RecurBot: Learn to Auto-complete GUI Tasks From Human Demonstrations

Thanapong Intharah
University College London
Gower Street, London, UK
t.intharah@ucl.ac.uk

Michael Firman
University College London
Gower Street, London, UK
m.firman@cs.ucl.ac.uk

Gabriel J. Brostow
University College London
Gower Street, London, UK
brostow@cs.ucl.ac.uk

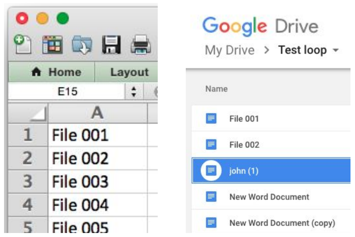
Acknowledgments: The authors were supported by the Royal Thai Government Scholarship, and NERC NE/P019013/1.

CHI'18 Extended Abstracts, April 21–26, 2018, Montreal, QC, Canada
ACM 978-1-4503-5621-3/18/04.
https://doi.org/10.1145/3170427.3188532

Abstract

On the surface, task-completion should be easy in graphical user interface (GUI) settings. In practice however, different actions look alike and applications run in operating-system silos. Our aim within GUI action recognition and prediction is to help the user, at least in completing the tedious tasks that are largely repetitive. We propose a method that learns from a few user-performed demonstrations, and then predicts and finally performs the remaining actions in the task. For example, a user can send customized SMS messages to the first three contacts in a school's spreadsheet of parents; then our system loops the process, iterating through the remaining parents.

First, our analysis system segments the demonstration into discrete loops, where each iteration usually included both intentional and accidental variations. Our technical innovation approach is a solution to the standing *motif-finding* optimization problem, but we also find visual patterns in those intentional variations. The second challenge is to predict subsequent GUI actions, extrapolating the patterns to allow our system to predict and perform the rest of a task. We validate our approach on a new database of GUI tasks, and show that our system usually (a) gleans what it needs from short user demonstrations, and (b) autocompletes tasks in diverse GUI situations.



	A
1	File 001
2	File 002
3	File 003
4	File 004
5	File 005

File 001
File 002
john (1)
New Word Document
New Word Document (copy)

Figure 1: A repetitive GUI task, where the user is renaming files on Google Drive to match names in Excel (a separate app). This type of task is long and tedious, and hard for a typical computer user to automate. Our system learns to complete such tasks by watching a user perform a few demonstration iterations.

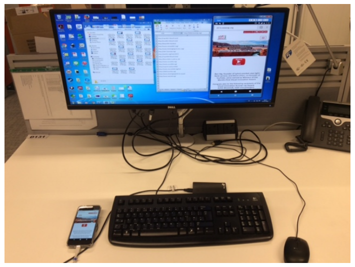


Figure 2: Mobile-phone testing: The task is to visit websites from a list, and to save screenshots of how they appeared on a mobile phone. The phone is connected to a PC using remote desktop software. GUI information is only shared through computer vision.

Author Keywords
Programming by Demonstration; Recurrent Action Recognition; GUI programming

ACM Classification Keywords
H.5.2. [Information Interfaces and Presentation (e.g. HCI)]: User Interface s- Graphical user interfaces (GUI); I.5.5. [Pattern Recognition]: Implementation - Interactive systems

Introduction
People who do *repetitive* tasks on their computer or mobile seemingly have a choice: they can either manually perform each loop until the task is complete, or they can program a macro to do the task for them. Except for short tasks, this is often a no-win situation in practice. Programming is not a universal skill, and most software applications (apps) must be controlled through a graphical user interface (GUI), rather than through an API or command-line hooks.

Software agents, known as bots, are still very far from fulfilling the seamless learning and skill-transfer dreams of Maes [9], Negroponte [11], and Kay [5]. We find that even computer-literate users over-estimate modern bot capabilities until they are asked to automate a repetitive task themselves. There are two main misconceptions. **Myth 1:** "Modern bots can see, but just lack good human interfaces." Bots can indeed grab screenshots, and they can attempt local Optical Character Recognition. But they can not systematically understand the GUI elements in terms of grouping widgets [3] or identifying interactive buttons. The diversity in mobile app GUIs makes this harder than ever. Without training data, scene-understanding of GUIs is not especially easier than scene-understanding of satellite images. **Myth 2:** "The bot can just ask the operating system (OS), skipping computer vision of the GUI." Even Open Source OS's like Android restrict developers (within an app) to the narrow parameters of accessibility API's, and those cannot retrieve the complete visual information of a GUI [7].

