HILC: Domain Independent PbD system via Computer Vision and Follow-up Questions

THANAPONG INTHARAH, University College London, United Kingdom and Khon Kaen University, Thailand
DANIYAR TURMUKHAMBETOV, University College London, United Kingdom
GABRIEL J. BROSTOW, University College London, United Kingdom

Creating automation scripts for tasks involving GUI interactions is hard. It is challenging because not all software applications allow access to a program's internal state, nor do they all have accessibility APIs. Although much of the internal state is exposed to the user through the GUI, it is hard to programmatically operate GUI's widgets.

To that end, we developed a system prototype which learns-by-demonstration, called HILC (Help, It Looks Confusing). Users, both programmers and non-programmers, train HILC to synthesize a task script by demonstrating the task. A demonstration produces the needed screenshots and their corresponding mouse-keyboard signals. After the demonstration, the user answers follow-up questions.

We propose a user-in-the-loop framework that learns to generate scripts of actions performed on visible elements of graphical applications. While pure programming-by-demonstration is still unrealistic due to a computer's limited understanding of user-intentions, we use quantitative and qualitative experiments to show that non-programming users are willing and effective at answering follow-up queries posed by our system, to help with confusing parts of the demonstrations. Our models of events and appearances are surprisingly simple, but are combined effectively to cope with varying amounts of supervision.

The best available baseline, Sikuli Slides, struggled to assist users in the majority of the tests in our user study experiments. The prototype with our proposed approach successfully helped users accomplish simple linear tasks, complicated tasks (monitoring, looping, and mixed), and tasks that span across multiple applications. Even when both systems could ultimately perform a task, ours was trained and refined by the user in less time. [1]

CCS Concepts: • **Human-centered computing → Graphical user interfaces**; **User interface programming**; • **Computing methodologies → Activity recognition and understanding**; • **Software and its engineering → Programming by example**;

Additional Key Words and Phrases: Programming by Demonstration; GUI Automation; Action Segmentation and Recognition; Visual-based Programming by Demonstration

[1]This is an extended version of [19]

Authors' addresses: Thanapong Intharah, University College London, Gower Street, London, WC1E 6BT, United Kingdom, Khon Kaen University, 123 Mittraphap Road, Khon Kaen, 40002, Thailand, t.intharah@ucl.ac.uk; Daniyar Turmukhambetov, University College London, Gower Street, London, WC1E 6BT, United Kingdom, daniyar.turmukhambetov.10@ucl.ac.uk; Gabriel J. Brostow, University College London, Gower Street, London, WC1E 6BT, United Kingdom, g.brostow@cs.ucl.ac.uk.

2160-6455/2018/8-ART0 $15.00
https://doi.org/0000001.0000001

ACM Reference Format:
Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J. Brostow. 2018. HILC: Domain Independent PbD system via Computer Vision and Follow-up Questions. *ACM Trans. Interact. Intell. Syst.* 0, 0, Article 0 (August 2018), 26 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Millions of person-years are spent laboring in front of computers. Although some of that work is creative, many tasks are repetitive. Able-bodied users find it increasingly tedious to repeat a task tens or hundreds of times. Even more severely, users with special needs, such as visual impairment, hand or finger motion problems, *etc.* , can find even a second or third repetition of a task to be prohibitively hard. We see computer vision in the domain of desktop and mobile Graphical User Interfaces (GUIs) as a prerequisite, needed to make effective virtual personal assistants.

Usually, for programmers to create a script which operates any GUI application, they need the application to grant them access, via custom accessibility APIs. Such APIs are available rarely, due to time and budget constraints. When an API is available, each new programmer needs a significant ramp-up time. As a workaround, programmers can use a GUI scripting framework such as Sikuli [34] to create scripts which operate GUI applications without the need to access a GUI's internal libraries. We go beyond that point by allowing *non-programming users* to create scripts, by demonstrating the intended task to our proposed system.

The proposed approach examines the many small and non-obvious challenges to learning-by-demonstration in a GUI world. While template-matching of icons and scripting of macros and bots are low-tech by modern vision standards, we start with these technologies because they are effective. The overall contributions of our approach are that:

- Non-programming users can teach a task to HILC, simply by demonstrating it, either on a computer with a sniffer program or through screencast-video.
- The system bypasses the need for accessibility APIs by using computer vision and user interaction techniques. This makes the system natively run across application boundaries, as shown in Figure 1.
- The system asks the human for help if parts of the demonstrated task were ambiguous (there are more than one similar looking targets).
- The same or other users can run the task repeatedly, giving them functionality that was previously hard to achieve (require programming knowledge) or was missing entirely, *e.g.,* looping.

Tasks range from short single-click operations to inter-application chains-of-events. Our informal surveys, interviewing participants and colleagues, revealed that each user had different tasks they wished to automate, but they agreed universally that the teaching of a task should not take much longer than performing the task itself. Most everyday tasks can be categorized into linear tasks where users just need to complete a sequence of actions, looping tasks where users need to apply a linear sequence multiple times on different items, and monitoring tasks where users need a program to wait for some signal to then execute a script. Existing PbD systems were designed to deal with a specific kind of task. However, in this paper, we show that HILC is flexible enough to create scripts for linear, looping, and monitoring tasks.

With the long-term aim of improving assistive technology, we separate the role played by the *instructor* from that of the *end-user*. For example, one person could use a mouse and keyboard (or an eye-tracker) to demonstrate a task and answer follow-on questions. Then the same or another person, end-users, could run that learned task with any convenient method, *e.g.,* using voice command or gesture. In this paper, we focus on algorithms to make the instructor effective.

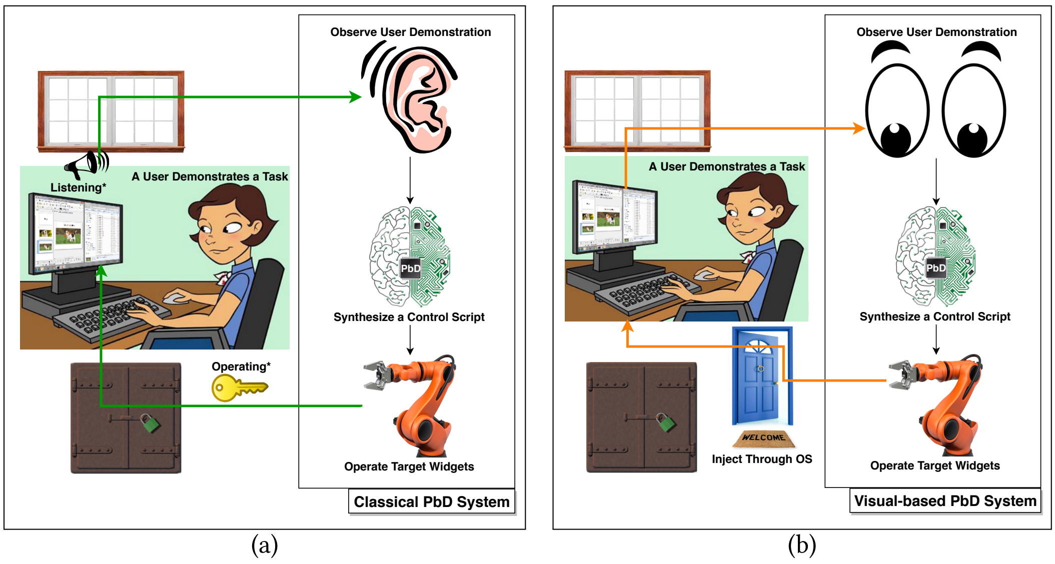

Observe User Demonstration
Listening*
A User Demonstrates a Task
Synthesize a Control Script
Operating*
Operate Target Widgets
Classical PbD System
(a)
Observe User Demonstration
A User Demonstrates a Task
Synthesize a Control Script
Inject Through OS
Operate Target Widgets
Visual-based PbD System
(b)

Fig. 1. Diagrams compare the concept of conventional PbD systems (a) against our proposed Visual-based PbD system (b). (a) A conventional PbD system listens to the demonstration via an Accessibility API (**listening***) and then the systems needs another API to operate application widgets (**operating***). (b) our proposed sniffer and visual-based system, on the other hand, watches the demonstration from the same screen as the user, via screenshot images (Section 4.2), and operates the GUI directly as instructed by HILC, which is looking for visual patterns (Section 5.2).

Moreover, we also split off the teacher role from the instructor role to support diverse demonstration inputs, *e.g.*, video tutorials *vs.* sniffer programs. Ultimately, using video tutorials as inputs will benefit end-users, especially with special needs, because the scripts can be generated from existing tutorials on the Internet.

Through our designated scenarios, as described in Table 1, we explore three issues of general Programming by Demonstration systems. For the linear tasks, we explore script generation; for the looping tasks, we explore generalization of the generated scripts; and for the monitoring tasks, we explore invocation of scripts.

1.1 Overview of Challenges

At a high level, our system, HILC, collects observations while the instructor performs a task, then it finds confusing looking patterns that call for the instructor to give more input. Once the task is learned and saved as a script, it can be called up by the end-user to run one or more times, or to monitor for some visual trigger before running.

A number of challenges make this a technical problem that relates to object recognition, action recognition, and one-shot learning. The instructor's computer can be instrumented with a sniffer program, that records mouse/keyboard events, and screen-appearance. But, for example, a click-drag and a slow click are still hard to distinguish, and hugely varying time-scales make sniffed observations surprisingly hard to classify. We also explore learning of tasks from pre-recorded screencast videos, which display noisy details of key/mouse events. We classify demonstrated actions using dynamic programming, the detail is in section 5.1. Further, clicking someplace like a File-menu usually means the task calls for that *relative* location to be clicked, but what if other

Scenario	Basic Actions + Typing: Click	Click Drag	Double Click	Right Click	Typing	Description
7.1.1 Mute audio playback	2	0	0	0	0	(linear) the task composes of 2 clicks: click the speaker icon, click the correct device to mute.
7.1.2 High-Contrast-mode	6	1	0	0	0	(linear) the task involves looking for Display setting icon in Control Panel, and then select the high contrast mode which helps visual impaired person.
7.1.3 Remote access (1)	11	0	0	0	4	(linear) the task is to access to remote computer via TeamViewer software by selecting log-in information from a spreadsheet program. (selecting information with keyboard shortcuts)
7.1.3 Remote access (2)	13	0	0	4	0	(linear) the task is to access to remote computer via TeamViewer software by selecting log-in information from a spreadsheet program. (selecting information with right click menu)
7.1.4 Skip YouTube ads.	1	0	0	0	0	(monitoring) the task is to click the skip add pop-up menu when it appears.
7.1.5 Close YouTube ads.	1	0	0	0	0	(monitoring) the task is to click the close icon of the ads. when YouTube displays the ads.
7.1.6 Create slides of Jpegs	2x	1x	0	0	0	(looping) the task is to create a presentation where each slide features one image from a given folder.
7.1.7 Create spreadsheet	4x	2x	0	0	4x	(looping) the task is to create a list of filenames in a spreadsheet program from a given folder.
7.1.8 Create Bibtex	9x	0	0	0	8x	(looping) the task is to copy each paper's title from a spreadsheet and then switch to Browser. In the Browser, the user uses Google Scholar to search for the paper BibTex. Finally, the user adds the BibTex code to the designated BibTex file on NotePad
7.1.9 Move files	4	1x	0	2	0	(looping-video) The scenario is using video as input. The task starts by creating a folder. After that, the user iteratively moves each Microsoft Word file to the created folder.

Table 1. The table describes in detail our designated scenarios which are used to qualitatively evaluate our system, HILC, against the closest available baseline system, Sikuli Slides [2]. The evaluation result can be found in Table 3.

parts of the screen have similar looking menus? The teacher can help find visual cues that serve as supporters. We address the challenge with Computer Vision by training pattern detectors and supporters which is described in section 5.2

Also, while demonstrating, the instructor runs through a linear version of the task, and can indicate if some part of the chain of actions should actually become a loop. Template matching may reveal that the to-be-looped segment was applied to one unique GUI element (an icon, a line of text, *etc.*), but the system can request further input to correctly detect the other valid GUI elements on the screen. This functionality is especially useful when a looping task must "step" each time, operating on subsequent rows or columns, instead of repeatedly visiting the same part of the screen.

2 RELATED WORK

The problem of learning-by-demonstration in a GUI world has challenges related broadly to four areas: action recognition from video, semantic understanding of GUI environments, analysis of video tutorials, and program synthesis.

2.1 Joint Segmentation and Classification of Action:

While most other action-recognition works just classified pre-segmented clips, Hoai et al. [17] proposed an algorithm to jointly segment and classify human actions in longer unsegmented videos. Like Hoai et al., our system takes long and unsegmented videos as the inputs.

