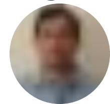
A Model For Technical Debt In Machine Learning Systems



Soumen Sarkar | August 30, 2022 at 5:45 pm

Machine Learning (ML) is a type of artificial intelligence (AI) that allows software applications to become more accurate at predicting outcomes without being explicitly programmed. Machine learning algorithms use historical data as input to predict new output values.

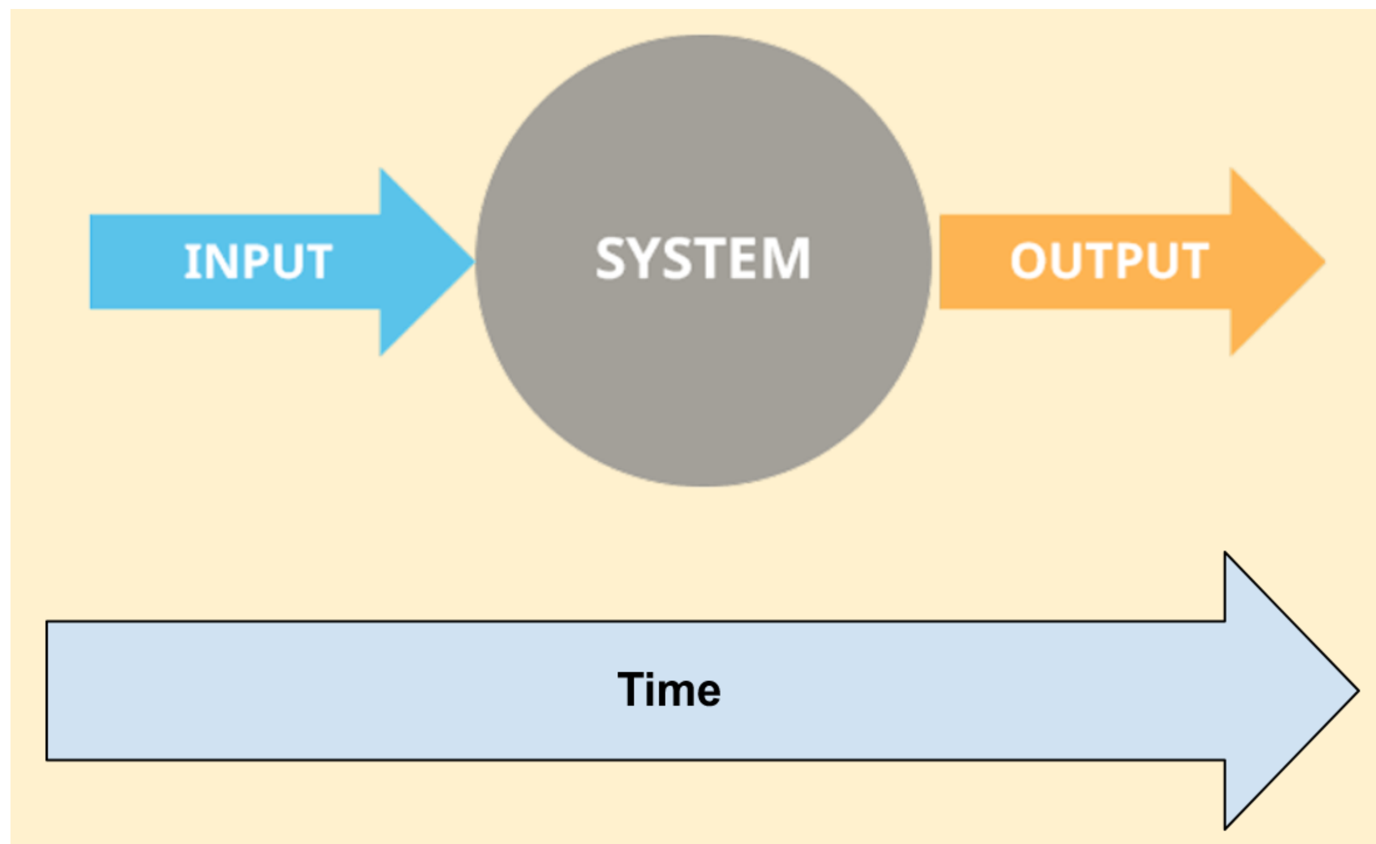
Technical Debt describes what results when development teams take conscious actions to expedite the delivery of a piece of functionality or a project which later needs to be remediated via refactoring. In other words, prioritizing speedy delivery over perfect code is the result.

This article will present a simple yet powerful Model of Technical Debt for Machine Learning Systems. The model is simple to remember, easier to extend, and provides a reliable means for reliable and maintainable Machine Learning Systems. This, in a nutshell, is the value proposition

reliable and maintainable Machine Learning Systems. This, in a nutshell, is the value proposition of this post.

