Quantifying Interest of Self-Admitted Technical Debt in Machine Learning Projects

Mayra Ruiz
Concordia University
Montreal, Canada
m_ruizro@live.concordia.ca

Rachna Raj
Concordia University
Montreal, Canada
r_rachna@live.concordia.ca

Mohammedhossein Malekpour
Polytechnique Montreal
Montreal, Canada
mohammadhossein.malekpour@polymtl.ca

Adenkule Ajebode
Queens University
Kingston, Canada
ajibode.a@queensu.ca

***Abstract*—Technical debt (TD) in software development is a metaphor representing the long-term costs incurred from the direct and indirect implementation of short-term solutions and workarounds. Developers often acknowledge the creation of TD through source code comments, commonly referred to as Self-Admitted Technical Debt (SATD). While many studies have examined SATD in traditional (non-machine learning) software, there is limited research focusing on SATD in Machine Learning (ML) applications. Therefore, the aim of this paper is to identify and quantify the types of TD in ML that present greater resolution challenges, utilizing specific software engineering metrics, such as commit size, fan-in, lenght of code, and so on for evaluation. Through the analysis of a previously studied dataset of SATD, we shed light on the software engineering metrics that define resolved SATDs. Additionally, we examined the variations in these characteristics between different ML development types and ML pipeline stages. The results of this study enhance understanding of the particular types of ML SATD types that get resolved, highlighting prioritization needs in ML development.**

***Index Terms*—technical debt, machine learning, self-admitted technical debt**

I. Introduction

Throughout the software development process, developers face the ongoing challenge of delivering high-quality products or services. However, owing to factors such as time constraints, market competitiveness, and cost considerations [1], developers frequently encounter the dilemma of whether to prioritize expedited completion or enhance software quality. This compromise often results in the accumulation of future burdens in the development lifecycle. This situation is aptly described by the metaphor of "technical debt (TD)," which was initially introduced by Cunningham in 1992 [2].

Existing research has consistently shown adverse effects of TD, including increased costs and negative impacts on product quality [3]–[5]. Consequently, numerous researchers have identified various types of TDs [6]–[8] and the examination of how developers address TD [4], [9]–[11].

More recently, Potdar and Shihab [12] introduced the concept of self-admitted technical debt (SATD), which pertains to situations in which developers are aware of suboptimal implementations and annotate their codes with comments highlighting the inadequacy of the solution.

Recent growth in the integration of machine learning (ML) models has provided developers with tools to address insurmountable challenges, including the domains of safety-critical systems, financial fraud detection, and medical diagnostics [13], [14]. However, the rapid emergence and adoption of ML models raises concerns about the potential development of highly planned, long-term deployed machine learning software [15].

A considerable body of research has focused on the domain of Self-Admitted Technical Debt (SATD). These investigations encompass various aspects of SATD, such as quantifying it using analytics [16], SATD detection employing text-mining techniques [17], the differences of SATD in machine learning software [18], the introduction and removal of TD within deep learning frameworks [19], SATD removal strategies [20], presence of Hidden TD in ML systems [15], and refactoring of TD within ML systems [21].

However, to the best of our knowledge there is no existing work that quantifies the effort of SATD removal in the context of ML applications, the machine learning types and pipeline stages that need more effort to remove SATDs and the reasons behind it.

Therefore, this study delves into the intricate process of quantifying SATD in machine-learning projects. First, we leverage the types of ML TD identified in the literature and calculate the effort required to remove such TD in ML projects. Second, given the multiple stages of ML pipelines, our assumption is that certain pipeline stages may accumulate more debt and take more time to remove. In this case, we examine the reasons why these identified stages accumulate more debt than the others. Third, recognizing that ML projects are not entirely free from debt, we assume that some SATD in these projects might accumulate more debt than others. Thus, we identify the reasons why a certain type of SATD accumulates more debt than others.

With these results, we address the following research questions:

- **RQ1:** Which types of SATD accumulates more interest?
- **RQ2:** Why do SATDs in specific stages of the machine learning pipeline accumulate more interest?
- **RQ3:** Why do certain types of SATD accumulate the most interest?

Our study reveals that within the field of ML, the types of SATD that demand more effort to resolve are Modularity and the Data Acquisition stage of the pipeline. We observed that addressing poor modularity, characterized by weak boundaries between ML subsystems, requires significant effort and often gives rise to challenges such as circular dependencies and training models with incorrect data. In contrast, TDs in the Data Acquisition stage, though promptly addressed, typically involve fewer modifications in terms of lines of code (LOC). This suggests a prioritization by developers towards refining data inputs for enhanced model performance, often resulting in a low number of lines of code modification.

II. Methodology

The main goal of our study is to identify the effort to remove SATDs in ML systems. To achieve this goal, we analyze a dataset involving open-source machine learning repositories presented in [18] which was based on a prior study examining ML repositories [22].