Shi et al. [30, 31] also segment and classify human actions. They use a Viterbi-like algorithm for inference, similar to ours, but they used Hamming distance [16] to measure the loss between labels of two consecutive frames. Other action-detection methods tend to be slow and ill-suited for GUI problems.

2.2 Semantic Understanding of GUI Environments:

Dixon et al. [10] study models of GUI widgets, *e.g.*, buttons, tick boxes, text fields, *etc.* The work uses just visual information to derive hierarchical models of the widgets. This work and its extension [11] aim to reverse-engineer GUI widgets to augment and enhance them. Hurst et al. [18] studied GUI elements that the user can manipulate, for human performance assessment and software usability analysis. While having an accessibility API simplifies task-interpretation, it assumes the softwares' developers want to invest extra effort.

Similarly, Chang et al. [7] developed an accessibility API based system which also uses visual analysis to establish the hierarchical structure of the GUI. This also improved text analysis and processing. In contrast, we aim to recognize cross-application actions without access to software internals, mimicking what a human personal assistant would learn only from observation.

2.3 Analysis of Video Tutorials:

Grabler et al. [13] presented a system that generates a photo manipulation tutorial from an instructor's demonstration in the GIMP program. The system generates each step of the tutorial by accessing changes in the interface and the internal application state. The source code of GIMP was modified to allow such access. The authors also proposed preliminary work on transferring operations from the user demonstration to a new image. Again, this approach relies heavily on privileged access to one application. *Chronicle* [14] is a system that allows users to explore the history of a graphical document, as it was created through multiple interactions. The system captures the relation between time, regions of interest in the document, tools, and actions. This rich information allows users to play back the video for a specific time or region of interest, to replicate the result or to understand the workflow.

The *Pause-and-Play* system [28] helps users better learn to use an application from a video tutorial. The system finds important events in the video tutorial, and links them with the target application. This allows the system to automatically pause the video by detecting events in the application, while the user is following the steps of the tutorial. However, the detection of events in an application is implemented again through the API of that application. The *Waken* system [3] processes video tutorials using consecutive frame differences to locate the mouse cursor and the application widgets on which the video tutorial is focusing. Moreover, the system also infers basic widget actions executed by the user, and characterizes widget behavior. They applied the system to a tutorial of a video player, allowing users to directly interact with the widgets in the application. However, this system mainly relies on pre-designed heuristic rules to detect cursors and widgets.

EverTutor [33] is a system that processes low-level touch events and screen-captured images on a smartphone to automatically generate a tutorial from a user demonstration, without binding itself to one application. A tutorial generated by that system can interactively guide the user through a process. Although this system approaches their problem in a similar way to HILC for processing low-level mouse and keyboard input events and screen-captured images, their algorithmic details are omitted.

2.4 Program Synthesis

Our work is related to two sub-areas of program synthesis from user inputs: Programming by Demonstration and Programming by Example. HILC fits best in the Programming by Demonstration category. Programming by Demonstration systems (PbD) have the user perform a task as an input, and output a computer program that performs that task. In contrast, Programming by Example systems (PbE) learn the model from input-output pairs provided by a user, and generate a computer program that does the task. Although research on Programming by Demonstration [9, 20] and Programming by Examples [15] shows promising results, the systems usually work within closed ecosystems where PbD can directly access software states and can manipulate the state. Automating user interaction with GUI objects, where working across different ecosystems is one of the most common scenarios, is still under-developed. The main contribution of our work is to make a PbD system be domain independent by relying on Computer Vision techniques rather than an Accessibility API. Table 2 compares existing PbD systems against our proposed system.

Feature Comparison	Sikuli [34]	Sikuli Slides [2]	Koala [25], CoScriptor [22]	Sheepdog [21], Familiar [27]	IBOTS [36]	HILC
Domain independent	✓	✓	✗	✗	✓*	✓
Non-programmer friendly	✗	✓	✓**	✓	✓	✓
Task variation	any	linear	linear	looping	linear	linear, looping, standby

Table 2. Feature comparison of existing PbD systems. ✓* indicates that the system is partially Domain independent. IBOTS can only work with pre-defined widgets. ✓** indicates that the systems partially require users to remember the simplified syntax, *Koala* and *CoScriptor*.

Trigger from Potter [9] is the first attempt which shows the potential of using pixels information to get access to on-screen objects' data for a GUI-based program synthesis. The first visual-based PbD system is *IBOTS* [36]. However, applications of the system are still limited to set of pre-defined widgets due to the system needs pre-defined heuristic rules to recognize the widgets.

Koala [25] and *CoScriptor* [22] built platforms to generate automation scripts and business processes with a loose programming approach, and to share the generated process across an organization. The aim was to personalize a process to a particular end-user's needs. Those systems focused only on web-based processes, since web-page source code is readily accessible and machine-readable.

Sikuli [34] is the first to provide programmers with its own API to process and interact with GUI applications using computer vision techniques. This easy-to-use system can interact with multiple applications and can be run on multiple platforms. It was extended to many applications [8, 35]. However, the automation script has to be programmed by the instructor, hence, *Sikuli Slides* [2] was developed to simplify the script generation process to allow less programming-skilled users to generate automation scripts. Instead of a coded script, *Sikuli Slides* represents a process as a Powerpoint presentation with annotations of basic actions on each of the slides. This presentation can be executed as a script. *Sikuli Slides* also provides a recorder application, so a user records a process and saves it as a starting draft of the desired presentation. Our user study compares our framework and this baseline on linear tasks.

Sheepdog [6, 21] and *Familiar* [27] are PbD systems that focus on looping tasks. HILC deals with looping tasks through a different approach, so instead of requiring multiple demonstrations, we ask users to demonstrate once and then give examples of iterators.

Nowadays, smartphones became a part of many people's everyday lives. There are attempts to build PbD systems for a smartphone, *Keep Doing It* [26] and *Sugilite* [24] are PbD system for a smartphone, both of them rely on Android accessibility API to listen to users demonstrated events and operate application elements in the Running Phase. *Keep Doing It* [26] is a PbD system that analyzes users' usage logs to generate automation scripts for users' intended tasks. Toby et al. [24] recently proposed a PbD system called *Sugilite*. The system gives users several programming functionalities: users can modify the generated scripts later when the GUIs change, users can create forks for conditional scripts, and the system learns to generalize the task by using Android accessibility API which provides the system with hierarchical information of the target application.

3 LEARNING DEMONSTRATED TASKS

Our system, HILC[2], has three phases: demonstration, teaching, and running. First, an instructor can choose to record their demonstration as either a screencast video (so highlighting mouse and key presses) or through custom-made sniffer software. Both methods have pros and cons, which we discuss further in the Demonstration Phase section. Next, during the Teaching Phase, the system performs joint segmentation and classification of basic actions. To generalize the observed actions, we introduce a human teacher (can be the same person as the instructor) who can help the system refine its pattern detectors. The challenges and proposed solutions of joint segmentation and classification, and for human in the loop training, are in the Teaching Phase section. Finally in the Running Phase, the system performs actions according to the transcript generated by the instructor and improved by the teacher. Our system also generates a Sikuli-like script for visualization purposes. A stand-alone runtime script in pure python is made using the PyAutoGUI package. Overview of HILC is displayed in Figure 2.

To start, we define a set of basic actions that HILC can perform, and HILC needs to jointly segment and recognize these actions from the input demonstration. The set of basic actions is composed of Left Click, Right Click, Double Click and Click Drag. Each basic action can be operated on a *target*, which is a GUI element, *i.e.,* a button, a text field, an icon, *etc.* . A target has a corresponding appearance or "pattern", which is an image patch around the location of the basic action.

4 DEMONSTRATION PHASE

In this phase, an instructor who knows how to complete the task demonstrates the task while only recording video or while running the sniffer program. If the user chooses to record the task as a video, the system will pre-process the video and create the unified format log-file (see Figure 3) from every frame of the video. On the other hand, if the user chooses to record the task via a sniffer, the system will record every signal produced by the user to the unified format log-file.

[2]Project page: http://visual.cs.ucl.ac.uk/pubs/HILC/

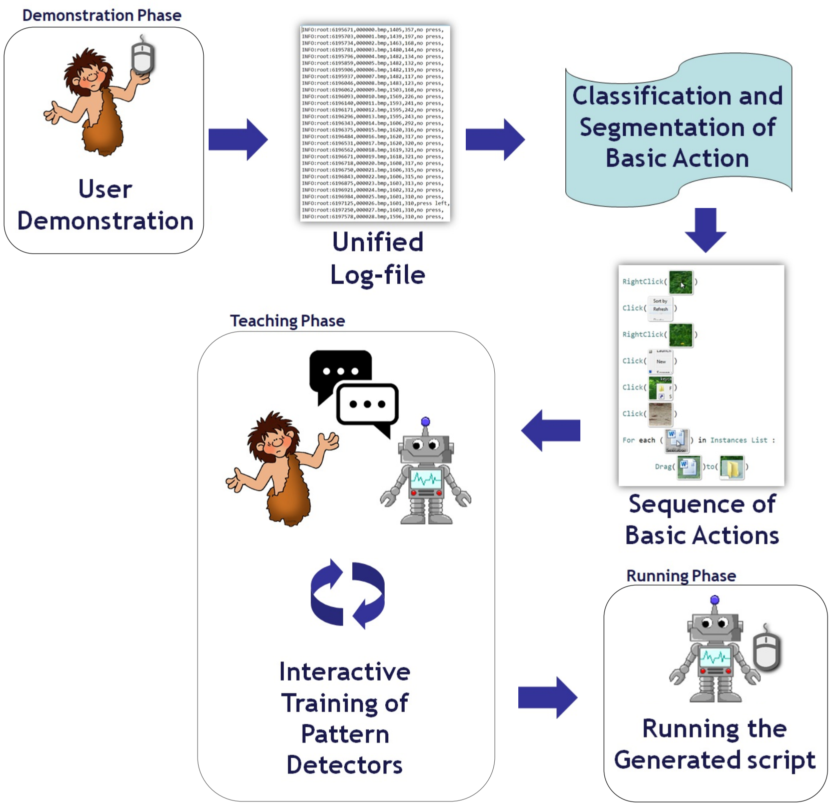
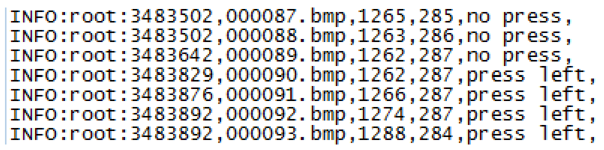

Demonstration Phase
User Demonstration
Unified Log-file
Classification and Segmentation of Basic Action
Sequence of Basic Actions
Teaching Phase
Interactive Training of Pattern Detectors
Running Phase
Running the Generated script

Fig. 2. The flow of the system. Details for each phase are in the text.



```
INFO:root:3483502,000087.bmp,1265,285,no press,
INFO:root:3483502,000088.bmp,1263,286,no press,
INFO:root:3483642,000089.bmp,1262,287,no press,
INFO:root:3483829,000090.bmp,1262,287,press left,
INFO:root:3483876,000091.bmp,1266,287,press left,
INFO:root:3483892,000092.bmp,1274,287,press left,
INFO:root:3483892,000093.bmp,1288,284,press left,
```

Fig. 3. Example of a log-file which merges our inputs from both video and sniffer data.

4.1 Recording the Task

For both input methods (video-only or sniffer) we define a unified sensor-data log format. Functioning like a log-file, each entry records the screenshot image and the low-level status of the machine: its mouse button status, mouse cursor location, and the keyboard's key press status. A sniffer can access useful information directly from the OS, but cannot determine the class of the basic action. It does naturally generate a log-file from the demonstration. However, one must take into account time intervals between each log entry, inconsistent machine lag, and storage space of screenshots. Details are presented in the Implementation section.

To help the system recognize whether a demonstrated segment is a linear, looping, or monitoring/standby task, we use special key combinations that the user was briefed on before using the system. We use the terms monitoring and standby interchangeably and they both refer to the same kind of task. The details of the recording step and the special key combination are presented in the User Study Scenarios section.

Although processing a video-only demonstration is slower, it is more versatile because end-users can leverage pre-recorded and internet-shared videos as the input demonstrations. However, processing of demonstration videos has many challenges, such as locating the mouse cursor, retrieving low-level mouse/keyboard events from screencast messages, removing the mouse pointer from the video's screenshot images (*i.e.*, capturing a template of the button without the cursor's occlusion), and noise/compression in the video recording.

