Heads in the Cloud: Challenges and Opportunities in Human Computation

Robert C. Miller, Greg Little, Michael Bernstein, Jeffrey P. Bigham, Lydia B. Chilton, Max Goldman, John J. Horton, Rajeev Nayak

Introduction

Crowd computing is rapidly becoming an essential part of the landscape of modern computing. Crowd computing encompasses the interaction among large numbers of people facilitated by software systems and increasingly ubiquitous networking technology. Crowds are powering intellectual enterprises (Wikipedia), real-time information media (Twitter), prediction markets (Intrade), and online labor markets (Amazon Mechanical Turk). One way to think about crowd computing is as the human analogue to *cloud computing*. Where the cloud provides access to elastic, highly available computation and storage resources out in the network, the crowd represents access to elastic, highly-available *human* resources, such as human perception and intelligence. Crowd computing offers the potential to build systems that combine the strengths of software with the intelligence and common sense of human beings.

The particular variant of crowd computing considered in this article is *human computation*, which we define as using software to orchestrate a process of small contributions from a crowd to solve a problem that can't be solved by software alone. Human computation was first popularized by Games With a Purpose (GWAP), in which the computation is a side effect of a fun game [8]. For example, the ESP Game asks two players to guess words associated with an image, scoring points when their words agree (which makes the game fun), but also generating useful labels to index the image for searching (which makes it human computation).

Since GWAP, other general platforms for human computation have begun to emerge. Amazon Mechanical Turk (www.mturk.com) is a marketplace for *paid* human computation, where people do short tasks for small amounts of money. CrowdFlower (crowdflower.com) also pays people for computation -- not only in real currency, but also in virtual currency for games like Farmville and Mafia Wars. Social networks like Facebook are also becoming platforms for human computation, motivated by social relationships rather than entertainment or monetary reward.

These platforms make it increasingly feasible to build and deploy systems that use human intelligence as an integral component. In this article, we discuss three challenges we face in exploring the space of human computation systems, and some initial steps that we have taken to address each one. The challenges are: *applications*, understanding what's appropriate for human computation and what isn't; *programming*, learning how to write software that uses human computation; and *systems*, learning how to get good performance out of a system with humans in the loop.

Applications

What application areas will benefit the most from human computation? What properties do certain problems possess that make them amenable to a successful solution by a hybrid human/software system? Since the end-user of such a system is also, typically, human, we can refine this question further: why

does a human end-user need to request the help of a human crowd to accomplish a goal, rather than just doing it themselves?

One reason is *differences in capability*, i.e., the crowd has abilities or knowledge that the end-user lacks, either innately or because of situational constraints. For example, VizWiz [1] helps blind users answer visual questions about the world by taking a photograph with a smartphone's camera, recording a spoken question, and then uploading the query to a crowd of sighted users on the net who are better able to answer it. Some actual VizWiz queries are shown in Figure 1. A related system, Sinch [7], draws on the crowd to help mobile web users, who experience *situational disabilities* caused by their mobility: limited ability to read on a small screen, "fat fingers" that make it hard to click on small web page targets, and slow networks. Sinch allows mobile users to speak a question into their phone and have crowd workers search the web for the answer, using their more capable desktop web access, and returning web pages with the answer highlighted.

