Feature Interactions on Steroids: On the Composition of ML Models

Christian Kästner (^*), Eunsuk Kang (*), and Sven Apel (+)
kaestner@cs.cmu.edu, eunsukk@andrew.cmu.edu, apel@cs.uni-saarland.de

(^) corresponding author
(*) Carnegie Mellon University, USA
(+) University of Saarland, Germany

From the Editor:
Is machine learning different? What lessons can we learn from SE that would help ML applications? Here, it is argue that "feature interaction" (a well studied problem in ML) is a natural framework within which to discuss developing ML apps.

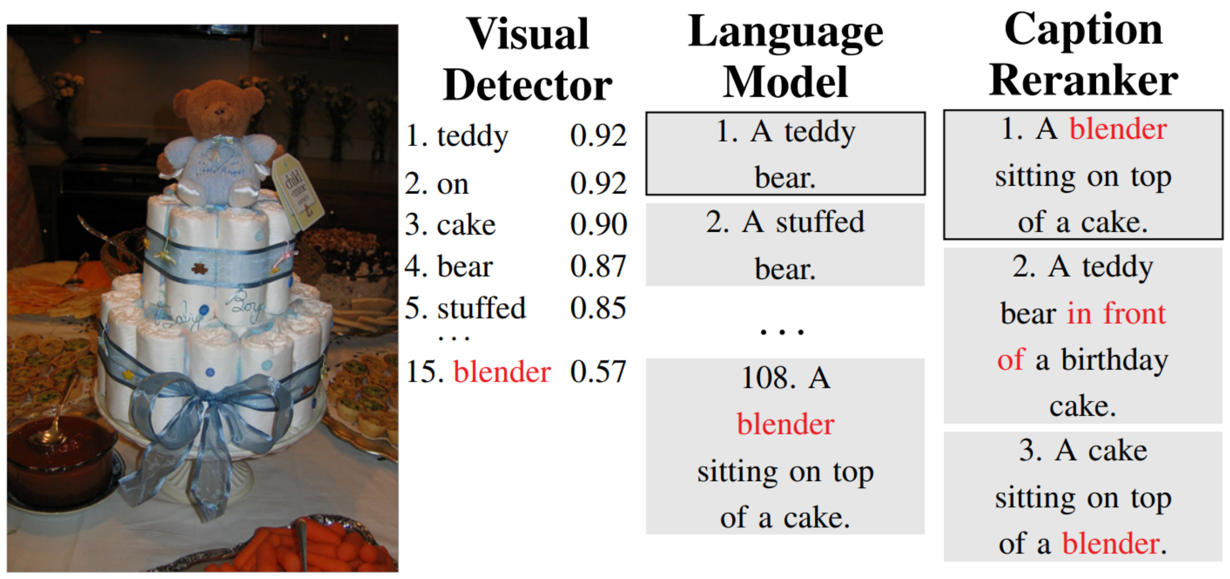
This ``SE for AI" column publishes commentaries on the growing field of SE for AI. Submissions are welcomed and encouraged (1,000–2,400 words, each figure and table counts as 250 words, try to use fewer than 12 references, and keep the discussion practitioner focused). Please submit your ideas to me at timm@ieee.org.—Tim Menzies

One of the key differences between traditional software engineering and machine learning (ML) is the lack of specifications for ML models. Traditionally, specifications provide a cornerstone for compositional reasoning and for the divide-and-conquer strategy of how we build large and complex systems from components, but these are hard to come by for machine-learned components. While the lack of specification seems like a fundamental new problem at first sight, in fact software engineers routinely deal with iffy specifications in practice: We face weak specifications, wrong specifications, and *unanticipated interactions* among specifications. Machine learning may push us further, but the problems are not fundamentally new. Rethinking ML model composition from the perspective of the *feature-interaction problem* highlights the importance of software design.

