and final precision compared to our incremental evaluation results. Furthermore, the discrepancy between average and final cumulative precision in LOOCV is larger compared to the incremental evaluation. To understand this difference, we inspected the prediction precision trends of individual domains. Figure 7 details the prediction precision performance on a select few domains according to our incremental evaluation (selected to demonstrate the range of observed results). Figure 8 then details the performance on each sequence tested during our LOOCV evaluation in one of the domains shown in Figure 7 (the *letsget* domain, a Web site for ordering restaurant food online). Notice that according to our incremental evaluation, our models are able to predict much of the repetition in the *letsget* domain (showing a final accuracy of 90.6%). However, performance on the 23 individual sequences in that domain varies. Examining the sequences, we see that our

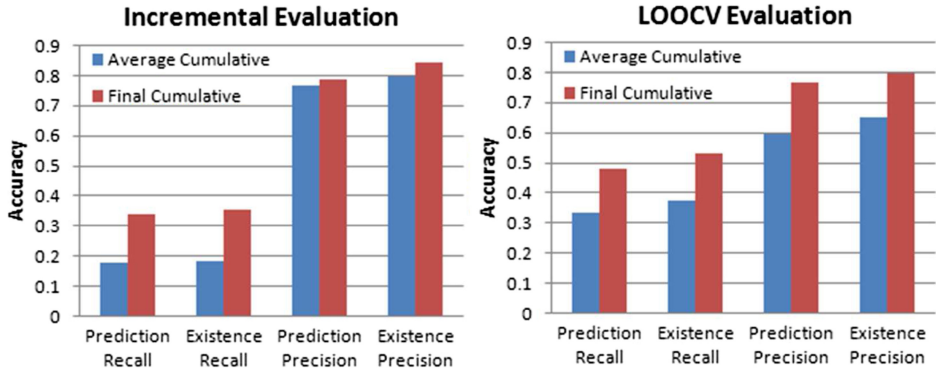

Fig. 6. Summary of incremental and LOOCV evaluation results, showing the mean and final accuracies averaged over all domains tested.

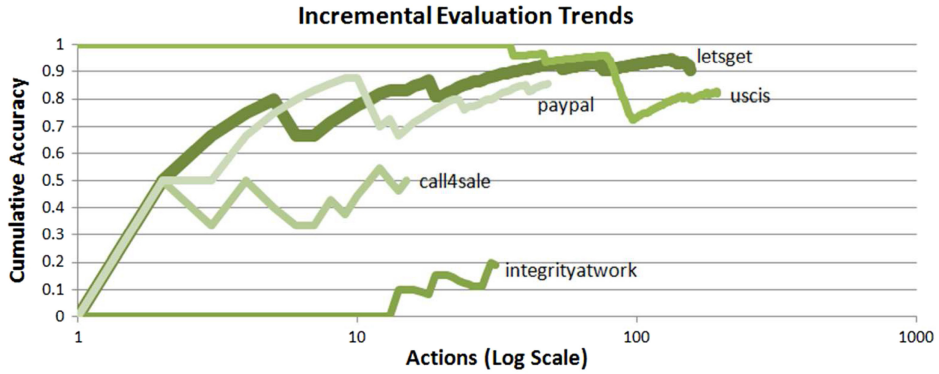

Fig. 7. Prediction precision performance trends for select domains tested during our incremental evaluation (selected to demonstrate the range of observed trends).

LiveAction models are able to predict 17 sequences with near perfect precision (see the multiple trend lines along the top of Figure 8). These sequences correspond to a person ordering the exact same meal online 17 times. The two trend lines with zero accuracy correspond to two behaviors not seen in any of the sequences used to build the models during our LOOCV evaluation: navigating the online menu and logging in (presumably the participant was logged in by default during their previous orders). The remaining four trend lines were predicted partially correctly and correspond to behaviors similar to the 17 behaviors that were exactly the same, but had a few deviations (e.g., adding one element to their normal order). This variation in performance on individual sequences accounts for the lower overall precision shown by LOOCV, even though our models are able to accurately predict most behaviors in the *letsget* domain. This justifies our incremental evaluation method for demonstrating the performance of our LiveAction models.

