23 Shades of Self-Admitted Technical Debt: An Empirical Study on Machine Learning Software

David OBrien
Dept. of Computer Science
Iowa State University
Ames, IA, USA
davidob@iastate.edu

Sumon Biswas*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
sumonb@cs.cmu.edu

Sayem Imtiaz
Dept. of Computer Science
Iowa State University
Ames, IA, USA
sayem@iastate.edu

Rabe Abdalkareem
School of Computer Science
Carleton University
Ottawa, ON, Canada
rabe.abdalkareem@carleton.ca

Emad Shihab
Concordia University
Montreal, QC, Canada
emad.shihab@concordia.ca

Hridesh Rajan
Dept. of Computer Science
Iowa State University
Ames, IA, USA
hridesh@iastate.edu

ABSTRACT

In software development, the term "technical debt" (TD) is used to characterize short-term solutions and workarounds implemented in source code which may incur a long-term cost. Technical debt has a variety of forms and can thus affect multiple qualities of software including but not limited to its legibility, performance, and structure. In this paper, we have conducted a comprehensive study on the technical debts in machine learning (ML) based software. TD can appear differently in ML software by infecting the data that ML models are trained on, thus affecting the functional behavior of ML systems. The growing inclusion of ML components in modern software systems have introduced a new set of TDs. Does ML software have similar TDs to traditional software? If not, what are the new types of ML specific TDs? Which ML pipeline stages do these debts appear? Do these debts differ in ML tools and applications and when they get removed? Currently, we do not know the state of the ML TDs in the wild. To address these questions, we mined 68,820 self-admitted technical debts (SATD) from all the revisions of a curated dataset consisting of 2,641 popular ML repositories from GitHub, along with their introduction and removal. By applying an open-coding scheme and following upon prior works, we provide a comprehensive taxonomy of ML SATDs. Our study analyzes ML SATD type organizations, their frequencies within stages of ML software, the differences between ML SATDs in applications and tools, and quantifies the removal of ML SATDs. The findings discovered suggest implications for ML developers and researchers to create maintainable ML systems.

*At the time this work was completed, Sumon Biswas was a graduate student at Iowa State University

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549088

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

technical debt, machine learning, open-source, data science

ACM Reference Format:
David OBrien, Sumon Biswas, Sayem Imtiaz, Rabe Abdalkareem, Emad Shihab, and Hridesh Rajan. 2022. 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 (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549088

1 INTRODUCTION

The recent uprising and rapid integration of machine learning (ML) models has empowered developers to tackle problems previously infeasible such as safety-critical systems, financial fraud detection, and medical diagnostics [3, 13]. However, the sudden emergence and adoption of ML models risks the creation of hastily-planned machine learning software deployed long-term [45]. Thus, the functionalities of future machine learning software depends upon the maintainability of present-day machine learning software.

Through decades of traditional software practices, the term "technical debt" (TD) was coined by Ward Cunningham to characterize short-term solutions, workarounds, or unfinished implementations that exist in long-term software. TD can worsen software's legibility, performance, and quality, consequentially complicating its maintenance lifetime. Similar to fiscal debt, TD becomes more "expensive" the longer an instance exists, costing a heavy amount of developers' time, effort, and knowledge to pay off. Therefore, the awareness of TD types has historically informed developers how to best manage their software to minimize the TD accumulated and prioritize their TD management [2, 19, 33, 53].

ML software contains additional components not found within traditional software development. Sculley et al. explored the hidden TD in ML to describe challenges encountered when Google engineers were maintaining ML software [44, 45]. Because ML software

is accompanied by challenges native to its own domain, it is susceptible to new forms of TD not encountered by traditional software practices. For example, because ML software has an implicit dependence upon the quality of its training data, problems in the data are reflected upon the resulting model behavior [1, 29]. Previous work has provided a framework to contain the contagion of TD in ML software through multi-level evaluations [11]. Moreover, Tang et al. [52] further studied TD in ML software through studying refactorings and their corresponding removal strategies found in Java ML software.

Historically, the notion of self-admitted technical debt (SATD) describes a developer's expressed beliefs on necessary improvements to current implementations of software [4, 27]. SATD is most commonly associated with source code comments documenting a potential change, although SATD has also been found in an analysis of software issue trackers [57]. SATD has been used as a proxy to study TD, since SATD contains natural language to communicate proposed changes to source code. A large financial organization in practice discovered that their developers used SATD to guide the management of their TD [54], further emphasizing the value of examining SATD.

Because previous studies have used SATD to indicate TD patterns, it may provide insights on ML specific TD symptoms. However, to the best of our knowledge no study has been conducted to understand the ML-specific SATD types and their characteristics present in the ML software.

Motivated by this, we created and filtered a dataset consisting of popular ML repositories written in Python which have been used in previous studies [26, 50]. We chose Python because it has been referred to as the de-facto language for ML development due to its popularity [5, 14], then performed a labeling process to create a statistically significant dataset of 856 labeled instances of SATD within these repositories to answer the following research questions:

RQ1: What types of SATD are found in our studied ML software? SATD can indicate the presence of TD, and is more human understandable by nature of natural language comments. Are there new types of SATDs that exist in ML software that previous works haven't discovered?
RQ2: What is the distribution of SATD types in the different ML pipeline stages? The ML pipeline splits up the ML development workflow into distinct objectives [3, 10]. Do different stages have unique frequencies of SATD types? Does a SATD appearance change in differing stages?
RQ3: Is there a difference in SATD types between ML applications vs ML tools? ML application developers and ML tool developers have different goals. Does this influence the SATD that they encounter?
RQ4: How much effort is needed to remove SATD types in ML software? Since SATD indicates problems affecting ML software's maintainability, it is important to understand which types of SATD comments have historically been difficult to remove.

Our findings indicate that ML developers are most concerned with improving how their software meets their project's functional and non-functional requirements. Moreover, our observations find a substantial focus on the configuration of data processing code across multiple pipeline stages. Additionally, our study finds that ML tools contain more SATD involving dependencies on outside code than ML applications. Finally, our observations suggests that various types of ML SATDs take varying degrees of effort to remove. To allow replication and future research, our labeled dataset of identified SATD in ML repositories is available to the public[1].

Table 1: Filtering Overview.

Stage Name	Remaining Comments
Extracted	9,725,127 changes
Length > 1	7,415,024 changes
SATD Detector / Keyword Search	193,787 changes
Removed Autogenerated	189,912 changes
Removed Non-Unique Removals	85,599 comments
Not in "site-packages" Folder	68,820 comments

The rest of the paper is organized as follows. Section 2 presents our methodology to gather and filter a dataset of source code comments, as well as our strategy to construct ML SATD taxonomies. The results of our four RQs are presented in Section 3. We discuss the implications of our work in Section 4. The limitations of our study are highlighted in Section 5. Section 6 discusses the related works, and Section 7 concludes the paper.

2 METHODOLOGY

In this study, we will manually analyze source code comments found in ML systems. This section describes the dataset utilized, our filtering and sampling processes, our classification scheme, and our classification process.

2.1 Dataset

Our study involves open-source machine learning repositories presented in [26] which has been used in a prior study examining ML bugs' associations with programming languages [50]. This dataset contains popular ML repositories, including ML tools (repositories supplying ML functionality) such as the scikit-learn [2] repository and ML applications (repositories applying ML to a problem) such as deepfake's faceswap repository[3]. The repositories found in the dataset have been filtered so that every software repository within has more than 5 stars *OR* $\geq$ 5 forks, must have been active in 2019, and must be a non-trivial software project. Although the repositories were gathered in 2019, we generate the data using the source code data as of January 2021. The resulting dataset contains repositories spanning across 439 topic labels on GitHub, indicating a representation of a variety of ML topics. Although the resulting dataset contains 5,224 repositories across a variety of programming languages, this study will only analyze the 2,684 which have been written in Python due to its popularity in the ML community [5, 14]. From the 2,684 repositories present in the dataset, 43 were not present on GitHub when our dataset was generated, resulting in 2,641 total projects being analyzed in this study.