Our proposed system addresses a desktop version of the Programming by Demonstration problem. It lies at the intersection of intention-inference, software usability, and action recognition and prediction. It lets a user teach a bot to perform a repetitive task, much like they'd teach a human. Consider the examples shown in Figures 1 and 2. In the first example, the user is renaming each file in Google Drive to match a list of names given in an Excel spreadsheet. This type of recurrent task is common to most computer users, and it is only the experts who have access to the scripting tools required to automate them. Our algorithm takes snapshots and the user's mouse/keyboard events as inputs, and from that initial user-demonstration of the task, it segments and extrapolates what was different about each loop, to complete the task automatically. Like Microsoft Excel's AutoFill, users want to extrapolate from these first few inputs, rather than cloning them. Aptly, Excel's FlashFill is touted as an important milestone [2] for practical inductive programming, because it extrapolates non-sequential patterns, *e.g.,* parsing of initials from people's names. Unlike Microsoft's apps, our input comes from many diverse apps, bitmaps of heterogeneous content, and *noisy time-series human demonstrations,* where order matters.

Recurrent Actions in User Input
We strive to recover and predict a user's intended loops by analyzing the data from their initial demonstration. We take as input a sequence of basic actions, $\mathbf{A} = (A_0, A_1, \ldots A_N)$, scraped using [4]. Figure 3 shows the input sequence for a simple example task, where the user had performed click actions in different locations to sequentially delete one type of file from a folder. Each basic action A_n is a tuple containing the *action type* a_n, together

Dataset: We used 15 GUI tasks to evaluate the system's ability to save users' time. Examples are shown in the video. One example task is visiting each "person" on a web-page of experts, where each one has a photo and a text name that links to that individual's homepage. Our system must detect the different photos that have nothing in common, localize the text, and then print the resulting homepage, before coming back to the start page and repeating the loop. Another example task has a user creating a list of local files based on a folder visible in a remote desktop.

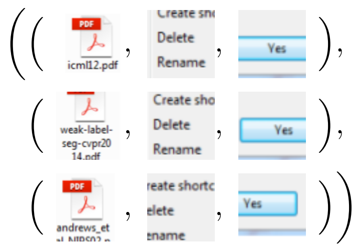


Figure 4: Expected result ($\mathbf{R}$) of an example input from Figure 3.

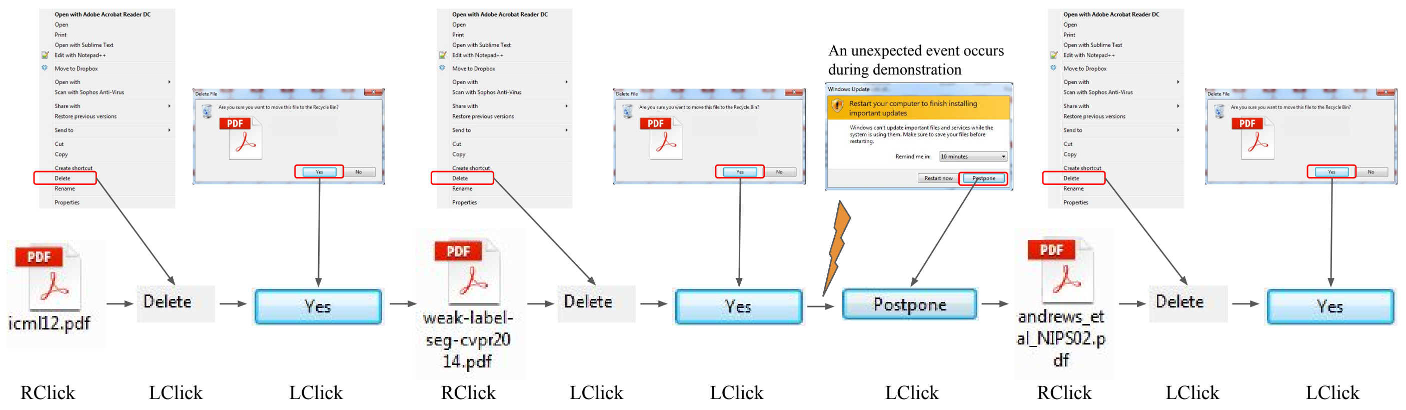


Figure 3: An example problem setup. This diagram shows that our system allows detour actions, which can accidentally happen during the demonstration process, without the user needing to re-record everything again from scratch. Here, the Windows pop-up asks to restart the system, and the user dismisses this dialog with a mouse click. Our system only has access to whole-screen screenshots and corresponding keystrokes and mouse pointer locations, and uses these to infer the intended recurrent behavior.

with, where appropriate, a screenshot and mouse cursor location. The action types obtained by [4] are: `LeftClick`, `RightClick`, `DoubleClick`, `ClickDrag`, and `Typing`.

