Data Science at Scale (DSCC402)

Hidden Technical Debt - Project Critique

Submitted By: Aradhya Mathur (Group 5)

Part a) Summary of the factors described in the paper that lead to technical debt in Machine Learning Systems.

The paper ***"Hidden Technical Debt in Machine Learning Systems"*** discusses several factors that lead to technical debt in machine learning systems.

The term technical debt refers to the cost of future rework that arises due to shortcuts taken during the development process. In machine learning systems, technical debt can manifest in different forms, such as poor data quality, suboptimal model selection, and inefficient deployment strategies. Below, I have tried to explain the factors that can lead to technical debt in machine learning systems, along with their implications and potential solutions.

- **Erosion of boundaries**
  - One common technical debt issue is related to the interdependencies among multiple applications. A change in one application can have a **cascading effect** on other dependent applications, causing unexpected errors and slowing down the system. This can occur when there are no strong abstraction boundaries in place to separate the different applications and their functionalities.
  - Another concern is the open access to the model and its output, which can lead to **hidden interdependencies** and potential security risks. Without proper access control and monitoring, unauthorized access to the model or its output can compromise the system's integrity and lead to data breaches. It is crucial to establish clear abstraction boundaries and access controls to prevent these issues and ensure the system's stability and security.

- **Dependency debt**
  - Data Dependencies refer to the debt related to the data flow, and there are several factors that contribute to it. Firstly, frequent changes in the source data format can lead to broken pipelines and disrupt the entire ETL process. Additionally, utilizing all data features without proper analysis can lead to increased computational complexity, which can impact the overall system's performance.
  - Moreover, not using **automated feature management systems** to detect data dependencies can lead to unidentified issues, which can cause problems down the line. It is crucial to have proper data management protocols in place, such as data lineage and data cataloguing, to ensure that data dependencies are identified and tracked throughout the data pipeline.
  - **Data security** is another critical aspect that needs to be considered when managing data dependencies. Unrestricted access to the model and its output can lead to hidden interdependencies and security concerns. Proper access controls, such as role-based access, should be implemented to ensure that only authorized personnel can access the data.
  - Overall, it is important to have a strong data governance strategy in place to manage data dependencies effectively. This includes having proper documentation, data profiling, and data quality checks to ensure that data dependencies are identified and addressed proactively.

- **Analysis debt**
  - Analysis debt is a type of technical debt related to feedback loops. This type of debt can arise when the model is learning from itself about its future training data without using any **randomization** (sampling) techniques. Without this, the model may become biased towards certain data points and may not generalize well to new data.
  - Another aspect of analysis debt is the **lack of proper documentation** for each flow. This can lead to hidden interdependencies among applications serving the same requirements, which can remain hidden until one application is being modified. This can affect other applications that are dependent on the modified application, leading to unexpected results and errors.

- **Code level debt**
  - Code level debt in ML-systems can arise due to not following **good coding practices**. For instance, blindly using generic packages or adding work-arounds to be able to use them may lead to glue code, which can become difficult to maintain over time. Not having a **clean pipeline** that is easy-to-understand and extend can lead to increased interdependencies between researchers and engineers, making it harder to accommodate any new changes that may arise. **Dead code** can also add to technical debt and should be identified and removed.
  - Following the **principles of distributed learning** appropriately can help utilize computational resources efficiently and save time, although there is a trade-off between implementation time and computational time. *Failing to use proper data types, coherent programming language across all applications, and writing experimental code instead of production-ready code* can all lead to code level technical debt. It is important to adhere to **coding standards and best practices**, to ensure that the code is maintainable, scalable and robust.

- **Configuration debt**
  - Configuration debt refers to the debt related to configuration management. It can arise due to several factors such as not having any specified rules or format to **add configurations**, not having **testing pipelines** for configurations, or not keeping **configurations updated** as the system evolves.

- **Real-time monitoring debt**
  - Real-time monitoring debt refers to the challenges posed by real-time changes in external sources. It is important to implement **self-learning thresholds** instead of static thresholds in order to take appropriate actions when the data changes dramatically over time.
  - Lack of **comprehensive live monitoring** of system behaviour with automated response and triggers or automated alerts based upon system limits can also lead to debt.
  - **Repeated human intervention** instead of real-time responses with real-time data changes can also add to this debt.

- **Other debts**
  - Lack of data testing pipelines to facilitate testing of major changes in input data
  - Absence of version control for the model, hyperparameters, and data, which makes reproducing results and debugging issues difficult
  - Absence of a team culture that values reducing technical debt in addition to improving accuracy or other success metrics.