2.2 Identifying SATD Comments

This section describes the filtering process used to extract SATD comments that have existed in the repositories described in the previous section. Table 1 provides an overview of the process.

[1] https://github.com/DavidMOBrien/23Shades
[2] https://github.com/scikit-learn/scikit-learn
[3] https://github.com/deepfakes/faceswap

We use Boa [15, 16], a language specifically designed to mine data from software repositories which has demonstrated capabilities to analyze data science repositories [7]. Boa analyzes each revision to determine when a source code comment has been introduced or removed [17, 18]. 9,725,127 comment changes are extracted from our dataset in this manner. After manual inspection of the dataset's contents, we found comments with a text length of one character likely do not represent SATD or contains enough information to classify an SATD comment. We remove such comments, leaving 7,415,024 comment changes remaining.

To filter our comment changes to include only SATD comment changes, we use both a previously trained classifier and a keyword search. First, we use the SATD detector classifier created by [27] which was trained upon comments from 8 active software repositories to obtain an average F1-score of 0.737. However, since SATD found in ML software may differ from SATD in traditional software, we measured the performance of this classifier on a sample of 1,255 ML source code comments, where 35 were labeled to be SATD by the authors. Previous work's classifier obtained a precision of 82.6% and a recall of 54.2% on this sample. However, we found that many of the false negatives from this experiment contained a keyword discovered to indicate a SATD ("todo", "fixme", "hacky", etc.) [36]. Therefore, we take the union of comments classified as SATD by previous work, and comments which contain a keyword presented in [36] to reach a recall of 82.6%. 193,787 comment changes remain after filtering in this manner.

Additionally, we remove comments which are likely irrelevant to SATD as done in previous work [36]. Therefore, we manually observe comments in our dataset which appear most often to identify cases such as comments describing software licensing, comments which have been automatically generated (i.e., *#generated protocol buffer*), and comments to assist linters (i.e., *#NOQA*) as cases which likely do not contain SATD. 189,912 comment changes remain after removing such comments.

We then organize our comment changes to separate comments which have been removed or not in their repositories lifetime. Prior work shows that data from GitHub has been found difficult to mine cleanly due to events such as merges, rebases, and rollbacks, among others [6, 22]. Additionally, a repository may have multiple copies of the same SATD comment. Therefore, we follow a similar procedure to [37] by considering a comment removed if there is a *unique* removal of a comment after its introduction. We find 85,599 unique comments in this manner.

Following manual evaluation, we identified the case where a source code comment is not native to the analyzed repository. Rather, the source code comment is found in the "site-packages" directory. Because the goal of this study is to analyze SATD comments in ML repositories, we choose to filter out comments which appear in this directory. After this step, we find 68,820 comments remaining in our dataset.

We took a statistically significant sample of our filtered dataset with a confidence interval of 95% and a margin of error of 3.33% to calculate the size of our sample dataset. Then, we sample comments similar to a previous study on SATD comments [25] where the SATD dataset is sampled so that the resulting sample's characteristics is proportional to the original dataset. Table 2 illustrates the respective proportions of the dataset and its sample.

Table 2: Total and Sampled SATD Comments

Dataset Bucket	# of SATD Comments	# of Sampled SATD Comments
Tool + Comment Removed	8,104	101
Tool + Comment Not Removed	19,724	245
App. + Comment Removed	10,215	127
App. + Comment Not Removed	30,777	383
Total	**68,820**	**856**

2.3 Building SATD Classification Scheme

In this section, we will describe the classification schemes used to label our sampled dataset.

Previous work on TD in ML applications argued that ML software can become indebted in similar ways to traditional software development, as well as in ways unique to ML systems [44]. Therefore, we review prior works on traditional software SATD types [2, 4, 42] to construct a taxonomy which this study refers to as "Software SATD Types". This study also presents a taxonomy consisting of SATDs which appear in ML-specialized software. To accomplish this, we identify TD types found in ML software from previous studies [45, 52]. Using these prior works, we construct a taxonomy of SATDs from the corresponding TDs which this study will refer to as "ML SATD Types".

We use the ML pipeline defined in a previous work [10] which analyzed 3 representations of the ML pipeline at varying levels to find 7 well defined pipeline stages which is used to answer RQ2.

We performed a pilot study using previous works' taxonomies [45, 52] on a sample of 100 comments from the dataset filtered in Section 2.1. Although every type in previous works' ML TD taxonomy was not reached in the 100 samples, we did notice beneficial improvements that could be proposed. For example, we observed that the *Configuration* debt [45] was too broad to effectively analyze and reason about its symptoms and effects. Therefore, we split the *Configuration* debt into five specific types of configurable options (*Data Configuration*, *Data Storage Configuration*, *Weight Configuration*, *Hyper-parameter Configuration*, *Layer Configuration*). Additionally, we also propose new types not found in previous studies (*Machine Learning Dependency*, *Machine Learning Knowledge*, *Machine Learning Reliability*, *Model Interpretability*, *Prediction Quality*).

2.4 Classifying SATD Related Comments

Once we created the classification schemes, we want to apply our classification scheme on the sampled SATD comments. We performed a coding process [46]. To do so, an initial training meeting with three of the authors to discuss the classification schemes and the pilot study described previously. Following this meeting, two of the authors would independently label 30-40 SATD comments, then discuss in the presence of the third author acting as a moderator. Since our classification is prone to human bias, we calculate the Cohen's Kappa coefficient to measure the agreement between two annotators. Cohen's Kappa coefficient is a common statistical method that is used to evaluate the inter-rater agreement level for classification scales by discarding the possibility of random agreement. The resulting coefficient is scaled to range between -1.0 and 1.0, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement.

The two labeling authors achieved a Cohen's Kappa coefficient of 63% between their first independently labeled sets, 77% on their second, and 80% on their third. Because the authors' Cohen's Kappa coefficient had reached very strong agreement, they labeled once more independently to achieve Cohen's Kappa coefficient of 83%, which is considered to be excellent agreement [21]. After each labeling session, the two labeling authors and the third moderating author would meet to reconcile any disagreements held in the classifications, while also discussing the cause of the disagreements.

After meeting to reconcile the fourth independently labeled sets, the authors then labeled the rest of the unlabeled data. In cases where an author was not confident on the appropriate label to assign to an SATD comment, other authors were consulted in order to reach a decision on the appropriate labels for those cases. Finally, the entire dataset was reviewed so that every comment was checked by at least two authors.

3 RESULTS

In this section, we investigate our labeled dataset to answer the four research questions.

As a result of our manual classification process and literature review, we ended up with 6 different Software SATD Types which are described and exemplified in Table 3.

After conducting our pilot study on SATD types in ML software, we found that some SATD comments were not well described by pre-existing works or are too broad to describe unique characteristics [45, 52]. To best fix these scenarios, we propose a new ML SATD classification scheme consisting of 23 ML SATD Types consisting of 13 pre-existing TD in ML software types [45, 52], and 5 new types as well as 5 split-up types proposed in this study which were used to label our dataset. All ML SATD Types presented upon have at least one comment in our labeled dataset. To further analyze the aspect by which a SATD in ML software is concerned with, we further organize the 23 ML SATD Types found in Section 2.3 into 9 high-level groups that will be referred to as "ML SATD Groups". We believe ML researchers and practitioners can use this hierarchical classification scheme to guide and evaluate ML software maintenance. Table 4 shows a detailed description of the ML SATD Types organized by ML SATD Groups.

3.1 RQ1: What types of SATD are found in our studied ML software?

In this section, we analyze the distributions within our dataset to present commonly found SATD types in ML software, their symptoms, and example comments.

