Voicify: User Demonstrated Voice Interfaces

Adam Vogel
Computer Science Department
Stanford University
av@cs.stanford.edu

Arda Kara
Computer Science Department
Stanford University
ardakara@cs.stanford.edu

ABSTRACT
We present *Voicify*, a framework for voice programming by demonstration which enables users to build their own speech interfaces. The user creates their own voice interface by demonstrating a voice command with the corresponding actions to take on the phone. The phone can then be put in a hands-free mode, where it responds to previously programmed voice commands. Transparency is a key issue, and we discuss the development of a voice-only keyboard for text entry paired with a custom screen reader for giving feedback to the user. We conducted a user study which shows that interfaces created using Voicify rival their touch counterparts. However, our study also shows that most users find programming by demonstration to be tedious, and that the natural language processing is too impoverished to be widely applicable.

Introduction
Speech interfaces have been gaining in popularity with the proliferation of mobile devices. Smartphones are frequently used in hands-free or eyes-free contexts, such as while driving or walking. Modern phones like the iPhone and Android phones include speech recognition and synthesis in the operating system, allowing users to make calls and send text messages by voice. However, there is a long tail of community authored phone applications which have no voice interface. This is a big restriction for the convenience of hands-free users and the sight disabled.

To address this problem, we introduce *Voicify*[1], a framework for creating speech interfaces through end-user demonstration. Voicify allows a user to create voice commands for Android applications through a "tell and show" interface: the user first *tells* the phone what to do in natural language and then *shows* the phone how to interpret this command. Once the user has demonstrated commands, the phone can execute commands in playback mode.

We further explore the efficacy of user demonstration and the usability of the resulting interface by conducting a user study. We *voicified* a popular open-source Android to-do list application called *Astrid* [1]. Using this version of Astrid, we tested the ease and competence of the demonstration and playback interfaces when compared to the traditional touch interface. Our results show that the voicified interfaces are usable, with a 60% command interpretation success rate, but still fall below the level needed for wide usage. Our user study revealed several weaknesses in the demonstration interaction that we implemented.

The remainder of the paper summarizes the large body of work on programming by demonstration, details the design and implementation of Voicify, and discusses the user study.

Related Work
Programming by demonstration has long been popular in the robotics community. Early work includes the Robot Programming System (RPS) of SRI [8] and Grossman [6], who taught robotic manipulators by manually controlling them. More recent machine learning approaches to programming by demonstration go by the name *apprenticeship learning*. Abeel et al. [2] used human demonstrations of helicopter maneuvers to automatically learn difficult flying tricks. These approaches treat the demonstrator as an agent acting in a reinforcement learning model and the system tries to infer the latent utility function the agent is acting to maximize.

One of the difficulties of programming by demonstration is handling errors in the demonstration. Chen and Weld [5] offer methods for detecting inconsistent demonstrations, where the same command yields seemingly opposite goals. They then use this detection to alert the user with disambiguating feedback. Further, they use a probabilistic framework to measure the consistency of hypotheses, treating uncertain parts of demonstrations as missing values.

Programming by demonstration has been studied in the user interface community as well. Bocionek and Sassin [3] improved room reservation systems by observing the emails a receptionist received and the corresponding actions he or she took to reserve a room. They specifically dealt with the natural language problem, and utilized a thesaurus to improve coverage.

In the natural language processing community there is an interest in learning to follow directions. Recent work which is applicable here is Branavan et al. [4] who learned to interpret help guides for the Windows operating system.

Lieberman [7] took programming by demonstration more literally, where users give examples of what they would like to

[1]The Voicify source code is available for download at `https://github.com/acvogel/astrid` and a demo video is available at `http://nlp.stanford.edu/acvogel/voicify.mov`.

do and the computer attempts to generate code to achieve this goal. The user then interacts with the generated code, in the hopes of making programming easier for beginners. In our case the user never modifies actual programming languages, but the principles involved are similar.

Programming by Demonstration

The Voicify framework requires three main components: the ability to capture low-level UI events, an interface for recording demonstrations, and lastly the ability to playback commands for interpretation.