We describe the implementation details of our approach for coping with the aforementioned issues for both sniffer and video inputs in the next section.

4.2 Implementation

The system was implemented in Python 2.7 and it was deployed on Microsoft Windows 7 64-bit machines with Intel Core i5-3317u @1.70GHz CPU and 8GB of RAM.

We implemented a custom sniffer which records low-level mouse and keyboard events. Log-file entries and screenshots were only saved before and after each of the mouse and keyboard status changes, to keep the hard drive i/o from slowing down the machine. We also re-sample records to have log-files where the time difference between records is 1000/30 milliseconds, to make sniffer log-files versions agree with video versions that sample screenshots at 30 frames per second.

Using an existing video as a demonstration requires that we address two important issues: how can the system retrieve low-level mouse and keyboard status information, and how can the system remove the mouse cursor from video frames for clean pattern extraction?

Video tutorials are commonly recorded with specialized screencasting software that renders visual indicators of mouse events, left-right button pressing, and keystrokes. Hence, we assume that the demonstration video was recorded with key-casting software. Each key-caster is different, but we trained for KeyCastOW for Windows and Key-mon for Unix. Others could easily be added. Thus, we extract key-cast display locations from the video and recognize the information for each frame via the open-source OCR software Tesseract [1]. The mouse status information displayed by the key-cast software is still low-level, similar to sniffer output. The mouse cursor location is located by a Normalized Cross Correlation detector [23] of the mouse pointer templates. Videos of demonstrations have an inherent problem of the mouse cursor occluding a target pattern during a basic action. To overcome this, we detect significant appearance changes of the screen and use a temporal median filter to remove the cursor to create a clean mouse-free screenshot.

In the Demonstration Phase, the instructor's basic actions are recorded by the system. Hence, to interact with the system, for example, to start or stop the recording, the instructor triggers explicit signals. In our implementation, these are special key combinations. The detection of loops or standby patterns is not automatic, and has to be flagged by the instructor. Therefore, we define three special key combinations for the different signals: End of Recording, Looping, and Standby, which are discussed in detail before we explain each designated scenario in the Evaluation and Results section.

5 TEACHING PHASE

This phase of the system takes as input the log-file from the Demonstration Phase, and then produces a transcribed script that consists of a sequence of basic actions. The system workflow of the Teaching Phase is in Figure 4.

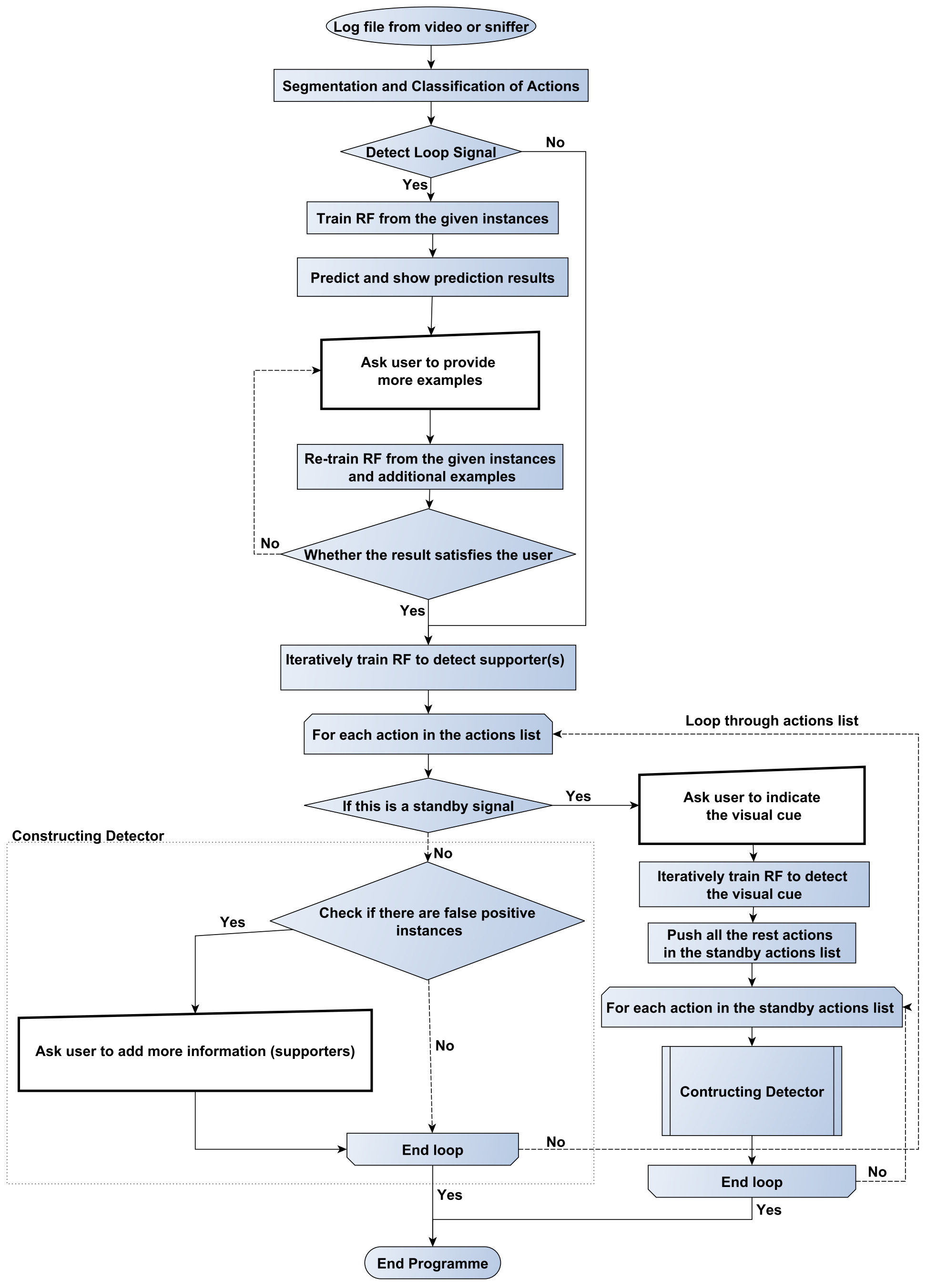

Log file from video or sniffer
Segmentation and Classification of Actions
Detect Loop Signal
No
Yes
Train RF from the given instances
Predict and show prediction results
Ask user to provide more examples
Re-train RF from the given instances and additional examples
No
Whether the result satisfies the user
Yes
Iteratively train RF to detect supporter(s)
For each action in the actions list
Loop through actions list
If this is a standby signal
Yes
Ask user to indicate the visual cue
Constructing Detector
No
Iteratively train RF to detect the visual cue
Yes
Check if there are false positive instances
Push all the rest actions in the standby actions list
For each action in the standby actions list
Ask user to add more information (supporters)
No
Contructing Detector
End loop
No
End loop
No
Yes
Yes
End Programme

Fig. 4. Our system workflow for the Teaching Phase. The thick-edged-white boxes indicate where the system poses questions to the teacher.

5.1 Classification and Segmentation of Basic Actions

Although the log-file contains all of the low-level information, identifying basic actions is not straightforward. For example, the "left mouse button pressed" status can be present in the log-file at multiple consecutive entries for a single "Left Click" basic action. Further, it is ambiguous if a left click is a single "Left Click" basic action, or a part of the "Double Click" basic action. Moreover, variability based on how individual users interact with the computer via peripheral devices makes it hard for deterministic rules to distinguish between different basic actions. This drives the need for training data, though each user provides only very little.

To create a system that can cope with ambiguities in recognizing basic actions, we treat the problem as a Viterbi path decoding problem: based on dynamic programming, our algorithm segments and classifies the basic actions concurrently. The Viterbi algorithm [32] is a dynamic programming algorithm which allows to compute the most likely path of hidden states from a noisy sequence of observed states.

Let $Y = \{y_1, ..., y_z\}$ be the set of all possible states, with a label for each temporal segment. x is the observation of a segment, a feature vector representing part of a basic action. The unary terms $U(y|x)$ are probability distribution functions over parts of basic actions, learned from pre-collected training data, The pairwise terms $P(\tilde{y}, y)$ are the constraints that force consecutive parts of actions to be assigned to the same basic action.

Let us define A_k as a basic action made from a sequence of parts of the action k, $A_k = \{y_1^k, y_2^k, ..., y_l^k\}$; $|A_k| = l$ and y_l^k is the last part of A_k. y_n^k is a part of the basic action k where n indicates a status change frame (key frame) in the log-file, *e.g.*, a frame where the mouse button status is changed, from press to release or vice versa. Pairwise term is defined as in Eq 1:

$$P(y_{v-1}^i, y_v^j) = \begin{cases} +1 & \text{if } i = j \text{ and } y_{v-1}^i \text{ follows } y_v^j, \\ 0 & \text{if } i \neq j \text{ and } y_v^i \text{ is the last part of } A_i, \\ -1 & \text{otherwise.} \end{cases} \tag{1}$$

5.1.1 Learning Probability Distribution Functions. One of the challenges of basic action classification is that each basic action has a different duration. For example, a "Left Click" usually spans a few milliseconds in the log-file, but "Click Drag" may span seconds. Moreover, each basic action also has its inter-variability in terms of action duration.

Our approach is to train a Linear Support Vector Machine (Linear SVM) [4] for each basic action using hard negative mining method [12]. Each basic action is predicted by a corresponding SVM, where the input time interval may be different for different actions (the time interval length is chosen empirically for each basic action). Due to the fact that the data is not linearly separable we train the SVMs by minimizing Hinge-loss function [29]. The Linear SVM is chosen here as we need to quickly process tremendous amounts of time intervals (parts of basic actions).

For an input time interval, we construct a feature vector by computing histograms of frequencies from encodings of a few records from the log-file. Each record is encoded by 3 binary low-level states: the status of each of the mouse buttons: whether the left and right button are pressed, and whether the mouse is moving. The feature vector also encodes context information of each action by adding a histogram of a small time interval after an action is done. Finally, our feature vector for the Linear SVM uses histograms of 3 equally divided time intervals around the action plus the context time interval, which means the feature vector has 4×2^3 dimensions. Figure 5 depicts an input part to the SVM.

The prediction scores of SVMs are not proportional to each other, so they must be scaled to a valid probability distribution before use within the Viterbi algorithm [32]. Hence, for each basic

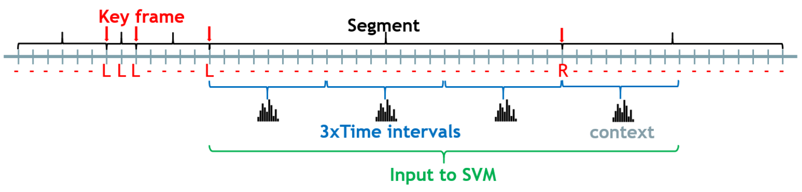

Key frame
Segment
- - - - - - L L L - - - - L - - - - - - - - - - - - - - - - - - R - - - - - - - - - - - - - -
3xTime intervals
context
Input to SVM

Fig. 5. An example segment of the log-file demonstrates input to an SVM. 'L' indicates the left mouse button is pressed, 'R' indicates the right mouse button is pressed, and '-' indicates none of the mouse buttons are pressed.

action, we train a Random Forest (RF) [5], that inputs the prediction scores of SVMs as features and outputs the probability of each basic action class.

For any unknown time interval, we can construct a unary matrix which maps a part of an action to the probability for that action using the trained RF. The unary matrix $U_{s\times\ c}$ has the shape (number of states×number key frames in the input sequences). The number of states is defined as Eq 2

$$\text{number of states} = \sum_{A_k \in B} |A_k| \, , \tag{2}$$

Please note that, we assign the same probability distribution to all parts of each basic action.

In addition, we learn, from the training data, basic actions' important statistics: minimum length, maximum length and the number of change signals of the action. The statistics are used for pre-filtering basic action before passing it through the classifier. We also detect status change frames in the log-file as frames where the mouse button status is changed, from press to release or vice versa. These status change frames are used as possible starting and ending points of basic actions.