The dataset contains repositories from various ML projects, where instances were generated from source code data dating back to January 2021. Topics and instances of SATD were labeled using different taxonomies to categorize types of ML SATD and the pipeline stages in which they occur. Furthermore, 856 instances of SATD were labeled, from which this study only utilized the resolved SATD instances. This approach was chosen as it provides the only means for calculating the efforts required to remove those SATDs. Consequently, this step resulted in 110 SATDs, which were further used as the basis for our analysis.

In order to compute different metrics for calculating each SATD effort, as detailed in Table II, we leverage the power of PyDriller. PyDriller is a Python framework specifically designed for mining software repositories. It offers a range of functionalities for analyzing version histories, extracting code changes, and determining code metrics. By utilizing PyDriller, we gain the ability to efficiently navigate through project histories, extracting relevant information required for our effort calculation metrics. This tool proves invaluable in facilitating a systematic and automated approach to gathering the necessary data points, ensuring accuracy and consistency in our effort estimation process. Additionally, we created a separate metric to extract the fan-in metric from each SATD file between the introduction and removal of SATD code lines. Our replication package can be found in our GitHub repository [1].

In software engineering, particularly in the analysis of code repositories, Fan-in is a crucial metric for assessing the coupling between software modules. This metric gauges the number of modules or classes that call or use a specific module or class. A high Fan-in indicates that the module is extensively utilized and plays a vital role in the software system. To calculate Fan-in for a module or class in a Python repository, several steps are involved. Firstly, the target module or class, denoted as ‘M‘, must be identified. Then, all other modules or classes in the repository should be listed. The next step is to count incoming dependencies for each module or class other than ‘M‘. These dependencies can include import statements, direct usage of functions, classes, or variables from ‘M‘, or inheritance involving a class extending a class in ‘M‘. The Fan-in is the sum of these dependencies across all modules or classes. The formula to represent Fan-in for a module or class ‘M‘, where other modules are denoted as ‘M1, M2, M3, ..., Mn‘, is given by:

$$\text{FI}(M) = \sum_{i=1}^{n} \text{dependency}(M_i, M)$$

Here, dependency(M_i, M) equals 1 if M_i depends on ‘M‘, and 0 otherwise. Additional considerations include the scope of analysis, handling transitive dependencies, and the automation of the process for large repositories. To gauge the overall coupling in a repository, various metrics can be employed. The total Fan-in is the sum of Fan-in values for all modules, providing a raw dependency count. It is calculated as:

$$\text{Total Fan-in} = \sum_{\text{all modules}} \text{Fan-in(module)}$$

Alternatively, the average Fan-in offers a balanced view by averaging these values. The average Fan-in is computed as:

$$\text{Average Fan-in} = \frac{\sum_{\text{all modules}} \text{Fan-in(module)}}{\text{Number of modules}}$$

Dimension	Name	Definition
Change	LA_S[mod]	# Lines added during the introduction of the SATD
	LD_S[mod]	# Lines removed during the introduction of the SATD
	LA_E[mod]	# Lines added during the removal of the SATD
	LD_E[mod]	# Lines removed during the removal of the SATD
Complexity	Complexity_S	Cyclomatic complexity of the modified file at the introduction of the SATD
	Complexity_E	Cyclomatic complexity of the modified file at the removal of the SATD
	Tokens_S	Total token number during the introduction of the SATD
	Tokens_E	Total token number during the removal of the SATD
	Fan-In_S	Total fan-in during the introduction of the SATD
	Fan-In_E	Total fan-in during the removal of the SATD
History	Commit count	Number of commits made to the file between introduction and removal of the SATD

