2021

# An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems

Yiming Tang
*CUNY Graduate Center*

Raffi T. Khatchadourian
*CUNY Hunter College*

Mehdi Bagherzadeh
*Oakland University*

Rhia Singh
*CUNY Macaulay Honors College*

Ajani Stewart
*CUNY Hunter College*

*See next page for additional authors*

## Authors

Yiming Tang, Raffi T. Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja

# An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems

Yiming Tang[*], Raffi Khatchadourian[†*], Mehdi Bagherzadeh[‡], Rhia Singh[§], Ajani Stewart[†], Anita Raja[†*]
[*]CUNY Graduate Center, [†]CUNY Hunter College, [‡]Oakland University, [§]CUNY Macaulay Honors College
Email: ytang3@gradcenter.cuny.edu, raffi.khatchadourian@hunter.cuny.edu, mbagherzadeh@oakland.edu,
rhia.singh@macaulay.cuny.edu, ajani.stewart42@myhunter.cuny.edu, anita.raja@hunter.cuny.edu

***Abstract*—Machine Learning (ML), including Deep Learning (DL), systems, i.e., those with ML capabilities, are pervasive in today's data-driven society. Such systems are complex; they are comprised of ML models and many subsystems that support learning processes. As with other complex systems, ML systems are prone to classic technical debt issues, especially when such systems are long-lived, but they also exhibit debt specific to these systems. Unfortunately, there is a gap of knowledge in how ML systems actually evolve and are maintained. In this paper, we fill this gap by studying refactorings, i.e., source-to-source semantics-preserving program transformations, performed in real-world, open-source software, and the technical debt issues they alleviate. We analyzed 26 projects, consisting of 4.2 MLOC, along with 327 manually examined code patches. The results indicate that developers refactor these systems for a variety of reasons, both specific and tangential to ML, some refactorings correspond to established technical debt categories, while others do not, and code duplication is a major crosscutting theme that particularly involved ML configuration and model code, which was also the most refactored. We also introduce 14 and 7 new ML-specific refactorings and technical debt categories, respectively, and put forth several recommendations, best practices, and anti-patterns. The results can potentially assist practitioners, tool developers, and educators in facilitating long-term ML system usefulness.**

***Index Terms*—empirical studies, refactoring, machine learning systems, technical debt, software repository mining**

## I. Introduction

In the big data era, Machine Learning (ML), including Deep Learning (DL), systems are pervasive in modern society. Central to these systems are dynamic ML models, whose behavior is ultimately defined by their input data. However, such systems do not only consist of ML models; instead, ML systems typically encompass complex subsystems that support ML processes [1]. ML systems—like other long-lived, complex systems—are prone to classic technical debt [2] issues; yet, they also exhibit debt specific to such systems [3]. While work exist on applying software engineering (SE) rigor to ML systems [4]–[12], there is generally a gap of knowledge in how ML systems actually evolve and are maintained. As ML systems become more difficult and expensive to maintain [1], understanding the kinds of modifications developers are required to make to such systems—our overarching research question—is of the utmost importance.

To fill this gap, we performed an empirical study on common refactorings, i.e., source-to-source semantics preserving program transformations—a widely accepted mechanism for effectively reducing technical debt [13]–[16]—in real-world, open-source ML systems. We set out to discover (i) the kinds of *refactorings*—both specific and tangential to ML—performed, (ii) whether particular refactorings occurred *more often* in model code vs. other supporting subsystems, (iii) the types of *technical debt* being addressed and whether they correspond to established ML-specific technical debt [1], and (iv) whether any *new*—potentially generalizable—ML-specific refactorings and technical debt categories could be derived.

Knowing the kinds of refactorings and technical debt typically associated with ML systems can, e.g., help improve existing—and drive new ML-specific—automated refactoring techniques, IDE code completion, and automated refactoring mining approaches. In general, the results (i) advance knowledge of how and why technical debt is actually manifested in ML systems and how refactorings are employed to alleviate such debt, (ii) help tool designers comprehend the struggles developers have with evolving ML systems, (iii) propose preliminary *recommendations*, *best practices*, and *anti-patterns* for practitioners in evolving long-lasting ML systems effectively, and (iv) assist educators in teaching techniques for combating technical debt in ML systems.

Our study involved analyzing 26 projects, consisting of 4.2 MLOC, along with 327 manually examined code patches. Refactorings were taxonomized, labeled as being performed in ML code or not, and related to the ML-specific debt they alleviated. Our study indicates that (i) duplicate code elimination—largely performed by introducing inheritance—was a major crosscutting theme in ML system refactoring that mainly involved ML *configuration* and *model* code, which was also the *most* refactored code, (ii) subtle *variation* of different yet related ML algorithms and their configurations were a major force driving code duplication, (iii) code generalization, reusability, and external interoperability—*essential* SE concepts—were among the *least* performed refactorings, and (iv) configuration, duplicate model code, and plain-old-data types were the *most* addressed technical debt.

Our contributions can be summarized as follows:

**Refactoring hierarchical taxonomy** From 327 patches of 26 projects manually examined, we build a rich hierarchical, crosscutting taxonomy of common generic and ML-specific refactorings, whether they occur in ML-related code—code specific to ML-related tasks (e.g., classifiers, feature extraction, algorithm parameters)—and the ML-specific technical debt they address.

TABLE I: Studied subjects.

| subject | dom | appl | KLOC | studied per | cmts | kws | exe |
|---|---|---|---:|---|---:|---:|---:|
| AffectiveTweets | NLP | Social | 5.59 | 2016–2019 | 308 | 1 | 1 |
| CoreNLP | NLP | Speech | 546.70 | 2013–2020 | 15,561 | 132 | 40 |
| DataCleaner | Analyt. | Vis. | 144.61 | 2008–2020 | 6,692 | 73 | 19 |
| deeplearning4j | DL | Math | 547.03 | 2019–2020 | 675 | 24 | 16 |
| DigitRecognizer | CV | Images | 1.29 | 2017–2018 | 69 | 2 | 2 |
| elasticsearch | Search | Outliers | 1,585.82 | 2010–2020 | 50,551 | 845 | 34 |
| elki | Data mine | Various | 189.93 | 2005–2020 | 9,993 | 754 | 58 |
| Foundry | ML | AI | 245.23 | 2011–2019 | 372 | 2 | 1 |
| grobid | NLP | Text | 661.28 | 2012–2020 | 1,825 | 18 | 7 |
| jenetics | GP | Optim. | 87.61 | 2008–2020 | 9,966 | 93 | 11 |
| knime-core | Analyt. | Vis. | 215.41 | 2005–2020 | 17,336 | 110 | 10 |
| liblevenshtein | NLP | Text | 7.48 | 2014–2016 | 244 | 2 | 2 |
| mahout | Dist. ML | Math | 122.06 | 2008–2020 | 4,391 | 84 | 24 |
| Mallet | NLP | Text | 76.90 | 2008–2019 | 693 | 8 | 6 |
| moa | Data mine | Streams | 100.62 | 2009–2019 | 1,145 | 3 | 2 |
| modernmt | MT | Speech | 37.83 | 2015–2020 | 3,187 | 128 | 21 |
| Mutters | NLP | Bots | 7.76 | 2016–2020 | 196 | 8 | 4 |
| neo4j-nlp | NLP | DB | 15.88 | 2016–2019 | 703 | 67 | 8 |
| neuronix | CV | Biomed | 3.33 | 2017–2018 | 143 | 3 | 3 |
| smile | ML | Stats | 101.95 | 2014–2020 | 1,853 | 115 | 11 |
| submarine | Dist. DL | Workflow | 45.11 | 2019–2020 | 240 | 2 | 1 |
| tablesaw | Analyt. | Vis. | 50.13 | 2015–2020 | 2,263 | 26 | 6 |
| Trainable_Seg | CV | Images | 23.42 | 2010–2019 | 1,274 | 1 | 1 |
| vespa | Dist. DL | Vis. | 1,439.60 | 2016–2020 | 34,884 | 349 | 29 |
| Weka | ML | Stats | 574.41 | 1999–2020 | 9,768 | 20 | 5 |
| Total | | | 6,879.16 | | 175,839 | 2,892 | 327 |

**New ML-specific refactorings & technical debt categories** We introduce 14 and 7 new ML-specific refactorings and technical debt categories, respectively.

**Recommendations, best practices, & anti-patterns** We propose preliminary recommendations, best practices, and anti-patterns for long-lasting ML system evolution from our statistical results, as well as an in-depth analysis.

Complete results of our study are available in our dataset [17].

## II. Methodology

We study common ML system refactorings using a (mostly) manual analysis. Refactorings unique to ML systems are extracted. From this study, we may find refactorings specific to ML systems that may assist both engineers and data scientists in effective evolution and management of ML technical debt.

### A. Subjects

Our study involves 26 open-source ML systems (tab. I), comprising ∼4.2 million lines of source code, 175,839 Git commits, and 183.76 years of combined project history, averaging 7.07 years per subject. They vary widely in their domain (column **dom**) and application (column **appl**), as well as size and popularity. All subjects have their sources publicly available on GitHub, exhibit non-trivial metrics, including stars, forks, and number of collaborators, and include a mix of ML libraries, frameworks, and applications. They have also been used in previous studies [18]–[28].

Subject criteria includes having at least one commit whose log message mentions "refactor," and at least a portion of the system must involve ML. We favored ML systems that were mostly written in Java, which—especially as a supporting language—is popular for large-scale ML [29]. However, although supporting subsystems were mainly written in Java, model code may be written in other languages, such as Python and C++. This was done to facilitate refactoring determination (statically-typed, single-parent inheritance) both manually and via the aid of assisting tools [30];[1] regardless of language, model and non-model code, alike, were manually examined.

To find changesets (patches) representing refactorings, we mined repositories for commit logs mentioning keywords; column **kws** of tab. I is the number of commits containing "refactor" in their log messages. While this may represent a proper subset of actual refactorings, this yielded 2,892 commits across 26 projects. We then randomly selected a subset of these commits to examine manually, as portrayed by column **exe**.