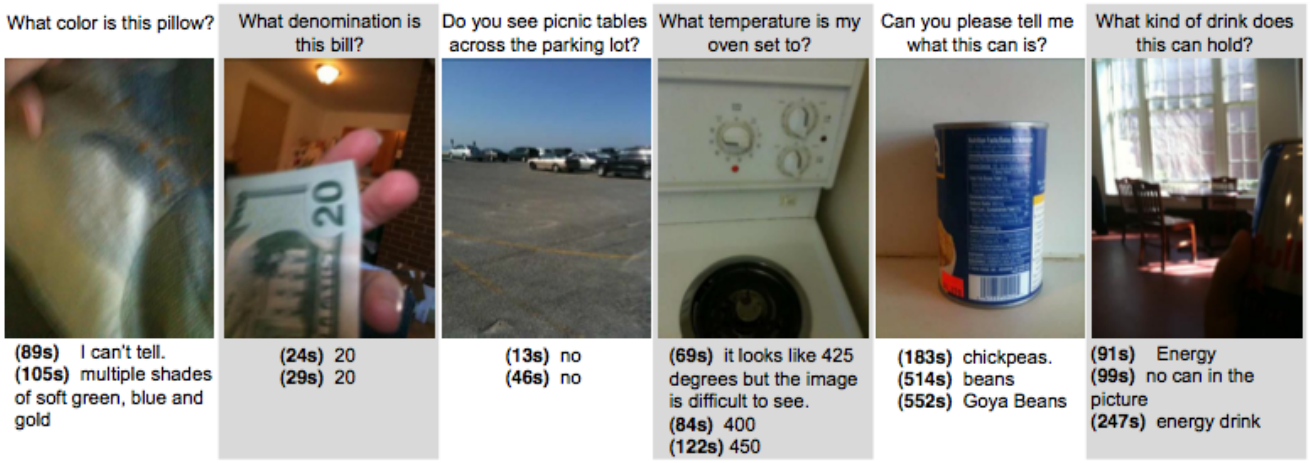

Figure 1: Questions asked by blind users of VizWiz, the photographs they took, and answers received (with latency in seconds).

Another reason to use a crowd is the *many eyes principle*, which has been claimed as an advantage of open source software development (i.e., "many eyes make bugs shallow"). We have exploited this principle in Soylent [2], a Microsoft Word extension that uses a crowd for proofreading, shortening, and repetitive editing. A typical run of Soylent may have dozens of people looking at each paragraph of a document, finding errors that a single human proofreader does not. In fact, a conference paper submitted about Soylent contained a grammatical error that was overlooked not only by Word's built-in grammar checker, but also by all eight authors and six reviewers who reviewed it for the conference. But when we passed the paper through Soylent, the crowd caught the error.

A corollary of the many eyes principle is *diversity*, the fact that the crowd has a wide range of ideas, opinions, and skills. Soylent not only finds writing errors, but suggests multiple ways to fix each one. It can also suggest places where the author can cut out text to save space -- a hard problem even for skilled authors, who are often attached to their writing and reluctant to cut it. Soylent can typically cut text down to 85% of its original length, by identifying words or phrases that can be removed or rewritten without changing the meaning of the text or introducing errors (Figure 2).

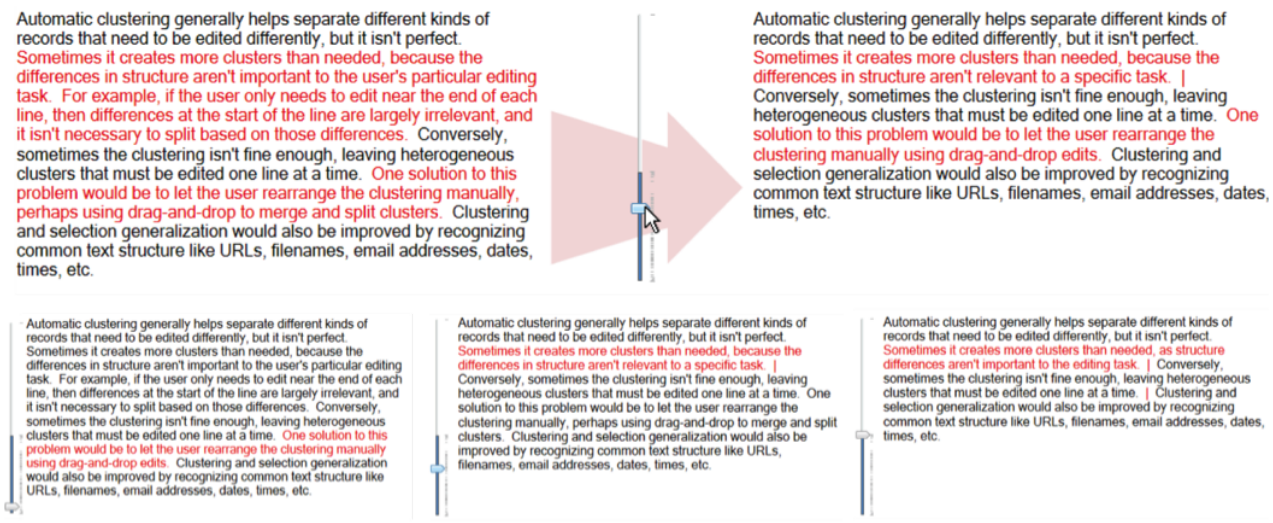

Figure 2. After a crowd has suggested words or phrases that can be rewritten or removed, Soylent allows the end-user to shorten a paragraph interactively with a slider. Red text indicates locations where cuts or rewrites have occurred. Tick marks represent possible lengths, and the blue background bounds the possible lengths.