TABLE I
METRICS USED TO ANALYZE THE PATTERNS OF SATDS AND ITS REMOVAL CHARACTERISTICS

[1]https://github.com/rachnaraj/SOEN-691-Project

Dimension	Name	Definition
AWR	Machine Learning Knowledge	Machine learning software carries plenty of unique challenges. Uneducated solutions by unaware developers may have to be revisited
	Model Interpretability	Machine learning models are a black box, causing poor understanding of model's functionality. This can lead to unknown behavior.
RDB	Model Code Comprehension	Model code carries extra legibility concerns that do not occur in traditional software. (i.e., poorly named temporary matrix variables).
DCE	Duplicate Model Code	Code duplication frequently occurs in model code.
	Duplicate Feature Extraction Code	Code duplication frequently occurs in feature extraction code. of model's functionality. This can lead to unknown behavior.
CFO	Weight Configuration	Editing code that involves the weights of a ML model, or configuring a ML model's weights directly
	Layer Configuration Code	Editing code that deals with ML models' layers, or configuring a ML model's layers itself
	Hyper-parameter Configuration	Configuring hyper-parameters of ML model, or editing the default values of off-the-shelf model.
DCE	Duplicate Model Code	Code duplication frequently occurs in model code.
	Duplicate Feature Extraction Code	Code duplication frequently occurs in feature extraction code. of model's functionality. This can lead to unknown behavior.
CDD	Machine Learning Dependency	When a needed change in ML software occurs because of its dependency on an external library or other piece of the ML software system. Usually indicates a condition that is waiting to be met before removal
	Glue Code	Supporting code written to interface with other code, inhibiting improvements due to peculiarities of dependent code.
	Custom Data Type	Using data types provided by general-purpose packages can cause extensive inter-operating with external libraries.
	Multiple Languages	Components written in other languages may introduce difficulties in ML development.
	Unnecessary Model Code	Model code that either bottlenecks performance, is unreachable or deprecated, or is unnecessary and should be removed.
DTD	Data Processing Configuration	Configuring the way that data is processed either by editing the data directly, or by adding in new processing steps.
	Plain Old Data Type	Using raw data types in ML causes confusion when interpreting processes.
	Data Storage Configuration	Configuring how data is represented within the source code (data structure) or how data is stored externally (database).
MDL	Abstraction	Lack of abstractions in ML systems and subsystems cause cascading changes when changes are introduced to one component.
	Boundary Erosion	Lack of boundaries between subsystems, creating difficulties when maintaining software and isolating changes made in ML software.
	Model Code Modifiability	Model code should be implemented in ways that enable easy maintenance and future modifications.
	Model Code Reusability	Model code should be generalized to be able to be reused in varying situations.
PRF	Prediction Quality	Previous work in evaluating ML workflows [23] shows that changes may affect performance
	Machine Learning Reliability	Machine learning models' functionalities are determined by the quality of their data, measures should be in place to ensure robustness.
SCL	Prototype	Small-scale prototypes being deployed into full systems can be dangerous.

TABLE II
DEFINITIONS OF THE TAXONOMY TO DEFINE ML SATD TYPES: AWARENESS (AWR), READABILITY (RDB), DUPLICATE CODE ELIMINATION (DCE), CONFIGURABLE OPTIONS (CFO), CODE DEPENDENCY (CDD), DATA DEPENDENCY (DTD), MODULARITY (MDL), PERFORMANCE (PRF), AND SCALABILITY (SCL) [18]

The median Fan-in can be useful to understand the typical coupling, especially in the presence of skewed data. Identifying the module with the highest Fan-in helps in understanding the upper limits of coupling. Analyzing the distribution of Fan-in values can offer a detailed insight into the coupling spread across the repository. Lastly, a weighted Fan-in, considering additional metrics like module size or complexity, can provide a nuanced understanding of dependencies.

Furthermore, to rank the ML TD, we utilize two approaches: calculating the weighted average (a classic statistical method) and Principal Component Analysis (unsupervised learning). This is because we want to ensure that we leave no stone unturned and can make efficient use of a limited dataset. Our assumption is that if the two approaches yield similar results, our method can be applied to a larger dataset. Principal Component Analysis (PCA) is a statistical procedure that enables us to summarize the information content in large data tables using a smaller set of 'summary indices,' making it more easily visualized and analyzed. Given our goal of using a combination of metrics to rank each instance of SATD, we believe PCA is a suitable choice because it is an unsupervised learning method that doesn't require any labels to produce meaningful results. PCA works by first finding the variances in the given SATD instances' metrics using this formula:

$$\mathrm{cov}(X) = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})(X_i - \bar{X})^T \tag{1}$$

Here, n is the number of observation, X_i represents the data points, and $\bar{X}$ is the mean vector of the data. T is the transpose of the vector.

After calculating the covariance, the Eigenvectors and Eigenvalues of the variances are calculated using the below formula:

$$\mathrm{cov}(X).\mathbf{v} = \lambda.\mathbf{v} \tag{2}$$

Here, v is an eigenvector, λ is the corresponding eigenvalue.

Finally, the original data matrix X (our feature values) is multiplied by the matrix of eigenvectors, as indicated below:

$$PC = X.V \tag{3}$$

Here, PC is the matrix of principal components, and V is the matrix of eigenvectors.

The final step yields scores assigned to each SATD instance, providing us with the opportunity for ranking. It's important to note that, during PCA processing, we requested the code to return only one component, despite the option to configure it to return one value for each feature. We opted for a single component for more effective ranking.

Similarly, We calculated weighted average for each ML SATD instance. From PCA, we had already obtained feature importance and score. However it was important to analyse the rank based on all features that we obtained and see if it encounters significant difference. To approach this idea, We assigned equal weights to all features and calculated the weighted average using the formula

The weighted average (W) is calculated using the formula: (4)

$$W = \frac{w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n}{w_1 + w_2 + \ldots + w_n} \tag{5}$$

Here, W is the weighted average, X_i represents the data points, and $\bar{w_i}$ are the score assigned to each feature.

After getting the top three ML SATD Types and the top three ML pipeline stages associated with higher resolution efforts, we manually analyze the code of select instances under those categories. This analysis aims to provide insights into the underlying reasons contributing to the elevated effort required for their removal.

III. Results

A. RQ1: Which types of SATD accumulates more interest?

Motivation: Machine learning applications have different development practices from traditional software development [24]. This is because ML has unique features like reliance on data and the unpredictability found in experiments and ML algorithms. Therefore, the effort to resolve SATD are likely to differ from traditional software development.