Focusing on our incremental evaluation, we see that our LiveAction models can predict actions with an average and final precision of 76.6% and 78.5%, respectively. If our models are used to present a person with a choice of possible next actions, our average and final (existence) precision improves to 79.6% and 84.2%, respectively.

Not surprisingly, recall performance of our models is relatively poor compared to precision. This is expected because we consider null predictions as incorrect according to recall. However, this penalizes our models even when no prediction is possible (e.g.,

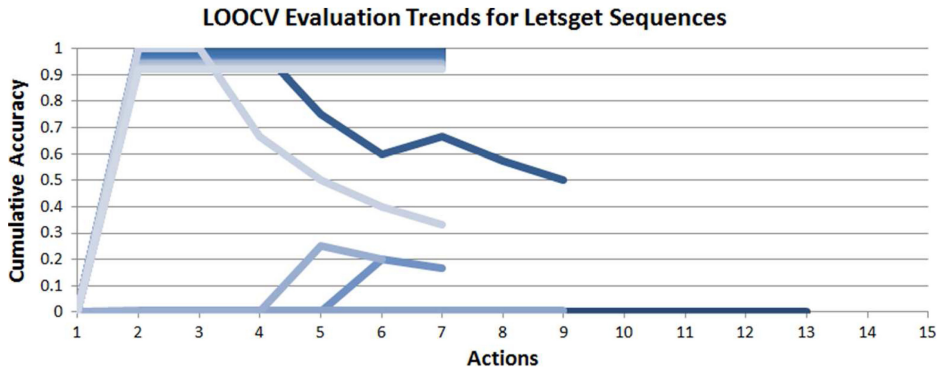

Fig. 8. Prediction precision performance according to LOOCV on individual sequences from the *letsget* domain tested.

no information exists or the current sequence of actions has never been seen before). Therefore, we recomputed accuracy after adjusting for null predictions and found that our average and final prediction recall improved to 62.4% and 63.4%, respectively.

6.3. Discussion and Future Work

There are many ways of accomplishing each step in our LiveAction approach to automatically generating task models (e.g., there are alternative heuristics for segmenting action sequences and mapping actions to action classes). While we empirically evaluated several of these alternatives and selected those that performed best, our choices, and corresponding evaluation results, demonstrate a proof-of-concept that could be improved upon. For example, our method of mapping actions to action classes via edit distance comparisons (less than three edits) will incorrectly map actions such as "Click the 'Inbox (26)' link" and "Click the 'Inbox (354)' link" to separate action classes although they should be equivalent. An improved mapping scheme could further increase performance of our LiveAction models. In this work we have used time-based heuristics for segmenting CRH actions into sequences, mapping actions to action-classes and then clustering sequences of action-classes. In the future, we would like to explore whether we can use other log analysis methods as described in [Hilbert and Redmiles 2000; Ivory and Hearst 2001].

Our evaluation results show that our current method of building LiveAction models is able to capture a variety of repetitive Web behaviors, varying in terms of the nature and complexity of the tasks. Note that understanding the types of tasks that our models were unable to capture is more difficult without labeled data about all of the tasks being performed (recall that labeled data is difficult to obtain because people have trouble identifying their repetitive tasks). Therefore, an analysis of failures is best answered in a deployed system and is left for future work.

In this research, we limit our task model representation to finite state automaton. In a dynamic Web application, where elements may be dynamically created and destroyed within a single page, an automaton representation may face further challenges. This is because dynamic changes to Web page elements result in changes to the action-class representation, which can make some old states (based on action-classes) of the automaton inaccessible. We have not investigated updating a learned finite state automaton in such cases, and leave this as a future work. However, we believe that such state changes would be minimal since most Web page elements are quite static (e.g., form elements such as labels of a button or a textbox).