Our results are organized as such: Table 3 presents the distribution of all Software SATD Type comments that occur. It is important to note, some comments were not well described by one of the 6 Software SATD Types, and was then left blank. Similarly, if an SATD comment in our labeled dataset was found to be described by multiple ML SATD Types, then it is counted as a unique debt for each ML SATD Type it received. For example, the comment `TODO: experiment with more layers` received the labels *Layer Configuration* and *Machine Learning Knowledge*, so it is counted in the totals for both ML SATD Types. The results from performing this study are shown in Table 5.

Finding 1: *Requirement* debt accounts for 40.68% of the Software SATD Types in ML software.

As seen in Table 3, more than a third of SATD comments in our labeled dataset which have a Software SATD Type received the *Requirement* debt label. *Requirement* debts are concerned with shortcomings involving incomplete functionality or non-supported features (functional requirements) as well as poorly performing code (non-functional requirements), ML-specific examples of these are provided in Table 3. Because ML is an active area both in industry and academia, there are rapid discoveries and improvements that can be made to ML software [26]. Cases such as algorithmic configurations, API updates, and developer improvement could contribute to the resolution of non-functional *Requirement* debts. Similarly, code reviews, issue trackers, and feature requests are methodologies which may identify functional *Requirement* debts. However, the abundance of *Requirement* debts in our studied ML software repositories may indicate that ML developers prefer to focus on other matters besides functional and non-functionally indebted subsystems, leading to the introduction of these self-admitted technical debts.

Our findings indicate that ML developers should be aware of how fast-paced ML development can be and how changes to requirement specifications could affect the evolution and maintenance of their software. Additionally, ML framework developers should also be aware that the functionalities they provide may lead to the introduction of *Requirement* debts in the applications which use them through API misuse, unclear intents, and cascading upgrades [45]. **We recommend that ML framework developers should design APIs such that its users are supplied with a complete picture of its requirements and limitations.** For example, the SATD comment `TODO add param for relative path vs just folder names` reveals the requirements and limitations found in an evaluation utilities class. Knowledge of this limitation can save developer time later in development. Furthermore, ML developers should carefully consider how their current implementations may hold up against future requirement changes [11, 44, 45].

Finding 2: When compared to traditional software, ML software contains less prioritized *Code* debts.

To put our results in perspective, the results from prior work on SATD types in traditional software development by Bavota and Russo are shown in Table 6 [4]. They analyzed 159 software repositories, resulting in the mining of 2 billion comments and the labeling of 273 SATD comments with the same classification scheme we used to construct our taxonomy of the Software SATD Types.

When we compare our results in ML software to Bavota and Russo, we see that our investigations indicate that ML developers might differ in their SATD management to traditional software developers. Instead of *Requirement* debt, *Code* debt is the SATD type which appears most in Bavota and Russo's study on traditional software SATD [4]. *Code* debt involves areas of poorly legible source code. Cases of *Code* debts include sections of code where code logic can be simplified to allow for easier understanding, variables or functions can be renamed, a function can be refactored, or code can be reformatted to adhere to expected coding standards. Example SATD comments of *Code* debts found in our dataset include: `# hackier; fixme to use regex or something` and `It's only`

Table 3: Definitions and Examples of the 6 Identified Software SATD Types.

SATD Types	Description	Example comment	# of Occ.	% of Occ.
Requirement	Requirement debts can be functional or non-functional. In the functional case, implementations are left unfinished or in need of future feature support. In the non-functional case, the corresponding code does not meet the requirement standards (speed, memory usage, security, etc...)	`TODO: handle channel modalities later, TODO: make efficient, TODO: Implement Conv Transpose.`	321	40.68%
Code	Bad coding practices leading to poor legibility of code, making it difficult to understand and maintain.	`TODO: This next code is dense and confusing. Clean up at some point.`	207	26.24%
Test	Problems found in implementations involving testing or monitoring sub-components.	`XXX: should we rather test if instance of estimator?`	84	10.65%
Defect	Identified defects in the system that should be addressed.	`TODO this will fail if a parameter cant handle size=(N;)`	82	10.39%
Design	Areas which violate good software design practices, causing poor flexibility to evolving business needs.	`TODO maybe improve this so it doesn't use a global`	80	10.14%
Documentation	Inadequate documentation that exists within the software system.	`TODO update doc above`	15	1.90%

`needed for a specific purpose in the short term; will go`. **Because the two types of developers differ in the types of SATD they accrue, it may be the case that the nature of the software domain affects the developers' maintenance patterns.**

As speculated by previous work [52], some ML developers may not be primarily software developers. Instead, they might be data scientists, researchers, or domain experts. Therefore, developers in these roles may not be as prepared to maintain software solutions. Regardless, a comparison between Bavota and Russo's study [4] and our own suggests that there are distinctions between the SATD patterns between both types of developers. Additionally, the languages which are used by the studied software in both studies are different. Bavota and Russo studied 159 repositories which are written in Java [4], while our study analyzes Python projects, both languages are studied together in a previous work on technical debt [51]. Therefore, the difference in programming language usage may cause a shift in SATD management.

Finding 3: 30.58% of ML developers' ML-specific SATD is due to *Data Dependency*.

As seen in Table 5, *Data Dependency* is the most used ML SATD Group. ML software functionality is heavily dependent upon the quality, structure, and consistency of models' training data. Therefore, the code that processes or stores this data may have became a natural focus for ML developers [45]. Because data is influenced and through ML models can influence the outside world, it is crucial that ML developers ensure their data is well-monitored and well-understood. Misunderstood data can cause unintended consequences, and as a result can effect the outside world when naively deployed [44, 45].

Based on our manual analysis, we observe that ML developers may leave more *Data Dependency* debts than any other group of ML SATDs. Technically indebted data can be harmful for ML models because it implies that the current data has identified shortcomings, similar to how code can be described with technical debt. For example, the comment text `# TODO: EXPAND PROPER NOUNS FOR COMMON WORDS AROUND WORD` admits a preprocessing task that should be improved upon. **Because this current short-term solution exists, not only is the software suffering from a SATD, but the data may be considered technically indebted as well.** Although many *Data Dependency* debts observed in our dataset involve data preprocessing tasks, there are other cases which *Data Dependency* can be concerned with. Other examples of *Data Dependency* involved debts include those associated with data visualizations (`TODO add general Distribution`), data storage management (`TODO check the local cache and cloud for different images of same name`), configuring qualities of training data (`TODO: weight by length here`), and interfacing with model output (`FIXME: Only keeping the first label`).

While *Data Dependency* SATD comments can involve how data is processed, represented, and used through ML software, we also find reoccurring symptoms in the *Data Dependency* SATD comments. These symptoms indicate an implied problem or suggested fix on ML code that interacts with data. We find that these symptoms typically indicate a change involving the addition, removal, improval, replacement, reapplication, or handling an edge case of some data involved component. An addition indicates a spot where a new data procedure can be used (`TODO: Add Hebrew-to-Aramaic converter`). A removal indicates a procedure that may need to be removed (`FIXME don't l2_normalize for any metric`). An improvement is an area where the data procedure can be modified to perform better (`NOTE: probably more efficient to sort then stride by nt_regions`). A replacement indicates where a different data-involved functionality can be used (`TODO: hard coded for now; looking for better extraction methods`). A reapplication involves the reuse of a pre-existing procedure (`TODO: It could be nice if this method was run on entire data; not just a sample`). Finally, symptoms can indicate cases where the current procedures fail or do not behave as expected (`Double newlines seem to mess with the rendering`).

Therefore, ML developers should consider how their data configuration implementations could evolve overtime. **If a particular component does not allow for the fix of one of the symptoms described above, then an underlying data-dependent TD may exist.** Since our data suggests that these problems are encountered commonly in ML software, preparation for these modifications can save maintenance activity.

