Human Computation
AAAI Technical Report WS-12-08

Using the Crowd to Do Natural Language Programming

Mehdi Manshadi, Carolyn Keenan and **James Allen**
Department of Computer Science
University of Rochester
{mehdih, ckeenan, james} cs.rochester.edu

Abstract

Natural language programming has proven to be a very challenging task. We present a novel idea which suggests using crowdsourcing to do natural language programming. Our approach asks non-expert workers to provide input/output examples for a task defined in natural language form. We then use a Programming by Example system to induce the intended program from the input/output examples. Our early results are promising, encouraging further research in this area.

1 Natural Language Programming

Deep understanding of natural language sentences is a challenging task. After around half a century work in this area, we don't know of a single robust deep understanding system except in extremely narrow domains (Liang, Jordan, and Klein 2011). When it comes to describing programming tasks, deep understanding becomes even more challenging, as no vagueness/ambiguity could be tolerated. The challenge is so real that in a decade when there was a lot of optimism about the future of natural language understanding, Dijkstra (1979) called the idea of natural language programming "foolish". Three decades later, with growth of programming forums, natural language programming has become very popular especially for the novice programmers. Hundreds of post are created every day describing one's intended task and asking for a piece of code in a specific programming language accomplishing the task. Some companies such as Microsoft hire many programming experts to address the programming needs of the users of their products such as Microsoft Excel (Gulwani 2011), often defined in natural language form (possibly accompanied by one or more examples of the input/output). As a result, natural language programming is becoming a real issue, which needs to be addressed in one way or another. Following the successful deployment of crowdsourcing to replace/help the AI systems, returning answers in nearly real-time (Bigham et al. 2010), we are proposing a novel idea involving crowdsourcing to approach the problem of natural language programming while avoiding deep language understanding.

2 Programming by Example

In the core of our model lies a Programming by Example (PbE) engine. PbE (a.k.a. Programming by Demonstration or PbD) is one of the most studied topics in the realm of automatic programming (Cypher et al. 1993). The idea of PbE is to provide a system with many examples of input/output of the programming task, and try to induce a program only by looking by analyzing examples. For example, consider the following three input/output examples for a particular task.

Input	Output
exe-file	exe
alpha-betas	alpha
orange-mango	orange

The intended task would probably be:

Print the letters before the dash.

The PbE system only looks at the input/output examples and tries to guess what the intended task is. It turns out that often people are aware of the ambiguities the natural language description may carry, so often they provide an example of input/output to resolve the ambiguities. With only one example, however, there is a huge space of solutions that would match the input/output. Every new input/output example, could drastically reduce the space of solutions. For the above task, the first example alone could come from programs that only print "exe", or that only print the first three characters, or only those that remove the last six characters. The second example doesn't fit any of those, so it reduces the number of potential programs. Theoretically speaking, an infinite number of input/outputs is required to unambiguously define a task. But in practice by first, limiting the power of the underlying language (Gulwani 2011) and second, making an intelligent guess to find the most probable intended task within the space of solutions (Liang, Jordan, and Klein 2010) it is possible to converge to the intended task with a limited number of examples.

3 Our proposal

Given the description of a natural language task, we offer to use crowdsourcing to provide a set of input/output examples, and to feed the examples into a Programing by Example system in order to induce a program. While providing programs

by human agents requires expertise and is expensive, providing examples of input/output for a programing task given its description in natural language form is trivial and could be done by regular non-expert workers, very cheaply. Note that providing rich (defined shortly) examples is a painful job, so often it is not in the interest of the users themselves to provide several examples of input/output. Most users prefer to give the natural language description of the task (possibly with one or two examples to clarify their purpose).