Programming

Prototyping a human computation system is hard if you have to entice a crowd to visit your website. GWAP handles this by making the experience fun, but not all human computation systems are fun enough to be self-motivating, particularly at the prototyping stage. We have found that Amazon Mechanical Turk is a good prototyping platform for many forms of human computation, since it offers a ready service for recruiting a crowd on demand. The first prototypes for VizWiz and Soylent were built on Mechanical Turk. Yet thinking about programming with human beings inside the system poses special problems. On Mechanical Turk, a request for a human to do a small task can take a few minutes and cost a few cents to get a result -- which is astounding in one sense, that you can obtain human assistance so quickly and so cheaply -- but is abysmally slow and expensive compared to a conventional function call.

We need new tools that help programmers experiment with human computation in their systems. For example, our TurKit toolkit [3] integrates Mechanical Turk calls in a traditional imperative/object-oriented programming paradigm, so that programmers can write algorithms that incorporate human computation in a familiar way. TurKit does this using a novel programming model, *crash and rerun*, which is suited to long running distributed processes where local computation (done by software) is cheap, and remote work (done by humans) is costly. The insight of crash-and-rerun programming is that if our program crashes, it is cheap to rerun the entire program up to the place it crashed. This is true as long as rerunning does not re-perform all of the costly external operations from the previous run. The latter problem is solved by recording information in a database every time a costly operation is executed. Costly operations are marked by a new primitive called *once*, meaning they should only be executed once over all reruns of a program. Subsequent runs of the program check the database before performing operations marked with once to see if they have already been executed. This model makes it much easier to code algorithms involving human computation: for example, a TurKit program can sort a list of images using human preference judgements by calling the human computation in the sort algorithm's comparison function, and wrapping those calls in *once* to make them persistent.

Another programming challenge is the development of algorithms and design patterns that handle the idiosyncrasies of human beings. Humans are not programmable machines, and they don't always follow instructions, unintentionally or otherwise. Sometimes this should be embraced and supported, to harness the creativity and diversity of the crowd; other times it simply produces noisy, erroneous, or useless results. For example, we have studied alternative algorithms for content creation [4]. Iterative processes are similar to Wikipedia or open source software development: people build on existing content created by others, with voting or independent review ensuring that the process stays on track. Parallel processes are often seen in design contests, like Threadless.com: people generate content independently, and then the best is chosen by a voting process. In experiments involving various kinds of work (handwriting transcription, image description, and brainstorming), our results show that iterative processes generally produce higher average quality than parallel processes. In the case of brainstorming, however, workers riff on good ideas that they see to create other good names, but the very best ideas seem to be generated by workers working alone. In transcription, showing workers guesses from other workers can lead them astray, especially if the guesses are self-consistent but wrong.

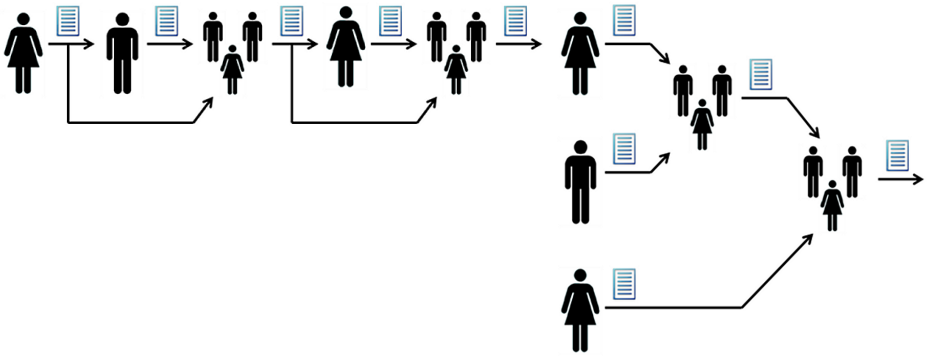

Figure 3. Some human computation processes are *iterative* (left), involving a succession of interleaved improvement steps (by one person) and voting steps (by several people). Other processes are *parallel* (right), in which individuals generate original content and voters simply choose among the alternatives.

Crowd workers exhibit high variance in the amount of effort they invest in a task. Some are Lazy Turkers, who do as little work as necessary to get paid. Others are Eager Beavers, who go above and beyond the requirements (either to be helpful or to signal that they aren't Lazy Turkers), but in counterproductive ways. We need new design patterns for algorithms involving human computation, that recognize and control this behavior. For example, Soylent uses a Find/Fix/Verify pattern to improve the quality of proofreading and document shortening (Figure 4). In this pattern, some workers find problems, other workers fix them, and still other workers verify the fixes. But questions remain. What other algorithms and design patterns are useful? How should algorithms involving human computation be evaluated and compared from a theoretical point of view?

