WASP Software Engineering and Cloud Computing
Module 2, Assignment 2

Leonard Papenmeier
(leonard.papenmeier@cs.lth.se)

1 My Research Area

My research topic is high-dimensional Bayesian Optimization. In particular, I am focussing on the optimization of high-dimensional black box functions. Such functions are assumed to be expensive-to-evaluate such that no gradients can be approximated. Furthermore, the function may be non-convex and can have multiple local minima. A common example for such a problem is hyperparameter optimization in machine learning. Here, the output of the function is the validation error of a machine learning model. The inputs to the function are the hyperparameters of the model, for example learning rate, dropout rate, and many more. Another example is neural architecture search where the inputs to the function are the number of layers, number of nodes in a layer, etc. While Bayesian Optimization is a general framework, I am focussing on Bayesian Optimization with Gaussian process surrogate models. Here, the goal is to find a probability distribution of the unknown function that is as "accurate" as possible. This involves evaluating the function at chosen points and updating the posterior distribution based on these observations. High-dimensional Bayesian Optimization is slightly more involved than this general approach but the general idea is similar.

2 Three Topics from the List

When working on a new algorithm, good research practices in the field of machine learning include publishing the source code. In fact, top-tier conferences encourage authors to submit the source code such that the results can be reproduced by reproducibility reviewers. When publishing the source code of an algorithm, one should make sure that the source code behaves as intended. One way of reducing software errors drastically is to use automated software testing. Automated software testing can be subdivided into different disciplines. For example, complex software products not only require unit tests, where one tests for small parts of the code whether it behaves as expected, but also more global forms of testing. If a software product consists of multiple modules, the interplay between these modules should be tested by integration tests. Examples are model-view-controller (MVC) or microservice architectures. Complex web-based applications usually require different testing methods. For example, the backend needs to be tested with unit tests, the frontend with front-end testing tools where one tests whether the fronted is usable with different web browsers. In academic settings, my opinion is that unit tests are usually sufficient, and it would help the field if they were more commonly used. In unit tests, the goal is to make sure that small parts of the code behave as expected. The general idea is that if every part behaves as expected, the overall program does as well. In

object-oriented programming, the common approach would be to test whether every method of a class returns the correct output for a given input. Unfortunately, many methods in scientific programming take arbitrary many discrete or real values. Therefore, testing for all possible inputs is generally impossible. One way of circumventing this problem is to identity finitely many "equivalence classes", i.e., sets of inputs where the behavior of the program does not change. Then, it is enough to test the method for each equivalence class. Additionally, it is often useful to test the boundaries of a equivalence class, e.g., the smallest and largest values in an equivalence class as errors often occur at these boundaries. In my opinion, functional programming facilitates unit testing. In object-oriented programming, methods can have side effects: when the state of an object changes, the output of a method can change for the same input. Functional programming forbids this. While pure functional programming has become rare, especially in business applications, I would argue that using as much functional code as possible is a good idea for scientific applications as functional programming is often a natural paradigm for such applications. Something similar can be achieved by using as many static methods as possible in object-oriented programming languages.

Requirements engineering (RE) is the discipline of identifying and documenting the requirements for a software product. This is often done by stakeholder analyses where, e.g., future users of a software product verbalize their requirements. Many more groups can be stakeholders, for example developers and all other persons that are connected to a software product. While stakeholder analyses are probably not very common for implementations of new algorithms, RE also includes the modelling of the software. I have seen often that implementations of new algorithms follow bad coding practices, probably because the algorithm was not known beforehand and code had to be changed during the development of the algorithm. Even though RE usually assumes that all requirements can be identified beforehand, which may be unrealistic in the beforementioned setting, many RE practices could improve scientific code. For example, once the algorithm has been worked out, developing a model of the implementation and changing the existing implementation to follow the plan could make code much cleaner and more easy-to-use for other researches working with that code.

A lot of scientific software is implemented in Python and many algorithm implementations of researchers not working at FAANG companies are not well maintained. One problem I see often when working with code from other researchers is that it relies on other Python packages that might change over time and break the code of the primary software. The main problem here is that dependencies are not well-defined. When installing new Python software, there is usually a list of other packages the software relies on. If the versions of the dependencies are not well-defined, the software might break even though its code did not change at all. Another occurs if the dependencies do not follow semantic versioning, i.e., the behavior of a package changes with a new patch version. Many problems with maintenance and evolution of software could be solved by encouraging developers to follow semantic versioning when specifying dependencies and when publishing a new version of a package.

3 Future Trends of Software Engineering

Many recent breakthroughs in machine learning have been achieved by models with an enormous number of parameters. For example, the largest version of GPT-3 has 175 billion parameters and training it on the cheapest cloud GPU would cost millions of dollars.[1] Training DALL·E

[1] https://lambdalabs.com/blog/demystifying-gpt-3/

would cost over 100,000$.[2] Even the development of the algorithm for my latest publication (coming soon!) required multiple thousand core hours. The reliance on large computational budgets has made the field of machine learning less democratic. While other fields like particle physics are also not possible without enormously expensive resources, many recent publications and ML products came from big companies like Meta or OpenAI. If publicly funded research should be able to compete with these companies, this requires considerable future investments in data centers. For example, training the largest version of GPT-3 on the NSC Berzelius cluster would still take months when occupying the entire cluster. Certainly, developing a model like GPT-3 requires more than training it once. Usually, during the development of machine learning models, multiple training iterations are required to end up with a usable model. During my own research, I have often experienced that many expensive computations turned out to be useless due to bugs in the code or conceptual errors.

To be able to compete with big players, we do not only need access to big compute clusters but also more careful development of machine learning software. I think that proper software testing tools for ML can help in reducing training costs by avoiding unnecessary errors. However, backing ML by more theory can also help as it would be known what can and cannot work during the development of ML models.

However, I cannot see how ML becomes just a another part of software engineering. The main difference I see is that ML but also other algorithms are often inherently probabilistic. This makes classical unit testing difficult as unit testing often relies on deterministic behavior of units. While there are certainly parts of the code that can and should be tested with unit tests, testing the overall behavior of a model is different from the way classical software products are tested. There are probably many ways in which insights from classical SE testing can influence ML testing and the other way round. Nevertheless, I think that the two fields will not grow together completely.

[2] https://app.subsocial.network/@creativeai/the-impact-of-dall-e-on-creative-work-610