WASP Software Engineering – Assignment Marcus Häggbom

My main areas of research are statistics and machine learning. I study financial time series and methods of approximating the underlying distributions and how to sample from them. Being an industrial PhD at SEB, the research questions stem from risk management in the context of portfolio management. This requires modelling of the distribution of the time series. The constraints imposed from the application is that the data available is not abundant and the models need to be interpretable in order to be useful in practice. With this in mind, the project studies data-driven machine learning models that have analytical properties unlike deep learning. The models in focus are energy-based generative models. The energy function is designed and not learned, given the limitation of data. Ways of sampling from this distribution is also studied. The samples are then used to evaluate the theoretical model with respect to the data. This requires handling data, coding, performing experiments and keeping track of results for different configurations of hyperparameters and so on. Thus, the research is a mix of "pen and paper" calculations and programming the implementation.

Since such a big part of the project involves writing code, it is important to abide by good coding standards, such as the *Pragmatic Software Engineering "Rules"*. By using git as a version control system, the temporal aspect of the development of the code is handled efficiently. The project usually unfolds as a stream of different ideas that are implemented and tested, so each idea becomes a "mini-project" with a corresponding separate branch. When the idea has been implemented, this branch is merged into the main branch. This, together with descriptive commit messages, results in a useful historization of the project, as well as enabling reversibility. It is also important for the code itself to follow these pragmatic rules. Of particular importance is *orthogonality* and *ETC* (Easier To Change). The nature of the project is such that additional features in the code are constantly required; this can range from something small like the introduction of a new hyperparameter, to something bigger like enabling hyperparameter optimization. The first versions of the code did not do this very well, and there came a point when a major refactoring was performed to make the code more suitable for this iterative workflow.

The coding exercises performed in the project are very much in the category of machine learning, rendering the testing mainly focusing on validation as opposed to verification (which is certainly still useful, such as metamorphic testing, e.g. asserting dimension/size of tensors). In the example of generating data, the focus of the project is naturally to attempt to generate data that is as close to the real world data in certain aspects. Thus, it is a problem of making sure the "right" (in some sense) model is made. This kind of analysis is made by evaluating different similarity metrics. It is then important that these cover a variety of characteristics that are central in the context of where this generated data is being used. At the same time, when using these measures as a proxy for the generative model being "right", one has to take Goodhart's law into consideration, suggesting that "when a measure becomes the target, it ceases to be a good measure".

On the topic of DevBots from Linda Erlenhov's lecture, there are three tools in particular that have a major impact on the productivity in the project when it comes to coding. The first is a good linter that is built in to the IDE (in this case PyCharm), which helps with rudimentary things such as typos but also predicting

some bugs, for instance when used together with type hinting. The second and third are GitHub Copilot and ChatGPT, the latter being used for getting clarification on certain libraries (like an interactive documentation) and design patterns. However, Copilot has probably had a greater impact, partly because of its integration into the IDE. The usefulness of Copilot becomes particularly prominent when evaluating it in the framework of SPACE. Besides increasing activity, when it works well it also removes friction, resulting in longer periods of flow. In addition, after learning its quirks, it can feel like a natural extension of you when you code, which enhances satisfaction and wellbeing.

From Fabio Calefato's lecture, the AutoML tool DataRobot which was demoed seemed interesting and it (or some similar tool) could be useful for imposing a better structuring of the experiments as well as increasing the research intelligence. Visualization is an important component when evaluating and comparing model results. It can almost be framed as a meta learning problem, where the system helps to find good models based on their result. In addition, it could help with small quality-of-life improvements, such as automated parameter sweeps.

Maintenance and evolution of ML software is a field in software engineering which characterizes challenges that arise on the practical side of the project, and provides useful methods for dealing with some of these issues. Maintenance of ML software differs from maintenance of traditional or non-ML software in multiple ways, a core reason being the ML software's reliance on data. Maintenance of an AI system therefore also requires maintenance of data, e.g. by versioning the data [1]. Due to the experimental nature of developing AI systems, often characterized by a fast-iteration workflow, it is also often necessary to version the model configurations, including model architecture and hyperparameters. There exist MLOps tools that attempt to facilitate this process [1]. Another aspect of maintenance is the observation that many ML models suffer from the "change anything, changes everything" principle, which renders some traditional methods used in non-ML software development for reducing coupling ineffective [1]. This principle, however, still necessitates low coupling, perhaps even at a level similar to non-ML software, as pointed out in [4]. In this empirical interview study, some of the interviewees claim ML as opposed to non-ML software requires *less* effort to maintain. Since performance of these AI systems can drift, they should include automatic maintenance to ensure consistency and robustness. This kind of automated evolution of the software is also discussed in [1], but there the level of autonomy and absence of a human in the loop is questioned.