1 Challenges in Composing ML Models

Many systems do not just use one ML model but compose multiple models to solve complex problems. To automatically generate captions for images, one could try to learn a model that directly takes an image and produces a caption, but a state-of-the-art solution decomposes the problem into three steps with different models [1]: First, a **visual detector** (convolutional neural network) predicts which of 1000 common objects is visible in the image, then a **language model** (maximum-entropy model) takes these objects and generates 500 plausible sentences with them, and finally a **caption ranker** (deep multimodal similarity model) takes both the original image and the generated sentences and scores the combination to pick the best sentence as caption.

At a first glance, this looks like a great divide-and-conquer story. We break down the problems into steps. Each model can be developed and tested independently, using different modeling and implementation techniques, possibly reusing models or training data from other domains. At a second glance though, it seems we cannot really reason modularly about problems. For example, as shown by Nushi et al. [2], all three models somehow contribute to the poor caption "A blender sitting on top of a cake" for an image of a diaper cake of a baby shower, similar to the one below: The visual detector detects a blender where there is none (albeit with low confidence), the language shows low common-sense awareness with "a blender sitting", and the ranking model picks that sentence, even though it includes a word with a low object detection score. So, no component is behaving perfectly or compensated for problems in others. There are no clear responsibilities or boundaries between the components that could be used to assign blame.



Problems in each component leading the to poor caption, from [2]

So, if we already have a problem with composing three models, how are we expecting to build reliable systems with, say, the 18 models in Baidu's self-driving car system [3].

2 The Root of the Problem: Lack of Specifications

The core of the problem is that we do not have clear specifications for what each model is supposed to do. For traditional software components, a specification tells us whether a component's output is *either correct or wrong* for a given input---not "pretty good" or "95% accurate." We also would not accept correct answers for 98% of all inputs, but would instead consider the component to have a *bug* if it produced a wrong result. In systems with multiple components, we can *assign blame* by checking which component produced a wrong output according to its *specification* (even if our specifications in practice are often weak and textual).

Specifications enable *modular reasoning* based on logic: We can understand the consequences from combining two modules in terms of the composed specifications. This is a cornerstone for a sound divide-and-conquer strategy, and *modularity* and enabled the vast reuse of libraries in building modern and complex software systems.

For ML components, we do not have any meaningful specifications in the traditional sense. For many problems, we already have a hard time capturing what it means for a single prediction of a model to be "correct" (humans may not agree). Even if we had a strict binary notion of correctness for any single prediction, we would *not* expect a model to make correct predictions for every possible input. In line with the aphorism *“all models are wrong, but some are useful,”* we do not evaluate the *correctness* of a model but whether the model *fits* our problem, usually approximated through *accuracy* measures. These do not compose nicely.

4 The Limits of Decomposition: Feature Interactions

While it may seem that modular decomposition in traditional software systems is obvious and clean, unfortunately, that is not always the case either. It is not even just a question of whether we write formal or informal, strong or weak specifications, but a question of whether this would be possible in the first place. The study of *feature interactions* has made this ample clear.

There are many examples of feature interactions, and they typically follow a common pattern: We decompose the system into components, then design, develop and test these components in separation, but finally observe unexpected behavior when composing them. The canonical example is a *call forwarding* feature in a phone system that competes with a independently developed *call waiting* feature on how to respond to the same call on a busy line. We can often blame interaction problems on weak specifications that did not anticipate the interaction, but in general the problem is much bigger.

In complex systems, it is often not possible to cleanly separate behavior and divide a problem into subproblems. Behavior is often *antimodular*. This is particularly apparent when the *real world*™ is involved. Consider a home automation system with a heating component, a ceiling fan, and a component to open windows. We could specify the behavior of each controller and reason about how each component interacts with the environment, but the actions of components may influence each other through physical processes in the environment (e.g., heated air moves through the ceiling fan to the open window). Components may also compete for the same resources, such as electricity or human attention. To truly understand how the components behave in concert, we would need to fully understand the environment, in addition to the actual components. Even if we could model the environment (room layouts, thermodynamics), we could not reason about components individually, but only about the system as a whole. As feature-interaction pioneer Michael Jackson framed it: *“the physical world has no compositionality.”*

A key insight from the study on feature interactions is that, even when systems are complex, *we decompose them anyway*, and, necessarily, make simplifying assumptions that may not actually hold. Humans simply cannot deal with complexity beyond a certain scale---we do not have the *cognitive capacity*. We need to abstract and decompose. We may make simplifications and create specifications that are weak or even wrong for specific cases. We may do this intentionally for good reasons, hoping that we can resolve issues at composition time.

