My Research

My field of research is *complexity theory*, which is a common subfield of theoretical computer science and mathematics that studies the limitations and capabilities of efficient computation. As hinted to by this description, the typical setting in complexity theory is that one has some *model of computation*, very broadly construed, as well as some kind of *complexity measure* of the computation performed in that model (such as time, space, randomness, non-determinism, etc.). One is then interested in how efficiently the model solves different computational problems, as measured by the complexity measure. In contrast to algorithms research, where one seeks increasingly efficient algorithms to solve a given computational task in some model of computation, complexity theory generally focuses on *lower bounds* on a given complexity measure. Thus, an archetypal result in the field shows that some task *cannot* be performed efficiently within a given computational model.

More specifically, my field is an instantiation of the template above called *proof complexity*. Here, the model of computation is some restricted mode of reasoning called a *proof system*; the computational problem to be solved is to prove a given mathematical theorem; and the complexity measure is, for instance, the size of the proof compared to the size of the theorem statement to be proved.

Complexity theory is an abstract and mathematical field: its research findings generally come in the form of mathematical theorems of rather theoretical nature, established through proofs, and the computational problems studied are abstracted to such a degree that results do not carry over immediately to real-world settings. Therefore, it seems fair to say that the vast majority of results and insights of complexity theory are of too theoretical a nature to have any significant bearing on a large-scale machine-learning engineering project. However, research in mathematics—and hence complexity theory—is becoming increasingly collaborative, and especially the process of writing mathematical papers is similar to that of collaboratively writing a complex piece of software. Therefore, software engineering insights may well be informative to mathematical practice and collaboration and vice versa, which I hope to illustrate below.

Concepts From Lectures

Robert's Lectures

DRY - Don't Repeat Yourself The first of my takeaways from the lectures is the DRY (Don't Repeat Yourself) principle of software development, originally conceived of by Hunt and Thomas [1]. It states that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." I find this principle highly relevant to mathematical writing.

A mathematical paper should present a proof of a mathematical claim, and this proof must be presented so as to have no logical gaps, but also be parsable by a human. A mathematical paper thus resembles a computer program where the reader plays the role of a compiler. This human compiler is more lenient with typos and minor errors but all the more sensitive to the quality of exposition than a usual compiler would be.

A mathematical proof of a theorem worthy of a research paper is often too long to present in one shot. Usually, the argument consists of a series of modular steps, each of which culminates in an auxiliary claim called a *lemma*. Lemmas are often rather uninteresting by themselves, but might be used repeatedly in the main argument. These lemmas should be clearly stated and pointed out when they are used in the main argument. In many ways, principled mathematical writing therefore applies the DRY principle, with lemmas playing the role of the "pieces of knowledge."

Version Control Another takeaway from the lectures is proper use of Version Control (VC). I have used version control in collaborative research projects, but never really learned to use it in a principled way. This probably lead to headaches among my collaborators. The resources provided in the lecture as well as the general pointers regarding branches, atomic commits, etc., will make a noticeable difference in how I use VC in my research projects going forward.

Guest Lectures

Behavioral Software Engineering The first concept from the guest lectures (also mentioned in Robert's lectures) that caught my attention was that of *Behavioral Software Engineering*, which focuses on the humans involved in writing software rather than the software itself. It really is quite a basic insight that whatever enterprise humans take on is profoundly influenced by the fact that it is indeed done by humans and not machines. But as can be seen from the massive influence of Daniel Kahneman's and others' work in behavioral economics and social science as a whole, we are still realizing the ramifications of this insight. In one sense this is quite mysterious: I have personally never been in a project or situation involving other humans where the personal characteristics and relations between those involved did *not* influence the project noticeably, and I'm sure my experience is not unusual.

It appears funny at first that such "objective" fields as mathematics or software engineering also have this characteristic, but it is not so strange: the correctness of a mathematical theorem or a piece of software might in some sense be universal, but they are only as useful as their ability to be internalized and understood by humans.

Over-investing The second concept from the guest lectures that I would like to highlight comes from how SAAB is tackling the AI boom: namely, by *over-investing* in it. More specifically, Per said that when faced with something new and uncertain like AI that SAAB can potentially benefit immensely from, they will invest large amounts of money on research and development in AI in order to ensure they do not fall behind, knowing full well that most of the projects they fund will not be worth the money invested. This approach is in many ways similar to that of a mathematician: results in mathematics are never certain, and there are countless examples of solutions to long-standing open problems that use techniques and insights from wholly disparate other fields of mathematics. Therefore, young mathematicians are often advised to read and think broadly and to be familiar with as many branches of mathematics as possible, even though most of that knowledge will not be useful when proving theorems of their own. I believe wholeheartedly in this approach of "over-investing" in mathematical knowledge, and it is encouraging to see large companies thinking in a similar vein when much more than one's own time is on the line.

