Software Engineering, ML and AI Reflection

Vincent Szolnoky

Introduction Due to my field of research being within applied statistics the topics relating to software engineering are not entirely relevant. Instead I will try to discuss these topics on their own without connecting it specifically to my own research as their is no obvious connection.

Automated Software Testing Software testing is sometimes a very misunderstood concept. Too often it used a simply a means to ensure that new code that is written fulfills it purpose which can be self fulfilling prophecy. Instead it's main purpose is to ensure that regressions aren't made and that code that works is not broken. To ensure this, automating software testing is a important step as it removes human laziness and bias. Testing should not be able to be skipped or easily bypassed and thus where automating it helps alleviate this.

A common problem with many code-bases is that they are written in such a way that does not allow efficient tests to be written and thus being excused from automated testing. The code is often not written in a modular way and components are not easily mocked. This means that to test it often requires reproducing a fully functioning deployment environment which can be a hefty task. Therefore, from the start, code needs to be written in a modular fashion where majority of the tests are unit tests that evaluate small components independently and a small number of functional tests that ensure overall functionality is met.

Additionally a new challenge is how to test ML code where the complexity lies not in the semantics of the code but what the code is actually implementing. For example, a machine learning model might be implemented in which the output lies in an infinite space but in reality should only be a specific subset. Semantically it will not be obvious from the code that this should be the case and requires a extra level of abstraction to test that the model should and should not do.

Continuous Deployment Continuous deployment (CD) is a practice common in software engineering where code is continuously deployed so that updates and testing happen are more in sync with development. While beneficial for traditional code bases, I see a larger benefit for new ML solutions. The reason for this is that ML code often involve other moving parts such as parameters and data. Just because the code works does not mean that the model itself will and therefore being able to continuously evaluate code, parameters and data together is important. Additionally a common use case is to test different variations of models where being able to rapidly deploy them in parallel is key. This enables one to do things like A/B testing between them more efficiently.

While there exist many good tools for CD of traditional software, such as Jenkins, I think there is a need for similar tools that are adapted for ML software. These, for example, should also take into account parameters and data as part of the deployment procedure.

Architecture and Design Functional programming has historically been a bit of a pariah when it comes to software development, especially within industry. However with the advent of data science, it has seen a rebirth. Many data science projects are written as pipelines that stream input data through multiple steps to produce a final output. Each of these steps are written as a functional piece. This also allows efficient automated testing as each step can be tested independently.

Projects such as Airflow and Spark are widespread in industry for implementing pipelines for example. While new concepts such as server-less computing also rely on concepts from functional programming. I would hope to see this spreading further where code bases exist simply as a collection of common functions and an overarching tool that simply orchestrates them together.

Software engineering together with machine learning (ML) and data science workflows I don't believe that software engineering will entirely absorb ML or vice-versa but instead they will complement each other. SE is well adapted to solving problems where the environment, constraints, rules, etc are well known. ML and data science however is useful when unknowns are involved and as such require modelling, imputation, predictive power, etc. Moving forward however they will need to deeply integrate with each other and therefore the line between might be blurred.

A general use case will be for ML systems to be built on-top of software systems and the data they generate. That's, however, where I see a huge gap as ideally for ML systems to work well they require large amounts of clean, feature-rich data. Most software systems are very sloppy however when it comes to capturing and storing data as they are mostly only interested in the explicit value of it and not the potential intrinsic properties. By this I mean that often data is stored based on how it is used in the software itself and not how it could be potentially used for other purposes such as ML. An example is storing timestamps without timezone information as I have seen in various software solutions due to the timezone not being necessary for the application itself. This removes a lot of information from the data that would otherwise have been very useful in a ML solution.

Another pitfall I see developing is using ML solutions in places where it is really not required. Often times ML methods, such as neural networks, come across as plug-and-play solutions that simply can perform any task given enough data. There are however many other considerations to be made that are often overlooked. Model certainty is a big one as it can be difficult in instances to deduce what the model will actually do and if it will provide expected results all the time. Software solutions on the other hand are by nature almost completely deterministic and easily debugged. I think it should be much more common practice to analyse problems from first principles and carefully deduce, with all factors weighed in, whether a software or ML solution would be more applicable.