5.1.2 Annotation of Log-files for Basic Actions. To annotate an unknown log-file, the system detects status change frames in the log-file and uses them as key frames. Records in the log-file are then grouped into N different time intervals indexed by the key frames and passed to the trained RF to construct the unary matrix U. Lastly, we do inference for $\mathbb{Y}^*$, which is the sequence of y_n^k that maximizes Eq 3 using the Viterbi dynamic programming algorithm [32], where

$$\mathbb{Y}^* = \arg\max_{\mathbb{Y}^*} \sum_{v=1}^{N} (P(y_{v-1}^i, y_v^j) + U(y_v^j | x_v)) \tag{3}$$

5.2 Interactive Training of Pattern Detectors with Few Examples

To perform any basic action, the system needs to learn the appearance of the target of the basic action. For linear tasks, the system needs to find a single correct location of the target pattern. However, for looping tasks, the system has to find multiple correct locations of the targets of the looping task, and thus the system needs to generalize about the target pattern appearance. In both cases, the target pattern may have appearance variations. For example the icon of the file may have moved on the desktop and has a different section of the wallpaper as its background.

Target patterns can also be categorized into two groups: patterns that are discriminative on their own and patterns that need extra information to be distinguishable, see Figure 6. Hence, the system needs to treat each of these possibilities differently and we also introduce a concept of supporters here.

5.2.1 Supporters. The supporters are salience patterns that have certain offsets from the target pattern. In linear tasks, a spreadsheet program's row and column names distinguish similar-looking cells, *e.g.*, Figure 6(c); or field names distinguish similar looking text fields, Figure 6(d). In the Running Phase, the system uses the same technique that is used for detecting the target pattern, to detect supporters. The target patterns give votes to each detection location, but supporters give votes to the offset locations.

Supporters for looping tasks work differently from the supporters for linear tasks, as a fixed offset is not informative for multiple looping targets. Hence, the supporters for looping provide x-axis and y-axis offsets to the targets, see Figure 7. The final result is the average of target pattern detectors and spatial supporters, see Figure 8.

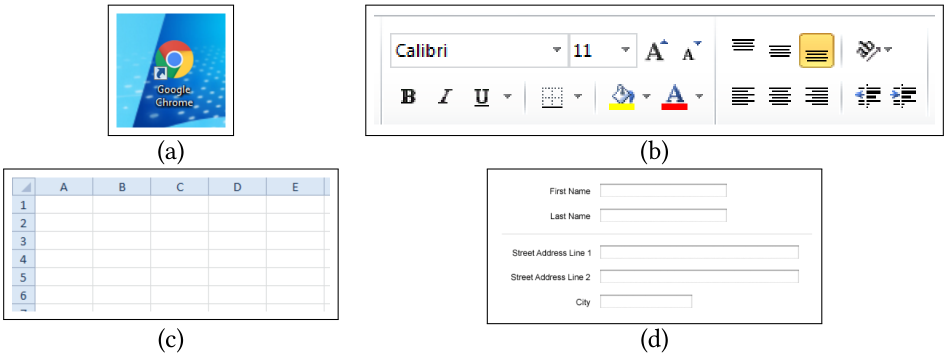
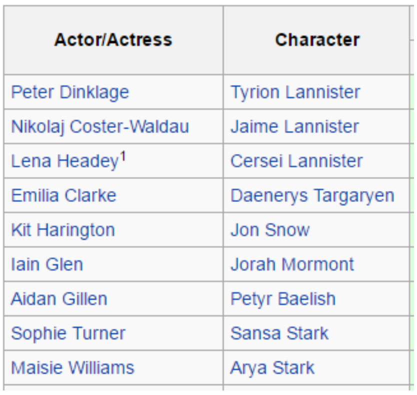


Fig. 6. Target patterns in (a) & (b) are distinguishable on their own. The spreadsheet cells in (c) need row and column names to differentiate between each other. Text fields in registration forms in (d) can be distinguished by the text field labels. Supporter help distinguish locally ambiguous patterns.



Actor/Actress	Character
Peter Dinklage	Tyrion Lannister
Nikolaj Coster-Waldau	Jaime Lannister
Lena Headey[1]	Cersei Lannister
Emilia Clarke	Daenerys Targaryen
Kit Harington	Jon Snow
Iain Glen	Jorah Mormont
Aidan Gillen	Petyr Baelish
Sophie Turner	Sansa Stark
Maisie Williams	Arya Stark

Fig. 7. An example of a supporter for a looping task. The table shows names of characters and actors/actresses of a popular TV show. The names in each column have similar appearance, so, if the user intended to loop through one of the columns, marking the column name as a supporter will help the system to distinguish between columns.

5.2.2 *Follow-up Questions:*

For different kinds of tasks, the system asks for help from the teacher differently.

Linear tasks: Every basic action of linear tasks has a unique target. The linear tasks can be executed multiple times, but each run performs the same task on the same unique targets. For each basic

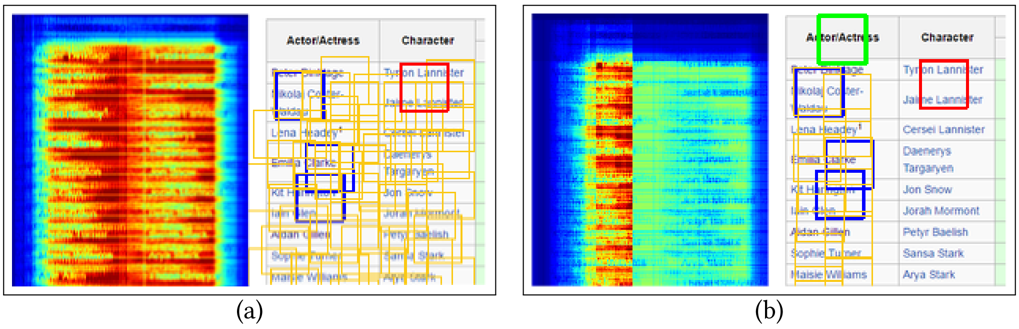


Fig. 8. An example of a spatial supporter. Blue boxes are user provided positive examples, red are user provided negative examples, yellow are target detections, and a green box indicates a user-provided supporter. (a) and (b) demonstrate detection performance without and with a spatial supporter; red means a high detection score. In (a), the left image shows the heatmap of target detection scores, and right shows detected targets. In (b), the left image shows the heatmap of target detection and spatial supporter scores, and right shows detected targets. The spatial supporter successfully suppressed all similar looking patterns under the Character column in the heatmap (b) so that the system is able to detect only desired targets under the Actor/Actress column

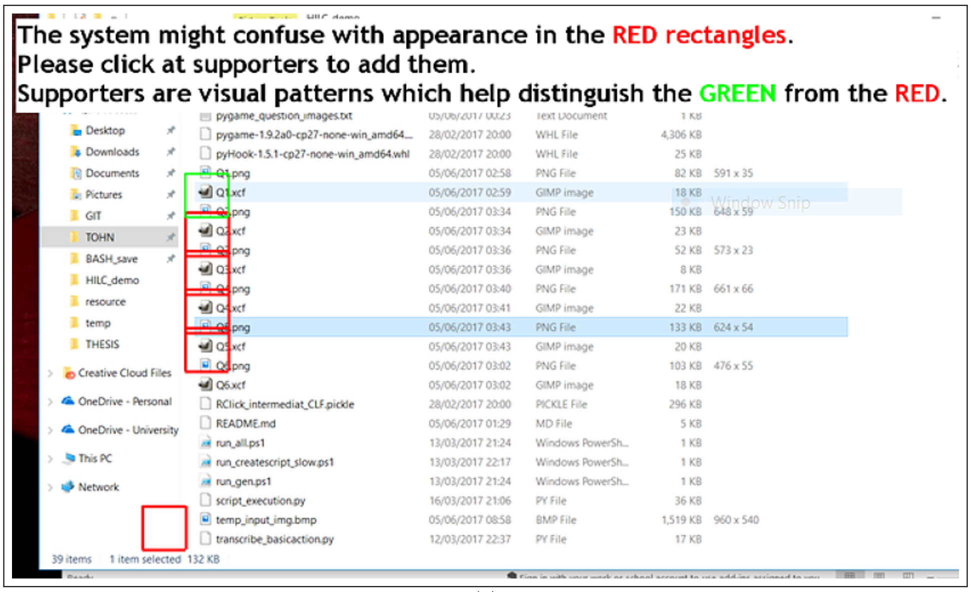
action of the linear task, the system has to learn the corresponding target pattern from only one positive example. To train the detector for the target pattern, the system initially performs pattern matching with the given positive example on the screenshot image when the basic action was about to be executed, to prevent the pattern from changing appearance after the action executed. We choose the most basic pattern matching algorithm, Normalized Cross Correlation (NCC) [23], since it is robust to some level of change and it needs less computation compared to more sophisticated matching algorithms. If there are false-positive locations with high NCC score, the system asks the teacher to help the system to distinguish between true and false positive examples by providing a supporter(s). Figure 9 shows screenshots when HILC queries for help from users.

After the teacher provides supporters, the system uses NCC as the detector for both the target pattern and the supporters. If the teacher does not provide any supporters, the system assumes that this pattern is distinguishable on its own. More false positive patches are mined to retrain the RF classifier until the system can distinguish between true positive and false positive patches. Each RF classifier uses raw RGB pixel values of every position in the patch as features.

Looping tasks: Looping tasks are a generalization of linear tasks, so that in the Running Phase, each run of the task iterates over a set of targets. For example, a linear task always prints a unique PDF file in a folder, but a looping task prints all PDF files in a folder by looping over each PDF file icon.

For looping tasks, the instructor shows the task once on a single looping target, but the system needs to repeatedly perform the task on all looping targets that are similar to the pattern the instructor, or the teacher, or the end-user had specified. In the Demonstration Phase, the instructor is asked to show more than one example of a looping target after demonstrating one complete iteration of a task. In the Teaching Phase, the system trains an RF classifier with the provided positive examples, and random patches as negative examples. Next, the system validates the RF predictions by asking the teacher to verify predicted positive and negative examples, and/or add supporters. Figure 10 illustrates an example screenshot when HILC queries users in a looping task.

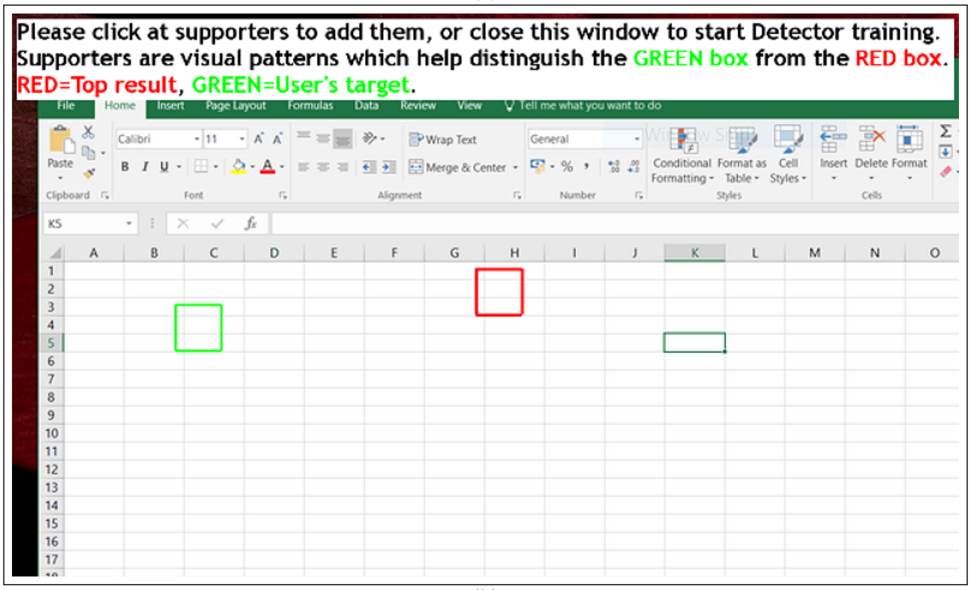
Monitoring tasks: In monitoring tasks, the system at the Running Phase perpetually runs in standby, looking for a specified visual pattern (visual cue), to invoke the rest of the script. In the Demonstration Phase, the instructor indicates when the visual cue appears by inputting a special



The system might confuse with appearance in the RED rectangles.
Please click at supporters to add them.
Supporters are visual patterns which help distinguish the GREEN from the RED.