In this article, we have not investigated scenarios when a sequence of actions can be matched with multiple learned automata. Our algorithm for matching a sequence to a set of automata does not capture the situation when the best matching may be obtained for multiple automata. In the simplest case, a random selection of such best matched multiple automata can be done. However, we leave this as a future work to deeply investigate such scenarios and design solutions to handle them.

Our LiveAction models are only able to make predictions if they have observed similar behaviors in the past. When predictions are possible, our models can recall 63.4% of next steps with 78.5% precision. Precision performance demonstrates the capacity of our models to automate repetitious behaviors on the Web (as only predicted actions can be automated). One conceivable application that could benefit from the ability of our LiveAction models to make one-step-ahead predictions is a Web task autocompletion system. For example, when a person navigates to a new domain, previously generated LiveAction models for that domain could be loaded for action prediction. Then, as a person interacts with a Web page, their sequences of actions could be passed to the autocompletion system for predicting next actions. If an action is predicted (i.e., the previous sequence of actions matches an existing model and next state predictions are computed), it could be presented to the person (e.g., via subtle interface highlighting to minimize distraction [Cypher 1991] or in pseudo-natural language in a browser sidebar as in [Leshed et al. 2008]). Note that, our automaton based models are nondeterministic, hence it is possible to reach multiple next states from a specific state. In such cases, the autocompletion system can highlight multiple next actions corresponding to multiple states.

When next actions are predicted and presented, the person could choose to automatically execute a predicted action (e.g., via a keyboard shortcut as with text field autocompletion). As a person continues to interact with pages in the domain (either on their own or by executing actions suggested by the system), their observed action sequence could be updated and new predictions could be computed and displayed. A future Web task autocompletion system such as this could also permit evaluation of our models with real users.

When considering existence of intended actions within our models, precision improves to 84.2%. This suggests that human guidance (via selection from multiple predictions in this case) could complement model predictions and improve automation accuracy. Alternatively, interactive human feedback could be used for manual model refinement and full task automation. For example, an intelligent task assistant could present a person with a sequence of LiveAction predicted steps necessary to carry out a task on their behalf. Those actions could then be manually edited by deleting irrelevant actions, typing or demonstrating necessary actions as in Li et al. [2010], or reordering the sequence. Manually editing and correcting a model could enable the assistant to fully automate the task in the future. A full exploration of LiveAction model refinement via interactive human feedback warrants further investigation and is left for future work.

7. CONCLUSION

In this research, we analyze element-level Web usage data, show that people do repeat behaviors on the Web, and show that automating these behaviors could reduce the amount of work needed to complete Web-based tasks. We then present LiveAction, a fully-automated machine learning-based approach to modeling repetitive behaviors on the Web. Our evaluations show that our LiveAction models can be used to automate repetitious tasks on the Web and have the potential to automatically populate the task model repositories required by many task automation systems existing today.

ACKNOWLEDGMENTS

We thank Julian Cerruti and our study participants for their help with this work.

REFERENCES

Eytan Adar, Jaime Teevan, and Susan T. Dumais. 2009. Resonance on the Web: Web dynamics and revisitation patterns. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'09)*. ACM Press, New York, 1381–1390.

James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. 2007. PLOW: A collaborative task learning agent. In *Proceedings of the 22th National Conference on Artificial Intelligence (AAAI'07)*. AAAI Press, 1514–1519.

John Annett and Keith D. Duncan. 1967. Task analysis and training design. *Occup. Psych. 41*, 211–221.

Vinod Anupam, Juliana Freire, Bharat Kumar, and Daniel Liuwen. 2000. Automating Web navigation with WebVCR. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks*, North-Holland Publishing Co., Amsterdam, 503–517.

Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. 2005. DocWizards: A system for authoring follow-me documentation wizards. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology (UIST'05)*. ACM Press, New York, 191–200.