Part b) Critique of how your project is setup to mitigate technical debt. Be sure to describe specific features of your code and the Databricks platform and specifically what technical debt those features help to mitigate.

Our primary objective from the outset was to develop a system that could be scaled efficiently. We recognized the importance of a robust platform like Databricks. This choice provided us with a significant advantage in managing various technical debts, as we shall elaborate below.

As a result of our comprehensive planning and implementation strategies, we were able to mitigate several technical debts effectively. We employed various techniques to manage these debts and successfully reduced them to a minimum level. In the following sections, we shall provide detailed insights into the techniques used to mitigate the technical debts.

Overall, our efforts have enabled us to create a scalable system that is not only efficient but also flexible and adaptable to future changes. By employing the right tools and techniques, we were able to reduce technical debts, enhance the system's performance, and ensure its long-term sustainability.

Station: University Pl & E 14 St

- **Erosion of boundaries**
  - Breaking down our pipeline into multiple modules such as **ETL, EDA, MDL, and APP**, each serving independent use cases, facilitated a more collaborative workspace. This approach also makes it easier for new team members to get onboarded quickly and understand the intricacies of the code flow.
  - By separating the pipeline into distinct modules, we created a more organized and manageable structure for our project. Each module can be worked on independently, allowing for easier collaboration among team members working on different parts of the project. This **modularity** also makes it simpler to debug and maintain the codebase since each module has its own purpose and functionality.
  - Furthermore, this approach significantly enhances the onboarding process for new team members. By dividing the pipeline into separate modules, new members can focus on one specific module at a time, allowing them to better understand the code flow and their specific responsibilities within the project. This also reduces the learning curve for new members, which can help them become productive team members more quickly.

- **Dependency debt**
  - To ensure that any real-time changes in the data do not have a drastic impact on the current system, we store all datasets at our end by creating **duplicates**. This approach provides the team with sufficient time to implement any necessary code changes to accommodate new data formats, for instance. It is worth noting that this approach comes with the added cost of storage. However, using the delta storage format helps to *improve storage efficiency.*
  - By creating duplicates of datasets, the system can operate independently of any changes in the original data source. This ensures that the system's performance and **functionality remain stable**, even when changes occur in the data source. Additionally, this approach provides the

team with more time to analyse and understand the changes before making any necessary adjustments to the code.

- While storing duplicates of datasets can increase storage costs, using **delta storage format** helps to mitigate this issue. Delta storage format only stores the changes made to the original dataset, *reducing storage requirements and minimizing the associated costs*.
- In conclusion, storing duplicates of datasets is an effective way to manage data dependencies and ensure that the system remains stable in the face of real-time changes. By using delta storage format, the approach becomes even more efficient and cost-effective.

❖ **Analysis debt**

- **Proper documentation** of the data flow helps to increase transparency and minimize the risk of any hidden interdependent issues. By documenting the data flow, the team gains a clear understanding of the data sources and how the data is being processed and analysed.
- This approach helps to prevent analysis debt, which refers to the cost of having to redo analysis work due to errors or issues that were not identified in the initial analysis. By having a clear understanding of the data flow and ensuring that it is properly documented, the team can identify potential issues early on, preventing analysis debt from occurring.
- Moreover, documentation also aids in knowledge transfer and facilitates **effective communication** among team members. When team members can easily access and understand the *data flow, they can collaborate more efficiently, leading to more effective problem-solving and decision-making.*

❖ **Code level debt**

- To avoid code-level debt, we have followed several coding principles while implementing our project on Databricks. First, we ensured that there is no dead code in the pipeline, meaning that all the code that we have written is used somewhere in the pipeline.
- Furthermore, we have set the resources efficiently for reading, querying, and storage. For example, we have used automatic **infer-schema**, which helps us avoid any run-time error due to new data columns. Though it may cause some inconvenience when there is any change in the data type of the existing columns, we can handle it in the future by mandating only a few columns and not allowing the pipeline to process any data without the mandated features.
- We have also set the **shuffle partitions** appropriately. The default shuffle partitions set by Databricks is 200, but we never utilized that amount of shuffle partitions. Thus, we set it to be the number of cores, as there was no need for parallel executions more than that for our use case. We can change it to a multiple of the number of cores in the future when implementing a stream-based system.
- To make the system scalable, we are storing our bronze bike information based upon station ID as a **partition criterion**. Though we worked on only one station currently, storing the data based on the station ID would allow processing all transformations efficiently in parallel at the station level.
- We also applied **z-ordering** while transforming the bronze to silver storage for historic bike data. Since all our queries were based upon windowing the timestamp at the hourly level and applying some maths for the resulting aggregations, we created a new hourly formatted timestamp and applied z-ordering for efficient querying.
- By implementing these coding principles, *we have reduced the code-level debt, making our pipeline more efficient and scalabl*e.