pygame_question_images.txt	05/06/2017 00:23	Text Document	1 KB	
pygame-1.9.2a0-cp27-none-win_amd64...	28/02/2017 20:00	WHL File	4,306 KB	
pyHook-1.5.1-cp27-none-win_amd64.whl	28/02/2017 20:00	WHL File	25 KB	
Q1.png	05/06/2017 02:58	PNG File	82 KB	591 x 35
Q1.xcf	05/06/2017 02:59	GIMP image	18 KB	
Q2.png	05/06/2017 03:34	PNG File	150 KB	648 x 59
Q2.xcf	05/06/2017 03:34	GIMP image	23 KB	
Q3.png	05/06/2017 03:36	PNG File	52 KB	573 x 23
Q3.xcf	05/06/2017 03:36	GIMP image	8 KB	
Q4.png	05/06/2017 03:40	PNG File	171 KB	661 x 66
Q4.xcf	05/06/2017 03:41	GIMP image	22 KB	
Q5.png	05/06/2017 03:43	PNG File	133 KB	624 x 54
Q5.xcf	05/06/2017 03:43	GIMP image	20 KB	
Q6.png	05/06/2017 03:02	PNG File	103 KB	476 x 55
Q6.xcf	05/06/2017 03:02	GIMP image	18 KB	
RClick_intermediat_CLF.pickle	28/02/2017 20:00	PICKLE File	296 KB	
README.md	05/06/2017 01:29	MD File	5 KB	
run_all.ps1	13/03/2017 21:24	Windows PowerSh...	1 KB	
run_createscript_slow.ps1	13/03/2017 22:17	Windows PowerSh...	1 KB	
run_gen.ps1	13/03/2017 21:24	Windows PowerSh...	1 KB	
script_execution.py	16/03/2017 21:06	PY File	36 KB	
temp_input_img.bmp	05/06/2017 08:58	BMP File	1,519 KB	960 x 540
transcribe_basicaction.py	12/03/2017 22:37	PY File	17 KB	

(a)

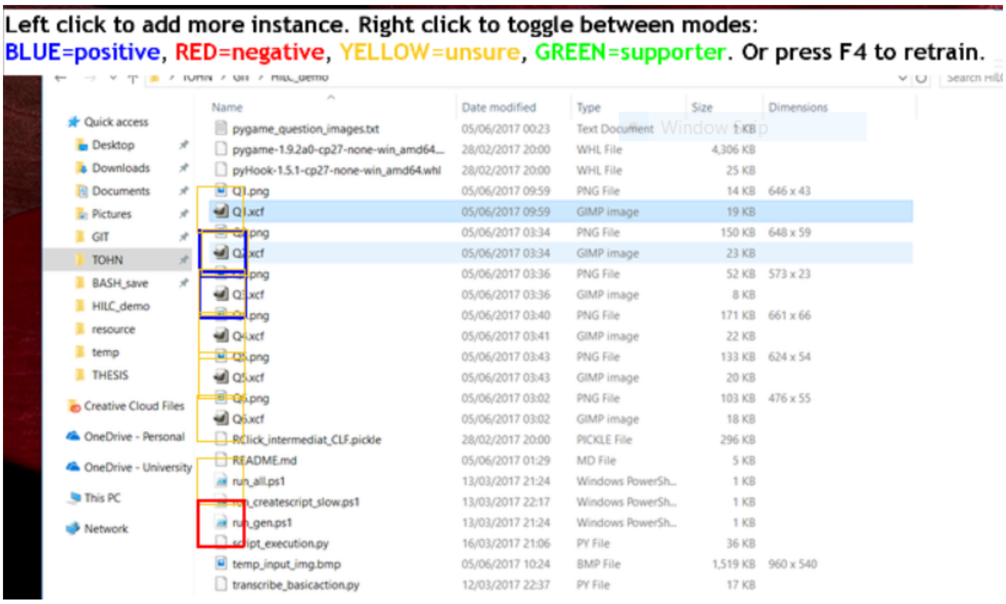


Please click at supporters to add them, or close this window to start Detector training.
Supporters are visual patterns which help distinguish the GREEN box from the RED box.
RED=Top result, GREEN=User's target.

(b)

Fig. 9. Example screenshots of HILC when it asks users for supporters. (a) HILC asks users to add supporters when there are confusing patterns (red boxes) which look similar to the user intended patterns (green box). In this case, NCC scores of the confusing patterns are higher than a threshold. The system uses only the NCC detector for the intended pattern at the Running Phase, unless the users provide supporter(s). (b) HILC asks users to add supporters when a confusing pattern (red box) has the same NCC score to the intended pattern (green box). Unless the users provide at least one supporter, HILC trains RF detector for the intended pattern.

key combination, then demonstrates the task itself. In the Teaching Phase, the system asks the teacher to point out which pattern needs to be detected as the visual cue.


Left click to add more instance. Right click to toggle between modes:
BLUE=positive, RED=negative, YELLOW=unsure, GREEN=supporter. Or press F4 to retrain.

Name	Date modified	Type	Size	Dimensions
pygame_question_images.txt	05/06/2017 00:23	Text Document	1 KB	
pygame-1.9.2a0-cp27-none-win_amd64...	28/02/2017 20:00	WHL File	4,306 KB	
pyHook-1.5.1-cp27-none-win_amd64.whl	28/02/2017 20:00	WHL File	25 KB	
Q1.png	05/06/2017 09:59	PNG File	14 KB	646 x 43
Q1.xcf	05/06/2017 09:59	GIMP image	19 KB	
png	05/06/2017 03:34	PNG File	150 KB	648 x 59
Q xcf	05/06/2017 03:34	GIMP image	23 KB	
png	05/06/2017 03:36	PNG File	52 KB	573 x 23
Q xcf	05/06/2017 03:36	GIMP image	8 KB	
png	05/06/2017 03:40	PNG File	171 KB	661 x 66
Q4.xcf	05/06/2017 03:41	GIMP image	22 KB	
Q5.png	05/06/2017 03:43	PNG File	133 KB	624 x 54
Q5.xcf	05/06/2017 03:43	GIMP image	20 KB	
Q6.png	05/06/2017 03:02	PNG File	103 KB	476 x 55
Q6.xcf	05/06/2017 03:02	GIMP image	18 KB	
RClick_intermediat_CLF.pickle	28/02/2017 20:00	PICKLE File	296 KB	
README.md	05/06/2017 01:29	MD File	5 KB	
run_all.ps1	13/03/2017 21:24	Windows PowerSh...	1 KB	
_createscript_slow.ps1	13/03/2017 22:17	Windows PowerSh...	1 KB	
ru_gen.ps1	13/03/2017 21:24	Windows PowerSh...	1 KB	
ipt_execution.py	16/03/2017 21:06	PY File	36 KB	
temp_input_img.bmp	05/06/2017 10:24	BMP File	1,519 KB	960 x 540
transcribe_basicaction.py	12/03/2017 22:37	PY File	17 KB	

Fig. 10. An example screenshot of HILC when it asks users for additional information during the Teaching Phase of looping tasks.

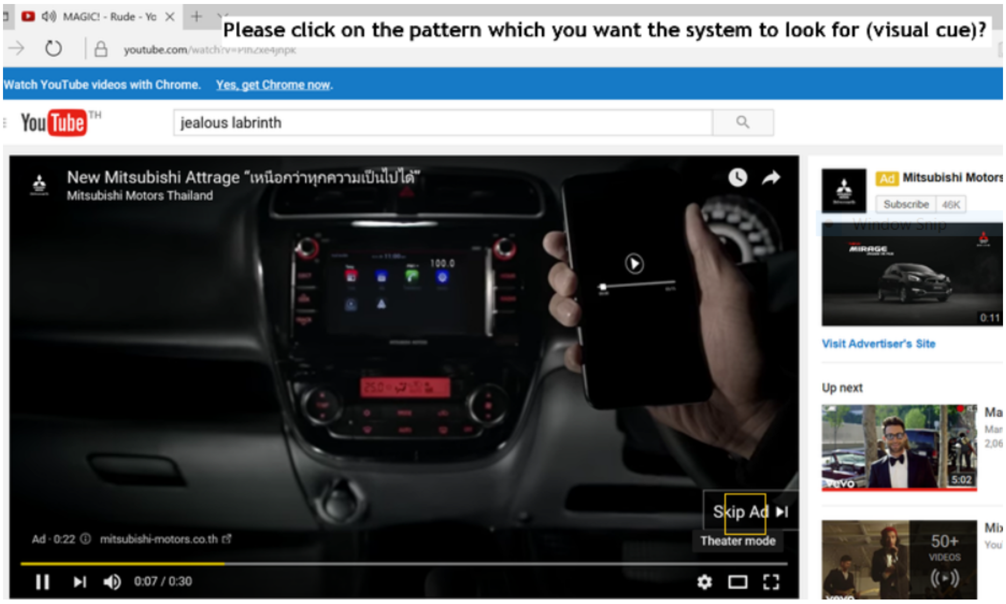

Please click on the pattern which you want the system to look for (visual cue)?

Fig. 11. An example screenshot of HILC when it requests a user to provide the visual cue (yellow box), the visual pattern which invokes the rest of the script whenever the system finds it.

Figure 11 displays an example screenshot while HILC is in the Teaching Phase, and asks for the visual cue. Examples of visual cues are illustrated in Figure 16 and 17.

6 RUNNING PHASE

The main reason we separate the Running Phase from the Demonstration Phase and the Teaching Phase is that our ultimate goal for the system is to help end-users who are non-regular computer users, or users with disabilities such as motor impaired persons, to complete tasks that might be hard for them but easy for others. The separated system is easy to execute by voice command or any other kind of triggering methods.

In the Running Phase, the system sequentially executes each action of the interpreted sequence of actions from the Teaching Phase. When a special signal like Looping or Standby is found, the system executes the specific module for each signal. The Running Phase's system flow is shown in Figure 12. For each normal basic action, the system starts by taking a screenshot of the current desktop and then looks for the target pattern and supporters (if available) using the trained detector(s). For the looping part, after taking a screenshot of the current desktop, the system evaluates every position on the screen with the trained detector RF, and applies the spatial voting from the supporter(s). After that, the non-maxima suppression and thresholding are applied respectively to the result, to get the list of positions to loop over. For the standby task, the system continuously takes a screenshot of a current desktop and checks if the target pattern appeared in the specified locations. When the target pattern is found, the system triggers the sequence of actions that the instructor designed, and then proceeds to the standby loop again.

7 EVALUATION AND RESULTS

We evaluated our algorithm quantitatively through a small user study, and qualitatively to probe our system's functionality. Here, we document nine use cases, and video of some of these is in the Supplemental Material. The only viable baseline PbD system available for comparison is Sikuli Slides[2], because it too assumes users are non-programmers, and it too has sniffer-like access to user events. We collected (through a survey) and prioritized a larger list of scenarios for which the participants would like a virtual personal assistant to complete a task. We picked scenarios that spanned the different basic actions, and sampled the space between having just two steps and up to 17 steps. Here we list the three scenarios tested in the seven-person user-study, and show its results in Table 3. Just these three linear tasks were picked because Sikuli Slides cannot handle looping or standby tasks. Further scenarios are discussed in the qualitative evaluation.

The task in each scenario was assessed in terms of 1) transcription accuracy (evaluating classification and segmentation), 2) task reproduction, *i.e.,* measuring pattern detection generalization, and 3) time users took to demonstrate and then refine the task model. In our studies, four of the participants had no programming exposure, two had taken a school-level course, and one was a trained programmer. Participants needed 1.5 - 2 hours because each completed all three tasks under both systems: they randomly started with either ours or Sikuli Slides, and then repeated the same task with the other system before proceeding to the next task. Before using both systems the users were briefed about the goal of the study as well as how to for using both systems for 20 minutes. In addition, the users were shown the videos of the instruction phase for each task before performing the task to ensure the users understand what are the tasks.

7.1 User Study Scenarios:

We evaluate the first three basic scenarios (linear tasks) quantitatively against Sikuli Slides and evaluate monitoring and looping tasks qualitatively.