Finding 4: *Awareness* account for 17.45% of our ML SATD.

Awareness consists of debts types where the lack of developers' knowledge or understanding negatively affects its associated software. SATD comments of this kind may be found in the form of a question, such as the comment `TODO - does this handle N-dim tensors correctly?`. Also, *Awareness* debts may be caused by the nature of working with ML models whose functionality can be considered a black box [39, 40]. When such cases arise, the demand for new functionality may arise to better understand model behavior. For example, the comment in our dataset `TODO: Model Precision` admits a new evaluation metric to be used to better understand the performance of a ML model, and was considered an *Awareness* debt under this context.

Table 4: Definitions and Examples of the 23 Identified 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). ♣ New ML SATD Type proposed in our study.

Dim.	Name	Definition	Example
AWR	Machine Learning Knowledge♣	Machine learning software carries plenty of unique challenges. Uneducated solutions by unaware developers may have to be revisited.	`FIXME: can I backprop error through both`
	Model Interpretability♣	Machine learning models are a black box, causing poor understanding of model's functionality. This can lead to unknown behavior.	`TODO: Model Precision`
RDB	Model Code Comprehension [52]	Model code carries extra legibility concerns that do not occur in traditional software. (i.e., poorly named temporary matrix variables).	`TODO refactor second part of if statement when implementing live model prediction`
DCE	Duplicate Model Code [52]	Code duplication frequently occurs in model code.	`TODO: Basically identical to 'test_intra_cv_target_transform' except for repeated KFold`
	Duplicate Feature Extraction Code [52]	Code duplication frequently occurs in feature extraction code.	`XXX de-duplicate this with code from Montage somewhere?`
CFO	Weight Configuration ♣	Editing code that involves the weights of a ML model, or configuring a ML model's weights directly.	`FIXME non-uniform sample weights not yet supported`
	Layer Configuration	Editing code that deals with ML models' layers, or configuring a ML model's layers itself.	`TODO: experiment with more layers`
	Hyper-parameter Configuration	Configuring hyper-parameters of ML model, or editing the default values of off-the-shelf model.	`TODO convert this to x/y params?`
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.	`TODO add test for keypoints once their handling was improved in Convolve`
	Glue Code [45]	Supporting code written to interface with other code, inhibiting improvements due to peculiarities of dependent code.	`FIXME XXX: Implement by rewriting functions above copied from autoresolve.py`
	Custom Data Type [52]	Using data types provided by general-purpose packages can cause extensive inter-operating with external libraries.	`TODO: Repeat_elements and tf.split doesn't support dynamic splits.`
	Multiple Languages [45]	Components written in other languages may introduce difficulties in ML development.	`TODO(sonots): Implement in C++`
	Unnecessary Model Code [52]	Model code that either bottlenecks performance, is unreachable or deprecated, or is unnecessary and should be removed.	`DEPRICATE? I don't think this is needed anymore`
DTD	Data Processing Configuration ♣	Configuring the way that data is processed either by editing the data directly, or by adding in new processing steps.	`TODO: normalize true states`
	Plain Old Data Type [45]	Using raw data types in ML causes confusion when interpreting processes.	`TODO: handle record value which are lists; at least error`
	Data Storage Configuration ♣	Configuring how data is represented within the source code (data structure) or how data is stored externally (database).	`TODO json?`
MDL	Abstraction [45]	Lack of abstractions in ML systems and subsystems cause cascading changes when changes are introduced to one component.	`TODO Split into separate functions`
	Boundary Erosion[45]	Lack of boundaries between subsystems, creating difficulties when maintaining software and isolating changes made in ML software.	`@todo nrows is for testing only!`
	Model Code Modifiability [52]	Model code should be implemented in ways that enable easy maintenance and future modifications.	`TODO init this somewhere else in a more principled way`
	Model Code Reusability [52]	Model code should be generalized to be able to be reused in varying situations.	`TODO: At the moment LHUC is RNN specific.`
PRF	Prediction Quality ♣	Previous work in evaluating ML workflows [11] shows that changes may affect performance.	`TODO: compare the performance!`
	Machine Learning Reliability ♣	Machine learning models' functionalities are determined by the quality of their data, measures should be in place to ensure robustness.	`TODO: extract an eval func more robustly`
SCL	Prototype [45]	Small-scale prototypes being deployed into full systems can be dangerous.	`TODO: This matching process is slow. Make it faster; avoid loops where possible.`

Table 5: Distribution of ML SATD Groups.

ML SATD Group Name	# of Occur.	% of Occur.
Data Dependency	170	30.58%
Code Dependency	127	22.84%
Awareness	97	17.45%
Modularity	55	9.89%
Configurable Options	51	9.17%
Scalability	31	5.58%
Readability	12	2.15%
Performance	11	1.97%
Duplicate Code Elimination	2	0.36%

Table 6: Results from prior work by Bavota and Russo [4].

SATD Type	# of SATD Comments	# of Sampled SATD Comments
Code Debt	81	29.67%
Defect Debt	55	20.15%
Requirement Debt	55	20.15%
Design Debt	34	12.45%
Documentation Debt	27	9.89%
Test Debt	21	7.69%
Total	**273**	**100%**

Symptoms of *Awareness* debts include doubts on the correctness of algorithmic procedures, doubts on the current design decisions, lack of knowledge of proper API usage, erroneous cases where a solution is not identified, and lack of domain knowledge. Doubts on algorithmic procedures, either by interfacing with other software or correctness of algorithms can lead to *Awareness* debt (`TODO figure out what exactly size does`, `FIXME: What if we never find a fail-high?`). Doubts on current design decisions can be due to questioning the qualities of a current implementation (`TODO: Something smarter?`) or questioning the design decisions in place (`TODO: do we want gainsbiases to be trainable?`). Lack of proper API understanding can also be a symptom of *Awareness* debts (`TODO: Check if self._mean_squared_error_w_precision can be used here`). Additionally, cases where errors are encountered or incorrect behavior is discovered, but a solution is not yet known can also be a symptom (`TODO: skip on error???`). Finally, since ML software can be applied in various domains, it is not surprising that a lack of domain knowledge contributes to questionable software behavior (`TODO Figure out how to distinguish oxidation states`).

Consistently updated documentation can alleviate such SATDs, disclosing proper API usage and possible design intentions. Since prior work shows that most SATD is addressed by a different developer than the one who self-admitted it [4, 37], including documentation details such as limitations, advantages, and defense of the current implementation can inform future developers of identifiable improvements. For example, the comment `TO-DO: this causes very long running when unique numbers are high. Find a workaround for this` describes the symptoms and limitations of present implementations, and gives an actionable starting point for future developers. Therefore, the more information which can be disclosed by a developer in a *Awareness* debt scenario may enable timely removals of these debts regardless of the symptoms they contain.

Our findings indicate that ML SATDs may reflect the fast-paced environment of ML software through a large amount of *Requirement* debts, setting itself apart from traditional software development. Furthermore, data dependent code and lack of developer awareness are major factors contributing to SATD.

3.2 RQ2: What is the distribution of SATD types in the different ML pipeline stages?

This section further analyzes distributions of ML SATDs by also considering which stage of the ML pipeline every SATD comment appears within. Because the ML pipeline consists of unique tasks which ML software commonly works through, we question whether some ML SATDs occur more frequently in varying stages. We utilized prior works studying the ML pipeline [3, 10, 30] to construct our taxonomy of ML pipeline stages which include *Data Acquisition, Data Preprocessing, Modeling, Training, Prediction, Evaluation,* and *Other*. The *Other* stage is reserved for tasks that can occur anywhere in the ML pipeline for a variety of reasons (i.e., data visualization and data storage) as well as cases where the authors were not confident on a pipeline label due to lack of distinguishable information. Table 7 presents the distribution of the ML SATD Groups amongst the pipeline stages.

Finding 5: *Data Preprocessing* is the pipeline stage with the most SATD.