Results: Table III and Figure 1 present the PCA analysis of SATD instances in ML projects. The results indicate that Model Code Modifiability (MCM), Unnecessary Model Code (UMC), Machine Learning Dependencies, and Data Configuration are the top four SATD instances that accumulate more technical debt in machine learning projects. Conversely, Machine Learning Knowledge, Dataset Configuration, Weight Configuration, and Model Code Comprehension are identified as the least impactful SATD instances in terms of debt accumulation. The implication of this result is that addressing and managing technical debt in machine learning projects should prioritize mitigating issues related to Model Code Modifiability, Unnecessary Model Code, Machine Learning Dependencies, and Data Configuration. These aspects contribute significantly to the accrued technical debt and, therefore, demand careful consideration in the development and maintenance processes.

Similarly, Table III shows the PCA rank of ML pipeline stages that accumulate more SATD. Figure 2 displays the plot of the top 3 pipeline stages that accumulate more debt. The results show that the Data Acquisition stage, represented as "Data Acq" in Figure 2, accumulates more SATD in ML, followed by the Prediction stage and then the Evaluation Stage. The implication of this result is that developers and stakeholders in ML projects should pay special attention to managing TD in the identified stages, particularly the Data Acquisition stage. Addressing SATD in these stages promptly and efficiently can contribute to a more streamlined and effective development process, potentially reducing challenges associated with TD accumulation.

We further present the result of our findings from weighted average method. The result in Table V and Table VI present

TABLE III
PCA RANKING OF ML SATD

ML TD Type	Principal Component Rank
Model Code Modifiability	11.5652
Unnecessary Model Code	8.7187
Machine Learning Dependency	5.1817
Data Configuration	2.1756
Prototype	-0.0643
Unnecessary Model Code (other)	-0.7246
Model Code Reusability	-0.9827
MLInter	-1.1452
Layer Conf	-1.2924
Hyper-parameter Configuration	-1.4663
Custom Data Type	-1.7581
Model Code Modifiability	-2.7480
Weight Configuration	-2.7497
DS Configuration	-3.3383
Machine Learning Knowledge	-11.3715

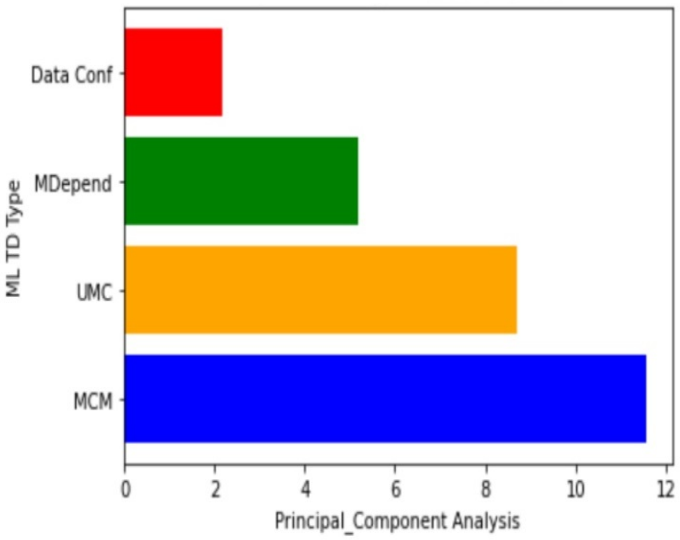

Fig. 1. PCA Ranking of ML SATD

TABLE IV
PCA RANKING OF ML SATD

Pipeline Stage	Principal Component Rank
Data Acquisition	8.2704
Prediction	6.2023
Evaluation	4.1836
Other	1.2518
Training	-2.8899
Modelling	-3.1816
Data Preparation	-13.8366

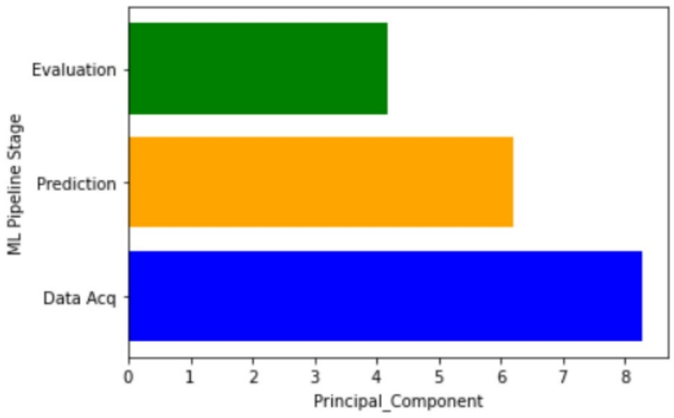

Fig. 2. PCA Ranking of ML Pipelines

the weighted average score of SATD instances in ML projects. The results indicate nice similarity with PCA. Here as well, Model Code Modifiability (MCM) ranked top in accumulating more ML SATD in machine learning projects. Machine Learning Dependencies (MDepend) and Unnecessary Model Code (UMC) rnked second and third.