UI Capture

To record a command demonstration, Voicify requires access to the user interface actions that a user engages in. Security restrictions on the Android platform make it impossible for a 3rd party application without system privileges to capture touch and key events from a different application as this would enable malicious key loggers to steal sensitive information. The Android Accessibility Team [9] confirmed that this is impossible to do without modifying the operating system and issuing a system certificate. To circumvent this problem, we chose an open-source application as a prototype, which allowed us to modify the program source itself, thus capturing UI events within the application.

The UI design of Android still makes capturing UI events across a whole application a difficult task. Within each screen, the interface elements, called *views*, are arranged in a tree structure. When the user touches the screen, the UI event is dispatched to the lowest element in the view tree which spans that portion of the screen. It is then passed up the view tree until a UI element consumes it. To capture these touch events, we programmatically traverse the view tree and insert an invisible UI layer at each node. The UI layer simply captures the UI interactions which pass through it.

Furthermore, some UI widgets like buttons and dialog boxes consume touch events without passing them along to the activity in which they reside. Complicating matters, these boxes are required to be leaves of the view tree, which doesn't allow for our trick of inserting invisible layers. To deal with these issues we overrode the UI widget classes to capture the interactions. This required changing the UI definitions for each of the screens to use our modified versions. This challenges our goal of making Voicify applicable to every application but is a necessary hack for this prototype.

Lastly, each screen in an Android application is run in its own process, which pauses when switching between screens in an application. This makes it difficult to share UI events across activities, which naturally happens when demonstrating some commands. To overcome this we implemented a *demonstration service* which runs in the background of the application and aggregates UI interactions across activities.

When recording UI events, we capture low-level touches. The information in touch events includes the (X,Y) coordinate on the screen, the pressure and size, and lastly whether it is initial contact, a dragging motion, or a release. Our various capturing shims transmit these events to the demonstration service, which associates them with the correct command.

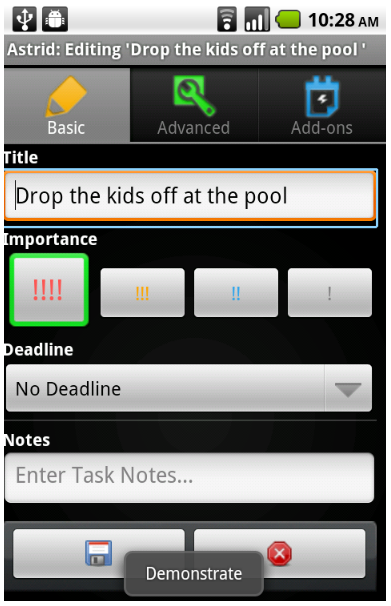
10:28 AM
Astrid: Editing 'Drop the kids off at the pool '
Basic | Advanced | Add-ons
Title
Drop the kids off at the pool
Importance
!!!! | !!! | !! | !
Deadline
No Deadline
Notes
Enter Task Notes...
Demonstrate

Figure 1: When the user presses the camera button a small UI notification appears to let them know to being demonstration.

Demonstration Interface

We now describe the user demonstration interface. We modified the hard camera button on the phone to serve as the "Voicify button". After loading the Astrid application, the user presses the camera button. This pops up a small notification which says "Demonstrate" (see Figure 1). From here the user says the command that they would like to record, for instance "create a new task". After running Google-powered speech recognition, the interface speaks back to the user what it thinks the command is, such as "recording command 'create a new task' ". The phone is now in capture mode, where all subsequent UI events are associated with this command. After executing the desired sequence of interactions, the user presses the camera button again to signal the end of demonstration, which triggers a small pop-up window to acknowledge the demonstration.

We use the hard camera-button to solve the *clutching problem*, which is how to determine when a trial has ended. We also experimented with a soft button in a context menu within the application, but users found it cumbersome to locate, whereas the hard camera button is able to be pressed by feeling alone. A better demonstration interface would be to always capture the audio and only record when the user says a key phrase. However, the current state of speech recognition is not good enough to enable this.