Lasse Bergroth, Harri Hakonen, and Timo Raita. 2000. A survey of longest common subsequence algorithms. In *Proceedings of the 7th International Symposium on String Processing Information Retrieval (Spire'00)*. IEEE, 39–48.

Stuart K. Card, Allen Newell, and Thomas P. Morgan. 1983. *The Psychology of Human Computer Interaction*. Lawrence Erlbaum Associates Inc., Hillsdale, NJ.

Vinay K. Chaudri, Adam Cheyer, Richard Guili, Bill Jarrold, Karen L. Myers, and John Niekrasz. 2006. A case study in engineering a knowledge base for an intelligent personal assistant. In *Proceedings of the Semantic Desktop Workshop*.

Andy Cockburn and Bruce Mckenzie. 2000. What do web users do? An empirical analysis of web use. *Int. J. Hum. Comput. Stud. 54*, 903–922.

Allen Cypher. 1991. Eager: Programming repetitive tasks by example. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'09)*. ACM Press, New York, NY, 33.

Anton N. Dragunov, Thomas G. Diettrich, Kevin Johnsrude, Matthew Mclaughlin, Lida Li, and Jonathan L. Herlocker. 2005. TaskTracer: A desktop environment to support multi-tasking knowledge workers. In *Proceedings of the 10th International Conference on Intelligent User Interfaces (IUI'05)*. ACM Press, New York, 75–82.

Patrick Dubroy and Ravin Balakrishnan. 2010. A study of tabbed browsing among Mozilla Firefox users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'10)*. ACM Press, New York, 673–682.

James Foley, Won Chul Kim, Srdjan Kovacevic, and Kevin Murray. 1991. UIDE: An intelligent user interface design environment. In *Proceedings of the Conference on Intelligent User Interfaces*. ACM Press, New York, 339–384.

Andrew Garland, Kathy Ryall, and Charles Rich. 2001. Learning hierarchical task models by defining and refining examples. In *Proceedings of the 1st International Conference on Knowledge Capture (K-Cap'01)*. ACM Press, New York, NY, 44–51.

Yolanda Gil, Varun Ratnakar, and Christian Fritz. 2011. TellMe: Learning procedures from tutorial instruction. In *Proceedings of the 16th International Conference on Intelligent User Interfaces (IUI'11)*. ACM Press, New York, 227–236.

Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK.

Jeffrey Heer and Ed H. Chi. 2002. Separating the swarm: Categorization methods for user access sessions on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'02)*. ACM Press, New York, NY, 243–250.

David M. Hilbert and David F. Redmiles. 2000. Extracting usability information from user interface events. *ACM Comput. Surv. 32*, 4, 384–421.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation* 3rd Ed. Addison-Wesley.

Scott E. Hudson, Bonnie E. John, Keith Knudsen, and Michael D. Byrne. 1999. A tool for creating predictive performance models from user interface demonstrations. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'99)*. ACM Press, New York, 93–102.

Darris Hupp and Robert C. Miller. 2007. Smart bookmarks: Automatic retroactive macro recording on the web. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'07)*. ACM Press, New York, NY, 81–90.

Melody Y. Ivory and Marti A. Hearst. 2001. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv. 33*, 470–516.

Xin Jin, Yanzan Zhou, and Bamshad Mobasher. 2005. Task-oriented Web user modeling for recommendation. In *Proceedings of the 10th International Conference on User Modeling (UM'05)*. Springer, 109–118.

Ron Kohavi, Llew Mason, Rajesh Parekh, and Zijian Zheng. 2004. Lessons and challenges from mining retail e-commerce data. *Mach. Learn. 57*, 83–113.

Raymond Kosala and Hendrik Blockeel. 2000. Web mining research: A survey. *ACM SIGKDD Explor. Newslett. 2*, 1, 1–15.