Similarly, Figure 3 shows the weighted average rank of ML SATD type that accumulate more SATD. Figure 4 displays the plot of the top 3 pipeline stages that accumulate more debt. We obtained slightly different result than PCA here. Based on weighted average score on pipeline stage, Prediction stage was ranked top to accumulate more SATD followed by Evaluation and Data acquisition stage Figure 2

TABLE V
WEIGHTED AVERAGE RANKING OF ML SATD

ML TD Type	Weighted Average Score
Model Code Modifiability	0.277
MDepend	0.18611
UMC	0.165
HP Conf	0.1277
Prototype	0.1241
Proto	0.12411
Data Conf	0.1237
Data Conf	0.1237
DS Configuration	0.1178
CDT	0.1130
UMC	0.1096
MCR	0.1075
Machine Learning Knowledge	0.0989
Layer Conf	0.094
MCC	-11.3715
Weight Conf	0.0701
MLInter	0.0558

The ML TD Types that accumulate more interest are Model Code Modifiability, ML Dependencies and Unnecessary Model Code. In the ML pipeline stage, the Data Acquisition and Prediction stages accumulate more interest.

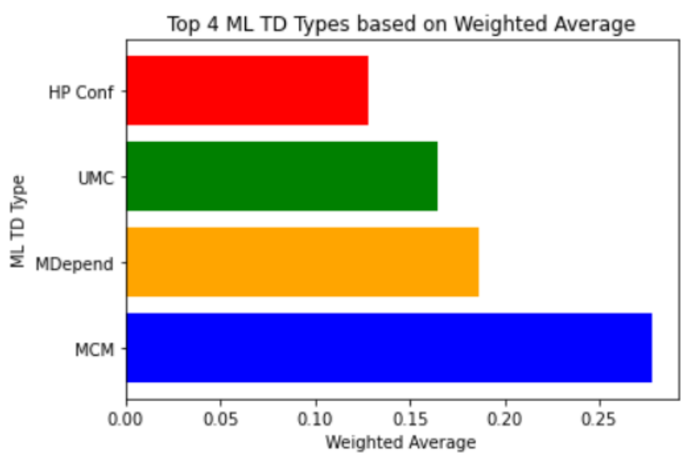

Fig. 3. Weighted Average Ranking of ML SATD

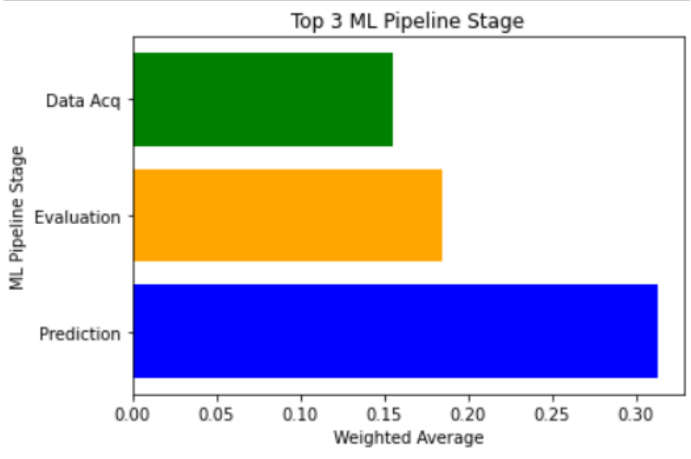

Fig. 4. Weighted Average Ranking of ML Pipelines

B. RQ2: Why do SATDs in specific stages of the machine learning pipeline accumulate more interest?

Motivation: ML applications usually follow a pipeline, which can shed light on the evolution of SATDs. In this section, we investigate why certain SATDs in specific stages of the ML pipeline have higher interest.

Approach: Following the taxonomy of ML pipeline stages defined in prior work: Data Acquisition, Data Preprocessing, Modeling, Training, Prediction and Evaluation [25], we conduct a manual examination of the code instances related to SATD within the Data Acquisition, Data Evaluation and Prediction stages. This investigation is based on our results from RQ1. We selected 23 datapoints of top three ranked ML pipeline stages from the dataset for manual analysis and assigned it to two evaluators (authors). Both evaluators conducted the analysis individually. We established a basic guidelines to follow while reviewing the code.Some of the guidelines and rationale that we followed to answer this research question are as follows:

- Documentation Standard: Lack of documentation can lead to difficulties in understanding, making it a potential source of technical debt.
- Commit Text: The actual comment at the time of commit to understand how that issue was resolved.
- The used library and imports, are they handling the external library downloading process.
- SATD Comments : What information are they giving for that SATD. Are they talking about making a change in prior pipeline stage. This could increase the work and effort to resolve the SATD
- No of Files Changed for that commit : How many files are changed in the last commit, are they realted to fixing the SATD.

These are a few areas that we decided to focus more while determining the reason of that ML SATD fix.

