Programming Self-Optimizing Workflows for Crowdsourcing – A Status Report

Christopher H. Lin Mausam Daniel S. Weld
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{chrislin,mausam,weld}@cs.washington.edu

August 15, 2013

Crowdsourcing requesters are trapped between a rock and a hard place. Typically they specify their crowdsourcing workflows procedurally, but current languages commit them to overly strict and static policies that waste human effort. While optimizing workflows with more sophisticated artificial intelligence tools can significantly reduce labor costs [1, 2], such techniques are hard to use and understand. We present CLOWDER, a system that allows users to easily procedurally program self-optimizing workflows for crowdsourcing.

CLOWDER provides an adaptive programming language (extending [5, 4] to handle partial observability and non-expert usability) that abstracts over and compiles into a partially observable Markov decision process. For instance, suppose a requester would like to write a dynamic workflow that uses crowdsourcing to label data. Specifically, the requester has a set of questions, possible answer choices, and a budget. We envision a language that allows the requester to write the program in Figure 1. The program first initializes a couple of arrays to count the number of votes for each answer choice given by the crowd. Then, while the budget has not been exhausted, the variable i is set to be one of several choices. If i is -1, the program terminates and returns the answers with the most votes. Otherwise, the program calls `crowd-vote`, an API call to some crowdsourcing platform that hires a worker to provide a label for question i.

While current methods can only allow users to program static policies (*e.g.*, ask 2 workers, and then ask a third to break ties), the `choose` functionality of CLOWDER enables intelligent and adaptive use of the budget. At run-time, CLOWDER will dynamically pick the best choice of the variable i. For instance, CLOWDER may decide that given the current history, question number 2 needs more input from voters, because the crowd has not been agreeing on the correct answer. Or perhaps at some point, CLOWDER will decide that question 9 is far too difficult, and will no longer expend any of its budget in obtaining labels for that question. CLOWDER can do this optimization using a single algorithm. In other words, given *any* program the user writes, CLOWDER automatically determines the best choices at runtime. Figure 2 shows another example of

```
// returns a list of the answers with the most votes
def vote(questions, answers0, answers1, budget):
  counts0 = [0,...,0], counts1 = [0,...,0]
  while (budget > 0):
    i = choose([-1, 0,...,|questions|])
    if i == -1:  break
    if crowd-vote(questions[i], answers0[i], answers1[i]):
       counts0[i] += 1
    else:  counts1[i] += 1
    budget -= 1
  return getBest(answers0, answers1, counts0, counts1)
```

Figure 1: Binary Labeling

a common workflow written in our language. It is a program that a user might write that uses iterative-improvement [3] to crowdsource a caption for an image.

CLOWDER works by relying on experts to define probabilistic models for primitive API calls like `crowd-vote` and `c-imp` as well as modules to elicit goals and utilities from users. Using a crowdsourced library of basic functions, CLOWDER alows end-users to optimize their crowdsourcing programs. We have currently implemented a first version of CLOWDER, which uses a Lisp-like language, for ease of interpretation. More details on related projects about decision-theoretic control of workflows can be found at the authors' webpages or at http://www.cs.washington.edu/node/3528/.

References

[1] Peng Dai, Mausam, and Daniel S. Weld. Decision-theoretic control of crowd-sourced workflows. In *AAAI*, 2010.

[2] Ece Kamar, Severin Hacker, and Eric Horvitz. Combining human and machine intelligence in large-scale crowdsourcing. In *AAMAS*, 2012.

[3] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *KDD-HCOMP*, pages 29–30, 2009.

[4] David McAllester. Bellman equations for stochastic programs, 1999.

[5] Jervis Pinto, Alan Fern, Tim Bauer, and Martin Erwig. Robust learning for adaptive programs by leveraging program structure. In *ICMLA*, 2010.

```
def iterative-improvement(image, budget):
  better-text = '', worse-text = ''
  while (budget > 0):
    i = choose([0,1,2])
    case i == 0:  //improve
      worse-text = better-text
      better-text = c-imp(image, better-text) //API Call
      budget -= 5
    case i == 1:  //vote
      if vote([image], [better-text],
          [worse-text], budget)[0] == worse-text:
          temp = better-text
          better-text = worse-text
          worse-text = temp
    case i == 2:
      break
  return better-text
```

Figure 2: Iterative-Improvement