### B. Commit Mining

To discover commits with changesets that included refactorings, we searched the commit logs, which were extracted via `git log`. A single keyword "refactor" was queried via the regular expression `\b(?i)refactor`, which matches strings containing the word `refactor` in a case-insensitive (`(?i)`) manor. The `\b` at the beginning of the expression indicates a word boundary at the start of the term. This allows the expression to match the term "Refactoring" but not, for example, "`ArabicFeatureFactory`"—a class in `CoreNLP`.

### C. Refactoring Identification

Random matching commits were chosen for manual inspection to verify whether they contained one or more refactorings; automated tools were not used in this process. Two of the authors are software engineering and programming language professors with extensive expertise in software evolution, technical debt, and empirical software engineering. Another author is a data mining and machine learning professor with substantial proficiency in artificial intelligence and software engineering. Although the researchers did not converse during the initial identification and classification process to avoid bias, this mix of expertise is effective in studying software engineering tasks in machine learning systems. The researchers convened regularly during the study, as well as at the end for finalization, to solidify the results. Cohen's Kappa coefficients [31] for refactoring identification, classification, and ML-related code identification were 0.64, 0.41, and 0.83, respectively.[2] As the authors did not always have detailed knowledge of the particular systems, only changes where a refactoring was extremely likely were marked as such. The authors also used commit comments and referenced bug databases to ascertain whether a change was a refactoring, a common practice [32]–[34]. Type annotations—when available—were also helpful in assessing semantics-preservation—a key characteristic of refactorings.

Only master branches were used. Refactorings in all parts of the system were considered, as opposed to only modules responsible for ML. This is done because "only a small fraction of real-world ML systems is composed of ML code" [1].

### D. Refactoring Classification

Once refactorings were identified, to comprehend the kinds of refactorings performed in ML systems, the authors studied the code changes to determine the refactoring category, whether

[1] `RefactoringMiner` [30] was only used for classification (cf. §II-D).

[2] Moderate agreement is expected; the team has mixed ML/SE expertise.

the refactoring took place in ML subsystems (ML-related code), and the ML-specific technical debt category, if any, the refactoring addresses. The ML-specific technical debt category may coincide with one put forth by Sculley *et al.* [1], or it may be a new ML-specific technical debt category of our own devise. Islam *et al.* [10] also make ML-specific categorizations in their work. Categories were then formed into a hierarchy.

To assist in the classification, fortunately, many commits reference bug reports detailing the task-at-hand. This information proved highly valuable in understanding the refactorings, their motivations, and how they relate to the system. On several occasions, we also contacted developers for clarification.

Refactorings combat technical debt, and different refactorings can reduce different kinds of debt. Therefore, some categories may appear under different parent categories in the hierarchy. Also, some of the refactorings were more isolated, i.e., a single changeset consisted mainly of one type of refactoring. For such cases, we used a more specific (sub)category where possible. Conversely, changesets containing several intertwined, related refactorings were grouped into more general (parent) categories. For changesets that were difficult to generalize, we relied more heavily on commit log messages and issue tracker discussions. To aid the manual verification, `RefactoringMiner` [30]—a tool for refactoring detection in commit history—was occasionally used to help isolate larger commits by identifying fine-grained refactoring clusters.

We used terms like “cluster” and “train” in the commit log messages to help identify whether the changesets were related to ML. We also considered matrix operations to be ML-related. While such operations may be more general, since the subjects were ML systems, it is likely that they were being used for ML. Package names were also used to decipher whether code was related to ML, e.g., `elasticsearch` has a specific ML plug-in, which is directly reflected in the package name.

## III. Results

In this section, we mainly summarize the study results using data—noting trends, exceptions, and unexpected outcomes. §IV, on the other hand, consolidates and comments on the main findings and connects the different parts of the results. Related discussion in §IV is referenced where appropriate.

### A. Quantitative Analysis

From the 327 commits manually examined (column **exe**, tab. I), we found 285 true refactorings, depicted in column **cnt** of tab. II. Of these, 165 appeared in ML-related code (column **MLc**, tab. II). Finding these refactorings and understanding their relevance required a significant amount of manual labor that may not be feasible in more large-scale, automated studies.

False positives—commits whose logs contained the keyword but were not refactorings—amounted to 42 (12.84%). Reasons for false positives varied and included using the keyword in a different context (e.g., as a reminder, “[s]hould refactor the training code, though” [35]). Others include situations where developers liberally used the term “refactor,” i.e., they were actually adding or altering existing functionality [36]. There were also two (0.61%) cases where we were not able to determine whether the commit was a refactoring due to a lack of domain knowledge and extremely large commit sizes.

TABLE II: Discovered refactorings (nonhierarchical).

| group | category | abbr | cnt | MLc |
|---|---|---|---|---|
| Generic | Defer execution | DEF | 1 | 0 |
| | Make immutable | IMM | 1 | 0 |
| | Make more reusable | RUS | 1 | 1 |
| | Generalization | GEN | 2 | 1 |
| | Make more interoperable | INT | 2 | 2 |
| | Simplify regex | RGX | 2 | 0 |
| | Concurrency | CON | 4 | 2 |
| | Safety | SAF | 5 | 2 |
| | Dead code elimination | DED | 6 | 4 |
| | Make more extensible | EXT | 11 | 8 |
| | New language feature | LNG | 14 | 5 |
| | Test | TST | 15 | 4 |
| | Unknown | UKN | 15 | 10 |
| | Improve performance | PRF | 27 | 18 |
| | Duplicate code elimination | DUP | 33 | 24 |
| | Clean up | CLN | 48 | 26 |
| | Reorganization | ORG | 81 | 41 |
| Total | | | 268 | 148 |
| ML-specific<br>(new) | Make algorithms more visible | VIZ | 1 | 1 |
| | Make matrix variable names more verbose | VRB | 1 | 1 |
| | Monitor feature extraction progress | MON | 1 | 1 |
| | Push down hyperparameters | HYP | 1 | 1 |
| | Pull up policy | PLC | 1 | 1 |
| | Remove unnecessary matrix operation | RMA | 1 | 1 |
| | Replace flags with polymorphic classifier | CLS | 1 | 1 |
| | Replace flags with polymorphic feature extraction | FET | 1 | 1 |
| | Replace primitive array with matrix | AMT | 1 | 1 |
| | Replace with sparse matrix | SMT | 1 | 1 |
| | Replace primitives with rich prediction | PRD | 2 | 2 |
| | Replace rich model parameter with primitives | RMP | 2 | 2 |
| | Replace primitives with rich model parameter | PRM | 3 | 3 |
| Total | | | 17 | 17 |
| Grand Total | | | 285 | 165 |

*1) Refactoring Categories:* From the manual changes, we devise a set of common refactoring categories. Refactorings were then grouped into these categories as shown in fig. 1 and tab. II (column **abbr** is the refactoring’s abbreviation).[3]

Fig. 1 presents a hierarchical categorization—with varying levels of detail—of the 285 refactorings found in the ML system subjects. Refactorings are represented by their category name, followed by their refactoring counts. Categories without instances are considered *abstract*, i.e., they *only* group together other categories. Some refactoring categories are crosscutting, appearing under *multiple* categories. For this reason, tab. II portrays a nonhierarchical view of fig. 1, including a column for each refactoring category regardless of its parent.

Refactorings are separated into two top-level categories (column **group** of tab. II), namely, those specifically related to ML systems (ML-specific) and those tangentially related, i.e., those that apply to general systems (generic). Categories in the former division are novel; they were formulated as a result of this study and are a key contribution of this work.

*a) Generic Refactorings:* Generic refactorings are further categorized into those related to code reorganization (ORG; e.g., modularization), improving performance (PRF; multi-threading, variable extraction [5]), those made within test code or making code more amenable to testing (“Test;” TST), and migration

[3] All ML-specific refactorings were performed on ML-related code; as such, column **cnt** = column **MLc** for ML-specific refactorings in tab. II.