Dealing with the maintenance and evolution of my code in my research is something that I put a good deal of time and thought into, considering both the ML and non-ML aspects. The experimental nature of the project requires flexible code of which modularity and decoupling is a big part. I often have to add new functionality which I had previously not considered. When it comes to the ML side, the number of ways to design models and configure hyperparameters is so staggering, requiring managing the experiments systematically. I have considered making use of SE4ML tools for this purpose, such as *Weights & Biases*, but I currently use my own simple framework for versioning, making it easily configurable but perhaps not so powerful. Also, by not using one of these management tools, I do feel like I'm missing out on the visual component in some regard, e.g. having dashboards providing condensed

overviews of the results of the experiments. Another issue with tracking experiments is model versioning. One way of dealing with this mentioned in [1] is of versioning the source code, reproducing old experiments by checking out the code from the corresponding timestamps. Simplifying a bit, I only have to store the hyperparameters from my experiments, which allows me to structure the code in such a way that an old experiment configuration works with the new code. This setup has its caveats, of course, for instance having to be meticulous about not breaking backwards compatibility. (One way to verify that this doesn't happen would be with automatic testing of previous experiments.)

Human-Computer Interaction (HCI) is a rich and multifaceted, cross-disciplinary field that combines sciences tied to humans, such as communication and psychology, with computer science. Browsing through the different sessions from the 2022 Conference on Human Factors in Computing Systems (CHI)[1] gives a perspective of the vastness of the field, with topics like accessibility, user simulation, gaming and medical applications. To limit the scope, I will stick to recommender systems and cognitive modelling, which I think could be relevant in the extension of my project in industry. Being an industrial PhD at a bank, one potential use case that could benefit from my research is recommending hedging strategies for traders. My research lies in methods for calculating risk, which can in turn be used as basis for hedging decisions. These suggested hedges could then be presented to the traders, leaving the trader fully in charge but with a "trading copilot" at their side. In this use case, well-designed HCI is paramount; no matter how good the risk model was, the traders would not use the copilot if the interaction was not user-friendly.

Methods for modelling the cognitive states of the user are described in [3], where the process is divided into two parts: the modelling itself, and estimation of the parameters of the model. It is pointed out that there is good progress being made in research of models that are rich enough to encode high complexity behaviours, taking into account e.g. user capabilities and interests, which is required for real-world applications. On the other hand, the inference part is lagging, thus being the authors' focus. I can see having a cognitive model of the user as being advantageous in the context of a trading copilot system, e.g. for inferring the trader's risk preference (in general) and risk appetite (contextually contingent).

Recommender systems model their users and monitor the context in order to provide suitable suggestions of possible actions for the user to take. The recommendations are learned by observing the user's historic actions. The recommendations could be prompted by the user, or pop up automatically when the system anticipates that the action(s) could be useful to the user given the current context [2]. These kinds of systems are very much intertwined with AI systems, often being trained with reinforcement learning. A challenge pointed out in [2] is that many recommender models, when trained on pre-existing data sets, are unable to incorporate knowledge about how the suggested actions affect the context. Having a model of how the recommendations affect the real world would be important in a trading copilot system. Another perspective that [2] brings up is that of privacy: A good recommender system is tailored to each used, which is made easier by collecting more data about the user. Herein lies a trade-off that needs to be considered when designing the systems.

[1] `http://st.sigchi.org/publications/toc/chi-2022.html`

Looking forward, and in general, I think that we will continue to see the expansion and adoption of these performance-enhancing tools, both within SE4AI and AI4SE. In the former case, I think this will lower the bar for many legacy industries in the process of making their business more data-driven, but also make them realize that simply storing data is not enough and that they have to invest more in data cleaning and maintenance. For the latter, I think GitHub Copilot will be used to write more code and more tests, leaving the developers spending more time on design choices and reviewing code. On the note of copilots, it seems likely that these will become more common in other places than your IDE (thinking of e.g. Microsoft and Clippy 2.0, also online retail, etc.), and that they will be personalized, putting high demands on HCI. It will also put high requirements on unbiasedness. Here, I think we will see two diverging patterns when it comes to what we require from the AI systems: For the first class of systems, including e.g. LLM's, I think people will overlook the faults and biases of the systems because the models are so good, even though these flaws could be (to some extent) harmful. For the second class of systems, these will go in the other direction, where requirements on fairness and explainability will be very high, especially where it has significant impact on individuals' lives, such as medical applications or automated sentencing in the juridical system. It is in this category that I think that the banking industry will primarily go. It will also be interesting to see, when legacy industries where the core product is not ML-related have gotten some experience in working with data and modelling, what role ML will have in their business strategies. One possible outcome is that some of these companies will realize how much effort it takes to have these systems running, when it comes to testing and maintaining and all other parts of the life cycle, that the gain doesn't cover the costs. Another possibility is that we will get increasingly capable "AI4AI" tools, cutting these costs instead.

References

[1] Görkem Giray. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software*, 180:111031, 2021.

[2] Giulio Jacucci, Pedram Daee, Tung Vuong, Salvatore Andolina, Khalil Klouche, Mats Sjöberg, Tuukka Ruotsalo, and Samuel Kaski. Entity recommendation for everyday digital tasks. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 28(5):1–41, 2021.

[3] Antti Kangasrääsiö, Kumaripaba Athukorala, Andrew Howes, Jukka Corander, Samuel Kaski, and Antti Oulasvirta. Inferring cognitive models from data using approximate bayesian computation. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 1295–1306, 2017.

[4] Zhiyuan Wan, Xin Xia, David Lo, and Gail C Murphy. How does machine learning change software development practices? *IEEE Transactions on Software Engineering*, 47(9):1857–1871, 2019.