BLOG@CACM

The *Communications* Web site, http://cacm.acm.org, features more than a dozen bloggers in the BLOG@CACM community. In each issue of *Communications*, we'll publish selected posts or excerpts.

twitter
Follow us on Twitter at http://twitter.com/blogCACM

DOI:10.1145/2240236.2240239 http://cacm.acm.org/blogs/blog-cacm

Machine Learning and Algorithms; Agile Development

John Langford poses questions about the direction of research for machine learning and algorithms. Ruben Ortega shares lessons about agile development practices like Scrum.

John Langford
"Research Directions for Machine Learning and Algorithms"
http://cacm.acm.org/blogs/blog-cacm/108385
May 16, 2011

S. Muthu Muthukrishnan invited me to the National Science Foundation's Workshop on Algorithms in the Field with the goal of providing a sense of where near-term research should go. When the time came, though, I instead bargained for a post, which provides a chance for other people to comment.

There are several things I did not fully understand when I went to Yahoo! about five years ago. I would like to repeat them as people in academia may not yet understand them intuitively.

1. Almost all the big-impact algorithms operate in pseudo-linear or better time. Think about caching, hashing, sorting, filtering, etc. and you have a sense of what some of the most heavily used algorithms are. This matters quite a bit to machine learning research, because people often work with superlinear time algorithms and languages. Two very common examples of this are graphical models where inference is often a superlinear operation—think about the n^2 dependence on the number of states in a Hidden Markov Model and Kernelized Support Vector Machines where optimization is typically quadratic or worse. There are two basic reasons for this. The most obvious is that linear time allows you to deal with large datasets. A less obvious but critical point is that a superlinear time algorithm is inherently buggy; it has an unreliable running time that can easily explode if you accidentally give it too much input.

2. Almost anything worth doing requires many people working together. This happens for many reasons. One is the time-critical aspect of development—in many places it really is worthwhile to pay more to have something developed faster. Another is that real projects are simply much bigger than you might otherwise expect. A third is that real organizations have people coming and going, and any project that is by just one person withers when that person leaves. This observation means the development of systems with clean abstractions can be extraordinarily helpful, as it allows people to work independently. This observation also means simple widely applicable tricks (for example, the hashing trick) can be broadly helpful.

A good way to phrase research directions is with questions. Here are a few of my natural questions.

1. How do we efficiently learn in settings where exploration is required? These are settings where the choice of action you take influences the observed reward—ad display and medical testing are two good scenarios. This is deeply critical to many applications because the learning with exploration setting is inherently more natural than the standard supervised learning setting. The tutorial we did detailed much of the state of the art here, but very significant questions remain. How can we do effective offline evaluation of algorithms? How can we be both efficient in sample complexity and computational complexity? Several groups are interested in sampling from a Bayesian posterior to solve these sorts of problems. Where and when can this be proved to work? (There is essentially no analysis.) What is a maximally distributed and incentive-compatible algorithm that remains efficient? The last question is very natural for marketplace

design. How can we best construct reward functions operating on different time scales? What is the relationship between the realizable and agnostic versions of this setting, and how can we construct an algorithm that smoothly interpolates between the two?

2. How can we learn from lots of data? We will be presenting a KDD survey/tutorial about what is been done. Some of the larger-scale learning problems have been addressed effectively using MapReduce. The best example I know is Ozgur Cetin's algorithm at Yahoo! It is preconditioned conjugate gradient with a Newton stepsize using two passes over examples per step. (A nonHadoop version is implemented in Vowpal Wabbit for reference.) But linear predictors are not enough; we would like learning algorithms that can, for example, learn from all the images in the world. Doing this well plausibly requires a new approach and new learning algorithms. A key observation here is that the bandwidth required by the learning algorithm cannot be too great.

3. How can we learn to index efficiently? The standard solution in information retrieval is to evaluate (or approximately evaluate) all objects in a database returning the elements with the largest score according to some learned or constructed scoring function. This is an inherently $O(n)$ operation, which is frustrating when it's plausible that an exponentially faster $O(log(n))$ solution exists. A good solution involves both theory and empirical work here as we need to think about how to think about how to solve the problem, and of course we need to solve it.

4. What is a flexible, inherently efficient language for architecting representations for learning algorithms? Right now, graphical models often get (mis)used for this purpose. It is easy and natural to pose a computationally intractable graphical model, implying many real applications involve approximations. A better solution would be to use a different representation language that was always computationally tractable yet flexible enough to solve real-world problems. One starting point for this is Searn. Another general approach was the topic of the Coarse-to-Fine Learning and Inference Workshop. These are inherently related as coarse-to-fine is a pruned breadth first search. Restated, it is not enough to have a language for specifying your prior structural beliefs; instead we must have a language that results in computationally tractable solutions.

5. The deep learning problem remains interesting. How do you effectively learn complex nonlinearities capable of better performance than a basic linear predictor? An effective solution avoids feature engineering. Right now, this is almost entirely dealt with empirically, but theory could easily have a role to play in phrasing appropriate optimization algorithms, for example.

Good solutions to each of these research directions would result in revolutions in their area, and every one of them would plausibly see wide applicability.

What's missing?

Ruben Ortega "Research in Agile Development Practices"

http://cacm.acm.org/blogs/blog-cacm/109811
June 20, 2011

I am an enthusiastic advocate of agile software development practices like Scrum. Its ability to allow teams to focus on delivering product and communicate status has made it one of the easiest and best software development techniques I have seen in a career that has used ad hoc, Waterfall, and everything in between.

Recent research from New Zealand has furthered the cause by performing a study that involved 58 practitioners in 23 organizations over four years. In reading a Victoria University of Wellington article on "Smarter Software Development" and then looking at Rashina Hoda's thesis "Self-Organizing Agile Teams: A Grounded Theory," there are two interesting takeaways:

1. Self-organizing scrum teams naturally perform a balancing act between:

▸ Freedom and responsibility: The team is responsible for collective decision making, assignment, commitment, and measurement, and must choose to do them.

▸ Cross functionality and specialization: The team is responsible for deciding when to distribute work across team members or have each focus on a certain part of the project.

▸ Continuous learning and iteration pressure: The team is responsible for delivering on its own schedule and the retrospectives to improve each sprint.

The advantage of giving this balancing act to the team is that it take ownership of the solution with the full understanding of all the trade-offs that will need to occur each sprint. By distributing the work to the team, it also makes team members accountable to one another for making sure the goals are achieved.

2. Self-organizing teams have their members assume some well-defined roles spontaneously, informally, and transiently to help make their projects successful:

▸ Mentor: Guides the team in the use of agile methods.

▸ Coordinator: Manages customer expectations and collaboration with the team.

▸ Translator: Translates customer business requirements to technical requirements and back.

▸ Champion: Advocates agile team approach with senior management.

▸ Promoter: Works with customers to explain agile development and how to collaborate best with the team.

▸ Terminator: Removes team members that hinders the team's successful functioning.

These roles are an emergent property that comes from using agile development methods. They are not prescribed explicitly as part of any of the agile development philosophies, but they arise as part of successful use of the methodology.

I am eager to see more research emerge as to where agile software development practices succeed and where they need improvement. There is a large body of evidence that shows it to be a successful strategy, and having the research to support it would help encourage its adoption. C

John Langford is a senior researcher at Microsoft Research New York. **Ruben Ortega** is an engineering director at Google.