We assume that there is a sequence of actions $\mathbf{T}$ which, in the user's mind, is the 'true' sequence. Template $\mathbf{T}$ contains the ground truth sequence of events that the user wishes to be repeated to complete their long chain of iterative tasks. If, in demonstrating the task, the user just perfectly performed $\mathbf{T}$ once as in [4], the problem would *still* be difficult, as:

1. The computer vision system seeks to find visual similarity between changing elements, and if the loop is only performed once, then we only have a single training example for predicting each future iteration.
2. The prediction of future actions relies on knowing about *iterative changes* between user actions in different loops. For example, loop two might require a click below the corresponding click in the first loop. This cannot be learned from a single loop.
3. In reality, even when a user tries to complete a sequence correctly, they typically deviate from the true sequence through no fault of their own. Multiple repetitions help us to discover the 'true' intention of the user.

We therefore ask that the user performs several loops. We then use our *motif-finding algorithm* to recover a set of divided subsequences, given $\mathbf{A}$. The only assumption we make is that our full sequence $\mathbf{A}$ contains at least one 'good enough' sequence $\tilde{\mathbf{T}}$ which is functionally equivalent to the true sequence $\mathbf{T}$, despite having some minor additions or deviations from the ideal. Other instantiations

Contribution overview: Our main contribution is an algorithm for programming by demonstration of recurrent GUI tasks. Our visual motif analysis overcomes three main challenges: (1) The user-demonstrated loops are non-identical. The community working on set-based Motif-finding has avoided visual problems, and seeks to identify a perfect subsequence that was repeated K times, given N actions and known K. (2) The demonstrated actions are typically iterating through the initial loops of a lengthy task. To automatically predict and execute the remaining loops, we must detect the one or more iterators implied by the demonstration. (3) Compared to experimental validation of passive action-recognition in computer vision, recognizing and then *predicting* GUI tasks requires a video dataset annotated with acceptable-interaction meta-information.

of $\mathbf{T}$ in $\mathbf{A}$ may be noisier, with extra, unwanted actions being performed, or missing actions. Our motif-finding algorithm is usually able to deal with these issues.

Recurrent Action Recognition

Our algorithm extends exact time series motif discovery [1, 10] by jointly examining a *set* of motifs instead of just a single pair. We assume that the user has provided K, the number of times they have performed the loop. The algorithm starts building a list of candidates $\mathcal{C}$ by forming K-tuples of similar single actions which have distances between all pairs of members smaller than a threshold. Each K-tuple of single actions is then extended into K-tuple of two-actions subsequences by adding subsequent actions from $\mathbf{A}$. The process continues until there are no more actions to add to each subsequence. The algorithm finally returns the set of motifs $\mathbf{R}$ which is the candidate with the lowest average distance from the set $\mathcal{C}$.

Artificial Subsequences for Robustness

Our algorithm, so far, has assumed that each version of $\mathbf{R}$ contained in $\mathbf{A}$ is a perfect, unmodified copy. However, as we have discussed, many users will miss out actions or include extra unneeded actions. The user variations can be categorized into three classes: (a) Missing actions, where the user omits a single step; (b) Noisy actions between two subsequences, and (c) Noisy actions within a subsequence. Figure 3 shows an example of a noisy action within a subsequence, where the user has dismissed a system dialogue with a click while demonstrating the loops.

To cope with the user variations, we generate *artificial subsequences* during our solving process. We extend subsequences in $\mathcal{C}$ with copies of items within $\mathcal{C}$, each of which has some user actions removed, or extra ones appended. These simulate user variations help to improve the matching between noisy input subsequences. Distance measures computed from or to any of these generated subsequences have an additional penalty added for each each appended or skipped action.