Linear Tasks Linear tasks are simply linear sequences of actions. They are the most basic type of task, and run only once. To record and edit all kinds of tasks, two common steps need to be done: First, at the end of a task-demonstration, the user presses the special keys combination `Shift+Esc`,

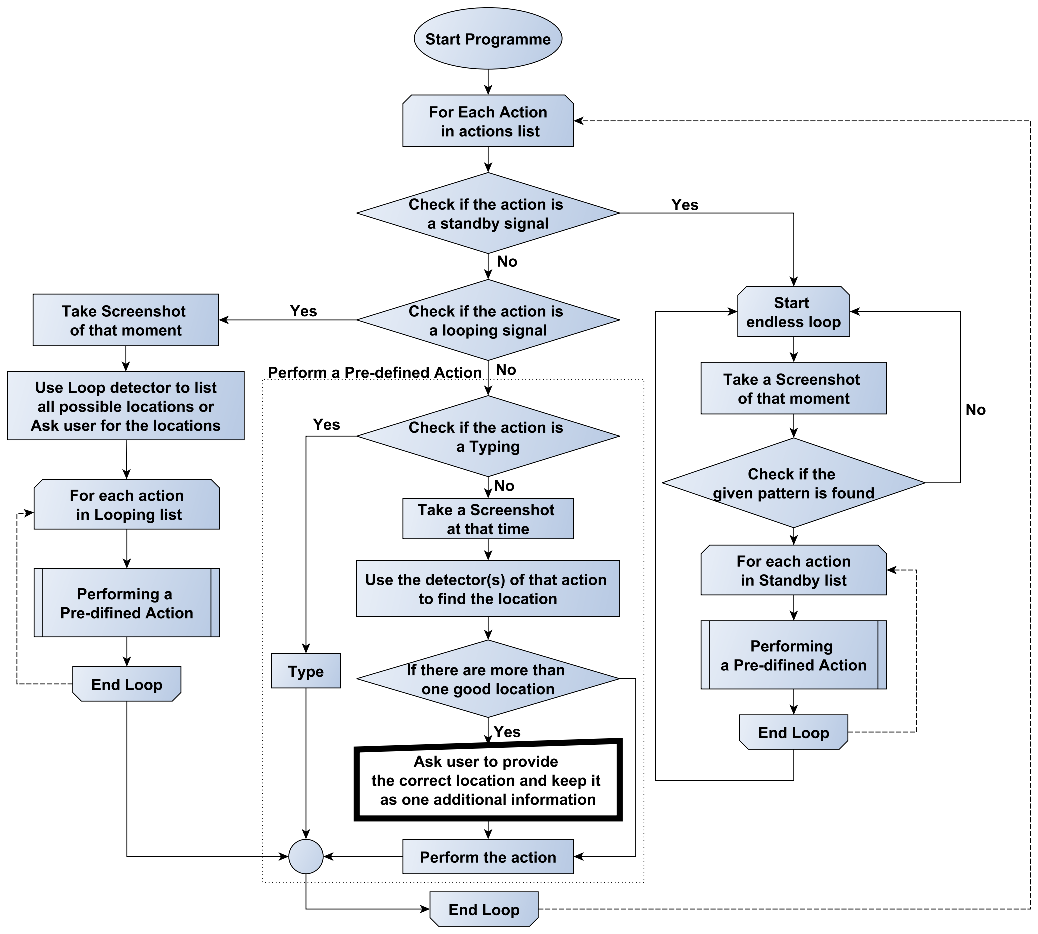

Start Programme
For Each Action in actions list
Check if the action is a standby signal
Yes
No
Check if the action is a looping signal
Yes
No
Take Screenshot of that moment
Use Loop detector to list all possible locations or Ask user for the locations
For each action in Looping list
Performing a Pre-difined Action
End Loop
Perform a Pre-defined Action
Check if the action is a Typing
Yes
No
Type
Take a Screenshot at that time
Use the detector(s) of that action to find the location
If there are more than one good location
Yes
Ask user to provide the correct location and keep it as one additional information
Perform the action
End Loop
Start endless loop
Take a Screenshot of that moment
Check if the given pattern is found
No
For each action in Standby list
Performing a Pre-difined Action
End Loop

Fig. 12. Our system workflow for the Running Phase. The thick-edged-white box indicates user interaction.

to indicate the end of the sequence. Second, also in the Teaching Phase, if the system cannot clearly distinguish between an input pattern and the other on-screen content, the system asks the teacher to click on supporter(s) near that pattern.

7.1.1 Mute audio playback. (Linear) This simple and short task was actually non-trivial because the speaker icon in some Windows installations is not unique (Figure 13). Half the users had to refine the model, which for our system meant adding a supporter. All users produced a working model of this task using our system, and some users were able to produce a working model of the task using Sikuli Slides (the right speaker icon was correctly selected by chance).

7.1.2 Turn on High-Contrast-Mode. (Linear) Some visually impaired *end-users* may want to trigger this task by voice command through a speech recognition system. Here, our sighted *instructors* were ultimately successful using both systems, but the study-supervisor had to walk users of Sikuli Slides through the extra steps of re-demonstrating the task and making and editing of screen-shots to refine that model. This task involves the Click Drag action, which Sikuli Slides was never able to recognize when transcribing. Figure 14 illustrates the High-Contrast vs default desktop modes, and

Scenario	Transcription		Reproduction		Demonstration Time VS Refining Time (average)	
	Sikuli Slides	HILC	Sikuli Slides	HILC	Sikuli Slides	HILC
7.1.1 Mute audio playback	✓	✓	✓*	✓	10s/49s	10s/27s
7.1.2 High-Contrast-mode	✓*	✓	✓**	✓	27s/10m	27s/170s
7.1.3 Remote access (1)	✓*	✓	✗	✓	40s/∞	40s/4m
7.1.3 Remote access (2)	✓*	✓	✗	✓	37s/∞	37s/7m
7.1.4 Skip Youtube ads.	✗	✓	✗	✓	N/A	10s/5.5m
7.1.5 Close Youtube ads.	✗	✓	✗	✓	N/A	12s/6.9m
7.1.6 Create slides of Jpegs	✗	✓	✗	✓	N/A	35s/10m
7.1.7 Create spreadsheet	✗	✓	✗	✓	N/A	60s/6.6m
7.1.8 Create Bibtex	✗	✓	✗	✓	N/A	86s/12.5m
7.1.9 Move files	✗	✓	✗	✓	N/A	25s/22m

Table 3. User study on our system compared to Sikuli Slides. Scenario "7.1.3 Remote access (2)" is an alternative way to perform Scenario 7.1.3, without pressing shortcut key combinations that Sikuli Slides is known to be missing. (✓= successful, ✓* = partially successful, ✓** = can be successful with guidance from the operator, ✗= cannot succeed at the task at all). x represents the number of repeated loops needed to complete the task. Please note that 90% of the refining time for Task 9 is offline - devoted to the time spent on processing video to produce the log-file.

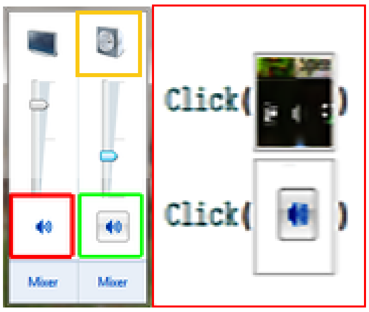


Fig. 13. The instructor clicked on the speaker in the green box, but the system also detected a similar pattern - the speaker in the red box. In this situation, our system asks the teacher for a supporter(s), the yellow box, to help with detecting the intended pattern.

our synthesized script for switching. Please note that High-Contrast-Mode also modifies the scale of objects on the screen.

7.1.3 Remote access with TeamViewer. (Linear) This sysadmin (or mobile-phone testing) task consists of running the TeamViewer application and logging into another device, using an ID and password provided in a spreadsheet file. The Teaching Phase of the task involves helping the system clarify ambiguous patterns by adding supporters. We illustrate the task and the transcribed steps in Figure. 15.

Inadvertently, this task proved impossible for Sikuli Slides users because it involves copy-pasting text, which that system does not capture keys combination shortcut. One user invented an alternate version of this scenario (7.1.3 Remote access (2)) where she tried to right-click and use a context menu to copy and paste, but we then realized that right-clicks are also not captured by Sikuli Slides.

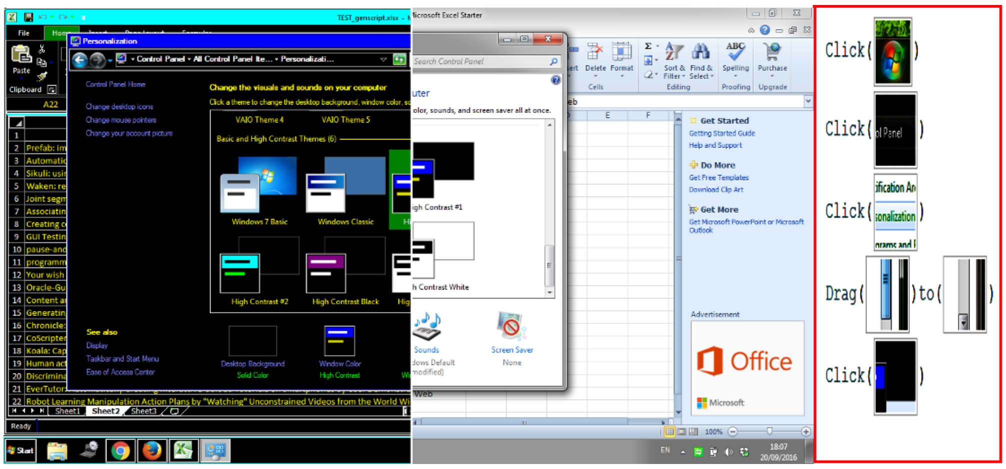


Fig. 14. High-Contrast-Mode comparing with Normal Mode and (in the red box) the transcribed steps of the task demonstrated by our system.

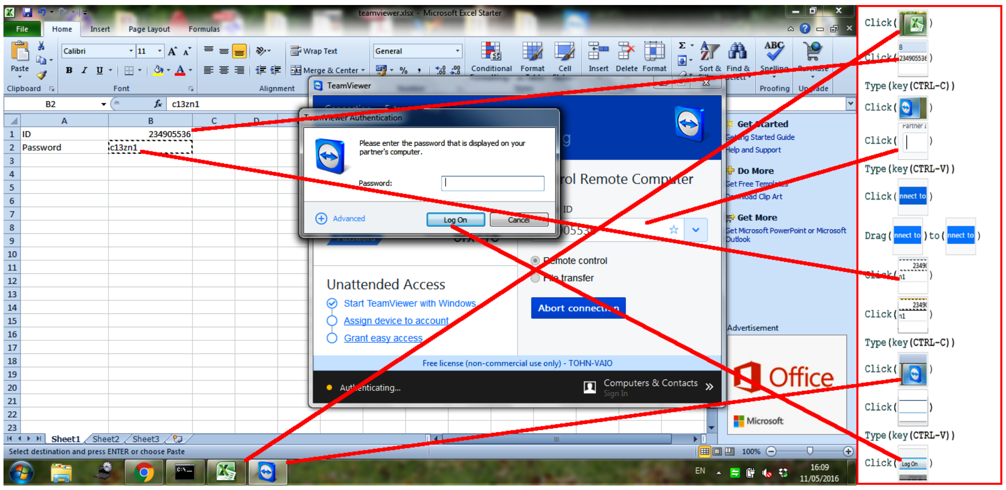


Fig. 15. Steps to complete remote access via TeamViewer. Red lines link related patterns on the screen with the pattern in the transcript. Please note that performing the basic action DragTo from and to the same pattern has a similar effect as performing the basic action Click on that pattern. Our system is robust to this type of different-but-interchangeable action.

Qualitative Evaluation: The remaining scenarios cannot be addressed using Sikuli Slides because they entail monitoring or looping tasks. We outline our new capabilities here, along with qualitative findings, and task illustrations, to better gauge success.

Monitoring Tasks Monitoring tasks run perpetually and then respond to specific patterns. When the specified pattern is detected, the script triggers the sequence of predefined actions.

In the Demonstration Phase, instructors press the special key combination (standby signal), `Ctrl+Shift+w` or `Ctrl+Shift+PrtScr`, to indicate that the pattern, which we want the system to detect, has appeared. The instructor then performs the desired sequence of actions, such as a linear task. There is an extra step in the Teaching Phase, where the system asks the teacher to indicate where the invocation pattern *can* occur (*e.g.*, anywhere, or in the taskbar).

7.1.4 Skip YouTube ads. (Monitoring) is a standby task that requires the system to click the text *Skip Ad* if/when it appears during a YouTube video. This task illustrates the need for spontaneous responses, because the *Skip Ad* advertisement banners appear randomly, for varying periods of time, ranging from 10 seconds to a few minutes, during playback of the requested content. Figure 16 demonstrates an example of the *Skip ad* task.