4 Experiments

In order to investigate whether our proposal is feasible in practice, we ran some experiments in a specific domain. Within the area of PbD/PbE, there has been a lot of work on automating text-editing tasks. The state of the art PbE system in this domain (Gulwani 2011) quite well on a restricted set of tasks. Therefore we decided to pick this domain and use Gulwani's engine in the core of our system. We put together a corpus of 773 text-editing tasks that can be described by Gulwani's engine. In order to provide such a corpus, we mainly used online resources such as Regex Programming Forum (http://forums.devshed.com/). We used Mechanical Turk to get input/output examples for each task. We allow each worker to provide only one example for each task. This is because we want to get variety of examples in order for the PbE system to converge to the intended task faster. The speed of the convergence (or in other words the size of the solution space) depends on how carefully the examples are designed. For this reason, we gave explicit instructions to the workers to provide rich examples: "[. . .] Your example must be RICH: there should be several locations that the edition applies and many locations that it does not [. . .]".

We created one HIT (Human Intelligence Task) per text-editing task, with 6 workers per HIT. This number was picked based on the average number of input/output examples required to converge to a solution for the tasks in our corpus, while leaving some margin for noise. The results from our first several published batches showed even though we only allowed users with high approval rate to do our HITs, we still got a lot of empty or gibberish answers. Therefore, we created a Qualification Type and gave the qualification to a list of all the workers who had given us rich examples in the first couple of runs. The following table contains the percentage of correct input/output examples provided by the workers, reviewed manually, for both before and after we limited the availability of the tasks to qualified workers. We published 1000 qualified HITs and 6000 unqualified HITs.

HITs accepted before qualification	59.9%
HITs accepted after qualification	96.1%

The 4% error rate on accomplished HITs looks very promising. Note that we can make the error rate arbitrarily low by increasing the number of HITs per task. We can automatically detect the incorrect answers and throw them out or even automatically fix them. This can be done in several levels. First, we can use some heuristics (or a classifier) to throw out empty or gibberish HITs. Second, the PbE engine can simply check whether there exists any solution for the given input/output examples. If not, it can throw out one example at a time (rotating through all examples) to find the largest subset of examples for which there exists a solution. Third, the PbE engine is able to to detect and fix small errors like having extra/missing whitespace/newline (a few extra/missing characters in general), etc. Fourth, a second type of HIT may be created to ask (possibly qualified) workers to verify/fix the input/output examples. Therefore even with no qualification assigned it is possible to achieve a high accuracy. It is a big advantage not to limit the list of workers, as it directly affects how fast we can have the HITs accomplished. The following table gives the average time workers spend on doing our HITs.

Average time to complete a HIT-unqualified	3m 24s
Average time to complete a HIT-qualified	4m 13s

Note that compare to the time human experts need to get back to the users (sometimes several days, depending on the volume of requests from users), this could be considered a nearly real-time response. Finally, based on our early experiments it costs less than 50 cents/task in average to gather the input/output examples, which is by far below what is paid to a programing expert to do the same task.

5 Future Work

In conclusion, our system depends upon a blend of traditional and human computation. We incorporate crowdsourced work with a PbE system in order to solve simple natural language programming tasks. Our early experiments show that the human computation portion can perform exceedingly well, even without much supervision. Our system drastically reduces the time and cost required to achieve the same task by human experts. While our early experiments are very encouraging, it remains to the future work to put the system into practice. We are optimistic about the success of this system, and we believe that it will demonstrate the effectiveness of human computation in traditionally difficult tasks.

References

Bigham, J. P.; Jayant, C.; Ji, H.; Little, G.; and et al. 2010. Vizwiz: nearly real-time answers to visual questions. In *Proceedings of UIST '10*, 333–342. New York, NY, USA: ACM.

Cypher, A.; Halbert, D. C.; Kurlander, D.; Lieberman; and et al., eds. 1993. *Watch what I do: programming by demonstration*. Cambridge, MA, USA: MIT Press.

Dijkstra, E. W. 1979. On the foolishness of "natural language programming". In *Program Construction, International Summer School*, 51–53. London: Springer-Verlag.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, 317–330. ACM.

Liang, P.; Jordan, M. I.; and Klein, D. 2010. Learning programs: A hierarchical bayesian approach. In *ICML '10*.

Liang, P.; Jordan, M. I.; and Klein, D. 2011. Learning dependency-based compositional semantics. In *ACL '11*.