Playback Interface

The Android operating system employs a sandbox-based security framework that prevents any interaction between applications. Each application has its own virtual machine, which makes it difficult to implement a system-wide voice control interface without modifying the OS. Our workaround was to use the UI testing features of the Android SDK unit testing environment. One downside of this approach is that the phone needs to be plugged into the debugging environment during testing; however, this didn't come up as a concern from our test users. The Astrid application was modified to receive a voice command upon a button press, look-up the closest matching voice command that it has in its database, and send the necessary UI interactions to the test suite using IPC. Our system for receiving commands used a word overlap index to find the closest command in its database. Just like the demonstration interface, the user would press a button and then utter the voice command they want to use. The phone then echoes the command that it thinks it received and carries out the UI interactions that were previously demonstrated.

When using Voicify in an eyes-free environment, the application needs to read information to the user in addition to interpreting commands. Android has a built-in screen reader tool called TalkBack that can be turned on as a system-wide setting and allows the user to get voice feedback from their interactions by traversing through UI elements using the d-pad on their phones (if available on the model). Our system did not use TalkBack, as it required the phone's accessibility mode to be turned on which intervened with custom accessibility objects that our code uses. Instead, we implemented a replacement that dispatches d-pad commands to traverse through visible UI elements, and simulates accessibility events to gather readable content at each step. This worked well for a uniform screen like the list of to-do tasks, but got in the way of user interaction in more complicated screens due to the simulated traversal. Therefore we only used the screen reader to read off the list of tasks.

We also wanted the voicified interface to be able to accept new input for text fields. For example, if the user demonstrated a voice command for creating a new to-do task, the title for the task should be customizable during playback. To achieve this, we implemented a voice keyboard for Android that is available across applications. Once the voice keyboard is selected as the system-wide keyboard, the phone asks you to speak, instead of displaying the keyboard, anytime an editable field is selected. This meant that any demonstration that ends with a text input field selected would know to prompt the user to speak again after the voice command and enter the transcribed text into the text field. Obviously, a limitation of this is that it did not allow demonstrations to accept input in between touch commands. Any demonstration that accepts input needed to do so as the very last step, which some users found confusing.

Method

We conducted a lab user study two questions:

1. Do users find our demonstration interface natural to use?
2. Are voice interfaces created by demonstration capable of achieving the same features as touch interfaces?

To do so, we recruited 10 members of Stanford University to participate in a user study. We focused on a representative set of seven tasks in the Astrid to-do app:

1. Create a new task
2. Add a title
3. Set the task priority to high
4. Add a note
5. Toggle the reminder
6. Delete a task
7. Save a task

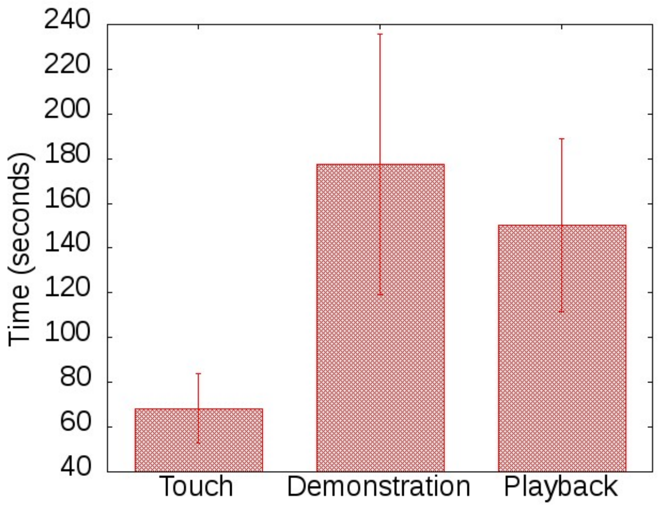

Figure 2: The amount of time each of the seven tasks took for each user interface. The speech interfaces are significantly slower than the touch interface.

The participants began by experimenting with the Astrid application for a few minutes to familiarize themselves with its layout. From here the participants were given the list of commands, written out in the above form, and asked to complete them using the touch interface. This served two purposes: to provide a baseline of the amount of time the tasks take, and to further familiarize the users with the parts of the interface they will be demonstrating. Here we recorded the UI interactions of the user to verify task completeness and recorded the amount of time it took.