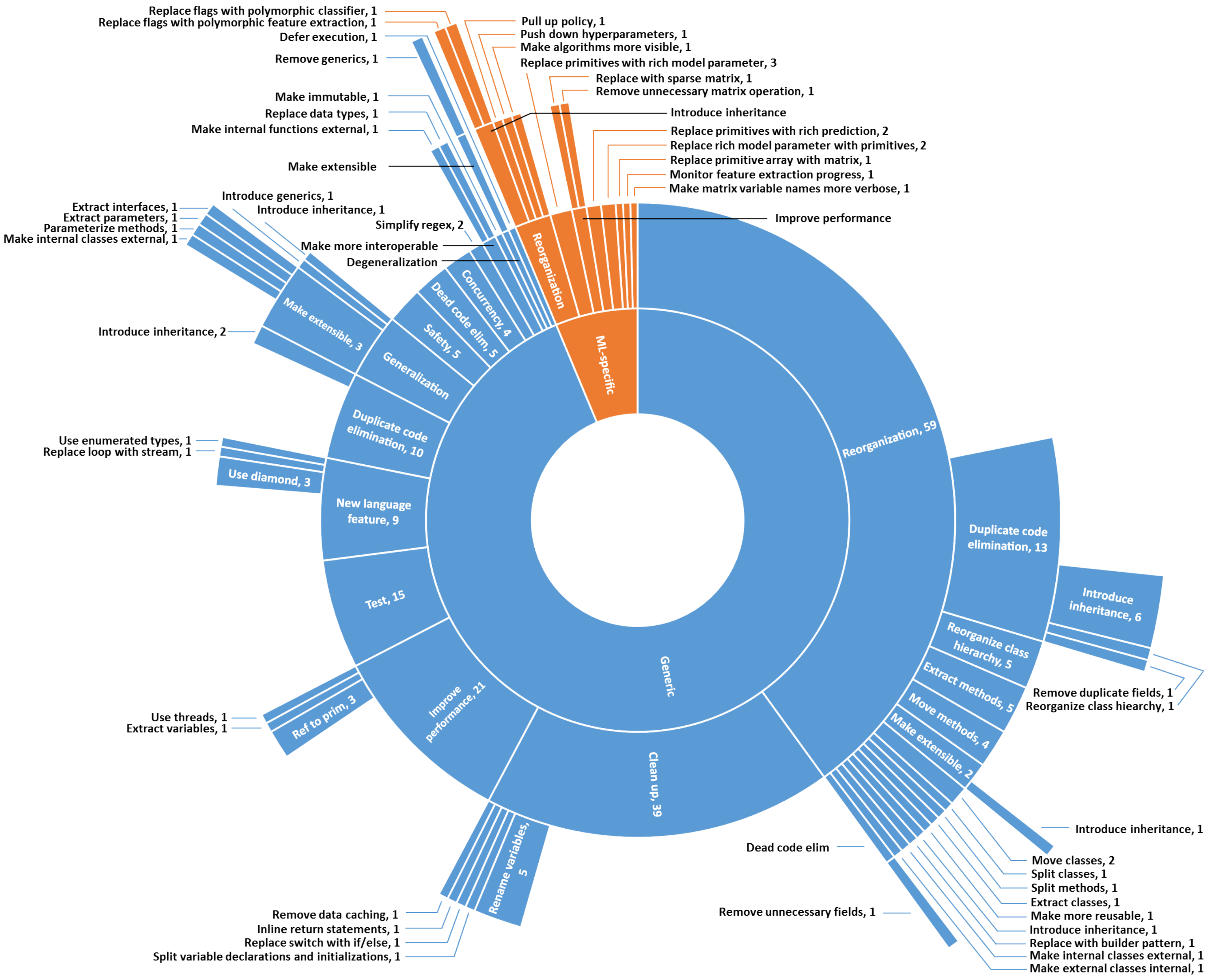

Fig. 1: Discovered refactorings (hierarchical).

to new language features (LNG; e.g., diamond syntax [37], multi-catch blocks, enumerated types [38], replacing loops with streams [39]). Others include duplicate code elimination (DUP; i.e., where redundant, possibly scattered code is centralized), making code more generally applicable (generalization; GEN), improving safety (SAF; e.g., allocating more memory for buffers holding tensors), eliminating dead code (DED), improving concurrency (CON; e.g., adding asynchrony [40]), regular expression simplification (RGX), making code more interoperable (INT; e.g., making private APIs public), code de-generalization (e.g., by removing generics), and deferring execution (DEF; e.g., making processing on-demand). "Clean up" (CLN) refactorings are general simplifications, e.g., removing unnecessary casts, while "unknown" (UKN; see tab. II) represents situations where the refactoring category was indeterminable without further domain knowledge or developer input. Only 5.26% of refactorings had unknown categories.

Generic reorganization (59) was the largest generic category; its largest subcategory was duplicate code elimination (13). Generic *reorganization* duplicate code elimination (13) differs from *generic* duplicate code elimination (10) as duplicate code elimination may or may not be part of a reorganization. For example, removing duplicate code by introducing inheritance or extracting methods can be considered a reorganization.

Duplicate code elimination was a major refactoring theme in ML system evolution, and we conjecture such systems are more prone to duplication due to slight variations in learning algorithms (see §IV-A). In general, categories crosscut, e.g., performance, because there are different ways to accomplish technical debt reduction, and there are different debt categories with the same "fix" (refactoring). Performance improvement refactorings, for instance, were both generic (PRF), e.g., converting reference types to primitives, and ML-specific, e.g., making matrices *sparse* (SMT), leading to finding 1 in fig. 2.

1) Performance improvement and reorganization (e.g., inheritance introduction) refactorings *crosscut* concerns, affecting *multiple* categories, both specifically and tangentially, associated with ML systems and were among the *most* frequent (37.89%).
2) Duplicate code elimination (11.58%) was a *major*, crosscutting ML system refactoring theme, combating debt in various ways.
3) We expected more dead code elimination [1]; however, though it crosscut, it was *not* usual (2.11%).
4) Making code more generalizable, reusable, and interoperable with libraries are essential SE tasks that were among the *least* performed refactorings (1.4%).
5) Inheritance introduction, appearing under six categories—the most of any other category—was a common and crosscutting way to eliminate duplication in ML systems and may be key in coping with subtle variations intrinsic to various ML algorithms.
6) Despite being the smallest subsystem [1], ML-related code was refactored the most (57.89%).
7) The majority of performance (66.67%), duplicate code elimination (72.73%), and extensibility (72.73%) refactorings were in ML-related code, while new language feature migration (35.71%) and test-related (26.67%) refactorings were among the least.
8) Although 66.67% of dead code elimination refactorings occurred in ML-related code, only one removed a dead experimental ML-related code path.
9) Configuration, duplicate model code, and plain-old-data type were the *most* tackled technical debt categories (36.84%, 18.95%, and 10.53%, respectively).
10) Configuration, duplicate model code, and plain-old-data type debts were mainly tackled by reorganization (42.86%), duplication elimination (94.44%), and replacing primitives with rich model parameters (30%).
11) Dead experimental code paths (1.05%), abstraction (2.11%), and boundary erosion (2.11%) were among the least addressed debts introduced by Sculley *et al.* [1]. Custom data types, duplicate feature extraction code, and model code reusability (3.16% combined) were among the least identified new categories.
12) Configuration debt (54.69%) was the most significant category from Sculley *et al.* [1], while duplicate model code was the most substantial of our newly introduced categories (58.06%).
13) Duplicate code elimination was a *major* refactoring (27.37%) in reducing ML-specific technical debt, overwhelming related to configuring and implementing different yet related ML algorithms (92.31%).
14) Inheritance and other reorganization refactorings were commonly (28.42%) used to reduce a variety of ML-specific debt, especially configuration (55.56%).

Fig. 2: Findings.

At 11.58%, duplicate code elimination was the largest category besides the umbrella-like categories of "clean up" and "reorganization" and crosscut several categories, meaning that it combated technical debt in several different ways. This leads to finding 2, fig. 2—further discussed in §IV-A.

Generic dead code elimination (DED), which may be accomplished via reorganization or deletion, was another refactoring that crosscut categories but was not prevalent (only 2.11%). However, we expected to see more of this category, as eliminating dead experimental code paths was a focal category of Sculley *et al.* [1]. ML systems typically use conditional branches for testing new experimental features and other ML algorithm improvements. Once branches are irrelevant, either because they were incorporated or deemed unnecessary, the corresponding code should be removed, leading to finding 3 in fig. 2. The relation of finding 3 to dead experimental code paths is discussed below with finding 8.

Generic generalization (GEN) refactorings introduced inheritance [41, Ch. 12] and generics [42] and made code more extensible (EXT), e.g., via extracting parameters [43, Ch. 11] and interfaces [44], [45]. Such EXT refactorings (3.86%) were also crosscutting—under generalization, de-generalization, and reorganization. Code was also made more interoperable (INT) by externally exposing internal C functions (**`extern`**) [46] and replacing custom data types with standard ones, e.g., to interface with `TensorFlow` [47], leading to finding 4, fig. 2.

*b) ML-specific Refactorings:* ML-specific refactorings are further divided into several categories corresponding to whether they involved reorganization (ORG), improving performance (PRF), e.g., removing unnecessary matrix operations (RMA, 1), and many new refactorings that we categorized as specifically applicable to ML-related code. These include replacing primitive types representing learning model parameters with objects (PRM, 3) and the opposite (RMP, 2), replacing primitive types representing model outcomes (predictions) with objects (PRD, 2), replacing primitive type arrays with matrix objects (AMT, 1), monitoring the progress of possibly lengthy feature extraction (MON, 1), and improving program comprehension by making the names of variables related to matrix calculations more verbose (VRB, 1). This last category emerged as we noticed many matrix calculations—a data structure highly used in ML—had numerous temporary variables. Improving these variable names can potentially facilitate matrix calculation evolution.

ML-specific reorganization again involved inheritance introduction. In fact, the "introduce inheritance" category appears six times in fig. 1, the *most* of any other category and is mostly used for duplicate code elimination through reorganization. This leads to finding 5 of fig. 2—discussed in §IV-B.

Two refactoring categories, both involving the conversion of "flag," i.e., intermediate boolean values [48, Ch. 17.2], checking to polymorphism, further divided ML-specific inheritance introduction. Specifically, the categories involve replacing many flags with polymorphic classifier (CLS, 1) and feature extraction (FET, 1) objects, respectively. These refactorings simplify the future addition and usage of new classifiers and features.

ML-specific reorganization also included two (new) ML-specific refactorings related to class hierarchy organization, namely, "pulling up" (clustering) policies (PLC, 1) and "pushing down" hyperparameters (HYP, 1). Learning algorithm variants may have similarities in their implementation. As such, PLC refactorings—similar to PULL UP MEMBERS [41, Ch. 12]—centralize otherwise scattered and duplicated code among classes representing the different policies (cf. § III-B1a). Hyperparameters, on the other hand, are used to configure ML algorithms, and HYP—similar to PUSH DOWN MEMBERS—is an ML-specific refactoring where hyperparameters are separated into individual algorithms. While this adds some duplication, it may improve cohesion and allow the hyperparameters to be used in ways more akin to the algorithms they configure (cf. § III-B2a). Both of these refactorings

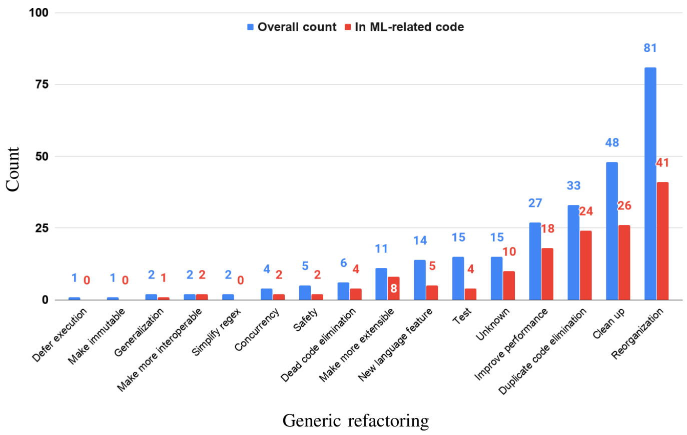

Fig. 3: Discovered generic refactorings (nonhierarchical).

operated on code that previously used inheritance, which is why there were not categorized under "inheritance introduction."