Papers

Challenges in Building Products With ML Components

In the paper *A Meta-Summary of Challenges in Building Products with ML Components – Collecting Experiences from 4758+ Practitioners* [2], Nahar *et al.* perform a meta-summary study of papers examining the specific challenges that Machine Learning (ML) components introduce into large-scale software projects in industry. Using guidelines for systematic literature review, Nahar *et al.* collect 50 relevant papers from the literature and organize the challenges in those papers into seven groups. A list of these groups and some of their associated challenges follows, paraphrasing the overview of Nahar *et al.*

Requirements Engineering: Challenges in this group include, for instance, dealing with unrealistic expectations on AI solutions from stakeholders, vagueness in ML problem specifications, and regulatory constraints.

Architecture, Design, and Implementation: Compared to traditional software engineering, the specific requirements of ML solutions requires transitioning to a more pipeline-driven view, which appears to be a difficult transition for many practitioners. Moreover, many of the tools of ML solutions induce various uncontrollable dependencies, and make scaling and monitoring difficult.

Model Development: Issues in this domain come from, for instance, the lack of standardization of code quality and limitations of engineering infrastructure in ML tooling.

Data Engineering: In traditional software engineering, data does not play the crucial role that it does in ML. Owing to this, many challenges arise in, for instance, ensuring data quality, adhering to privacy regulation, and versioning.

Quality Assurance: ML systems are by their nature hard to test and debug, and it is in general not clear how pipelines and models interact with each other in large-scale ML systems.

Process: Compared to the more mature field of software engineering, the development of ML systems lacks well-defined processes and comes with a large degree of uncertainty, making it hard to plan ahead and estimate efforts and costs.

Organization and Teams: ML systems are often developed by teams with more varied background than in traditional software engineering. Therefore, different team members often have conflicting ideals regarding, for instance, code quality, and it is often the case that no one team member has all the interdisciplinary skills required to oversee all parts of the system development. Moreover, ML systems are often costly and so often suffer from resource limitations.

Nahar *et al.* account for the most common challenges faced in each of these domains in the investigated papers. Regarding requirements engineering, they find that 17 out of the 50 considered papers mention lack of AI literacy and vagueness in ML problem specifications as challenges. In the domain of architecture, design, and implementation, the transition from model-centric to pipeline-driven development and ML-incurred design complexity are mentioned as challenges in 11 out of the 50 considered papers. In terms of model development, the most commonly mentioned challenge (19 out of 50 papers) was that ML infrastructure and technical support are limited. The most common challenge in the domain of data engineering was found to be ensuring data quality (17 out of 50 papers), with many of them stating that current tools do not support this well.

This paper consolidates what seems to be a rather large body of knowledge about the challenges incurred by ML solutions, and synthesizes the main takeaways in a clear and understandable way. The findings of this paper indicate that some challenges resurface again and again in large-scale ML systems. This might be valuable for practitioners, who can then be aware of the challenges that might turn up during the course of development and, hopefully, be able to plan ahead and mitigate them.

My research does not rely on any data or code whatsoever, and is thus not amenable to ML solutions. Therefore, this paper does not have any immediate consequences for my research, but there are still many general takeaways to consider. For instance, I am not aware of any meta-summary of challenges that turn up during mathematical collaboration. I am sure that many of the issues that turn up there have a similarly specific nature to those in ML systems, and a systematic overview of them would be helpful for me to be aware of. It is often the case that different collaborators in a project in mathematics come from different parts of mathematics, with differing views on, for instance, what facts are "well-known" and therefore need no explanation, and conflicting views on the balance between intuition and rigor in mathematical exposition. A systematic overview of such challenges, similar to the one of Nahar *et al.*, would be very helpful in this regard.

In a large-scale AI engineering project, the systematic overview of Nahar *et al.*would be very helpful in raising awareness of the challenges that might—or indeed, probably *will*—surface, and there can even be concrete meetings before the project begins that decide on how each of them will be tackled should they emerge. Regarding my own research, concrete results in my field have close to zero bearing on a large-scale AI engineering project, but again the general principles of mathematical practice and collaboration can be helpful. For instance, the ideal in mathematics of clear, modular exposition directed to a *human* reader is desirable in a large software project as well. Moreover, the fairly flat hierarchy of a mathematical collaboration could also be a good fit for some software projects, especially if the team of developers is interdisciplinary as is common in ML projects.

