Hidden Technical Debt – Project Critique

Submitted by Richa Yadav (Group 05)

Project Overview:

This project critique report is focused on discussing the different factors that lead to technical debt in Machine Learning systems. In the field of software engineering, technical debt is the unforeseen cost of hefty reiterations required when deploying a simple/quick but insufficient solution instead of a better approach that could take more time. The overall resources and time required to "re-pay" this technical debt can be daunting (sometimes fatal too) for organizations. Analogous to technical debt in software engineering, technical debt can accumulate in a modern Machine Learning (ML) system too. The authors of this paper uncover the different types of risks that ML engineers should account for before developing and deploying pipelines in live environments.

Objectives:

- To report a short summary of the different factors that lead to technical debt in ML systems.
- To report how our group's data architecture and system flow are designed to mitigate technical debt.

1. Overview of the causes of technical debt in ML:

- The authors argue that an ML system is much more vulnerable to technical debt than a traditional software stack. This happens as gaps can appear at a code or even at a system level. The authors highlight 7 different kinds of technical debt that developers must be wary of as illustrated below:
    - **How complex models erode boundaries** – Traditional software can be easily isolated and maintained by enforcing strict abstraction boundaries. However, it is much more difficult to perform something similar for an ML system as it relies on external data for producing results.
    - **Data dependencies cost more than code dependencies** – Identifying code dependencies is much easier by using tools like static analysis by compilers, etc. The authors state that data is prone to build up many large dependency stages that are difficult to resolve.
    - **Feedback loops** – As the definition of an ML system is that it learns from data given to it over time, it can sometimes lead to a risk called analysis debt. Analysis debt states that a model's behavior is unpredictable before it gets released.

- **ML-System Anti-Patterns –** In real-world ML operations, actual modeling and prediction is a relatively tiny portion of the entire architecture.
- **Configuration debt** – This debt is related to the efficiency of an ML system to handle additional configurations and whether there is a guideline or format in which the configurations can be tested.
- **Dealing with changes in the external world** – This debt is related to the changes in the input sources due to external factors. Any model is as good as the accuracy of the input. This debt can come in form of static thresholds instead of dynamic ones, not having a live system tracking and monitoring, etc.
- **Other areas of ML-related debt** – This miscellaneous ML-related debt can appear when there is no versioning of hyperparameters to reproduce older results. It is also noticed when a team culture rewards an employee who not only identifies technical debt in the system but also mitigates it.

2. How we mitigated the risks related to technical debt in our project:

As clearly said by the professor, our focus was entirely on building a scalable system that gives our future bike inventory. We did perform an adequate level of hypermeter tuning to ensure that the predictions are acceptable, however, our main focus was to create an intact system. Using Databricks as a platform to implement our project was a big advantage to manage various technical debts by itself, as will be discussed below.

- **Data Dependency –** Duplicating the data sources at our end will ensure that any changes made at the input level get reflected in our system without any inconsistency.
- **Abstraction –** Creating a different workbook to tackle different problems makes the code and it's working totally isolated. Managing code in this way will help to onboard any new team member.
- **Analysis debt –** We have proactively defined schemas of our bronze, silver, and gold tables with consistent datatypes for common columns. We have also commented on the different code blocks and thus increasing transparency.
- **Code level debt –** In our system, there is no block of code that can turn into a dead code. Databricks provides intelligent solutions to fundamental problems as follows
    - Infer-schema of input file - It aids in preventing any runtime errors brought on by new data columns. We can address it in the future by not demanding a few columns and not allowing the pipeline to process any data without the mandatory characteristics, even though it can be extreme when there is any change in the data type of the present columns.
    - Z-Ordering – We created a new hourly formatted timestamp and applied z-ordering for effective querying because all of our queries for historical bike data were based on windowing the timestamp at the hourly level and using some math for the resulting aggregations.

  - Shuffle partitions – Although Databricks has a 200-shuffle-partition limit by default, we never use that many. Since there was no requirement for more parallel executions than that for our use case, we configured it to be the number of cores. When creating a stream-based system in the future, we can alter it to a multiple of the number of cores.
  - Data partitions – We are storing our bronze bike informa+on based upon station id as a partition criterion. This is done to make the system scalable. Though we worked on only one station currently, so it won't give any advantage as of now, but in the future, it would allow the processing of all transformations efficiently.
- **Configuration debt -** We added all our configurations in the same format as the already provided ones, and included them in the table showing the output when "includes" is ran in each and every notebook for initial configurations. This helps to abstract any configuration-level changes to be applied in a single file.
- **Realtime debt** – We have added the functionality of promoting any model to production easily with just a button click, which can be decided easily based on the visualization and descriptive statistics provided in the "app" notebook.

- ***How easily can an entirely new algorithmic approach be tested at full scale?***
  - As previously stated, any improvement in the transformation stage can be done efficiently due to the application of data partitioning and effective querying using Z-Ordering. Additionally, any new algorithmic approach will be logged using MLFlow along with all previous modeling strategies, making it simple to advance the best algorithm to the production stage.
- ***What is the transitive closure of all data dependencies?***
  - Any new data change can be promptly updated to the silver storage using the ETL pipeline. By calling the Gold table, this can now simply help to be used in MDL and APP notebooks.
- ***How precisely can the impact of a new change to the system be measured?***
  - Impact of any change in the system can be easily measured by changes in the accuracy score like MAE or any sudden changes in processing time.
- ***Does improving one model or signal degrade others?***
  - Due to MLFlow in Databricks, managing different models based upon tags(version or timestamp-based) makes it easy to provide stable environment and not be affected by any new modeling approach being tried at any given point in time.
- ***How quickly can new members of the team be brought up to speed?***
  - New members can be brought up to speed fairly easily and quickly. As the workbooks are well-documented with their own functionality, it is much easier to understand the code and its importance in the entire tech Stack.
- Project Link: https://github.com/Rishabhkandoi/G05-final-project/tree/master