*2) Generic Refactorings Performed on ML-related Code:* As seen in tab. II, all ML-specific refactorings were made to ML-related code, i.e., the code directly involved with learning processes. As there were a significant number of generic refactorings made to the ML systems, we were also interested in understanding the kinds of generic refactorings that were being performed to model code in these systems. While the ML-specific refactoring categorization aims to unveil *new* refactorings specific to ML systems, this section sets forth to understand which *existing* refactorings are made to this code. Such information may provide insight into the struggles that developers have in maintaining and evolving ML systems and the refactorings that can help. For comparison purposes, fig. 3 diagrammatically portrays only the generic refactorings, including their **overall counts** (left/blue bars) and counts of the refactorings appearing **in ML-related code** (right/red bars).

Larger, more definitive categories with the most ML-related code changes were performance improvements (66.67%), duplicate code elimination (72.73%), and extensibility improvements (72.73%). Most of these took place in ML-related code. In some respects, it surprising that the *majority* (57.89%) of *all* refactorings were performed in ML-related code as ML subsystems typically the *smallest* subsystem of ML systems [1], leading to finding 6, fig. 2. Finding 6 coincides with that of Dilhara *et al.* [49]—that developers update ML libraries more often than traditional libraries. New language feature migration (LNG) and test-related (TST) refactorings were some of the least performed on ML code, leading to finding 7 in fig. 2. Finding 7—and its relation to finding 2—is discussed in §IV-A.

Per finding 3, dead code elimination (2.11%) was minimal; nevertheless, most such refactorings occurred within ML-related code (66.67%). One might expect these are the ML-related code path eliminations discussed by Sculley *et al.* [1]; however, only one of the four refactorings did indeed remove an experimental code path (cf. tab. III), leading to finding 8, fig. 2.

*3) ML-specific Technical Debt vs. Refactorings:* Recall that refactorings were classified on three fronts, i.e., their categories (fig. 1), whether they took place in ML subsystems (tab. II), and the ML-specific technical debt category, if any, the refactoring addresses. Tab. III presents the identified ML-specific debt categorization (rows) and juxtaposes them with their corresponding refactoring categories (columns; abbreviations from tab. II). Debt categories are grouped by existing [1] and new categories that have been formulated as a result of our study.

*a) Technical Debt:* Finding 9, fig. 2 summarizes the most tackled debt categories. Configuration and plain-old-data type ML-specific debt categories are classical [1], while duplicate model code is new. Configuration debt deals with configurable ML system options, including the features and data utilized, algorithm-specific learning settings, pre- and post-processing, and evaluation methods employed [1]. Duplicate model code occurs when code duplication exists in core learning code, e.g., classification, prediction, and makes adding new and changing existing learning algorithms error-prone. It is especially prone to situations where many learning algorithm variants are utilized. Plain-old-data type debt occurs when rich information used and produced by ML systems is encoded using primitives, making, for example, the purpose of hyperparameter indecipherable and predictions less explainable [1], [50].

Configuration debt was addressed by several refactorings, e.g., duplication elimination (DUP; 20%), extensibility (EXT; 8.57%). Plain-old-data types was more even and widespread, spanning six refactorings, including replacing (i) primitives with rich predictions (PRD; 20%), and (ii) primitive arrays with matrix objects (AMT; 10%), leading to finding 10.

Dead experimental code paths [1] are those used to prototype new learning algorithm variants. If successful, they are eventually incorporated into the mainline logic, making the experimental paths irrelevant (and disabled). Leaving such paths in code hinders developers' ability to later add new and modify existing algorithms. Abstraction debt arises from a lack of standard interfaces and constructs [9] (e.g., those in relational databases [51]) that may be subtly corrupted or invalidated by the fact that data influences ML system behavior, while boundary erosion amounts from a lack of modular boundaries between ML subsystems [1]. Finding 11—discussed in §IV-C—summarizes the least tackled debt.

Overall, only three of the categories established by Sculley *et al.* [1] were prevalent, i.e., configuration (36.84%), plain-old-data type (10.53%), and multiple language debt (7.37%). We also found that duplicate model code (18.95%), model code comprehension (5.26%), and model code modifiability (4.21%) were the only prevalent new categories, leading to finding 12. Model code modifiability is specific to ML algorithm encapsulation as opposed to traditional data encapsulation. ML developers must incorporate a variety of learning algorithms that are subsequently evaluated and compared. The inability to abstract learning algorithm variations and make learning components extensible may be detrimental to ML systems.

*b) Refactorings:* Duplicate code elimination (DUP; 27.37%) was among the refactorings that tackled the *most* technical debt, spanning such categories as duplicate model code (65.38% of DUP refactorings), configuration (26.92%),

TABLE III: Discovered ML-specific technical debt vs. refactoring categories.

| group | technical debt \ refactoring | AMT | CLS | FET | GEN | HYP | LNG | MON | PLC | RMA | RUS | SMT | VIZ | VRB | PRD | DED | INT | PRF | SAF | CLN | PRM | EXT | DUP | ORG | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Existing | Dead experimental code paths | | | | | | | | | | | | | | | 1 | | | | | | | | | 1 |
| | Abstraction | | | | | | | | 1 | | | | | | | | | 1 | | | | | | | 2 |
| | Boundary erosion | | | | | | | | | | | | | | | | | | | | | | | 2 | 2 |
| | Glue code | | | | | | | | | | | | | | | | 1 | | | | | | | 1 | 2 |
| | Prototype | | | | | | | | | | | | | | | | | | | | | | | 2 | 2 |
| | Monitoring and testing | | | | | | | 1 | | | | | | | | | | | | | | | 1 | 1 | 3 |
| | Multiple languages | | | | | | 1 | | | | | | | | | 1 | 1 | | 2 | | | | | 2 | 7 |
| | Plain-old-data type | 1 | | | | | | | | | | | | | 2 | | | | 1 | 1 | 3 | | | 2 | 10 |
| | Configuration | | 1 | 1 | 1 | 1 | | | | | | 1 | | | | 1 | | 2 | | 2 | 2 | 3 | 7 | 15 | 37 |
| Total | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | 1 | | | 2 | 3 | 2 | 3 | 3 | 3 | 5 | 3 | 8 | 25 | 66 |
| New | Custom data types | | | | | | | | | | | | | | | | 1 | | | | | | | | 1 |
| | Duplicate feature extraction code | | | | | | | | | | | | | | | | | | | | | | 1 | | 1 |
| | Model code reusability | | | | | | | | | | 1 | | | | | | | | | | | | | | 1 |
| | Unnecessary model code | | | | | | | | | 1 | | | | | | | | | | | | | | | 1 |
| | Model code comprehension | | | | | | | | | | | | 1 | 1 | | | | | | 1 | | | | 1 | 4 |
| | Model code modifiability | | | | | | | | | | | | | | | | | | | | | 5 | | | 5 |
| | Duplicate model code | | | | | | | | | | | | | | | | | | | | | | 17 | 1 | 18 |
| Total | | | | | | | | | | 1 | 1 | | 1 | 1 | | | 1 | | | 1 | | 5 | 18 | 2 | 31 |
| Grand Total | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 3 | 3 | 4 | 5 | 8 | 26 | 27 | 97 |

**Listing 1** Commit `3eba6f26` in `Mahout`: Refactored ClusteringPolicies into hierarchy under new AbstractClusteringPolicy ...

```
1 +public abstract class AbstractClusteringPolicy
2 +  implements ClusteringPolicy {
3 +  public Vector classify(Vector d, ClusterClassifier p){
4 +  List<Cluster> models = p.getModels(); /*..*/ }}
5  public class CanopyClusteringPolicy
6 -  implements ClusteringPolicy {
7 +  extends AbstractClusteringPolicy {
8 -  public Vector classify(Vector d, List<Cluster> models){
9 -    Vector pdfs = new DenseVector(models.size());/*..*/}}
10 public class DirichletClusteringPolicy
11 -  implements ClusteringPolicy {
12 +  extends AbstractClusteringPolicy {
13 -  public Vector classify(Vector d, List<Cluster> models){
14 -    Vector pdfs = new DenseVector(models.size());/*..*/}}
```

duplicate feature extraction code (3.85%), and monitoring and testing (3.85%), which deals with ML evaluation. From these results, a central theme emerges; code duplication is extensive in ML systems and presents itself mainly on two fronts—in configuration and in model code. In other words, code duplication infects configuring learning algorithms and in the implementation of the learning algorithms themselves, leading to finding 13 of fig. 2. Finding 13—in the context of findings 2 and 7—is discussed in §IV-A.

Reorganization—including inheritance introduction—was also a common way to reduce technical debt in ML systems, accounting for 28.42% of refactorings combating ML-specific technical debt. Reorganization also spanned 9/16 technical debt categories with a major focus on configuration debt (55.56%), leading to finding 14. Finding 14—along with its relation to finding 5—is discussed in §IV-B.

1) Favor inheritance to abstract learning algorithm variations, thereby reducing redundant model code.
2) Adding some duplicate code via class hierarchy reorganization may help focus ML algorithm configuration, especially when using dependency injection.
3) Favor polymorphism over flags when many ML algorithm variants exist to reduce to configuration debt.
4) To facilitate ML system evolution, use descriptive (temporary) variable names, especially for matrices.
5) Since ML has many algorithms for similar tasks [52], restructure code (e.g., method extraction) for greater reusability among learning algorithm variants.
6) ML libraries imposing custom numeric data types should include conversion code to built-in types.

Fig. 4: Best practices.