After analyzing the assessments provided by both evaluators, we have synthesized the rationale for each stage of the ML pipeline by merging their evaluation results. Through a collaborative process, we have reached a consensus on a unified rationality to articulate and categorize the findings. This common rationality encapsulates the insights gathered from the evaluations, providing a cohesive and comprehensive understanding of the strengths, weaknesses, and considerations associated with each stage of the machine learning pipeline. This approach ensures a harmonized interpretation of the evaluation results.

Results: Our analysis, as visualized in Figure 2, suggests that the Data Acquisition stage is susceptible to accumulate more interest. This observation is based on the Github version history of the file which created the debt. For instance, developers frequently modify data types impacting training samples, as shown in Listing 1. Such modifications are not only frequent but also removed promptly, typically resolved within 0-8 commits. This quick resolution of technical debt implies a prioritization by developers to refine the quality and diversity of data inputs to the model.

TABLE VI
WEIGHTED AVERAGE RANKING OF ML SATD

Pipeline Stage	Weighted Average Score
Prediction	0.313
Evaluation	0.1843
Data Acquisition	0.1553
Other	0.1250
Training	0.117
Modelling	0.113
Data Preparation	0.111

```
# TODO: make the dtype configurable; see
    callback()
    self.stream = self.pa.open(rate=self.
        sample_rate,
        channels=self.num_channels,
        format=pyaudio.paInt16, input=True,
```

Listing 1. Data Acquisition Example 1

Another example of quick resolution is illustrated in Listing 2, where developers seek to improve diversity of data by incorporating a wider range of Operators. Despite the minor number in lines of code for these changes - averaging an addition of 43 lines of code to resolve the SATD - their quick removal in terms of number of commits suggests the importance that developers attribute to the enhancement of data quality.

```
@find_domain.register(ops.Op)  # TODO this is
    too general, register all ops
@find_domain.register(ops.ReciprocalOp)
@find_domain.register(ops.SigmoidOp)
@find_domain.register(ops.TanhOp)
@find_domain.register(ops.AtanhOp)
```

Listing 2. Data Acquisition Example 2

ML debt in Data Acquisition step of the pipeline accumulates debt that is promptly resolved and is usually part of adding more information to the model. Our analysis suggest that these SATDs accumulate more interest since these are promptly removed and the data affects the accuracy of the models.

C. RQ3: Why do certain types of SATD accumulate the most interest?

Motivation: Traditional software development SATDs have been categorized in different taxonomy, but this does not apply directly to ML. Recently, there has been research which worked to create a taxonomy for ML SATD types. Currently, we do not know which types of SATD in ML involve more effort to be removed. In this section, we dive into why the types of SATD we selected on RQ1 involve more effort to be removed.

Approach: Based on our results for RQ1, we manually investigate the code of the instances of SATD for Model Code Modifiability, ML Dependencies and Unnecesary Model Code.

Results: MCM SATDs take in average 5 commits from introduction to removal. In the related code instances which are quicker to remove there are cases related to use different data types to improve the accuracy of the model. Other example which are quickly removed is refractoring code related to building the model. In Example 2, there is a removal of these hyperparameters definition from the file but the commit involves refractoring this code and extracting it in another function in another file.

```
    # TODO: make the dtype configurable; see
        __init__()
        data = np.fromstring(data, 'int16').
            astype(self.dtype)
```

Listing 3. MCM Example 1

```
    # TODO: maybe refactor this
        if self.multi_label:
            self.hyper_parameters['
                compile_params']['loss'] = '
                binary_crossentropy'
            self.hyper_parameters['
                compile_params']['metrics'] =
                ['categorical_accuracy']
            self.hyper_parameters['
                activation_layer']['activation
                '] = 'sigmoid'
```

Listing 4. MCM Example 2

ML Dependency instances refer to having ML Dependency issues with internal or external dependencies. These dependencies require more effort to resolve since they are harder to detect. As part of our manual analysis we extracted one example in which an internal dependency caused issues in the training of the data [Listing 5]. This led to a quickly removal after 1 commit in the file.

```
    # The original InceptionResNetV2 has been
        trained by mistake with
        preprocess_input using caffe scale
        similar to resnet50
    # TODO: it's worth to switch back to the
        correct preprocess_input when
        InceptionResNetV2 model is re-trained
    from tensorflow.python.keras.applications.
        resnet50 import preprocess_input as
        preprocess_input_inception_resnet_v2
```

Listing 5. ML Dependency Example 1

In Listing 6, we find an instance in which exploiting a bug worked for 148 days, but once the dependency fixed this exploit it impacted this project by getting wrong scores and causing inconsistencies between scoring and inference.

```
# TODO: we currently exploit a bug in the
    implementation of unravel_index to not
    require knowing the first shape
        # value. See https://github.com/apache
            /incubator-mxnet/issues/13862
        unraveled = F.unravel_index(indices,
            shape=(C.LARGEST_INT, self.
            vocab_size))
```

Listing 6. ML Dependency Example 2

ML Dependencies accumulate more interest due to creating circular dependencies between the models. Some external dependency issues might take longer to resolve but they accumulate more effort to resolve, while affecting scoring and inference.