❖ **Configuration debt**

- We have taken measures to minimize configuration debt by ensuring that all our configurations are added in the same format as the ones already provided. These configurations are included in a table displayed when the "includes" command is run in each

notebook for initial configurations. This approach helps to **avoid code duplication** and ensures that standard coding patterns are followed throughout the project.

- Additionally, it allows for any configuration level changes to be easily applied to a single file, increasing maintainability and reducing the risk of errors caused by scattered configurations.

❖ **Real time debt**

- Real-time debt refers to any potential issues or areas of improvement in the system's ability to handle and respond to real-time data. In this context, while the current implementation does not include any automated tooling to make decisions based on generated results, there is scope for such tools to be added in the future.
- Additionally, the team has added the functionality to easily **promote** a model to production with just a button click, which can be decided based on the visualizations provided in the "app" notebook, including a residual plot comparison of staging and production.
- Furthermore, the team has included a **visualization** of forecasted inventory in the "app" notebook. This feature can be used as a reference to generate automated triggers for over or understocking of bikes at any forecasted duration, which is also configurable. By implementing these features, the system can better handle and respond to real-time data, and any potential issues or areas of improvement can be addressed proactively.

❖ **Other debts**

- It is worth noting that Databricks offers MLFlow, a powerful tool that allows us to manage all model versions in a file system, enabling us to easily track all model history at any given time. This feature helps with better debugging, reproducibility, and undoing any detrimental behaviour that may occur in production.

The conclusions section had several questions that should be answered, such as:

a. **How easily can an entirely new algorithmic approach be tested at full scale?**
   The data partitioning and efficient querying using Z-Ordering make it possible to efficiently implement any improvements in the transformation step. Additionally, the use of MLFlow provides the ability to track and manage all the historic modelling techniques and versions, which allows for easy promotion of the best model to the production environment.
   The abstraction of code components makes it straightforward to test individual pieces of code independently whenever a new approach is being implemented. This means that changes to the model can be made more easily without worrying about affecting the entire system. As a result, implementing and testing new algorithmic approaches in the system is efficient and streamlined.

b. **What is the transitive closure of all data dependencies?**
   The ETL pipeline provides a quick and efficient way to update any new data changes to the silver storage. This ensures that the latest data is readily available for use in the MDL and APP notebooks. The updated data can be easily accessed and used for further analysis and model development. The ETL pipeline enables seamless integration of new data, which helps to ensure the accuracy and relevancy of the models and applications.

c. **How precisely can the impact of a new change to the system be measured?**
Any new changes made to the system can be evaluated based on their impact on the system's performance, which can be measured in terms of the time taken to run all the operations or the end outcome's accuracy, such as the Mean Absolute Error. The system's performance can be monitored by tracking the metrics and analyzing them regularly to identify any issues or areas for improvement. This allows for continuous optimization and refinement of the system to ensure that it is running efficiently and accurately. Additionally, by tracking the system's performance over time, it is possible to identify trends and patterns that can inform future decision-making and improvements to the system.

d. **Does improving one model or signal degrade others?**
MLFlow in Databricks provides the ability to manage models based on tags, which ensures a stable environment and prevents any new modelling approach being tried from affecting the current setup. This allows for better experimentation and testing of new approaches without disrupting the current production system. With the ability to track and log all modelling techniques and historic versions, it becomes easier to analyze and compare different models, leading to better decision-making in terms of selecting the best model for the job. Overall, MLFlow plays a crucial role in maintaining a stable and efficient production environment.

e. **How quickly can new members of the team be brought up to speed?**
With properly documented pipelines, abstracted functionalities, managed configurations, outcome visualizations, and version control, onboarding new team members can be relatively straightforward. They can quickly understand the workflow, which would help them navigate the codebase and quickly get up to speed with the project. The documentation and version control also ensure that the new team member can review the project's history and understand how the system has evolved over time. Overall, having a well-organized and well-documented codebase makes it easier for new team members to contribute and be productive quickly.