**Listing 2** Commit `59f39c7b` in `DataCleaner`: Refactored components to have a "Training analyzer" per algorithm.

```
-public class MLTrainingAnalyzer /*...*/ {
+public abstract class MLTrainingAnalyzer /*...*/ {
-  @Configured @NumberProperty(negative=false,zero=false)
-  int epochs = 10; /*...*/ }
+public class RandomForestTrainingAnalyzer extends
+  MLTrainingAnalyzer {
+  @Configured @NumberProperty(negative=false,zero=false)
+  int epochs = 10; /*...*/ }
+public class SvmTrainingAnalyzer extends
+  MLTrainingAnalyzer {
+  @Configured @NumberProperty(negative=false,zero=false)
+  int epochs = 10; /*...*/ }
```

### B. Qualitative Analysis

We highlight refactorings and ML-specific technical debt with examples, summarize causes, symptoms, and fixes in tab. IV, and propose preliminary best practices and anti-patterns. Rows in tab. IV correspond to debt categories discussed below.

#### 1) Duplicate Model Code Debt:

*a) ML→ORG→PLC:* Duplicate code elimination dominated the refactorings in ML-related code and crosscut multiple categories. Consider a Pull Up Policy (PLC) refactoring in lst. 1. There are multiple classes, e.g., lines 5 and 10, representing different clustering algorithm policies. Each class previously implemented a common interface; however, as interfaces do not contain functionality, an abstract class is introduced on line 1 that encapsulates the common policy functionality. As a result, the duplicated model code on lines 8–9 and 13–14 are replaced with polymorphic calls to `classify()` on line 3, leading to best practice 1, fig. 4.

#### 2) Configuration Debt:

*a) ML→ORG→HYP:* While PLC refactorings centralize ML-related code, others do the opposite. Consider the Push

TABLE IV: Common attributes of ML-specific technical debt categories discussed in §III-B.

| debt | situation | cause | symptoms | fixes |
|---|---|---|---|---|
| Duplicate model code | Code duplication in learning code, e.g., classification, prediction. | Learning algorithms have many variants with subtle differences. | Adding new/changing existing model code is error-prone. | Inheritance introduction, class hierarchy reorganization. |
| Configuration | Learning algorithms have many configurable options. | Configuration is treated as an afterthought [1]. | Each configuration line has a potential for errors. | Class hierarchy reorganization, duplicate code elimination, etc. |
| Model code comprehension | Many temporary matrix variables, perf vs. comprehension trade-offs. | Variables poorly named, unnecessarily sacrificing comprehension. | Reasoning about and evolving model code is made difficult. | More verbose matrix variable names, inheritance introduction. |
| Model code reusability | Adding new models requires duplicating existing code. | Model code is insufficiently modularized. | Reusing existing model code is made difficult and error-prone. | Reorganization, method extraction. |
| Unnecessary model code | Matrix calculations may have performance bottlenecks. | Unnecessary matrix APIs. | Poor performance. | Replace expensive APIs with calculations in existing traversals. |
| Custom data types | Project-specific data types used instead of built-in types in ML. | Library dependencies may impose custom data types. | Interoperating with other libraries can be difficult. | Widespread modifications involving type replacement or conversion. |

**Listing 3** Commit `32546890` in `CoreNLP`: merged remote branch crf_stochastic_fix, refactored CRFClassifier.

```
1  + CRFClassifier<CoreLabel> chooseCRFClassifier(
2  +     SeqClassifierFlags flags) {
3  +   CRFClassifier<CoreLabel> crf = null;
4  +   if (flags.useFloat)
5  +     crf = new CRFClassifierFloat<CoreLabel>(flags);
6  +   else if (flags.nonLinearCRF)
7  +     crf = new CRFClassifierNonlinear<CoreLabel>(flags);
8  +   else if (flags.numLopExpert > 1)
9  +     crf = new CRFClassifierWithLOP<CoreLabel>(flags);
10 +   // ...
11 +   return crf;
12 + }
13   // ...
14   Properties props=StringUtils.argsToProperties(args);
15 - CRFClassifier<CoreLabel> crf=new CRFClassifier<>(props);
16 + SeqClassifierFlags flags=new SeqClassifierFlags(props);
17 + CRFClassifier<CoreLabel> crf=chooseCRFClassifier(flags);
```

**Listing 4** Commit `5c3dcd35` in `ELKI`: refactoring feature extraction for images.

```
1 -for (int k = 0; k < DISTS.length; k++) {
2 +for (int k = 0; k < DISTANCES.length; k++) {
3 -  int d = DISTS[k];
4 +  int d = DISTANCES[k];
5 -  // horizontal
6 +  // horizontal neighbor
7 +  // TODO Pete: What is sum?
8    sum[k] += 2;
```

**Listing 5** Commit `6dd54317` in `ELKI`: Huge Pair refactoring.

```
1  List<Integer> currentCluster = new ArrayList<>();
2 - for (ComparablePair<D, Integer> seed : seeds) {
3 + for (DistanceResultPair<D> seed : seeds) {
4 -   Integer nextID = seed.getSecond();
5 +   Integer nextID = seed.getID();
```

Down Hyperparameters (HYP) refactoring snippet in lst. 2. Several hyperparameters (e.g., line 4) were de-centralized from the parent and copied into subclasses of different learning algorithms (lines 8 and 12). While adding some duplication, it "allows us to have much more specific hyperparameters [that] apply to the particular algorithm instead of trying to make a one-size-fits-all parameter selection" [53]. Though the field declarations above are identical, note the annotations on lines 7 and 11. In this case, inheritance may make it more difficult to configure hyperparameters when, e.g., using dependency injection and different hyperparameters require varying values, leading to best practice 2 in fig. 4.

*b) ML→ORG→INH→CLS:* Configuration debt was the largest discovered technical debt category. Especially interesting was the management of flags corresponding to ML configuration parameters, as ML system configuration is increasingly unwieldy [1], giving way to a configuration "parameter server" design pattern [54]. Consider the Replace Flags with Polymorphic Classifier (CLS) refactoring in lst. 3, where large portions of parameter flag code in `CRFClassifier` were replaced with polymorphic objects. The factory method `chooseCRFClassifier()` accepts `flags` and returns a subclass instance. Instead of passing flags directly to the constructor (line 15), a separate `SeqClassifierFlags` parameter object is passed to `chooseCRFClassifier()` (line 17). Algorithmic flag checking is then replaced with polymorphism, leading to best practice 3 of fig. 4.

### *3) Model Code Comprehension Debt:*

*a) ML→VRB:* Matrix algebra is central to ML, and matrix calculations often include the use of many temporary variables. Consider the Make Matrix Variable Names More Verbose (VRB) refactoring snippet in lst. 4 that is performed on feature extraction code for image classification. On lines 2 and 4, `DISTS` is renamed to `DISTANCES`. While this is a minor refactoring, `DISTS` may also have referred to "distributions" in such analytical-based software. Although poor variable name quality can cause confusion and inhibit effective software evolution in general, it is especially problematic in ML systems due to the high reliance on matrix calculations that may involve many temporary variables thus compounding the issue. Further refactoring motivation is on lines 6 and 7, where a comment is diluted and a variable clarification is requested, leading to best practice 4, fig. 4.

*b) GEN→ORG→INH:* Model code is particularly performance-sensitive due to the number of iterations ML systems typically perform on (large) datasets. In such cases, there may be trade-offs between performance and comprehension; however, they can be misguided, sacrificing readability unnecessarily. Consider the generic (GEN) refactoring snippet in lst. 5 performed on ML-related code. On line 3, `ComparablePair`, which was previously deemed to be more performant, is replaced with the more specific `DistanceResultPair` type—allowing for the more readable `getID()` accessor on line 5 instead of the more ambiguous `getSecond()`. The author

1) Unnecessarily sacrificing ML model code clarity for performance gains.
2) Expensive multidimensional matrix APIs are used for single-dimensional vectors.
3) Project-specific numeric data types are used in model code, decreasing interoperability with ML libraries.

Fig. 5: Anti-patterns.

**Listing 6** Commit `4432e319` in `Mahout`: MAHOUT-846: Minor refactoring to eliminate unnecessary `vector.times(SQRT2PI)`.

```
1   public double pdf(VectorWritable vw) { // ...
2     Vector s = getRadius().plus(0.0000001);
3     return Math.exp(-(divSquareAndSum(x.minus(m),s)/2))
4  -     / zProd(s.times(UncommonDistribs.SQRT2PI));
5  +     / zProdSqt2Pi(s);
6   }
7  -private double zProd(Vector s) {
8  +private double zProdSqt2Pi(Vector s) {
9     double prod = 1;
10    for (int i = 0; i < s.size(); i++)
11 -     prod *= s.getQuick(i);
12 +     prod *= s.getQuick(i) * UncommonDistribs.SQRT2PI;
```

proclaims the following, leading to anti-pattern 1, fig. 5:

> Java performance studies have shown no cost in making `Pair` non-final; hotspot-VMs will optimize that very well. Since we can get `getDistance()` and `getID()` for free, we [a]re go[ing to] use them to increase readability of the code [55].

For anti-pattern 1, it is understandable that developers strive for peak runtime performance in model code; it is beneficial for large datasets to be efficiently processed. However, performance improvements that degrade code clarify, particularly in (complex) model code, should be carefully scrutinized, e.g., via performance testing, for whether they are, in fact, notably enhancing performance. In the above example, `ComparablePair` is a general type containing general methods to retrieve consistent components (`getFirst()` and `getSecond()`). These components can represent *any* entity in an ML algorithm, but this type was previously deemed more performant than using a more specific type; `ComparablePair` is **`final`**, thus disallowing any subtypes and forgoing multiple dispatch. Performance testing of multiple alternative constructs, especially in model code, may reveal particular VMs optimizations, e.g., those that allow for non-**`final`** types that improve readability, such as those with the more specific method `getDistance()` that can be used instead of `getSecond()`. A consequence of anti-pattern 1 is that (already) complex model code is made more difficult to comprehend and consequently difficult to evolve.