As a result of our Recurrent Action Recognition algorithm, the example input from Figure 3 will be mapped to the output in Figure 4. Note that our system automatically identified and segmented the three separate loops, and then aligned the equivalent actions in each loop. The extra action (of dismissing the popup window) has been correctly identified as noise, and is therefore not shown in this final result. An ability to deal with user variations gives our algorithm much greater scope for use compared to Familiar [12].

Prediction of Future Actions

We use the discovered sets of loops to predict the user's intended actions. Our best discovered set of subsequences $\mathbf{R}$ effectively forms a training set to enable this inference. As the user confirms or corrects the system at run-time, the training set will grow. See the Human-in-the-loop sidebar.

We find, for each discovered action A_i, the set of corresponding actions from each of the K subsequences in $\mathbf{R}$. This gives us up to K cropped *training images* for each action in the loop. We denote this set of crops associated with action A_i as $\mathcal{H}_i$. Our system iteratively plays back each action A_i at time t at a predicted screen location (x^*, y^*), computed using Bayes' Theorem as

$$\begin{aligned} x^*, y^* &= \arg\max_{x,y} P(x, y \,|\, \mathcal{H}_i, I) \\ &= \arg\max_{x,y} P(\mathcal{H}_i, I \,|\, x, y) P(x, y), \end{aligned} \tag{1}$$

where I is the current screenshot, $P(\mathcal{H}_i, I \,|\, x, y)$ is the visual term, computed by normalized cross-correlation, and

Human-in-the-loop
A good Programming by Demonstration (PbD) system should let users know the next action that the system is going to execute, and allow users to approve or, if necessary, modify the action [6]. After predicting the most likely action location x^*, y^* at time step t, our system shows the user an animation of the proposed action and asks them to approve or correct the action. The user's response is added to the information 'bucket' for this action, used for refining the prediction model for the action at the next iteration. This means that, as our system interacts with the human, it learns more about the task and improves its predictions.

$P(x, y)$ is the location prior, inferred by maximum likelihood estimation. This strategy allows our algorithm to deal with loop variations as described in Figure 6.

Validation of the Algorithm
We first measure our system's ability to recognize recurrent actions in the user's demonstration loops, using our **Demonstration Dataset**. The dataset comprises 55 GUI tasks. Each sequence was recorded by asking 7 experienced computer users to perform the first four or so loops of specific repetitive GUI tasks. While working and with their knowledge, they were recorded by sniffer-software that captured both mouse/key events, and screenshots throughout each task. The mouse/key events in this dataset, and all sniffer-events observed at test-time, are converted into basic actions thanks to [4], and serve as inputs to our system.

We then annotated each task's action-transcript, identifying the boundaries between loops, and tagging the parts of each loop that included either extra actions or were missing actions, as compared to the other loops in the task. On average, each loop in this dataset has 4.33 actions. A test user performed the first 3 or 4 loops of each task, and these were labeled to quantify performance. Because real users are not perfect, loops within the same sequence naturally differ from each other by having extra, missing, or iteratively changing actions. On average, 65% of sequences have noisy actions and 29% have missing actions.

Our recurrent action recognition algorithm is able to produce a ranked list of possible answers, sorted by the average value of normalized distances between every pair of motifs inside the answer. We count up the fraction of tasks where the correct answer is within the top k answers returned by the algorithm, and plot this success rate against

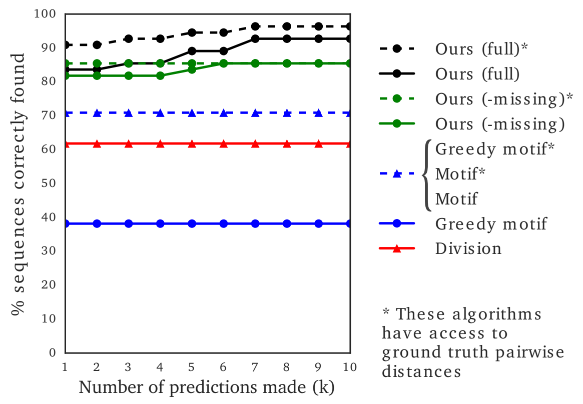


Figure 5: Quantitative results of our recurrent action recognition algorithm on the Demonstration Dataset. Algorithms with an asterisk, also shown with dashed lines, have access to the ground truth distance measures. We compare our algorithm to three baselines: *Division*, *Greedy motif* and *Motif*. *Greedy motif* and *Motif* achieve the same performance so are shown as one line here. *Ours (-missing)* is our ablation study, showing the degraded result when we remove the artificially appended actions.

different values of k. Figure 5 demonstrates the performance of our full algorithm compared to three baselines: *Division*, *Motif*, and *Greedy motif*.