Model

The model for an ML System is quite simple and generic. For Technical Debt, the dimension of Time is a major consideration since unmanaged debt becomes a debilitating impediment to progressive velocity. ML systems act on Input to produce output based on machine learning theory and practice.



Generic Model Of Machine Learning System

From this, we may draw the following System Behavior expectations as Time passes:

- We expect the System to produce similar Output when presented with similar Input – **no surprises in Output**
- We expect to change the System incrementally to accept different Inputs to produce new Output as per certain knowledge of the domain – **Incremental Refinement**
- We do not expect to be overburdened by the System Complexity or Understandability in our quest to map Input to Output – we expect the System to be a **Maintainable System**

We say that a System is loaded with Technical Debt when any or all of these expectations do not hold over time as we seek Return On Investment (ROI) in the System by deploying it in production. The accumulation of suboptimal decisions starts holding the System back. This phenomenon calls for systematic and scientific management of accumulated Technical Debt.

Python Technical Debt: Global Interpreter Lock (GIL)

What could be a very good example of Technical Debt in the universe of machine learning?

Look no more beyond Python. Python interpreter has a Global Interpreter Lock (GIL). The GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously. This lock is necessary mainly because CPython's memory management is not thread-safe. This holds back concurrency design in Python, leading to wasted time waiting on the global lock. The situation is serious enough to motivate Microsoft to entice Python BDFL (Benevolent Dictator For Life), Guido Van Rossum, to come out from retirement. Back in 2018, Python Benevolent Dictator For Life Guido van Rossum retired, but a couple of years later, he changed his mind and returned to work – at Microsoft.

Why is Python GIL a Technical Debt? It is so since this decision was taken consciously, which takes us to the next section. This Technical Debt is threatening Python's future since CPUs are inherently multi-core due to power density issues. Python cannot take advantage of multiple cores efficiently due to GIL.

Technical Debt Registry

It is very important to recognize that the act of assuming Technical Debt should be a very **conscious** one. Following is a list of examples of items that we do not consider tech debt:

- We didn't deliberately make errors in design as part of the trade-off to speed time to market (impacting scalability, resiliency, user experience, etc.)
- Errors in coding/defects Inadequate documentation Ignorance of the right way(s) to do something
- Failure to unit test UNLESS it was a conscious decision to speed time to market

TEMPORARILY.

As a result, it is nonnegotiable to require that all Technical Debt must be consciously maintained in a Technical Debt Registry. Whatever is not present in the registry is not under consideration for planning and budgeting exercises.

To summarize: **Neglect Debt is not Technical Debt.** Neglect Debt (not keeping up on maintenance, software patching, etc.) is an act of omission – Tech debt is an act of commission we knowingly make in favor of faster time-to-market with a plan to address later.

The practice of maintaining a registry for Technical Debt is not new or novel. There is already established practice of maintaining a registry for Architectural Decisions – it is known as:

Architectural Decision Records

An Architectural Decision (AD) is a software design choice that addresses a functional or non-functional requirement that is architecturally significant. An Architecturally Significant Requirement (ASR) is a requirement that has a measurable effect on a software system's architecture and quality. An Architectural Decision Record (ADR) captures a single AD, such as often done when writing personal notes or meeting minutes; the collection of ADRs created and maintained in a project constitute its decision log. All these are within the topic of Architectural Knowledge Management (AKM).

Technical Debt is an Architectural Decision (just like Python GIL), and it is no surprise that system development and maintenance best practices require a dedicated registry for Technical Debt.

Technology Delta

There is another viewpoint, known as *Technology Delta*. Your technical delta is the difference between your systems' internals today and your systems' ideal internals for that same functionality.

I like to pair it with another term—the functional delta of your systems, which is the difference between what your systems currently do and what they would ideally do to move your business forward (think workarounds and new capabilities the business needs developed). The functional delta is apparent to people who use the systems; the technical delta is only apparent to those looking at their internals.

What risks and limitations does your current technology pose? I'm talking here specifically about the internals of your systems—the functional limitations are already addressed by the questions above. Are you using an operating system version that's about to be deprecated? Are there bugs in your code that are likely to disrupt business operations? Is your availability adequate? Is your code written in a programming language that no one knows anymore? Will your infrastructure scale to meet next year's plan for increasing market share? How will your current technology hold you back? What will you have to spend money on in the future? These are aspects of the technology asset that pose risks or affect future cash flows.

The second part of this article will do a deep dive into the Technical Debt Analysis of machine learning systems.