Data Preprocessing is the stage in the ML pipeline that has the most SATD comments out of all pipeline stages. RQ1 found that *Data Dependency* is the most used ML SATD Group. However, since *Data Preprocessing* contains the most SATD in our dataset, it suggests that ML developers leave SATD comments of a wide variety in data preprocessing code. For example, the comment `TODO validation_split is not used` exemplifies an *Unnecessary Model Code* debt that was found in the *Data Preprocessing* stage. Similarly, the comment `FIXME: does pytorch has something similar to tf.add_n which sum over a list?` is a *Machine Learning Knowledge* and *Custom Data Type* debt that appeared in the *Data Preprocessing* stage as well. The high SATD activity in the *Data Preprocessing* stage could be caused by feature engineering components having the largest body of code, therefore there is more code to self-admit technical debts. Sculley et al. [45] showed that a mature ML system may be at most 5% ML code. The rest of the system may consist of subsystems such as process managing tools or feature engineering code, among others. Regardless, the high activity of SATD within *Data Preprocessing* code stresses the importance for rigorous review on code which handles data preprocessing.

Finding 6: *Data Dependency* is the SATD Type which is dominant in 5 out of the 7 pipeline stages.

Our observations find that *Data Acquisition, Data Preprocessing, Evaluation, Prediction, and Other* stages exhibit *Data Dependency* as their primary SATD Type. This observation indicates that subsystems across the ML pipeline encounter SATD amounts which involve adjustments to data interactions. To demonstrate that *Data Dependency* debts can take different shapes amongst the pipeline stages consider the *Evaluation* stage comment `FIXME don't l2_normalize for any metric` which involves the normalization means in the *Evaluation* stage of a speaker diarization pipeline. Meanwhile, the comment `TODO combine these values to get a final prediction!` describes functionality to be implemented for an SVM model in the *Prediction* stage. **Our manual analysis suggests that potential solutions to repairing data-dependent**

Table 7: ML SATD Groups by Pipeline Stage

ML SATD Groups	ML Pipeline Stages						
	Acqu.	Prep.	Mod.	Train.	Pred.	Evalu.	Oth.
Awareness	12	39	18	9	2	5	12
Code Dependency	19	39	33	13	1	6	16
Configurable Options	-	3	23	23	-	1	1
Data Dependency	26	82	8	13	4	6	31
Duplicate Code Elimination	-	2	-	-	-	-	-
Modularity	7	12	20	9	3	-	4
Performance	-	-	2	2	-	5	2
Readability	1	2	4	3	1	1	-
Scalability	6	11	4	5	-	2	3

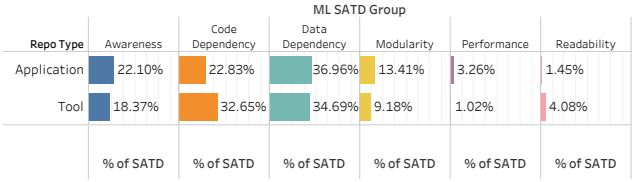

Figure 1: ML SATD Groupings Distribution by Repo Type

SATDs may be completely different between pipeline stages, despite their broad similarities. Another explanation to this observation is the concept of Pipeline Jungles introduced by Sculley *et al.* [45] where there is little independent responsibility given to each pipeline stage. Because of this, data preprocessing is not isolated to one coherent stage, rather data preprocessing is performed "as needed" [10]. Thus, the placement of data preprocessing code within other ML pipeline stages accrues technical debt, since the debugging or refactoring must consider the data transformation implementations across multiple stages.

We observe that *Data Preprocessing* is the most popular pipeline stage, stressing the importance of mindful data handling implementations. Similarly, *Data Dependency* debts are the biggest contributor to 5 pipeline stages in our dataset, the two exceptions being the *Modeling* and *Training* stages.

3.3 RQ3: Is there a difference in SATD types between ML applications vs ML tools?

In this section, we separate our findings into the ML repositories that apply ML towards a task (applications) and ML repositories that provide ML functionalities (tools) as labeled by the original dataset creators [26].

Figure 1 illustrates the ML SATD group distributions by repository type to visualize differences between them. For viewing purposes, only the ML SATD Groups which differ by more than 2% are shown.

Finding 7: Our analysis of SATD comments suggests that *Code Dependency* debts appear more often in ML tools than ML applications.

According to Figure 1, *Code Dependency* related debts may be more common in ML tools than ML applications. *Code Dependency* refers to a similar concept to "On-Hold SATD" explained by Maipradit *et al.* [35]. In these instances, an SATD is waiting for a condition to be met before taking action (e.g., waiting for an update or other development tasks to complete) such as the comment `TODO Modify mu and sigma once feature scaling is built into the logistic regression`. We describe *Code Dependency* as SATD which depends upon other code. Cases such as interfacing with specific data types (`TODO: Repeat_elements and tf.split doesn't support dynamic splits`), changes to software and APIs which cause versioning updates (`TODO: assertWarns exist only for Python 3.2+; test in all versions`), or cascading changes throughout software systems (`TODO: it's worth to switch back to the correct preprocess_input when InceptionResNetV2 model is re-trained`).

ML tools find themselves in a competitive environment amongst themselves to provide efficient techniques for their users. Previous work has described the temptation that a shorter time-to-market brings to deep learning framework developers [34]. It could be that in order to stay relevant, ML tools' implementations must interoperate with other evolving ML tools. Thus, *Code Dependency* debts may occur when other tasks take priority above resolving these identified problems. Our observations suggest ML tool developers suffer from these occurrences more than ML application developers.

We believe that ML tool developers can benefit from these observations by further evaluating the implementations which they depend on, and what the advantages of evolving in correspondence with those implementations may bring. Furthermore, we encourage ML tool developers to consider how severe of a refactoring that an API change may have upon depending systems, and to allow for upgrading versions to be performed with ease.

Finding 8: In our dataset, ML applications encounter more *Modularity* debts.

Modularity debts describe cases where weak modular boundaries exist between ML subsystems. An example debt in the *Modularity* group is *Boundary Erosion* [45], an example comment of which is `@todo nrows is for testing only!`, which describes a case where a parameter intended for testing purposes was instead used within data visualization code. Another *Modularity* example is the *Abstraction* debt comment `TODO: refactor as independent function` which admits the task of creating a new abstraction within a feature engineering process.

Our analysis suggests that *Modularity* debts are found more often in ML applications than ML tools, hinting at a possible difference between their software maintenance patterns. A possible explanation may lie in the common use of Jupyter Notebooks by application developers as explored by Pimentel *et al.* [41]. In a Jupyter Notebook, application developers use an interactive environment to debug and monitor their code. **However, the transition from Jupyter Notebook to an Object Oriented (OO) design may be difficult, and instances of *Modularity* SATD may be left where these transitions were attempted.** Tang et al. [52] also speculates how ML developers may not be familiar with Object Oriented Programming best practices, leading to the introduction of such technical debts.

It was found that ML tools suffer from much more *Code Dependency* debts, possibly a side effect from inter-operating with other libraries. Additionally, our results suggest that ML application developers incur more *Modularity* debts.

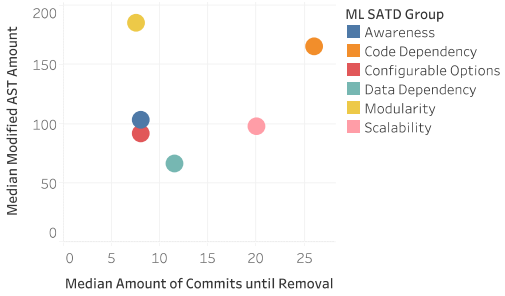

Figure 2: ML SATD Groupings Removal Characteristics.