IV. Discussion and Implications

The results of our study offer actionable guidance for ML developers aiming to manage more effectively their technical debt. In this section, we describe how our results can be used to guide practitioners and researchers at improving their prioritization of machine learning SATDs.

Implication for Practitioners. Our analysis, as depicted in Figures 1 and 2, highlights the primary ML SATD categories which accumulate more interest: Model Code Modifiability, Unnecessary Model Code, and ML Dependencies. These categories predominantly pertain to issues of Modularity and Code Dependency. The development community should use these findings and integrate these insights into their design processes. Prior research indicates that ML developers may not be familiar with Object Oriented Programming [21], thus making them more vulnerable to incur into introducing this type of debts. Awareness on why these issues occur could be crucial in mitigating these risks.

Implication for Researchers. The findings of our study gather insights on how ML technical debt is resolved. Researchers should investigate how these debts affect metrics like precision, recall and F1-score. This could help prioritize technical debt which impacts precision of the model. Also, researchers should investigate if the introduction of SATDs are related to ML code smells and propose guidelines to manage SATDs in ML projects.

V. Related work

OBrien et al. [18] conducted a study on TDs in ML software, mining 68,820 SATDs from 2,641 ML repositories on GitHub. They developed a taxonomy of ML SATDs, identifying 8 new ML SATD groups and analyzing their characteristics, frequency, and removal efforts. This work is significant because it provides a framework for identifying and managing SATDs in ML systems, aiding the development of more maintainable ML solutions. This paper is relevant to our project as it provides insights into the nature and management of SATDs in ML software, aligning with our aim to quantify and understand SATDs in ML projects.

Kamei et al. [16] proposed a method using software product metrics to quantify the interest on SATD. Their case study on the Apache JMeter project revealed that about 42-44% of TD incurs positive interest. This work is crucial as it provides a tangible way to measure the impact of SATD, aligning closely with our project.

Zampetti et al. [26] conducted an in-depth study on the removal of SATD in Java open source projects. Their research revealed that 20%-50% of SATD comments were removed accidentally during class or method deletions, with only 8% of SATD removals documented in commit messages. They also observed that resolving SATD often involved complex changes to method calls or conditionals. This work is significant for highlighting the unintentional removal and documentation practices of SATD, offering valuable insights for managing SATD in software development. This paper is relevant to our project as it sheds light on the practical aspects of SATD removal, informing our exploration of SATD in ML projects.

Maldonado et al. [20] conducted an empirical study on the removal of SATD. They analyzed five open-source projects, finding that most SATD is removed, often by the same person who introduced it, and typically lasts between 82 to 613.2 days. This work is significant because it sheds light on SATD's lifecycle and removal practices, which is crucial for understanding and managing SATD in ML projects.

Detofeno et al. [27] introduced PriorTD, a method for prioritizing TD in software projects. Their approach combines source code analysis with project importance, aiding decision-making for TD payment. They found that PriorTD effectively prioritizes TD, identifies dead code, and enhances team-manager communication. This work is significant as it provides a comprehensive TD prioritization method, bridging technical and business considerations.

Bavota et al. [28] investigated SATD across 159 open-source systems, mining over 600K commits and 2 billion comments. They found that SATD is prevalent, with an average of 51 instances per system, mostly in the form of code, defect, and requirement debt. Notably, SATD tends to increase over time and survive long (over 1,000 commits on average) in systems. This work is important because it provides empirical evidence on the persistence and types of SATD, underscoring the need for effective management strategies. This study is relevant to our project as it offers foundational insights into SATD's prevalence and longevity, informing our analysis of interest accumulation in ML systems' SATD.

Sculley et al. [15] proposed a framework to address hidden TD in ML systems, highlighting risk factors like boundary erosion and system-level anti-patterns. They emphasized that addressing TD in ML systems incurs high maintenance costs and requires a cultural shift. This work is significant for its focus on the often-overlooked long-term challenges in ML system maintenance. This paper is essential to our project as it offers key insights into TD in ML systems, informing our study on quantifying SADT.

Tang et al. [21] explored refactorings and TD in ML systems, analyzing 26 projects and 327 code patches. They discovered various reasons for refactorings and introduced 14 new ML-specific refactorings and 7 TD categories.This paper is relevant to our project because it provides insights into TD in ML systems, which aligns with our focus on quantifying SADT in such systems.

Zampetti et al. [29] explored how developers admit TD in both industry and open-source contexts. Their study, involving interviews and surveys, revealed that while TD annotation practices are similar across sectors, the admittance of SATD in industry is often influenced by organizational guidelines or fear of responsibility. This research is significant as it sheds light on the human and organizational factors affecting SATD admissions. Its findings are particularly relevant to our project as they provide crucial insights into the dynamics of SATD management, essential for understanding SATD accumulation and impact in ML projects.

De Lima et al. [30] proposed a prioritization approach to support decision-making in the payment stage of SATD. Their approach combines SATD descriptions and issues found in the source code to prioritize SATD, demonstrating greater precision compared to using SATD descriptions alone. This paper is relevant to our project as it introduces a novel method for prioritizing SATD, essential for managing TD in ML systems.