Usability Testing
We conducted a user study to evaluate the usability of our prototype system. Ten users, who are white-collar workers, participated in this study. The group consisted of 7 females and 3 males, their ages varying between 24 to 40. One of the participants works as a programmer, and the rest have little to no programming skills (the average score of ten participants on the question "I can do programming" is 2.4 on 7-point Likert scale). All of the participants reported that they use computers in their daily work.

Category	Score / 7
Usefulness	5.65 ± 0.36
Ease of use	5.51 ± 0.30
Ease of learning	6.05 ± 0.06
User satisfaction	5.86 ± 0.14
Overall average	5.70 ± 0.32

Table 1: Results of the user study on our full system. Each question was scored on a Likert scale out of a maximum of 7 (strongly agree).

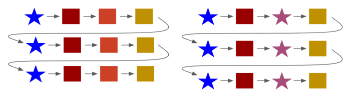


Figure 6: Loop Variations: Some actions have the same visual appearance on each loop – we show these here as squares. This might include clicking the browser's 'back' button. Other actions change appearance, and even location, on each loop. These are shown here as stars, to indicate intentional variations. Our system can correctly predict for both of these scenarios, and when actions are missing or added, while HILC [4] can only deal with the scenario where there is only a single star at the start of the loop, and that example must be perfect.

The participants were introduced to our system and each of them was asked to complete two recurrent tasks in rounds: the first round has them complete a task manually (without our system), while they complete the task using our system in the second round. The tasks we set the users were "creating list of filenames from the files in a folder", and "creating slides of images from a folder of images". After completing the tasks, participants were asked to fill out the USE questionnaire [8], which measures usefulness, ease of use, ease of learning, and satisfaction aspects of the system. Each item was rated on a 7-point Likert scale, with values from "Strongly Disagree:1" to "Strongly Agree:7".

The overall averaged score of the system was 5.70, and the breakdown of scores across categories is shown in Table 1. Participants stated in the open ended comments, that the system reduced tedium in completing recurrent work, is easy to learn and use, and gains users' trust by asking the users when a prediction is uncertain. Users expressed that they would like the initial analysis to run faster, and for the pattern matching to be robust (*e.g.,* to occlusions).

Conclusions and Future Work

We have shown how to recognize repeated actions in hybrid visual-sniffer data, when a user interacts with one or more GUIs. The key has been to extend existing motif-finding algorithms to deal with sets, and to cope with noisy and real user interaction data. Beyond parsing a user's GUI demonstration, we were able to make an interactive prediction and correction system that users can guide.

In future work, we need to detect and control screen-scrolling, and parse text. Even without these enhancements, our system demonstrates a new visual-interaction problem within programming by demonstration. That many repetitive GUI-based tasks might be automatable for non-programmers is significant, and could be especially helpful for building interfaces for motor-impaired and hands-free computer users.

REFERENCES

1. A. Bagnall, J. Hills, and J. Lines. 2014. Technical Report CMPC14-03: Finding Motif Sets in Time Series. (2014).
2. S. Gulwani et al. 2015. Inductive programming meets the real world. *Commun. ACM* (2015).
3. A. Hurst et al. 2010. Automatically Identifying Targets Users Interact with During Real World Tasks *(IUI)*.
4. T. Intharah, D. Turmukhambetov, and G. J. Brostow. 2017. Help, It Looks Confusing: GUI Task Automation Through Demonstration and Follow-up Questions *(IUI)*.
5. Alan C. Kay. 1984. Computer Software. *Scientific American* (1984).
6. T. Lau. 2008. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*.
7. T. JJ Li, A. Azaria, and B. A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration *(CHI)*.
8. A. M. Lund. 2001. Measuring Usability with the USE Questionnaire. *Usability interface* 8, 2 (2001), 3–6.
9. P. Maes. 1994. Agents that reduce work and information overload. *Commun. ACM* (1994).
10. A. Mueen, E. Keogh, Q. Zhu, S. S. Cash, and M. B. Westover. 2009. Exact discovery of time series motifs. In *(SDM)*.
11. N. Negroponte. 1970. *The Architecture Machine*. MIT.
12. G W Paynter. 2000. *Automating iterative tasks with programming by demonstration*. Ph.D. Dissertation.