It is not clear how my research could be changed to benefit ML engineering. Some of the computational problems routinely solved by ML are intensively studied in my field, but the results are usually of so theoretical a nature that they do not impact practical projects in any significant way. Perhaps a theoretical result saying that some problem is very difficult to solve might help with the challenge of dealing with unrealistic expectations on ML solutions if a stakeholder has a mathematical background.

Impact of SOLID Design Principles on ML Code Understanding

In the paper *Investigating the Impact of SOLID Design Principles on Machine Learning Code Understanding* [3], Cabral *et al.* seek to understand the impact of the five tenets of the SOLID set of design principles (described in the next paragraph) on the understanding of machine learning code. To this end, Cabral *et al.* set up a controlled experiment with 100 data scientists, sourced from two gradutate programs in data science and one software company. The control group was presented with a snippet of ML code that did not adhere to the SOLID principles, while the experimental group was instead shown a restructured piece of code, adhering to the SOLID principles, with the same functionality. The participants were then asked a series of questions about the readability and functionality of the code. The paper found that each of the five SOLID principles significantly improved understanding of the code snippets and also allowed for further extensibility of the code.

SOLID is an acronym for five principles that aim to make code more understandable, reusable, and maintainable. Here follows a brief overview of these principles, paraphrasing that of Cabral *et al.* The ***S**ingle Responsibility Principle* says that each class in the code should do one thing, with the goal of keeping design simple and future changes less dramatic. The ***O**pen-Closed Principle* advocates that classes should be open for extension but closed for modification. The ***L**iskov Substitution Principle* states that if a class A extends some other class B, then all instances of class B in the code should be replaceable by instances of class A without issues. As an (unrealistic) example, all parts of the code that apply to rectangles should still work when occurrences of "rectangle" are replaced by "square" since squares are rectangles. The ***I**nterface Segregation Principle* says that interfaces should be developed separately for each client, meaning that clients should never be forced to implement interfaces they do not use nor depend on unused methods. Finally, the ***D**ependency Inversion Principle* roughly states that whenever two modules or classes need to depend on each other, the less stable one should depend on the more stable one and not vice versa.

The SOLID design principles can help to standardize code in a large-scale project and to increase readability and extensibility. They also provide a standardized and clear template for code review. This paper shows via experiment that adherence to the principles indeed increases ease of understanding code, which might help team members in ML projects argue for why they should be used in the project and to increase motivation for adhering to them.

My research does not involve writing any code, so the SOLID principles and the conclusions of this paper do not really apply directly to my research. Nevertheless, if one replaces the word

class in the SOLID principles with the word *lemma* (explained earlier) or *definition*, the SOLID principles give a robust and reasonable set of guidelines for mathematical writing. For instance, a definition in a paper should, ideally, be open to extension but not modification, since then all lemmas involving that definition will still hold. In general, a similar set of principles specifically designed for mathematical writing would be very helpful. I am not aware of any such set of principles.

As noted previously, my research has little to offer a large-scale ML engineering project, but the principles of modular and human-focused exposition in my field are a good complement to the SOLID principles in writing good code. The SOLID principles themselves and the conclusions in this paper that they indeed help code understanding are on the other hand immediately applicable to a large-scale software project. They might even be especially helpful in a still-nascent field like large-scale ML engineering for standardizing code and helping to increase understanding and extensibility. The scientific findings in this paper might also motivate stakeholders to ensure that the code in the project adheres to the SOLID principles.

As I said when discussing the previous paper, the only way I can think of changing my research to help ML engineering is to study problems that ML solutions aim to solve in order to guide practical projects toward which problems are feasible to tackle. However, these problems are rather far from my area of expertise, and results that would be of immediate practical significance are far out of reach for current methods.

References

[1] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Reading, Massachusetts: Addison-Wesley, 2000.

[2] N. Nahar, H. Zhang, G. Lewis, S. Zhou, and C. Kästner, "A meta-summary of challenges in building products with ML components – collecting experiences from 4758+ practitioners," in *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 171–183.

[3] R. Cabral, M. Kalinowski, M. T. Baldassarre, H. Villamizar, T. Escovedo, and H. Lopes, "Investigating the impact of SOLID design principles on machine learning code understanding," in *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, ser. CAIN '24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 7–17.