VI. Threats to Validity

Internal Validity: Internal Validity concerns factors that could have influenced our results. Our methodologies of measuring effort by weighted and PCA approach depend on our metrics to measure effort. Although ML applications might need to consider different metrics than traditional software engineering ones, we argue that our metrics gives insights into the complexity and history of the SATDs.

Construct Validity: Construct validity considers the relationship between theory and observation. Manual analysis of the SATDs that result in more effort to resolve according to our study might not reflect the overall issues that occur in these cases. To minimize this threat, we manually analyzed all instances of SATDs according to the type and pipeline stage which requires more effort to resolve. These insights were discussed with a second author to understand the root of the SATD and how it was resolved.

External Validity: Threats to external validity concern the generalization of our findings. In this study, we examined the effort of resolving SATDs based on previous work which utilized open-source Python repositories. Although the original dataset included 2,641 ML Python repositories, our study involves a smaller subset of 110 instances of SATDs. Future work could work on mining latest ML repositories and increase the data set to complement our findings.

VII. Conclusion

In this study we analyze machine learning SATDs to identify the types of technical debt which need more effort to be payed. We find that issues related to Modularity and Code dependencies are the ones which accumulate the most interest if left unresolved. We also find that the pipeline stages which incur the technical debt that accumulate the most interest are Data Acquisition and Prediction. Our results suggest that practitioners should create software components which are easy to modify in the future and carefully consider the specific types of data that are needed in the Data Acquisition step of the pipeline.

Our study opens the door for researchers and practitioners to further understand how technical debt is payed in Machine Learning applications and understand which types of technical debt could further impact the accuracy of ML models. For this end, future work could focus on analyzing SATD comments and developers commit messages using NLP technique to categorize level of prioritization of each SATD. Future work could also take surveys from ML experts to gather insights on the trade offs of incurring technical debt and how these are different from traditional software engineering. Lastly, an increased data set could be used to analyze latest software repositories since the attention to ML applications has increased in the last few years.

References

[1] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE software*, vol. 29, no. 6, pp. 22–27, 2012.

[2] W. Cunningham, "The wycash portfolio management system," *ACM Sigplan Oops Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[3] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 179–188.

[4] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd workshop on managing technical debt*, 2011, pp. 17–23.

[5] F. A. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, 2012, pp. 15–22.

[6] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *2014 Sixth International Workshop on Managing Technical Debt*. IEEE, 2014, pp. 1–7.

[7] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, 2013, pp. 42–47.

[8] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[9] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 50–60.

[10] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An enterprise perspective on technical debt," in *Proceedings of the 2nd Workshop on managing technical debt*, 2011, pp. 35–38.

[11] R. O. Spínola, A. Vetrò, N. Zazworka, C. Seaman, and F. Shull, "Investigating technical debt folklore: Shedding some light on technical debt opinion," in *2013 4th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2013, pp. 1–7.

[12] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 91–100.

[13] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.

[14] M. Dilhara, A. Ketkar, and D. Dig, "Understanding software-2.0: A study of machine learning library usage and evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–42, 2021.

[15] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, 2015.

[16] Y. Kamei, E. d. S. Maldonado, E. Shihab, and N. Ubayashi, "Using analytics to quantify interest of self-admitted technical debt." in *QuA-SoQ/TDA@ APSEC*, 2016, pp. 68–71.

[17] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "Satd detector: A text-mining-based self-admitted technical debt detection tool," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 9–12.

[18] D. OBrien, S. Biswas, S. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, "23 shades of self-admitted technical debt: an empirical study on machine learning software," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 734–746.

[19] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks," *Empirical Software Engineering*, vol. 26, pp. 1–36, 2021.

[20] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt,"

in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 238–248.

[21] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja, "An empirical study of refactorings and technical debt in machine learning systems," in *2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*. IEEE, 2021, pp. 238–250.

[22] D. Gonzalez, T. Zimmermann, and N. Nagappan, "The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github," in *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, 2020, pp. 431–442.

[23] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ml test score: A rubric for ml production readiness and technical debt reduction," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 1123–1132.

[24] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?" *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1857–1871, 2019.

[25] S. Biswas, M. Wardat, and H. Rajan, "The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2091–2103.

[26] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal? an in-depth perspective," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 526–536.

[27] T. Detofeno, A. Malucelli, and S. Reinehr, "Priortd: a method for prioritization technical debt," in *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, 2022, pp. 230–240.

[28] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 315–326.

[29] F. Zampetti, G. Fucci, A. Serebrenik, and M. Di Penta, "Self-admitted technical debt practices: a comparison between industry and open-source," *Empirical Software Engineering*, vol. 26, pp. 1–32, 2021.

[30] B. S. de Lima, R. E. Garcia, and D. M. Eler, "Toward prioritization of self-admitted technical debt: an approach to support decision to payment," *Software Quality Journal*, vol. 30, no. 3, pp. 729–755, 2022.