*4) Model Code Reusability Debt:*

*a) GEN→ORG→RUS:* Essential to ML system evolution is the addition of new models—ideally—via code reuse. One generic MAKE MORE REUSABLE (RUS) refactoring in ML-related code uses method extraction "for code reuse in other SNN functions" [56], leading to best practice 5, fig. 4.

*5) Unnecessary Model Code Debt:*

*a) ML→PRF→RMA:* As model code is performance-sensitive; seeming innocuous refactorings in such regions can impact performance [5]. Consider the REMOVE UNNECESSARY MATRIX OPERATION (RMA) refactoring snippet in lst. 6. This refactoring helped solve a performance issue [57] related to Gaussian clustering scalability by "eliminat[ing an] unnecessary call to `vector.times(SQRT2PI)`" [58] on line 4. This matrix API implementation includes several layers of method calls dealing with multiple dimensions. Since `s` is a single dimensional vector, the calculation can instead be inlined into the existing traversal (line 12), resulting in code that "is significantly faster with no new [v]ectors created." [57]. This leads to anti-pattern 2 of fig. 5.

**Listing 7** Commit `19057519` in `Deeplearning4j`: Refactored pad and mirror_pad ops to conform with TF. (#100)

```
1 -auto paddings = NDArrayFactory::create<Nd4jLong>({1,0});
2 +auto paddings = NDArrayFactory::create<int>({1LL,0LL});
```

The problem w.r.t. anti-pattern 2 is an impedance mismatch between the *expected* API argument's complexity and the *actual* argument. Specifically, APIs processing multidimensional matrices may involve multiple layers of method calls, which is unnecessary when the actual argument is single-dimensional. A consequence of anti-pattern 2 is that the added method call layers can degrade performance, especially in model code, which is hypersensitive to performance impact as many iterations of (large) datasets occur in such (critical) areas. Common contexts of anti-pattern 2 involve model code dealing with single-dimensional vectors. A common fix for anti-pattern 2 is to replace the expensive API calls with calculations using operators (e.g., `*` for multiplication), inlining those calculations into existing loops.

*6) Custom Data Types Debt:*

*a) GEN→INT:* ML systems may depended on learning libraries for which they must interoperate. Using project-specific ("wrapped") data types, however, can impede interoperability, leading to anti-pattern 3, fig. 5. Consider the generic MAKE MORE INTEROPERABLE (INT) refactoring snippet in lst. 7 performed on ML-related C++ code, where line 2 replaces a custom data type (`Nd4JLong`) with a built-in primitive (**`int`**). Although specifying array literals, in this case, may be more cumbersome (`1` vs. `1LL`), the code can now freely interoperate with `TensorFlow`. Dependencies themselves may impose custom data types—`Nd4jLong` is from the highly-related yet external `ND4J` scientific computing library—in which case, conversion code may be necessary.

The issue w.r.t. anti-pattern 3 is that "wrapper" types like `Nd4JLong` from scientific computing or native (GPU-oriented) libraries are used in performance-sensitive model code contexts to enhance performance in these critical areas. However, there is a trade-off; such types may not interoperate with other (ML) libraries and frameworks (e.g., `TensorFlow`). Moreover, the custom types may stem from other library dependencies, possibly making their use necessary. The consequences are that developers may need to (i) make a difficult choice to forgo depending on particular libraries, (ii) make widespread modifications to replace or modify the type, and (iii) write

1) Automated refactorings especially designed for migrating "linear" ML algorithm and configuration code to use inheritance constructs may be advantageous in avoiding code duplication.
2) More ML-specific refactoring tool-support may encourage more refactoring of model and configuration code, potentially reducing technical debt.
3) More *automated* client-side matrix calculation refactorings may replace *manual* model code performance enhancements.

Fig. 6: Recommendations.

their own conversation code. To alleviate such consequences, we suggest best practice 6 of fig. 4.

## IV. Discussion

### A. Code Duplication in Configuration & Model Code

With duplicate code elimination being one of the top overall and crosscutting refactoring categories (finding 2), as well as the top refactoring performed on ML-related code (finding 7), ML systems seem to exhibit a significant amount of code duplication, particularly in *configuration* and *model code* regions (finding 13). Feasible explanations include (i) data scientists—potentially untrained as software engineers and thus not fully aware of advanced modularization techniques—may be responsible for model code, (ii) model code is highly-configurable—containing a substantial number of different yet related hyperparameters—which are configured in similar ways, and (iii) many different ML algorithms share a significant amount of commonality, giving way to code duplication.

Further research is needed to uncover different developer roles in ML systems to fully understand the phenomenon underpinning Item (i). As configuration debt was the largest technical debt category, for Item (ii), it is apparent that ML code involves many flags, and developers are finding ways to deal with them so that both comprehension and extensibility are improved. Configuration debt was a major theme of Sculley *et al.* [1]; thus, it was not surprising that the majority of refactorings aim at reducing it. Although parameter servers [54] help, without language-level modularization techniques, they are simply *moving* the problem. As for Item (iii), as demonstrated in §III-B, language-level modularization techniques can help reduce some of the redundancy resulting from variant learning algorithm implementations. While our findings coincide with those of Lopes *et al.* [59], i.e., there is a non-trivial amount of code duplication, their findings are inter-project focused, whereas ours are *intra*-project. Also, per finding 7, we find that most duplication is addressed *within* ML-code in ML systems, a comparison between components.

### B. Combating Code Duplication Debt in ML Systems

We identified the two ML system areas that exhibit the most duplication, i.e., configuration and model code. Amershi *et al.* [8] also note issues with model code reuse. Fortunately, per finding 5, inheritance was a centrally used technique in eliminating code duplication, particularly with algorithm variations, and finding 14 shows that it was especially useful to reduce duplicate configuration code. As such, inheritance may be a key in reducing duplicate code in ML systems. A problem, however, is that model code is not always written in an Object-Oriented (OO) style, as scripting languages are popular. Or, if such code *is* written in OOP, developers may either not be aware (i) of inheritance techniques, (ii) that inheritance can help avoid duplicate code, or (iii) of (or cannot use) tool-supported refactorings that can help with inheritance introduction. As such, for ML code currently taking advantage of OO and especially those either implementing ML algorithm variations or configuring hyperparameters, more awareness and specialized tool-support may be necessary, leading to recommendation 1, fig. 6. More tool-support is also advocated by Arpteg *et al.* [9], while Bavota *et al.* [60] warn against hierarchy refactorings.

Conversely, for situations where code is not written in an OO style but OO is available, recommendation 1 may help promote inheritance usage. For example, although dynamic languages, e.g., Python, are popular for writing model code [61], such languages may have inheritance mechanisms available, e.g., `abc` [62]. Automated refactorings that are custom-tailored to ML development may promote more usage of such packages. Unfortunately, due to the static analysis typically required in such refactorings, adapting existing automated refactoring algorithms to dynamic settings is non-trivial. A possible solution is to leverage a tractable, speculative analysis that is customized to ML contexts [5] to provide accurate and useful dynamic resolution.

### C. Generic vs. ML-specific Refactorings

Generic refactorings (94.03%) vastly outnumbered those of our new ML-specific refactorings (5.97%). A feasible explanation is (i) model code is among the smallest ML subsystems [1]; thus, we would expect less ML-specific refactorings, (ii) data scientists—potentially not versed in refactoring—may be responsible for ML-related code maintenance and evolution, and (iii) a lack of ML-specific *automated* refactorings may deter developers as they must refactor *manually*, leading to recommendation 2, fig. 6. The lack of ML-specific refactoring occurrences—along with finding 11—does not necessarily indicate that technical debt is not present; it may be that it is simply not being addressed. Also, good solutions for these problems may not yet exist [1]. Nevertheless, future work consists of extracting generalizable refactoring exemplars from the refactorings presented earlier that can serve as a basis for refactoring preconditions in varying contexts.

### D. ML-related Code Performance

Model code needs to be fast; thus, it is not surprising that 66.67% of performance enhancements occurred in ML-related code (finding 7). We came across several refactorings that converted reference types to primitives for performance reasons. Our findings coincide with that of Kim [11] and Zhang *et al.* [18], i.e., performance is essential yet challenging in ML systems. Additional client-side tool-support focused on improving matrix calculations (e.g., [5]) may alleviate developers from making manual performance enhancements, leading to recommendation 3, fig. 6. Future work also consists of automating the RMP refactoring (tab. II) by reversing the approach taken by Khatchadourian [38], which converts primitives to reference types.

## V. Threats to Validity

Subjects may not be representative of ML systems. To mitigate this, subjects encompass diverse domains and sizes and have been used in previous studies. Various GitHub metrics and ML-related keywords/tags were used to assess popularity and in choosing subjects, respectively. Although Java was favored (cf. §II-A), many subjects were written in *multiple* languages, particularly for model code, which was also analyzed. For example, ∼30% of `Deeplearning4j` is written in C++.

Our study involved many hours of manual validation to understand and categorize the refactorings. To mitigate bias, we investigated referenced bug reports and other comments from developers to help us understand changes more fully.

Larger refactorings may be non-atomic, spanning multiple commits [63]–[65]. In such cases, it may be difficult to assess the task-at-hand for accurate categorization. To mitigate this, we examined referenced bug tracker reports, which often mentioned multiple commits, allowing us to understand the overall goals. It is also possible that developers performed refactorings but did not mention so in commit log messages, potentially causing us to miss refactorings. Nevertheless, our study still involved manually examining 327 commits.

The heuristics applied in determining whether refactorings were related to ML-code may not be accurate; however, the researchers thoroughly examined each changeset and conversed regularly. `RefactoringMiner` [30], which aided some manual classification—particularly with larger commits—may not be accurate. However, *all* commits were still *manually* analyzed, and this tool has been used extensively [66]–[68].

## VI. Related Work

Sculley *et al.* [1] identify common SE issues surrounding ML systems based on their experiences at Google. Arpteg *et al.* [9] also detail several ML-specific technical debt categories. Our work—in part—can be seen as an open-source data-driven complement to theirs. In addition to technical debt, we also explore ML system *refactorings*, correlate them to ML-specific technical debt, and introduce 14 and 7 new ML-specific refactorings and technical debt categories, respectively.