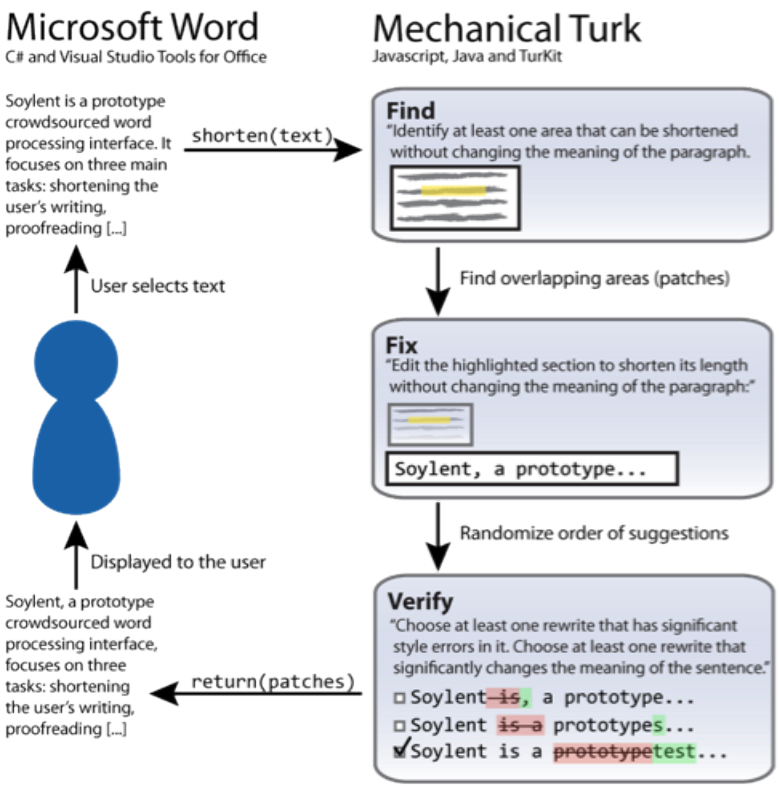

Figure 4. The Find-Fix-Verify algorithm in Soylent identifies patches in need of editing, suggests fixes to the patches, and votes on those fixes.

Systems Problems

Moving from prototyping to actual deployment requires facing questions about how to obtain a reliable and well-performing source of human computation for the system. How can we recruit a crowd to help, and motivate it to continue to help over time, while optimizing for cost, latency, bandwidth, quality, churn, and other parameters? For paid crowds, such as those recruited on Mechanical Turk, these questions intersect with labor economics. Some of our recent work has found that workers in human computation markets like Mechanical Turk behave in unusual ways. For example, instead of seeking work that provides a target *wage* rate, they often seek a target earning amount, and simply work until they reach that target, consistent with game-playing behavior [5]. Another difference in these markets is the overwhelming importance of searchability. Workers' ability to find tasks they want to do is strongly affected by the kind of interface the market offers. Mechanical Turk, for example, typically displays a list of thousands of available tasks, divided into hundreds of result pages, with few effective tools for searching or filtering this list. We have found that most workers simply choose a particular sort order and work their way through the list; they most often sort by newest task, or most tasks available, and surprisingly *not* by price. The speed of completion of a task is strongly affected by its ability to be found -- which may not be strongly related to the monetary reward it offers [6].

We can also think about human computation in computer systems terms, such as cost, latency, and parallelism. Services like VizWiz and Sinch need to return answers quickly. To support that, we have developed an approach (and accompanying implementation) called quikTurkit that provides a layer of

abstraction on top of Mechanical Turk to intelligently recruit multiple workers before they are needed. In a field deployment of VizWiz, users had to wait just over 2 minutes to get their first answer on average, but wait times decreased sharply when questions and photos were easy for workers to understand. Answers were returned at an average cost per question of only $0.07 for 3.3 answers. Given that other visual-assistance tools for the blind can cost upwards of $1000 (the equivalent of nearly 15,000 uses of VizWiz), we believe that human computation embedded in an inexpensive software system can not only be more effective but also competitive with, or even cheaper than, existing pure software solutions. When set to maintain a steady pool of workers (at a cost of less than $5 per hour), quikTurkit can obtain answers in less than 30 seconds.