Tessa Lau, Lawrence Bergman, Vittorio Castelli, and Daniel Oblinger. 2004. Sheepdog: Learning procedures for technical support. In *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI'04)*. ACM Press, New York, 109–116.

Tessa Lau, Clemens Drews, and Jeffrey Nichols. 2009. Interpreting written how-to instructions. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1433–1438.

Tessa Lau, Julian Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey P. Bigham, and Jeffrey Nichols. 2010. A conversational interface to web automation. In *Proceedings of the 23rd Annual ACM Symposium on User Interface Software Technology (UIST'10)*. ACM Press, New York, 229–238.

Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's what i did: Sharing and reusing web activity with ActionShot. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'10)*. ACM Press, New York, 723–732.

Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'08)*. ACM Press, New York, 1719–1728.

Nizar R. Mabroukeh and Christie I. Ezeife. 2009. Semantic-Rich Markov models for web prefetching. In *Proceedings of the IEEE International Conference on Data Mining Workshops (ICDMW'09)*. IEEE, 465–470.

Jalal Mahmud and Tessa Lau. 2010. Lowering the barrier to website testing with CoTester. In *Proceedings of the 15th International Conference on Intelligent User Interfaces (IUI'10)*. ACM Press, New York, 169–178.

Jalal Mahmud, Yevgen Borodin, I. V. Ramakrishnan, and C. R. Ramakrishnan. 2009. Automated construction of web accessibility models from transaction click-streams In *Proceedings of the 18th International Conference on the World Wide Web*. ACM Press, New York, 871–880.

Nuria Oliver, Greg Smith, Chintan Thakkar, and Arun C. Surendran. 2006. SWISH: Semantic analysis of window titles and switching history. In *Proceedings of the 11th International Conference on Intelligent User Interfaces (IUI'06)*. ACM Press, New York, 194–201.

Laila Paganelli and Fabio Paternò. 2003. A tool for creating design models from web site code. *Int. J. Software Eng. Knowledge Eng. 13*, 2, 169–189.

Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. 1997. ConcurTaskTree: A diagrammatic notation for specifying task models. In *Proceedings of the International Conference on Human-Computer Interaction (INTERACT'97)*. Chapman & Hall, Ltd., London, UK, 362–369.

Dimitrios Pierrakos, Georgios Paliouras, Christos Papatheodorou, and Constantine D. Spyropoulos. 2003. Web usage mining as a tool for personalization: A survey. *User Model. User-Adapted Interact. 13*, 4, 311–372.

James Pitkow and Peter Pirolli. 1999. Mining longest repeated subsequences to Predict World Wide Web surfing. In *Proceedings of the 2nd Conference on USENIX Symposium on Internet Technologies and Systems (USITS'99)*. USENIX Association, Berkeley, CA, 13–13.

Jianqiang Shen, Erin Fitzhenry, and Thomas G. Dietterich. Discovering frequent work procedures from resource connections. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI'09)*. ACM Press, New York, 277–285.

Jianqiang Shen, Lida Li, Thomas G. Dietterich, and Jonathan L. Herlocker. 2006. A hybrid learning system for recognizing user tasks from desktop activities and email messages. In *Proceedings of the 11th International Conference on Intelligent User Interfaces (IUI'06)*. ACM Press, New York, 86–92.

Aaron Spaulding, Jim Blythe, Will Haines, and Melinda Gervasio. 2009. From geek to sleek: Integrating task learning tools to support end users in real-world applications. In *Proceedings of*

the 14th International Conference on Intelligent User Interfaces (IUI'09). ACM Press, New York, 389–394.

Alexander Strehl. 2002. Relationship-based clustering and cluster ensembles for high-dimensional data mining. Ph.D. dissertation, The University of Texas at Austin.

Zan Sun, Jalal Mahmud, I. V. Ramakrishnan, and Saikat Mukherjee. 2007. Model-directed web transactions under constrained modalities. *ACM Trans. Web 1*, 3, Article 12.

Received January 2012; revised December 2012; accepted March 2013