The good news is that, *most of the time*, a divide-and-conquer approach pays off and imperfect decompositions work. For example, most home-automation components work well together, most of the time, and control mechanisms can often dynamically adjust for unanticipated interactions. The problems that reach the surface are the remaining unanticipated interactions that surprise us.

5 Coping with Feature Interactions

Feature interactions have been actively studied, at least, since the 90s. The community has learned how to build systems that work reasonably well despite weak or even partially wrong component specifications. Dedicated analysis, design, and control techniques can anticipate and compensate for some modularity violations.

While not a silver bullet, it is likely that there are insights from a long tradition of coping with feature interactions that may help us to better understand and build systems composed of multiple ML components and traditional components. In both worlds, we explicitly deal with modularity and composition problems stemming from weak or missing specifications. Let’s look at four strategies to manage feature interactions.

Detection through testing. It is widely recognized that unit testing or component verification are clearly not sufficient, even in traditional software systems, but that *integration testing* and *system testing* are important too. This observation holds also for systems with ML components, which is just another reminder to evaluate the entire system (often in production) and not just evaluate prediction accuracy of individual models.

Detection through better specifications. A long history of research on feature interactions has shown that better requirements engineering can help to anticipate interactions, often through systematic inspection of potential interaction points, or model checking specifications in concert. Even weak specifications can be useful to detect problems through inspection, such as *goal models* and *resource models*, e.g., to analyze whether multiple home automation components might compete for electricity or human attention. Nhlabatsi et al. provide a concise overview of common kinds of conflicts and different kinds of interactions that can help to guide an inspection [4]. In an ML setting, we may be able to reason to some degree about goals, resources, and maybe even weak specifications to detect certain kinds of interactions, especially if models interact through the environment and shared resources. However, given how hard it is to provide even weak specifications of ML components, leveraging system design might be a more promising solution.

Design for interactions. Likely the most powerful strategy in addressing the feature-interaction problem is to prepare for the possibility of interactions as part of the *system design*, designing the system to (1) prevent certain interactions by *isolating* components and (2) prepare for *resolving* interactions when they eventually occur.

Isolation is a common design strategy to shield components from each other, such as Android apps that cannot access each other's internal state or files. However, *full* isolation is rarely desired, as components should often work together to achieve a goal. Designs will therefore typically allow specific kinds of interactions, but often require to use permitted and possibly controlled communication channels; the system can intercept, modify, or block messages at runtime if it serves the system specification (e.g., not making phone calls without the corresponding app permissions).

Once an interaction has been detected, it can be resolved with additional *coordination logic*, such as one component overwriting the behavior of another. Anticipating the need for resolution as part of system design, will make it easier for developers or end users to resolve interactions when found. If we anticipate that interactions may happen, clever system designs can automatically select *default resolutions*, that is, design the system to handle *unknown* interactions gracefully. In several domains, automatic *domain-specific default resolution* mechanisms have been successful: For example, in self-driving cars, multiple components (cruise control, emergency braking, map) may provide possibly conflicting suggestions for the target speed, and anticipating such conflicts, the system can be designed to resolve such conflict by *always* picking the lowest (safest) suggested speed [5]. In Android, the system is designed to ask the user which of multiple apps should open a link if a conflict is detected at runtime. It is important to note that these strategies usually need to be designed for a specific problem, which is their strength and weakness at the same time: The system can resort to resolutions that leverage domain knowledge, but it is difficult to transfer this type of solution to other domains.