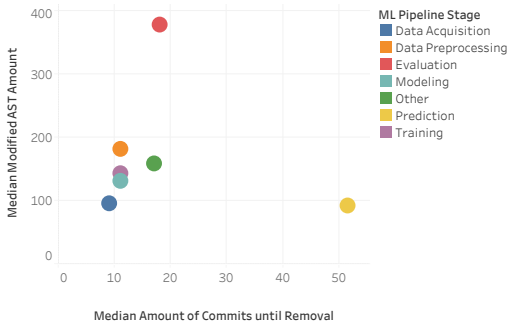

Figure 3: ML SATD Removal Amongst Pipeline Stages

3.4 RQ4: How much effort is needed to remove SATD types in ML software?

This section analyzes the effort conducted to remove SATD in ML software. In order to accomplish this, we only analyze our SATD comments which have been removed at the time of data generation.

We define effort of removing an SATD comment as a comparison between the time passed since its introduction and size of the commit which removes it. Since an SATD may take multiple commits to remove because of unknown solutions or lower prioritization, we follow the advice presented by Bird et al. [6] to connect a comment-removing commit to a comment-introducing comment of the same text. Because all commits are not the same size, we also use the Boa language and its infrastructure created by Dyer et al. [15] to perform the GumTree algorithm [20] to compute the edit script of a source code change at the Abstract Syntax Tree (AST) level. With this methodology, we count the number of AST nodes modified in the revision which removes the SATD comment to get an indication of work which removed it. Due to larger repositories causing timeout errors, some commit revisions are unable to be quantified in our dataset.

Figure 2 illustrates our results on the ML SATD Groups. Since we want to present on data with larger bodies, the ML SATD Groups shown are those which contain 10 or more removed instances. In addition, Figure 3 illustrates our results across the ML pipeline stages. We measure our statistics in median because it is unaffected by outliers, since it has been shown that an SATD comment may have been removed without any additional code change or alongside large refactorings. [37].

Finding 9: *Data Dependency, Configurable Options, and Awareness* may require less removal effort.

The low number of commits and low AST count in the removing commit suggest that these debts are more convenient to remove. For example, the SATD comment: `TODO: modify this later` is a *Data Dependency* debt describing a sampling modification was removed in 9 commits with 20 modified AST nodes. Because of these lower quantities involved in these comments' removals, it could be that ML developers understand these SATDs better, since much of ML software involves operating with data dependent code, configurable APIs, and unknown solutions. Dilhara *et al.* examined common code changes made in ML software by reapplying RefactorMiner to Python programs [14]. However, further work is needed to understand the challenges involved when ML developers identify and implement differing code changes.

Finding 10: *Modularity* debts have a higher amount of AST nodes in their removing commits.

Our analysis suggests that *Modularity* debt removals have larger commits and shorter removal times. Therefore, high activity in a low amount of time may suggest that identified *Modularity* debts are the most urgent to remove. Consider the comment `TODO Promote this to a Funsor subclass.` which involves reorganizing software classes whose removal involved 355 modified AST nodes. A possible explanation is that the heavier refactoring of *Modularity* debts are viewed as more valued improvements to ML developers, and are prioritized sooner despite their larger change size.

Therefore, ML developers can benefit from this finding by carefully considering their modular decisions over potentially "simpler" decisions earlier in development. Since our results indicate heavier activity is required from *Modularity* debt resolutions, earlier decisions involving abstractions and modular design are more important in a software's lifetime than issues which are lighter to resolve.

Finding 11: Our analysis suggests that SATD in Prediction and Evaluation stages have larger removal characteristics.

Figure 3 shows the complexities of debt removals by the ML pipeline stages. Our study suggests that debts removed in the *Evaluation* stage have a large amount of modified AST nodes, and debts removed in the Prediction stage are removed after a larger number of commits have passed.

These findings may indicate that debts removed in the *Evaluation* stage may require larger activity, possibly due to reused evaluation metrics across multiple model codes. Therefore, a change made in this stage may have a wide reach of consequential changes. If a debt in this stage goes unnoticed, its resolution may take a heavier amount of work, as is suggested by the larger amount of AST nodes modified in comment removals in *Evaluation* stage code.

Our observations indicate that debts found in the *Prediction* stage go unresolved for larger number of commits, possibly hinting at ML developer's priorities when working across the pipeline for larger periods of time. Rather than improve upon predicting involved code, it may be that ML developers choose to prioritize the immediate removal of SATDs elsewhere. Thus, the *Prediction* stage may be an area of ML software where the removal of an SATD is not as urgent.

Our study suggests that *Awareness*, *Configurable Options*, and *Data Dependency* debts are easier to remove than *Modularity* debts. Moreover, there are heavier debts to pay off in the *Evaluation* pipeline stage, and debts in the *Prediction* stage last the longest.

4 DISCUSSION

ML techniques are increasingly adopted by developers to solve previously difficult-to-solve problems, thus there is a benefit to understanding how these systems evolve. Although Sculley et al. [45] previously explored the hidden technical debts encountered at Google and Tang *et al.* examined how refactoring accompanies technical debts in ML software [52], to the best of our knowledge no previous study has examined how ML-specific technical debts permeate as self-admitted technical debt. The study of SATD has previously been fruitful to practitioners and researchers since an organization has reported that SATD guides their technical debt management [54]. This large-scale study of SATD in ML software can also provide researchers and practitioners with insights on ML software evolution from our analysis of ML SATD.

The ML SATD taxonomy depicted in Table 4 is a synthesis of prior works studying TD in ML software and our manual analysis of 856 SATD comments. This taxonomy not only includes new types of TD through SATD analysis, but also splits previous types which were previously described a large assortment of issues. Our proposed taxonomy is hierarchical in nature, providing ML developers and researchers a classification scheme for reasoning about issues in ML software maintenance. We believe this taxonomy can guide ML developers in ML development, as well as provide researchers with areas to further explore.

Additionally, the ML pipeline has been a focus of researchers recently [3, 10], yet no prior study has examined the SATD which persist through the individual stages' evolution. Our analysis of SATD in these pipeline stages is a step in understanding the unique challenges each stage encounters overtime. Moreover, this study is the first to analyze how differing types of ML projects encounter SATD to researchers interested in improving ML developer experience. Finally, we provide a study quantifying the efforts needed to remove various SATDs in ML software, although further work can complement these findings.

4.1 Implications

(1) Our study suggests that SATD in ML software can be inspected on various levels, such as how traditional software evolves in addition to ML specific ways. Our proposed taxonomy can guide ML practitioners in their ML software maintenance, describing 23 unique problems which have become evident through our SATD analysis. A key finding is that ML software encounters debts frequently involve the manipulation of data processing means, including the introduction, removal, improvement, and reuse of code which handles the data from data acquisition to model evaluation. Therefore, ML developers should consider the data requirements (distribution, organization, appearance, etc... [23]) for every pipeline stage in their software and where these requirements are not met.
(2) Traditional software developer and ML developers differ in their SATD activity. ML developers may prefer to evolve their software through functional and non-functional requirement changes as new capabilities become available or their software interacts with the real world. Because of this, ML software may benefit from distinct solutions and tools than traditional software. Consequentially, researchers may be interested if the reuse of traditional developer-assisting tools may not adequately address the SATD issues encountered by ML developers.
(3) We find that ML developers' SATD changes depending on the type of ML software being maintained. This is an interesting direction, showing that different types of developers may gain greater benefits from various techniques. For instance, we find that ML tool developers accrue larger debts from interacting with fellow evolving libraries [13] and ML application developers can benefit from more focused modular designs implemented earlier in production.

5 THREATS TO VALIDITY

In this section, we discuss the threats to internal, construct and external validity of our study.

Internal validity: concerns factors that could have influenced our results. Our findings depend largely on the data labeled. In order to ensure that our labeled data is precise, two authors went through an extensive period where their Cohen's Kappa coefficient was calculated, and disagreements in labeling were settled in the presence of a third author until their Cohen's Kappa coefficient was above 80%.