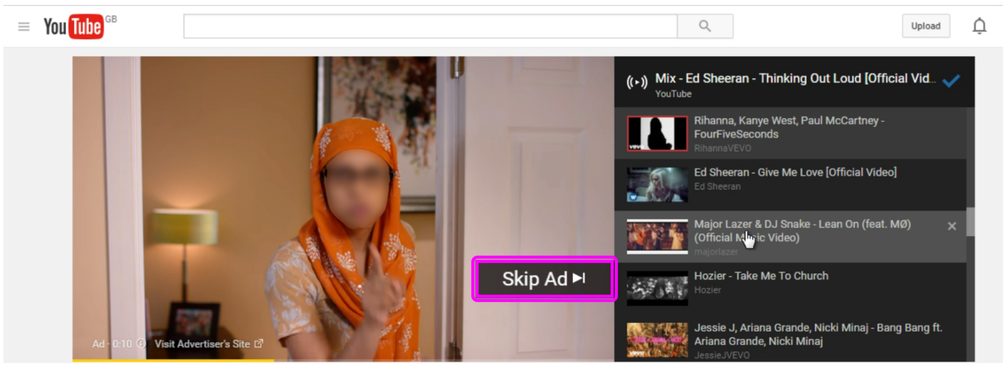
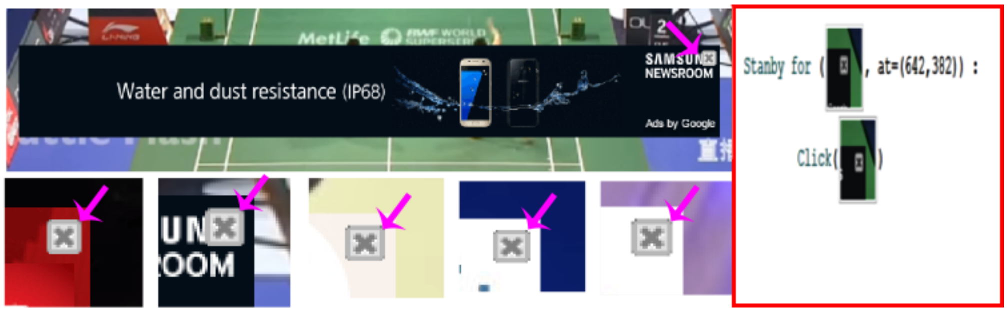

YouTube GB
Upload
Mix - Ed Sheeran - Thinking Out Loud [Official Vid...
YouTube
Rihanna, Kanye West, Paul McCartney - FourFiveSeconds
RihannaVEVO
Ed Sheeran - Give Me Love [Official Video]
Ed Sheeran
Major Lazer & DJ Snake - Lean On (feat. MØ) (Official Music Video)
majorlazer
Hozier - Take Me To Church
Hozier
Jessie J, Ariana Grande, Nicki Minaj - Bang Bang ft. Ariana Grande, Nicki Minaj
JessieJVEVO
Skip Ad ▸|
Ad · 0:10 Visit Advertiser's Site

Fig. 16. YouTube Skip Ad. These advertisements are shown before or during a playing video for varying periods of times, and our system successfully closes them in Scenario 7.1.4, as soon as the text appears. The visual cue is highlighted by the magenta rectangle.

7.1.5 Close YouTube ads. (Monitoring) creates a standby script to close advertisements that may appear, despite various changing backgrounds, as shown in Figure 17. The first line of the script directs the system to monitor an area where the given pattern can appear. When the system detects the pattern, the system triggers a script, in the second line, to click on that pattern.



```
Stanby for ( , at=(642,382)) :
    Click( )
```

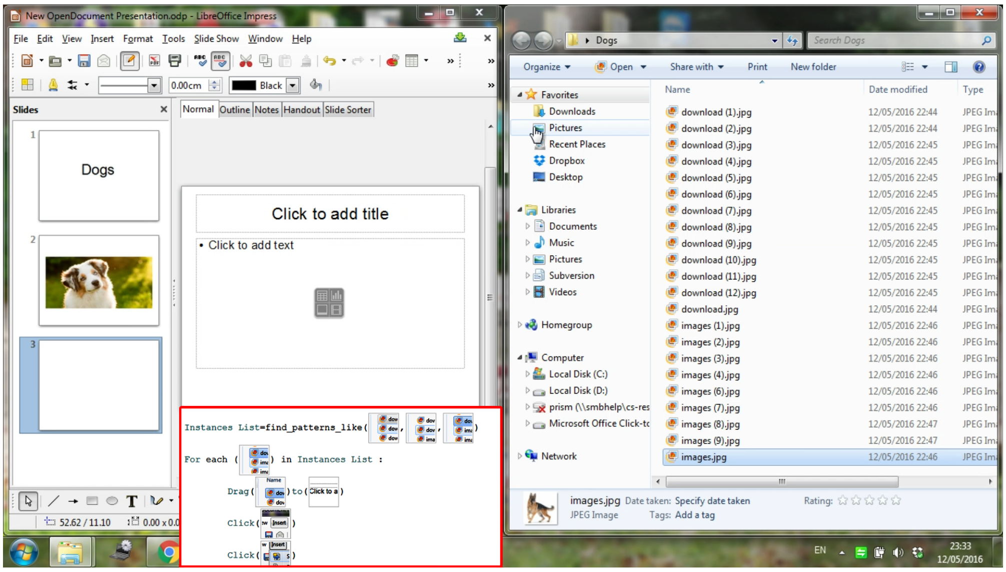
Fig. 17. Close-ups of YouTube Ads. These ads appear at the bottom of a playing video, and our system detects and successfully closes them in Scenario 7.1.5. The visual cues are pointed by magenta arrows.

Looping Tasks Our system allows a loop to be a step in a linear action, or to be a stand-alone script. Looping tasks are the tasks that execute the same sequences of linear actions multiple times on similar looking yet different objects.

To indicate the start and stop of a loop, the demonstrator inputs the looping signal key combination Ctrl+Shift+l or Ctrl+Shift+Break, before and after performing one sequence of actions that need to be repeated. Thereafter, the instructor gives examples of patterns that needed to be a starting point of the loop by pressing a Ctrl key while clicking on an example pattern. When the instructor is happy with the examples, they then input the looping signal once more. The script can be followed by linear actions or can finish right after the third looping signal.

The Teaching Phase of a looping task has one additional step. The system displays the result of the trained Random Forest, and lets the teacher add positive examples, remove false positive results and provide supporters. This triggers re-training.

7.1.6 Create slides out of jpgs folder. (Looping) The task is to create a presentation where each slide features one image from a given folder. To create the script, a demonstrator only shows how to create one slide from one jpg, and gives a few examples of what the jpg file-icon looks like. In the Running Phase, the system steps through all the jpg files in a given folder, making each one into a separate slide of the LibreOffice Impress presentation. Not only does this show that the system can loop, the task also demonstrates that the system can help the user complete repeated steps across different applications (LibreOffice Impress and Windows Explorer). We show an example screen of the task and the generated script in Figure 18.

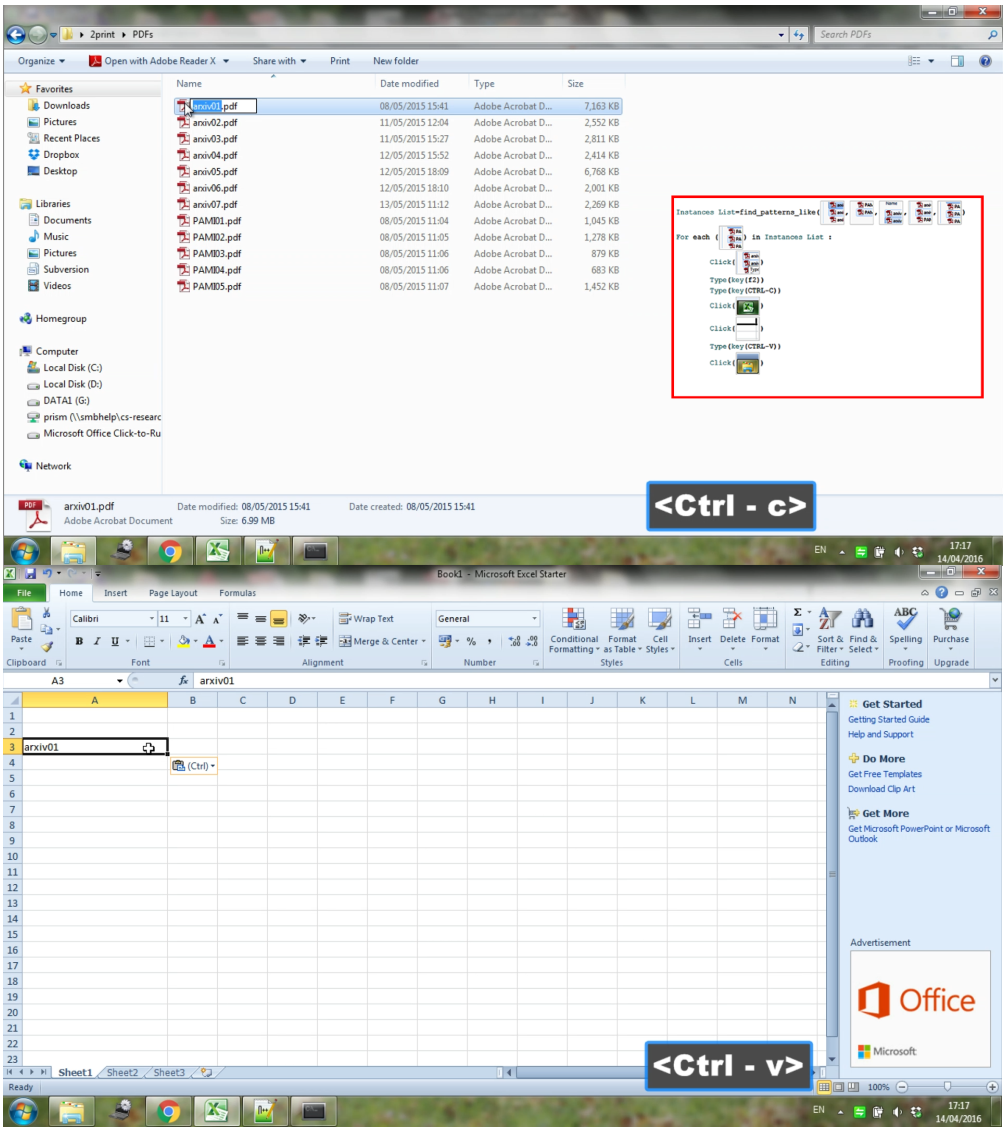


```
Instances List=find_patterns_like( , , )
For each ( ) in Instances List :
        Drag( ) to ( )
        Click( )
        Click( )
```

Fig. 18. Scenario 7.1.6: create slides from folder full of images. The generated script is shown in the red frame. The system starts by building a list of locations that will be the starting points for each iteration. The list is formed by the Trained RF, which trained and refined in the Teaching Phase with a few examples stemming from the Demonstration Phase. The system then iteratively executes a sequence of actions from line three to five (DragTo, Click, Click). In this scenario, the two applications are displayed side-by-side.

7.1.7 Create a spreadsheet of filenames. (Looping) The purpose of this scenario is to create a list of filenames in a spreadsheet program, Figure 19. In the Running Phase, the system copies the filenames from within a given folder into successive Microsoft Excel cells. While repeatedly successful, the paste operation targeted the cell below the previously-selected (dark outline) cell on the spreadsheet. So the first entry will always be pasted below the initially selected cell.