Systems with multiple ML components usually already naturally isolate the models and communicate through messages where resolution can be focused. *Data fusion* can be considered as a resolution strategy that is already common in many system architectures. The fusion strategy can either be defined manually (like picking the lowest speed) or learned with another ML model. In a sense, the ranking component of our initial image captioning scenario can be seen as a domain-specific fusion mechanism that combines outputs of object detection and language model, trained on task-specific data. The ranking component is carrying out coordination logic to resolve interactions that have been learned itself!

We suspect there are many opportunities to think very deliberately about communication channels and formats to restrict the kind of data exposed from models and more importantly data fusion steps to define or even learn default resolutions for interactions.

6 A Call for Systems Thinking

Even though the lack of proper specifications may make ML models appear special as compared to traditional software systems, there are many parallels around how to design a system to anticipate interactions with a healthy dose of system thinking. That is, we need to focus on *system* design, not just the design of ML model architectures.

The key point is to realize that decomposition without perfect modularity is okay. Decomposition is a best-effort approach, but we need to anticipate interactions, prepare for them as part of the system design and development process, and make feature interactions a first-class concern to reason about them when they occur. We need to embrace design methods of managing interactions, and machine learning itself may provide a powerful tool in our toolbox for learning feature-interaction-resolution strategies.

At the same time, it is worth exploring what kind of specifications, however partial, we can provide for ML models. Describing goals of models or assumptions made in training data selection or modeling can help to reason at least partially about compositions. Recent adoption of more structured documentation such model cards [6] and datasheets [7] can provide inspiration for providing structured and possibly even machine-readable information about models.

An extended version of this article can be found in [8].

Acknowledgement: Apel's work has been funded by the German Research Foundation (DFG), Grant 389792660, as part of TRR 248 -- CPEC; see https://perspicuous-computing.science/.

[1] H. Fang *et al.*, "From captions to visual concepts and back," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1473–1482.
[2] B. Nushi, E. Kamar, E. Horvitz, and D. Kossmann, "On Human Intellect and Machine Failures: Troubleshooting Integrative Machine Learning Systems," *AAAI*, vol. 31, no. 1, Feb. 2017, [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10633.
[3] Z. Peng, J. Yang, T.-H. (peter) Chen, and L. Ma, "A first look at the integration of machine learning models in complex autonomous driving systems: a case study on Apollo," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event, USA, Nov. 2020, pp. 1240–1250, Accessed: Mar. 18, 2021. [Online].
[4] A. Nhlabatsi, R. Laney, and B. Nuseibeh, "Feature Interaction: The Security Threat from within Software Systems," *Progress in Informatics*, pp. 75–89, 2008.
[5] C. Bocovich and J. M. Atlee, "Variable-specific resolutions for feature interactions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong, China, Nov. 2014, pp. 553–563, Accessed: Mar. 18, 2021. [Online].
[6] M. Mitchell *et al.*, "Model Cards for Model Reporting," in *Proceedings of the Conference on Fairness, Accountability, and Transparency*, Atlanta, GA, USA, Jan. 2019, pp. 220–229, Accessed: Apr. 10, 2021. [Online].
[7] T. Gebru *et al.*, "Datasheets for Datasets," *arXiv [cs.DB]*, Mar. 23, 2018.
[8] C. Kästner, E. Kang, and S. Apel, "Feature Interactions on Steroids: On the Composition of ML Models," *arXiv [cs.SE]*, May 13, 2021.

Prof. Dr. Sven Apel holds the Chair of Software Engineering at Saarland University & Saarland Informatics Campus, Germany. Prof. Apel received his Ph.D. in Computer Science in 2007 from the University of Magdeburg. His research interests include software product lines, software analysis, optimization, and evolution, as well as empirical methods and the human factor in software engineering.

Christian Kaestner is an associate professor in the School of Computer Science at Carnegie Mellon University. His research includes studying the limits of modularity and complexity caused by variability in software systems, sustainability of open source, and how software engineering changes when machine learning is inserted into the mix

Eunsuk Kang is an Assistant Professor in the School of Computer Science at Carnegie Mellon University. His research interests include software specification and verification, modeling, safety, security, and cyber-physical systems.