When associating comment-introducing revisions with comment-removing revisions, we only consider the cases where the comment appeared the same during both commits. The comment could have been modified in-between, or other unusual cases might have occurred [6]. Because of this, our separation of comments that have been removed and comments that still exist within their projects may be inaccurate. This should only affect cases where a comment was incorrectly placed as not removed. All comments that were found to have been removed should be true positives, which are the cases analyzed in RQ4.

It is possible our methodologies of measuring "effort" rather measures "priority" or "system impact". However, we argue that there may be no perfect way to measure "effort" since development patterns and activity likely differ across software projects. For this reason, we utilized multiple measures (change-size and time) to quantify effort, which can be adopted in future research to complement our findings further.

Construct validity: considers the relationship between theory and observation. It has been shown that the removal of an SATD comment is a naive indication that a TD was also removed [37, 58]. A comment could be removed without any fix to the SATD or a SATD comment could be removed at a later time than the resolution, or the comment could never be removed at all. To mitigate this, the authors examined the code around the SATD comment and were able to identify a few cases where the SATD comment is no longer relevant. These cases were then removed from our labeled data. Additionally, the quantitative data of RQ4 was measured by its median to avoid influence by outliers.

The projects analyzed within this project may not be indicative of present ML software practices. However, the dataset was created

through extensive queries of the GitHub API with a variety of keywords consisting of topics, subtopics, technologies, and techniques related to ML [26]. Because of this, we believe our study has a wide representation of ML topics.

Although the SATD comment classifier used has historically performed well, it still may produce false predictions when our dataset was filtered. However, when labeling, the authors would remove any false positive samples. The inclusion of using a keyword search along with using the comment classifier's output was used to recover any false negative samples the comment classifier produced.
External validity: concerns the generalization of our results. In this study, we examined only open-source Python repositories that are divided into applications and tools used in previous studes [26, 50]. Hence, our findings may not be generalized to to other non-Python repositories or non-open-source repositories. That said, our decision of studying Python repositories was made by its popularity among the ML community [26]. Also, our dataset size of 2,641 ML Python repositories that may not represent the whole population of ML software written in Python.

6 RELATED WORK

In this section, we describe the related work. We divided the related work into two sections: work related to technical debt management in general and work related to the management of technical debt in machine learning repositories.

General management of technical debt. Several earlier works studied the management and detection of technical debt (e.g., [12, 19, 32, 47]). For example, Brown et al. [12], Kruchten et al. [32], and Seaman and Guo [47] made several observations about the term 'technical debt' and mentioned that it is regularly used to communicate development issues to managers. Similarly, Zazworka et al. [59] performed a study to measure the impact of technical debt on software quality. Other works by Fontana et al. [24] investigated design technical debt indicated in the form of code smells. Furthermore, Ernst et al. [19] conducted a survey involving more than 1,800 participants and found that architectural decisions are the most important source of technical debt. Similarly, in this paper, we study technical debt in software applications. However, our study focuses on gaining a better insight into the existence of the technical debt in ML repositories.

Recently, [42] proposed the concept of self-admitted technical debt (SATD), which considers debt that is intentionally introduced or identified by developers. [42] analyzed more than a hundred thousand code comments from four projects to come up with 62 patterns that indicate self-admitted technical debt. Also, their findings reveal that approx. 31% of the files in a project contain self-admitted technical debt. More specifically, they found that 1) the majority of the self-admitted technical debt is removed in the immediate next release; 2) developers with higher experience are mostly the ones who introduce the self-admitted technical debt; 3) release pressure does not play a major role in the introduction of self-admitted technical debt. In a follow-up study, Bavota and Russo [4] replicated the study of SATD on a large set of projects and confirmed the findings observed by Potdar and Shihab in their earlier work [42].

Other works investigate different aspects related to SATD, including; automatically identify SATD from source code (e.g., [36, 43]), examine the maintenance and the removal of SATD [37, 58], and discuss how the existence of SATD may lead to the rejection of pull requests by developers [49]. We refer the reader to a recent survey by [48] for a more comprehensive list of studies on self-admitted technical debt. Similar to the work mentioned above, our study uses source code comments to detect self-admitted technical debts in ML repositories.

Management of technical debt in machine learning applications. In recent years, examining the characteristics of machine learning applications has gained momentum. One of the first studies related to technical debt in ML applications introduced the concept of implicit TD in ML software through experiences encountered during ML projects at Google [45]. This work was then furthered by [52], where new ML-specific refactoring and TD categories were introduced, accompanied by recommendations of best practices to facilitate long-term ML software development. Similarly, the work by [11] discusses a workflow to evaluate ML software development practices at the data, model, infrastructure, and monitoring levels that was then performed across multiple ML projects at Google. Liu et al. [34] examined the existence of TD in deep learning frameworks. Their study showed that design debt, defect debt, documentation debt are most presented TD in deep learning frameworks.

Several studies examine the development of ML repositories in general. Humbatova et al. [28] and Islam et al. [30, 31] examined deep learning systems to systematically build a taxonomy of bugs that impact deep learning systems, and recent works propose techniques to address these identified issues [55, 56]. Nguyen et al. [38] proposes a technique to leverage repositories of models to assist ML developers. Amershi et al. [3] investigates the unique challenges faced by software developers when developing ML applications. ML models can exhibit new qualities unfamiliar to traditional software such as model fairness [8, 9]. Dilhara et al. [13] conducted an empirical study to examine the evolution and usage of ML libraries.

7 CONCLUSION

Nowadays, ML solutions are being adopted by software developers to accomplish otherwise infeasible tasks. Similar to traditional software, there are unique costs to impractical design decisions in ML software that result in "technical debts". In this study, we have analyzed self-admitted technical debts in ML software through a statistically significant sample of SATD comments from open-source repositories. Furthermore, we propose a classification scheme consisting of Software SATD Types, ML SATD Types and 8 new ML SATD Groups. This classification scheme can provide practitioners and researchers a structure for discovering and managing their SATD. Additionally, we provide an analysis of SATD characteristics as they appear amidst pipeline stages and repository type as well as an analysis of SATD removal effort. The results discussed can assist developers and researchers to create more maintainable and improvable ML software solutions.

ACKNOWLEDGMENTS

This work was supported in part by US NSF grants CNS-21-20448 and CCF-19-34884 We also thank the reviewers for their insightful comments. All opinions are of the authors and do not reflect the view of sponsors.

REFERENCES

[1] Reem Alfayez, Wesam Alwehaibi, Robert Winn, Elaine Venson, and Barry Boehm. 2020. A Systematic Literature Review of Technical Debt Prioritization. In *Proceedings of the 3rd International Conference on Technical Debt* (Seoul, Republic of Korea) *(TechDebt '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3387906.3388630

[2] Nicolli S.R. Alves, Leilane F. Ribeiro, Vivyane Caires, Thiago S. Mendes, and Rodrigo O. Spínola. 2014. Towards an Ontology of Terms on Technical Debt. In *2014 Sixth International Workshop on Managing Technical Debt*. 1–7. https://doi.org/10.1109/MTD.2014.9

[3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[4] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *International Conference on Mining Software Repositories*. ACM, 315–326.

[5] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The Open-Closed Principle of Modern Machine Learning Frameworks. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 353–363.

[6] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. 2009. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. 1–10. https://doi.org/10.1109/MSR.2009.5069475

[7] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. 2019. Boa Meets Python: A Boa Dataset of Data Science Software in Python Language. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 577–581. https://doi.org/10.1109/MSR.2019.00086

[8] Sumon Biswas and Hridesh Rajan. 2020. Do the Machine Learning Models on a Crowd Sourced Platform Exhibit Bias? An Empirical Study on Model Fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 642–653. https://doi.org/10.1145/3368089.3409704

[9] Sumon Biswas and Hridesh Rajan. 2021. Fair Preprocessing: Towards Understanding Compositional Fairness of Data Transformers in Machine Learning Pipeline. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 981–993. https://doi.org/10.1145/3468264.3468536

[10] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. 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 *ICSE'22: The 44th International Conference on Software Engineering* (Pittsburgh, PA, USA).