Next the experimenters explained the demonstration interface and programmed a sample command to show the user how it works. The users were then instructed to program voice commands to complete the tasks. The printed instructions were still available to users, but surprisingly few directly read the printed instructions. When speech recognition errors occurred, some users re-recorded the command, and others simply moved on. The quality of speech recognition and demonstration in this phase is critical for creating a good interface. However, not all errors are fatal. Sometimes the transcription errors are systematic, meaning that if the user says the same thing at playback the command will still work, despite the fact that the speech is misinterpreted.

After completing the demonstration phase, the phone was put into playback mode by starting the UI test environment on the connected laptop. The users then did the seven tasks once again, and the success of each completing each step was recorded. Users were instructed that if they could not complete a task after several attempts to continue to the next one.

Results

Figure 2 shows the amount of time users spent in each mode. As expected, users completed the tasks in less time using the touch interface. The demonstration period took the longest, as users had to both state their actions and execute them in the UI. Playback took slightly less time due to the phone automatically executing touch actions. We found a relatively wide range of time spent by participants. This can be attributed to systematic differences in speech recognition quality across users.

In playback mode, users successfully completed tasks 58.6% of the time, with a standard deviation of 23.6%. This is a relatively high variation across users. Several users were able to complete all but one of the tasks successfully, and several completed almost none.

Discussion

Errors in playback mode can be grouped into several classes:

- Speech recognition errors: sometimes the system mis-interprets the command given.
- Demonstration mistakes: some users incorrectly demonstrated how to carry out commands.
- Cascading errors: the system makes an error in the middle of a larger task. The user can become uncertain of the state of the application, causing an incorrect command to be played.

The errors in speech recognition are difficult to deal with. By echoing the user command both at demonstration and playback time we hoped to provide *transparency* to the user. However, even if users are aware that the system makes an interpretation error, it can be difficult to recover from without appropriate commands. Some users found that the interface was overly verbose - listening to the system read back commands can be tedious. Possible approaches to this problem follow trends in normal dialog-systems, which frequently use the confidence of speech recognition to decide when to ground user commands [10].

Other errors were caused by the brittleness of our demonstration representation. Low-level touch events are highly context sensitive and do not generalize across interfaces. For instance, correctly pressing the 'save' button on one screen can differ from another. Furthermore, we did not represent the screen on which a command was demonstrated, thus it sometimes executes commands which are not applicable to a given UI state. A first step in this direction would be to represent the specific UI element that was manipulated instead of just recording (x, y) coordinates. We could further leverage cues like the labels on buttons or dialog boxes to aid command interpretation, similar to [4].

Conclusion

We presented *Voicify*, a demonstration framework for mobile Android applications. Our user study provided several key insights into successful programming by demonstration. Users require an abstract representation of instructions, above the UI level. Demonstrations should take into account program context and the overall goals of the user. Our study showed that speech recognition quality is critical to good performance, but that existing technology is almost good enough. Despite the fact that the speech interfaces learned through demonstration are not yet at the level of their traditional touch counterparts, this work shows a promising route towards providing wide-coverage voice interfaces to mobile applications that currently lack them.

REFERENCES

1. Astrid homepage. `http://weloveastrid.com`.

2. Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, ICML '04, pages 1–, New York, NY, USA, 2004. ACM.

3. Siegfried Bocionek and Michael Sassin. Dialog-based learning (dbl) for adaptive interface agents and programming-by-demonstration systems. Technical report, Pittsburgh, PA, USA, 1993.

4. S. R. K. Branavan, Harr Chen, Luke S. Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *ACL/AFNLP*, pages 82–90. The Association for Computer Linguistics, 2009.

5. Jiun-Hung Chen and Daniel S. Weld. Recovering from errors during programming by demonstration. In *Proceedings of the 13th international conference on Intelligent user interfaces*, IUI '08, pages 159–168, New York, NY, USA, 2008. ACM.

6. D.D. Grossman. Programming a computer controlled manipulator by guiding through the motions. Technical report, 1977.

7. H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

8. W.T. Park. The SRI robot programming system (RPS). In *13th ISIR*, pages 12–21, 1983.

9. Alan Viverette. Personal Communication, 2011.

10. Steve Young, Milica Gasic, Simon Keizer, François Mairesse, Jost Schatzmann, Blaise Thomson, and Kai Yu. The hidden information state model: A practical framework for pomdp-based spoken dialogue management. *Computer Speech & Language*, 24(2):150–174, 2010.