7.1.8 Create a BibTex file from a spreadsheet. (Looping) This is the most complex of all the scenarios listed here. The task involves switching between three different applications (eight different screens). An instructor needs to plan out the task's steps, to ensure each application is in a state that is ready for the same action of the next loop to be executed. In the Running Phase, the



Name	Date modified	Type	Size
arxiv01.pdf	08/05/2015 15:41	Adobe Acrobat D...	7,163 KB
arxiv02.pdf	11/05/2015 12:04	Adobe Acrobat D...	2,552 KB
arxiv03.pdf	11/05/2015 15:27	Adobe Acrobat D...	2,811 KB
arxiv04.pdf	12/05/2015 15:52	Adobe Acrobat D...	2,414 KB
arxiv05.pdf	12/05/2015 18:09	Adobe Acrobat D...	6,768 KB
arxiv06.pdf	12/05/2015 18:10	Adobe Acrobat D...	2,001 KB
arxiv07.pdf	13/05/2015 11:12	Adobe Acrobat D...	2,269 KB
PAMI01.pdf	08/05/2015 11:04	Adobe Acrobat D...	1,045 KB
PAMI02.pdf	08/05/2015 11:05	Adobe Acrobat D...	1,278 KB
PAMI03.pdf	08/05/2015 11:06	Adobe Acrobat D...	879 KB
PAMI04.pdf	08/05/2015 11:06	Adobe Acrobat D...	683 KB
PAMI05.pdf	08/05/2015 11:07	Adobe Acrobat D...	1,452 KB

arxiv01.pdf Date modified: 08/05/2015 15:41 Date created: 08/05/2015 15:41
Adobe Acrobat Document Size: 6.99 MB

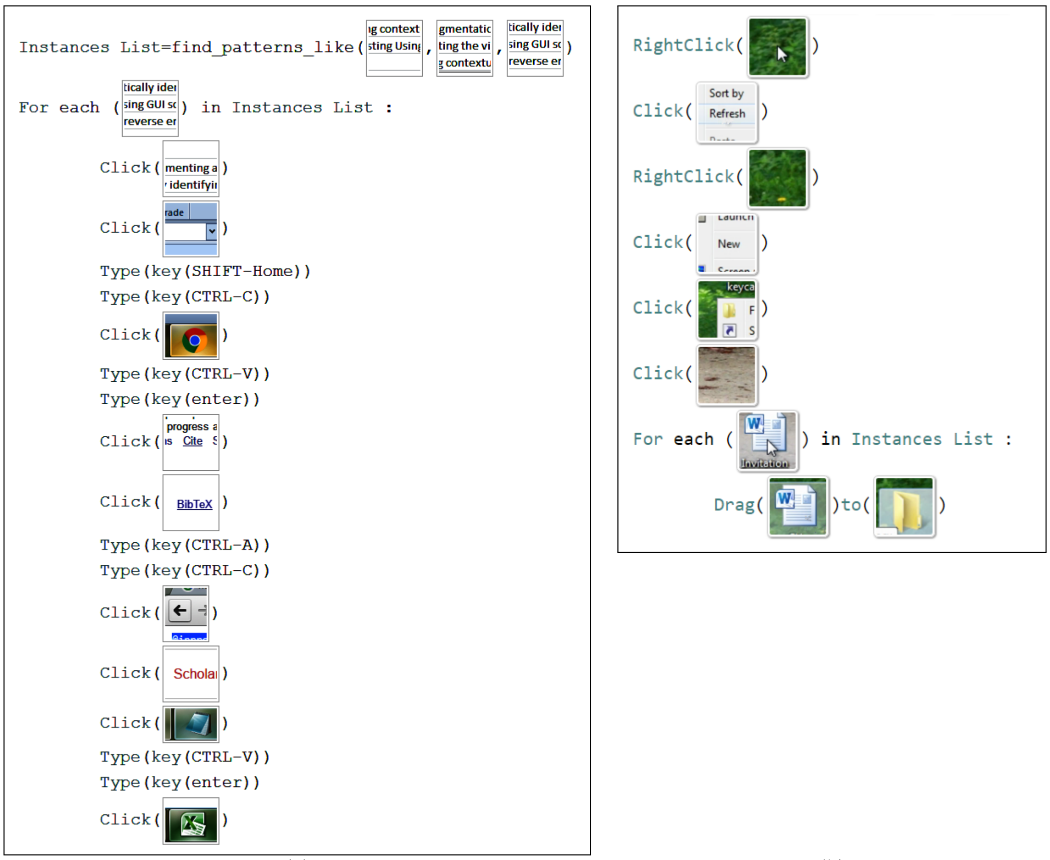
```
Instances List=find_patterns_like( , , , , )
For each ( ) in Instances List :
        Click( )
        Type(key(f2))
        Type(key(CTRL-C))
        Click( )
        Click( )
        Type(key(CTRL-V))
        Click( )
```

A3 arxiv01

Fig. 19. Two application's screens from Scenario 7.1.7, where file names are being collected into a spreadsheet. The script of the task, in the red frame, involves switching back and forth between the two applications, and pasting the text into similar-looking cells.

system works through an Excel file that lists titles of papers that should be cited. The system then uses Google Scholar to look for the BibTex of each paper, and produces a single BibTex file listing all the citations using the Notepad program. Figure 20 (a) illustrates the generated script.

7.1.9 Move MSWord files to a folder. (Looping-Video) In this scenario, we tested a further proof of concept of our system. The system successfully uses *only* video from a screencast as input, instead of data from the sniffer, illustrating that instructors could post how-to-videos online, which can then easily be refined into a working script.



```
Instances List=find_patterns_like( , , )

For each ( ) in Instances List :

        Click( )

        Click( )

        Type(key(SHIFT-Home))
        Type(key(CTRL-C))

        Click( )

        Type(key(CTRL-V))
        Type(key(enter))

        Click( )

        Click( )

        Type(key(CTRL-A))
        Type(key(CTRL-C))

        Click( )

        Click( )

        Click( )

        Type(key(CTRL-V))
        Type(key(enter))

        Click( )
```

```
RightClick( )

Click( )

RightClick( )

Click( )

Click( )

Click( )

For each ( ) in Instances List :

    Drag( )to( )
```

(a) (b)

Fig. 20. A synthesized scripts (a) result of Scenario 7.1.8, where a BibTex file is automatically constructed from a list of paper titles. Three different desktop GUI's were involved. The user was able to train the system quite easily, and can just run the task without further instructions when writing their next research paper. (b) The system successfully uses videos as the inputs of the system, instead of generating the input log-file from the sniffer, to create a working script. Please note that the system failed to remove mouse pointer from the target patterns in the first and the seventh lines.

The scenario starts by executing a sequence of linear basic actions to create a new folder. It then continues to iteratively Drag and Drop each Microsoft Word file into the newly created folder. The script is shown in Figure 20 (b).

8 DISCUSSION AND FUTURE WORK

The two sets of evaluation scenarios showed that our approach substantially extends the programming by demonstration functionality that was available to non-programming users of desktop-automation tools. The main innovation is the sanity-check performed when the instructor demonstrates their task: given a cooperative human, it allows the system to transition from a winner-takes-all template-matching view of targets and actions, onto a supervised-classification interpretation of the instructor's intentions.

This prototype has important opportunities for improvements. Basic actions are occasionally misclassified, when none of them has a high probability. Tests showed the joint segmentation and classification algorithm has an average accuracy of 95.4% for classifying each basic action. Our system allows users to fix misclassified actions instead of requiring a user to re-record the instruction again. Users were more successful and could do more with our system, but found the concept of supporters somewhat foreign, at least as presented in our instructions.

Currently, the system works without the awareness of states of the computer. For example, if a task expects to work with a pre-opened folder (or to open a closed one), the end-user must prepare their desktop appropriately. This could be addressed in the future by intelligently chaining tasks together, and showing temporary pop-up messages to indicate success or failure of a task.

In addition, we insert short fixed-length sleep() commands after each action to account for loading time of the computer because the system cannot know if the OS task has finished/web-page has loaded. Therefore, shorter sleeps would make automated tasks go faster, but the system could operate actions before the GUI is ready. This could be addressed in the future by training the system to recognize computer states from visual signals.

Finally, the current appearance models have fixed size and aspect ratio, which can hurt accuracy when items in a list are short and wide. Learned appearance features, even spanning across devices, could emerge, given enough training footage.

Although we have a working algorithm for the video input pre-processing, the usability is still very limited because of two main obstacles which need to be addressed in future research. Firstly, processing video tutorial takes long time because the system has to analyze every frame for mouse and keyboard button status using Tesseract [1], and the mouse pointer location using NCC [23]. Secondly, videos, especially videos from the Internet, have noise and compression artifacts due to recording software-device and policies of video sharing websites. This leads to an inaccurate log-file generation which propagates errors to the rest of the pipeline.

ACKNOWLEDGMENTS

We thank all the volunteers, and all reviewers, who provided helpful comments on previous versions of this paper. Authors gratefully acknowledge the Ministry of Science and Technology of Thailand Scholarship and the EPSRC grants EP/K023578/1 and EP/K015664/1.

REFERENCES

[1] 2006. Tesseract. https://github.com/tesseract-ocr/tesseract/. Accessed: 26th June, 2018.
[2] 2014. Sikuli Slides. http://slides.sikuli.org/. Accessed: 26th May, 2017.
[3] Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2012. Waken : Reverse Engineering Usage Information and Interface Structure from Software Videos. *UIST '12* (2012), 83–92.
[4] Christopher M Bishop. 2006. *Pattern recognition and machine learning*. springer.
[5] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324
[6] Vittorio Castelli, Lawrence Bergman, Tessa Lau, and Daniel Oblinger. 2010. Sheepdog, Parallel Collaborative Programming-by-Demonstration. *Knowledge-Based Systems* (2010).
[7] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. 2011. Associating the Visual Representation of User Interfaces with Their Internal Structures and Metadata. *UIST '11* (2011), 245.
[8] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. 2010. GUI Testing Using Computer Vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010), 1535–1544.
[9] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch What I Do: Programming by Demonstration.* MIT press.
[10] Morgan Dixon, Daniel Leventhal, and James Fogarty. 2011. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011), 969.
[11] Morgan Dixon, A. Conrad Nied, and James Fogarty. 2014. Prefab Layers and Prefab Annotations: Extensible Pixel-Based Interpretation of Graphical Interfaces. *UIST '14* (2014), 221–230.

[12] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. 2010. Object Detection with Discriminatively Trained Part-Based Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32, 9 (Sept 2010), 1627–1645. https://doi.org/10.1109/TPAMI.2009.167

[13] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating Photo Manipulation Tutorials by Demonstration. *ACM Transactions on Graphics* 28, 3 (2009), 1.

[14] Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2010. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. *UIST '10* (2010), 143–152.

[15] Sumit Gulwani. 2016. Programming by Examples (and Its Applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press.

[16] R. W. Hamming. 1950. Error detecting and error correcting codes. *The Bell System Technical Journal* 29, 2 (April 1950), 147–160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

[17] Minh Hoai, Zhen-Zhong Lan, and Fernando De la Torre. 2011. Joint Segmentation and Classification of Human Actions in Video. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition* (2011), 3265–3272.

[18] Amy Hurst, Scott E Hudson, and Jennifer Mankoff. 2010. Automatically Identifying Targets Users Interact with During Real World Tasks. *IUI '10* (2010), 11–20.

[19] Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J. Brostow. 2017. Help, It Looks Confusing: GUI Task Automation Through Demonstration and Follow-up Questions. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces (IUI '17)*. ACM.

[20] Tessa Lau et al. 2008. Why PBD Systems Fail: Lessons Learned for Usable AI. In *CHI 2008 Workshop on Usable AI*.

[21] Tessa Lau, Lawrence Bergman, Vittorio Castelli, and Daniel Oblinger. 2004. Sheepdog: Learning Procedures for Technical Support. *IUI '04* (2004).

[22] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter : Automating & Sharing How-To Knowledge in the Enterprise. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2008), 1719–1728.

[23] John P Lewis. 1995. Fast normalized cross-correlation. In *Vision interface*, Vol. 10. 120–123.

[24] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration *(CHI '17)*.

[25] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2007), 943–946.

[26] Rodrigo de A. Maués and Simone Diniz Junqueira Barbosa. 2013. Keep Doing What I Just Did: Automating Smartphones by Demonstration. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services (MobileHCI '13)*. ACM, 295–303.

[27] Gordon W Paynter. 2000. Automating Iterative Tasks with Programming by Demonstration. (2000).

[28] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F. Cohen. 2011. Pause-and-Play: Automatically Linking Screencast Video Tutorials with Applications. *UIST '11* (2011), 135–144.

[29] Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. 2004. Are Loss Functions All the Same? *Neural Comput.* 16, 5 (May 2004), 1063–1076. https://doi.org/10.1162/089976604773135104

[30] Qinfeng Shi, Li Cheng, Li Wang, and Alex Smola. 2011. Human Action Segmentation and Recognition Using Discriminative Semi-Markov Models. *International Journal of Computer Vision* 93, 1 (2011), 22–32.

[31] Qinfeng Shi, Li Wang, Li Cheng, and Alex Smola. 2008. Discriminative Human Action Segmentation and Recognition Using Semi-Markov Model. *26th IEEE Conference on Computer Vision and Pattern Recognition* (2008).

[32] A. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* 13, 2 (April 1967), 260–269. https://doi.org/10.1109/TIT.1967.1054010

[33] Cheng-Yao Wang, Wei-Chen Chu, Hou-Ren Chen, Chun-Yen Hsu, and Mike Y Chen. 2014. EverTutor: Automatically Creating Interactive Guided Tutorials on Smartphones by User Demonstration. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2014), 4027–4036.

[34] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. *UIST '09* (2009), 183–192.

[35] Tom Yeh, Tsung-Hsiang Chang, Bo Xie, Greg Walsh, Ivan Watkins, Krist Wongsuphasawat, Man Huang, Larry S. Davis, and Benjamin B. Bederson. 2011. Creating Contextual Help for GUIs Using Screenshots. *UIST '11* (2011), 145.

[36] Luke S. Zettlemoyer, Robert St. Amant, and Martin S. Dulberg. 1999. IBOTS: Agent Control Through the User Interface. In *Proceedings of the 4th International Conference on Intelligent User Interfaces (IUI '99)*. ACM, New York, NY, USA, 31–37. https://doi.org/10.1145/291080.291087