[11] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. 2017. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. In *Proceedings of IEEE Big Data*.

[12] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. 2010. Managing Technical Debt in Software-reliant Systems. In *FSE/SDP Workshop on Future of Software Engineering Research*. ACM, 47–52.

[13] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 55 (2021), 42 pages.

[14] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering Repetitive Code Changes in Python ML Systems. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 736–748. https://doi.org/10.1145/3510003.3510225

[15] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. 422–431. https://doi.org/10.1109/ICSE.2013.6606588

[16] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2015. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 7 (2015), 7:1–7:34 pages.

[17] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE'14)*. 779–790.

[18] Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. 2013. Declarative Visitors to Ease Fine-grained Source Code Mining with Full History on Billions of AST Nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (Indianapolis, IN) *(GPCE)*. 23–32.

[19] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 50–60.

[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[21] Joseph L Fleiss, Bruce Levin, Myunghee Cho Paik, et al. 1981. The measurement of interrater agreement. *Statistical methods for rates and proportions* 2, 212-236 (1981), 22–23.

[22] Samuel W. Flint, Jigyasa Chauhan, and Robert Dyer. 2021. Escaping the Time Pit: Pitfalls and Guidelines for Using Time-Based Git Data. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 85–96. https://doi.org/10.1109/MSR52588.2021.00022

[23] H. Foidl, M. Felderer, and R. Ramler. 2022. Data Smells: Categories, Causes and Consequences, and Detection of Suspicious Data in AI-based Systems. In *2022 IEEE/ACM 1st International Conference on AI Engineering – Software Engineering for AI (CAIN)*. IEEE Computer Society, Los Alamitos, CA, USA, 229–239. https://doi.ieeecomputersociety.org/

[24] F. A. Fontana, V. Ferme, and S. Spinelli. 2012. Investigating the impact of code smells debt on quality code evaluation. In *International Workshop on Managing Technical Debt*. IEEE, 15–22.

[25] Gianmarco Fucci, Nathan Cassee, Fiorella Zampetti, Nicole Novielli, Alexander Serebrenik, and Massimiliano Di Penta. 2021. Waiting around or job half-done? Sentiment in self-admitted technical debt. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 403–414. https://doi.org/10.1109/MSR52588.2021.00052

[26] Danielle Gonzalez, T. Zimmermann, and N. Nagappan. 2020. The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub. *Proceedings of the 17th International Conference on Mining Software Repositories* (2020).

[27] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying Self-Admitted Technical Debt in Open Source Projects Using Text Mining. *Empirical Softw. Engg.* 23, 1 (feb 2018), 418–451. https://doi.org/10.1007/s10664-017-9522-4

[28] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 1110–1121.

[29] Nick Hynes, D. Sculley, and Michael Terry. 2017. The Data Linter: Lightweight Automated Sanity Checking for ML Data Sets. http://learningsys.org/nips17/assets/papers/paper_19.pdf

[30] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[31] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1135–1146. https://doi.org/10.1145/3377811.3380378

[32] Philippe Kruchten, Robert L. Nord, Ipek Ozkaya, and Davide Falessi. 2013. Technical debt: towards a Crisper Definition. Report on the 4th International Workshop on Managing Technical Debt. *ACM SIGSOFT Software Engineering Notes* 38, 5 (2013), 51–54.

[33] Valentina Lenarduzzi, Terese Besker, Davide Taibi, Antonio Martini, and Francesca Arcelli Fontana. 2021. A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software* 171 (2021), 110827. https://doi.org/10.1016/j.jss.2020.110827

[34] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2020. Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks *(ICSE-SEIS '20)*. 1–10.

[35] Rungroj Maipradit, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2019. Wait For It: Identifying "On-Hold" Self-Admitted Technical Debt. *CoRR* abs/1901.09511 (2019). arXiv:1901.09511 http://arxiv.org/abs/1901.09511

[36] Everton Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *IEEE Transactions on Software Engineering* (2017), to appear.

[37] Everton Da S. Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. 2017. An Empirical Study on the Removal of Self-Admitted Technical Debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 238–248. https://doi.org/10.1109/ICSME.2017.8

[38] Giang Nguyen, Md Johirul Islam, Rangeet Pan, and Hridesh Rajan. 2022. Manas: Mining Software Repositories to Assist AutoML. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1368–1380. https://doi.org/10.1145/3510003.3510052

[39] Rangeet Pan and Hridesh Rajan. 2020. On Decomposing a Deep Neural Network into Modules. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 889–900. https://doi.org/10.1145/3368089.3409668

[40] Rangeet Pan and Hridesh Rajan. 2022. Decomposing Convolutional Neural Networks into Reusable and Replaceable Modules. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 524–535. https://doi.org/10.1145/3510003.3510051

[41] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 507–517. https://doi.org/10.1109/MSR.2019.00077

[42] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 91–100.

[43] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural Network-Based Detection of Self-Admitted Technical Debt: From Performance to Explainability. *ACM Trans. Softw. Eng. Methodol.* 28, 3, Article 15 (2019), 45 pages.

[44] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*.

[45] D. Sculley, Gary Holt, D. Golovin, Eugene Davydov, Todd Phillips, D. Ebner, Vinay Chaudhary, Michael Young, J. Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *NIPS*.

[46] C.B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[47] C. Seaman and Y. Guo. 2011. Measuring and Monitoring Technical Debt. *Advances in Computers* 82 (2011), 25–46.

[48] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software* 152 (2019), 70–82.

[49] Marcelino Campos Oliveira Silva, Marco Tulio Valente, and Ricardo Terra. 2016. Does technical debt lead to the rejection of pull requests? *arXiv preprint arXiv:1604.01450* (2016).

[50] Sebastian Sztwiertnia, Maximilian Grübel, Amine Chouchane, Daniel Sokolowski, Krishna Narasimhan, and Mira Mezini. 2021. Impact of Programming Languages on Machine Learning Bugs. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis* (Virtual, Denmark) *(AISTA 2021)*. Association for Computing Machinery, New York, NY, USA, 9–12. https://doi.org/10.1145/3464968.3468408

[51] Jie Tan, Daniel Feitosa, and Paris Avgeriou. 2022. Does It Matter Who Pays Back Technical Debt? An Empirical Study of Self-Fixed TD. *Inf. Softw. Technol.* 143, C (mar 2022), 15 pages. https://doi.org/10.1016/j.infsof.2021.106738

[52] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 238–250. https://doi.org/10.1109/ICSE43902.2021.00033

[53] Dimitrios Tsoukalas, Miltiadis Siavvas, Marija Jankovic, Dionysios Kehagias, Alexander Chatzigeorgiou, and Dimitrios Tzovaras. 2018. Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey. In *2018 International Conference on Intelligent Systems (IS)*. 698–705. https://doi.org/10.1109/IS.2018.8710521

[54] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. 2016. Continuous Delivery Practices in a Large Financial Organization. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 519–528. https://doi.org/10.1109/ICSME.2016.72

[55] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 561–572. https://doi.org/10.1145/3510003.3510071

[56] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault Localization for Deep Neural Networks. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 251–262. https://doi.org/10.1109/ICSE43902.2021.00034

[57] Laerte Xavier, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. 2020. Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 137–146. https://doi.org/10.1145/3379597.3387459

[58] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was Self-Admitted Technical Debt Removal a Real Removal? An In-Depth Perspective. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 526–536.

[59] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the Impact of Design Debt on Software Quality. In *International Workshop on Managing Technical Debt*. ACM, 17–23.