Beyond payment, many other reasons may entice people to participate in a human computation system, including altruism, entertainment, and friendship. How do those motivations influence system performance, and how should human computation systems be designed to encourage some motivations, and perhaps discourage others? After demonstrating that VizWiz was feasible using paid strangers on Mechanical Turk, we also ported it to Facebook, so that a blind user's sighted friends can help. We are currently studying how people (at least in this context) choose to trade off the strengths and weaknesses of each service. Mechanical Turk is fast but costs money. Facebook is free, and your friends might be more motivated to answer, or even more capable since they know more about you, but you may also be less willing to ask certain personal questions to your friends, rather than asking a Mechanical Turk worker who will forever remain anonymous.

Conclusion

The gap between what software can do and what people can do is shrinking, but the gap will continue to exist for long time. Automatic techniques need to be able to fallback to people when necessary to fill in the gaps, enabling interactions that automatic techniques alone can't yet support and helping us design for the future.

Wizard of Oz prototyping is a venerable technique in human-computer interaction and artificial intelligence, referring to hiding a human being "behind the curtain" to make an intelligent system (or even a not-so-intelligent one) appear to work even though a software backend isn't ready yet. With platforms like Mechanical Turk and Facebook that make human computation practical, we are now at the point where Wizard of Oz is not just for prototyping anymore. We can build useful systems with human power inside, and actually deploy them to real users. These systems will stretch the limits of what software can do, and allow us to find out whether the ideas even work and how people would use them. In addition, we can collect data from actual system use -- like VizWiz queries and photos -- that might eventually help to replace some or all of the human power with artificial intelligence. From this perspective, AI would speed up performance and reduce labor costs. But human computation made the system possible in the first place.

Acknowledgements

Ideas and work discussed in this article come from many students and collaborators, including Michael Bernstein, Mark Ackerman, David Crowell, Bjoern Hartmann, David Karger, Marc Grimson, Katrina Panovich. This work was supported in part by Quanta Computer, NSF, and Xerox.

References

1. Bigham, J.P., Jayant, C., Ji, H., Little, G., Miller, A., Miller, R.C., Miller, R., Whyte, B., White, S., Yeh, T. VizWiz: Nearly Real-time Answers to Visual Questions. *UIST 2010*, to appear.
2. Bernstein, M., Little, G., Miller, R.C., Hartmann, B., Ackerman, M.S., Karger, D.R., Crowell, D., Panovich, K. Soylent: A Word Processor with a Crowd Inside. *UIST 2010*, to appear.
3. Little, G., Chilton, L., Goldman, M., Miller, R.C. TurKit: Human Computation Algorithms on Mechanical Turk. *UIST 2010*, to appear.
4. Little, G., Chilton, L., Goldman, M., Miller, R.C. Exploring Iterative and Parallel Human Computation Processes. *HCOMP 2010*, to appear.
5. Horton, J.J. and Chilton, L. The Labor Economics of Paid Crowdsourcing. *EC 2010*.
6. Chilton, L., Horton, J.J., Miller, R.C., Azenkot, S. Task Search in a Human Computation Market. *HCOMP 2010*, to appear.
7. Nayak, R., *et al*. Sinch: Searching Intelligently on a Mobile Device. *CHI 2011*, in submission.
8. von Ahn, L., and Dabbish, L. Designing Games with a Purpose. *CACM*, v51 n8, August 2008.

About the Authors

Robert C. Miller is an associate professor of computer science at MIT. He grew up in rural Louisiana and plays the accordion, although extremely poorly.

Danny “Greg” Little is a PhD student in computer science at MIT. He is probably asleep right now.

Michael Bernstein is a PhD student in computer science at MIT. He has no spare time, really, but if he did he would spend it krumping.

Jeffrey P. Bigham is an assistant professor of computer science at University of Rochester, and he wants to be mayor of your living room. Don’t let him in.

Lydia B. Chilton is a PhD student in computer science at University of Washington. When not studying human computation, she is a consultant to the United Federation of Planets and makes the occasional journey on the USS Enterprise with her old pals Kirk, Spock and McCoy.

Max Goldman is a PhD student in computer science at MIT. When he gives a talk, his performance is so lively and engaging that it distracts the audience from the actual research results, but everybody goes away happy.

John J. Horton is a PhD student in public policy at Harvard University. He is having trouble thinking of a public policy angle for his human computation research and needs to defend his dissertation soon. This is a growing problem that, to date, turkers have been unable to solve.

Rajeev Nayak is a masters student in computer science at MIT. He bats .400, shoots .575 from the field, sings a cappella, and watches “Glee” religiously.