Several studies involve ML and DL systems. Amershi *et al.* [8] conduct a study at Microsoft, observing software teams as they developed AI applications. They also put forth best practices to address challenges specific to engineering ML systems; albeit, many are organizational or process-based. Lwakatare *et al.* [12] also classify SE challenges for ML systems at six different companies, focusing mainly on deployment issues. Zhang *et al.* [18] present a large-scale empirical study of DL questions on Stack Overflow. Zhang *et al.* [7] and Islam *et al.* [10] study DL bug characteristics and present anti-patterns but to avoid bugs. Dilhara *et al.* [49] study ML library usage and evolution. In contrast, our focus is on the *non-functional* qualities of ML systems, the technical debt they cause, and the refactorings that address them.

Other work studies and categorizes refactorings. Tsantalis *et al.* [30] automatically detect refactorings in commit history; however, their approach is currently limited to fine-grained analysis of classical refactorings, supports only Java, which is problematic for multilanguage ML systems, and does not correlate technical debt. Kim *et al.* [36] study refactoring challenges and benefits at Microsoft, while Vassallo *et al.* [69] perform a large-scale refactoring study on open-source software, and Murphy-Hill *et al.* [70] study general refactoring at the IDE level. Sousa *et al.* [71] characterize composite refactorings, Hora and Robbes [72] explore the characteristics of method extraction refactorings, Peruma *et al.* [73] investigate refactorings of unit tests in Android, and Bavota *et al.* [60] and Ferreira *et al.* [74] study fault inducing refactoring activities.

Technical debt has also been studied. Tom *et al.* [2] propose the concept for general systems. Potdar and Shihab [75] explore self-admitted, e.g., via code comments, technical debt (SATD), while Bavota and Russo [76] investigate the diffusion and evolution of SATD and its relationship with software quality. Huang *et al.* [77] and Rantala *et al.* [78] identify SATD using advanced techniques, and Christians [79] examines the relation between SATD and refactoring in general systems. The refactorings we have identified that correlate to debt categories may be considered a form of (ML-specific) SATD. Code smells can also indicate technical debt, and Aversano *et al.* [68] study the evolution of smells and their tendencies to be refactored.

There are many empirical studies of software [80]. Lopes *et al.* [59] study (inter-project) code duplication. Mazinanian *et al.* [67] research lambda expressions in Java, Khatchadourian and Masuhara [24] explore refactoring as a proactive tool for empirically assessing new language features, Bagherzadeh and Khatchadourian [81] investigate common questions asked by big data developers, and Khatchadourian *et al.* [22] examine the use and misuse of Java streams.

## VII. Conclusion & Future Work

This study advances knowledge of refactorings performed and the technical debt they alleviate in ML systems. We have explored refactorings specific and tangential to ML and occurring within and outside of ML-related code. A hierarchical taxonomy of refactorings in ML systems was formulated, 14 and 7 new ML-specific refactorings and technical debt categories, respectively, were introduced, and preliminary recommendations, best practices, and anti-patterns were proposed. In the future, we will explore juxtaposing our findings with developer specialties and expertise and integrating our results into automated refactoring detection techniques [30].

## Acknowledgments

We would like to thank Erich Schubert and the anonymous reviewers for feedback. Support for this project was provided by Amazon Web Services (AWS) and PSC-CUNY Awards #617930049 and #638010051, jointly funded by The Professional Staff Congress and The City University of New York.

# References

[1] 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," in *Neural Information Processing Systems*, vol. 2, MIT Press, 2015, pp. 2503–2511.

[2] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013. DOI: 10.1016/j.jss.2012.12.052.

[3] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high interest credit card of technical debt," in *SE4ML: Software Engineering for Machine Learning*, NIPS 2014 Workshop, 2014.

[4] K. Pei, S. Wang, Y. Tian, J. Whitehouse, C. Vondrick, Y. Cao, B. Ray, S. Jana, and J. Yang, "Bringing engineering rigor to Deep Learning," *SIGOPS Oper. Syst. Rev.*, 2019.

[5] W. Zhou, Y. Zhao, G. Zhang, and X. Shen, "HARP: Holistic analysis for refactoring Python-based analytics programs," in *International Conference on Software Engineering*, 2020.

[6] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static analysis of shape in TensorFlow programs," in *European Conference on Object-Oriented Programming*, 2020.

[7] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on TensorFlow program bugs," in *International Symposium on Software Testing and Analysis*, ACM, 2018, pp. 129–140. DOI: 10.1145/3213846.3213866.

[8] 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 *International Conference on Software Engineering: Software Engineering in Practice*, IEEE, 2019, pp. 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042.

[9] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, "Software engineering challenges of Deep Learning," in *Euromicro Conference on Software Engineering and Advanced Applications*, 2018, pp. 50–59. DOI: 10.1109/SEAA.2018.00018.

[10] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on Deep Learning bug characteristics," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2019, pp. 510–520. DOI: 10.1145/3338906.3338955.

[11] M. Kim, "Software engineering for data analytics," *IEEE Softw.*, vol. 37, no. 4, pp. 36–42, 2020. DOI: 10.1109/MS.2020.2985775.

[12] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson, and I. Crnkovic, "A taxonomy of software engineering challenges for Machine Learning systems: An empirical investigation," in *Agile Processes in Software Engineering and Extreme Programming*, P. Kruchten, S. Fraser, and F. Coallier, Eds., Springer International Publishing, 2019, pp. 227–243. DOI: 10.1007/978-3-030-19034-7_14.

[13] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. Morgan Kaufmann, 2014.

[14] 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, "Managing technical debt in software-reliant systems," in *FSE/SDP Workshop on Future of Software Engineering Research*, ACM, 2010, pp. 47–52. DOI: 10.1145/1882362.1882373.

[15] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[16] W. N. Behutiye, P. Rodríguez, M. Oivo, and A. Tosun, "Analyzing the concept of technical debt in the context of agile software development: A systematic literature review," *Information and Software Technology*, vol. 82, pp. 139–158, 2017. DOI: 10.1016/j.infsof.2016.10.004.

[17] 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*, Aug. 2020. DOI: 10.5281/zenodo.3841195.

[18] T. Zhang, C. Gao, L. Ma, M. R. Lyu, and M. Kim, "An empirical study of common challenges in developing Deep Learning applications," in *International Symposium on Software Reliability Engineering*, IEEE, 2019. DOI: 10.1109/issre.2019.00020.

[19] A. Ketkar, A. Mesbah, D. Mazinanian, D. Dig, and E. Aftandilian, "Type migration in ultra-large-scale codebases," in *International Conference on Software Engineering*, IEEE, 2019, pp. 1142–1153. DOI: 10.1109/ICSE.2019.00117.

[20] F. Falcini, G. Lami, and A. M. Costanza, "Deep Learning in automotive software," *IEEE Softw.*, vol. 34, no. 3, pp. 56–63, 2017. DOI: 10.1109/MS.2017.79.

[21] M. Song and T. Chambers, "Text mining with the Stanford CoreNLP," in *Measuring Scholarly Impact: Methods and Practice*, Y. Ding, R. Rousseau, and D. Wolfram, Eds. Springer International Publishing, 2014, pp. 215–234. DOI: 10.1007/978-3-319-10377-8_10.

[22] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and B. Ray, "An empirical study on the use and misuse of Java 8 streams," in *Fundamental Approaches to Software Engineering*, ETAPS, Springer, 2020, pp. 97–118. DOI: 10.1007/978-3-030-45234-6_5.

[23] R. Khatchadourian, Y. Tang, and M. Bagherzadeh, "Safe automated refactoring for intelligent parallelization of Java 8 streams," *Science of Computer Programming*, vol. 195, p. 102 476, 2020. DOI: 10.1016/j.scico.2020.102476.

[24] R. Khatchadourian and H. Masuhara, "Proactive empirical assessment of new language feature adoption via automated refactoring: The case of Java 8 default methods," *The Art, Science, and Engineering of Programming*, vol. 2, no. 6, 6:1–6:30, 3 Mar. 27, 2018. DOI: 10.22152/programming-journal.org/2018/2/6. arXiv: 1803.10198v1 [cs.PL].

[25] ——, "Automated refactoring of legacy Java software to default methods," in *International Conference on Software Engineering*, IEEE, 2017, pp. 82–93. DOI: 10.1109/icse.2017.16.

[26] S. M. Basha and D. S. Rajput, "Evaluating the impact of feature selection on overall performance of sentiment analysis," in *International Conference on Information Technology*, ACM, 2017, pp. 96–102. DOI: 10.1145/3176653.3176665.

[27] G. O. Campos, A. Zimek, J. Sander, R. J. G. B. Campello, B. Micenková, E. Schubert, I. Assent, and M. E. Houle, "On the evaluation of unsupervised outlier detection: Measures, datasets, and an empirical study," *Data Mining and Knowledge Discovery*, vol. 30, no. 4, pp. 891–927, 2016. DOI: 10.1007/s10618-015-0444-8.

[28] V. R. Eluri, M. Ramesh, A. Salim Mohd Al-Jabri, and M. Jane, "A comparative study of various clustering techniques on big data sets using Apache Mahout," in *International Conference on Big Data and Smart City*, MEC, 2016, pp. 1–4. DOI: 10.1109/icbdsc.2016.7460397.

[29] U. Kamath and K. Choppella, *Mastering Java Machine Learning*. Packt Publishing, 2017.

[30] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *International Conference on Software Engineering*, 2018.

[31] A. J. Viera and J. M. Garrett, "Understanding interobserver agreement: The kappa statistic.," *Family medicine*, vol. 37, pp. 360–363, 5 2005.

[32] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray, "Assert use in GitHub projects," in *International Conference on Software Engineering*, IEEE, 2015, pp. 755–766.

[33] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2008, pp. 329–339. DOI: 10.1145/1346281.1346323.

[34] P. S. Kochhar and D. Lo, "Revisiting assert use in GitHub projects," in *International Conference on Evaluation and Assessment in Software Engineering*, ACM, 2017, pp. 298–307. DOI: 10.1145/3084226.3084259.

[35] J. Bauer. (May 19, 2014). "Make the featurefactory an option, get the parser's objects from the ...· stanfordnlp/corenlp@616d524," Stanford NLP, [Online]. Available: http://git.io/JfowK (visited on 05/28/2020).

[36] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, 2014. DOI: 10.1109/TSE.2014.2318734.

[37] Oracle. (2020). "Type inference for generic instance creation, Java SE documentation," [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html (visited on 08/18/2020).

[38] R. Khatchadourian, "Automated refactoring of legacy Java software to enumerated types," *Automated Software Engineering*, vol. 24, no. 4, pp. 757–787, Dec. 1, 2017. DOI: 10.1007/s10515-016-0208-8.

[39] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2013, pp. 543–553. DOI: 10.1145/2491411.2491461.

[40] Y. Lin, S. Okur, and D. Dig, “Study and refactoring of Android asynchronous programming,” in *International Conference on Automated Software Engineering*, IEEE, 2015, pp. 224–235. DOI: 10.1109/ASE.2015.50.

[41] M. Fowler, *Refactoring: Improving the Design of Existing Code*, English, 2nd ed. Addison-Wesley, Nov. 30, 2018.

[42] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, “Refactoring using type constraints,” *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 3, 9:1–9:47, 2011. DOI: 10.1145/1961204.1961205.

[43] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.

[44] F. Steimann, “The Infer Type refactoring and its use for interface-based programming,” *J. Object Technol.*, vol. 6, pp. 99–120, 2 Feb. 2007. DOI: 10.5381/jot.2007.6.2.a5.

[45] L. White. (Jul. 15, 2018). “Make column type an interface · issue #310 · jtablesaw/tablesaw,” Tablesaw, [Online]. Available: http://git.io/JJNur (visited on 08/18/2020).

[46] S. Audet and A. Black. (Jul. 22, 2019). “Refactor NativeOps.h to export C functions · eclipse/deeplearning4j@dcc72e2,” Eclipse, [Online]. Available: http://git.io/JJNuF (visited on 05/12/2020).

[47] S. Gazeos. (Dec. 3, 2019). “Refactored pad and mirror_pad ops to conform with TF. (#100) · eclipse/deeplearning4j@1905751,” Eclipse, [Online]. Available: http://git.io/JJNuO (visited on 08/21/2020).

[48] K. L. Busbee, *Programming Fundamentals: A Modular Structured Approach Using C++*, English, 1st ed. OpenStax CNX, Jan. 10, 2013. [Online]. Available: https://openlibrary-repo.ecampusontario.ca/jspui/handle/123456789/693 (visited on 08/20/2020).

[49] M. Dilhara, A. Ketkar, and D. Dig, “Understanding Software-2.0: A study of Machine Learning library usage and evolution,” Oregon State University, Nov. 16, 2020. [Online]. Available: https://ir.library.oregonstate.edu/concern/defaults/3b591h056 (visited on 02/12/2021).

[50] R. Roscher, B. Bohn, M. F. Duarte, and J. Garcke, “Explainable machine learning for scientific insights and discoveries,” *IEEE Access*, vol. 8, pp. 42 200–42 216, 2020. DOI: 10.1109/ACCESS.2020.2976199.

[51] A. Zheng, “The challenges of building machine learning tools for the masses,” in *SE4ML: Software Engineering for Machine Learning*, NIPS 2014 Workshop, Dec. 13, 2014.

[52] E. Schubert and A. Zimek. (Feb. 16, 2019). “ELKI: Environment for developing KDD-applications supported by index-structures, Open-source data mining with Java,” ELKI Data Mining Toolkit, [Online]. Available: http://git.io/JUvul (visited on 08/22/2020).

[53] K. Sørensen. (Mar. 24, 2019). “Refactored components to have a “training analyzer” per algorithm. · datacleaner/datacleaner@59f39c7,” DataCleaner, [Online]. Available: http://git.io/JJAgT (visited on 08/19/2020).

[54] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, “Scaling distributed Machine Learning with the parameter server,” in *Operating Systems Design and Implementation*, USENIX, 2014, pp. 583–598.

[55] E. Schubert. (Mar. 31, 2009). “Huge Pair refactoring. · elki-project/elki@6dd5431,” ELKI, [Online]. Available: https://git.io/JUUHG (visited on 08/25/2020).

[56] ——, (Jan. 9, 2009). “Code refactoring, to allow code reuse in other SNN functions · elki-project/elki@eb13202,” ELKI Data Mining Toolkit, [Online]. Available: http://git.io/JUvue (visited on 08/22/2020).

[57] J. Eastman. (Oct. 11, 2011). “[MAHOUT-846] improve scalability of Gaussian cluster for wide vectors - ASF JIRA,” Apache, [Online]. Available: http://issues.apache.org/jira/browse/MAHOUT-846 (visited on 08/22/2020).

[58] ——, (Dec. 22, 2011). “MAHOUT-846: Minor refactoring to eliminate unnecessary…· apache/mahout@4432e31,” Apache, [Online]. Available: http://git.io/JUv2G (visited on 08/22/2020).

[59] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, “Déjàvu: A map of code duplicates on GitHub,” in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, 2017. DOI: 10.1145/3133908.

[60] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, “When does a refactoring induce bugs? an empirical study,” in *International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2012, pp. 104–113. DOI: 10.1109/SCAM.2012.20.

[61] Global App Testing. (Apr. 16, 2019). “Most popular programming languages on Stack Overflow bar chart race,” [Online]. Available: http://youtu.be/cKzP61Gjf00 (visited on 04/23/2020).

[62] Python Software Foundation. (Aug. 26, 2020). “abc, Abstract Base Classes,” Python 3.8.5 documentation, [Online]. Available: http://docs.python.org/3/library/abc.html (visited on 08/26/2020).

[63] T. Winters, “Non-atomic refactoring and software sustainability,” in *International Workshop on API Usage and Evolution*, IEEE, 2018, pp. 2–5. DOI: 10.1145/3194793.3194794.

[64] H. Wright, “Lessons learned from large-scale refactoring,” in *International Conference on Software Maintenance and Evolution*, IEEE, 2019, pp. 366–366. DOI: 10.1109/ICSME.2019.00058.

[65] H. Wright, “Incremental type migration using type algebra,” in *International Conference on Software Maintenance and Evolution*, IEEE, 2020, pp. 756–765. DOI: 10.1109/ICSME46990.2020.00085.

[66] D. Silva, N. Tsantalis, and M. T. Valente, “Why we refactor? confessions of GitHub contributors,” in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2016, pp. 858–870. DOI: 10.1145/2950290.2950305.

[67] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, “Understanding the use of lambda expressions in Java,” in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, vol. 1, ACM, 2017. DOI: 10.1145/3133909.

[68] L. Aversano, U. Carpenito, and M. Iammarino, “An empirical study on the evolution of design smells,” *Information*, vol. 11, no. 7, p. 348, 2020. DOI: 10.3390/info11070348.

[69] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, “A large-scale empirical exploration on refactoring activities in open source software projects,” *Science of Computer Programming*, vol. 180, pp. 1–15, 2019. DOI: 10.1016/j.scico.2019.05.002.

[70] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” in *International Conference on Software Engineering*, IEEE, 2009, pp. 287–297. DOI: 10.1109/ICSE.2009.5070529.

[71] L. Sousa, D. Cedrim, W. Oizumi, A. Bibiano, A. Oliveira, A. Garcia, D. Oliveira, and M. Kim, “Characterizing and identifying composite refactorings: Concepts, heuristics and patterns,” in *International Conference on Mining Software Repositories*, 2020.

[72] A. Hora and R. Robbes, “Characteristics of method extractions in Java: A large scale empirical study,” *Empirical Software Engineering*, 2020. DOI: 10.1007/s10664-020-09809-8.

[73] A. Peruma, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, “An exploratory study on the refactoring of unit test files in Android applications,” in *International Workshop on Refactoring*, 2020. [Online]. Available: http://git.io/JJAgD (visited on 08/19/2020).

[74] I. Ferreira, E. Fernandes, D. Cedrim, A. Uchôa, A. C. Bibiano, A. Garcia, J. L. Correia, F. Santos, G. Nunes, C. Barbosa, and et al., “The buggy side of code refactoring: Understanding the relationship between refactorings and bugs,” in *International Conference on Software Engineering: Companion Proceeedings*, ACM, 2018, pp. 406–407. DOI: 10.1145/3183440.3195030.

[75] A. Potdar and E. Shihab, “An exploratory study on self-admitted technical debt,” in *International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 91–100. DOI: 10.1109/icsme.2014.31.

[76] G. Bavota and B. Russo, “A large-scale empirical study on self-admitted technical debt,” in *International Conference on Mining Software Repositories*, ACM, 2016, pp. 315–326. DOI: 10.1145/2901739.2901742.

[77] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, “Identifying self-admitted technical debt in open source projects using text mining,” *Empirical Softw. Engg.*, vol. 23, no. 1, pp. 418–451, 2018. DOI: 10.1007/s10664-017-9522-4.

[78] L. Rantala, M. Mäntylä, and D. Lo, “Prevalence, contents and automatic detection of KL-SATD,” in *Euromicro Conference on Software Engineering and Advanced Applications*, 2020. arXiv: 2008.05159v1.

[79] B. Christians, “Self-admitted technical debt–an investigation from farm to table to refactoring,” Rochester Institute of Technology, Sep. 2020.

[80] D. E. Perry, A. A. Porter, and L. G. Votta, “Empirical studies of software engineering: A roadmap,” in *International Conference on Software Engineering*, ACM, 2000, pp. 345–355. DOI: 10.1145/336512.336586.

[81] M. Bagherzadeh and R. Khatchadourian, “Going big: A large-scale study on what big data developers ask,” in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2019, pp. 432–442. DOI: 10.1145/3338906.3338939.