Empirical Software Engineering (2023) 28:60
https://doi.org/10.1007/s10664-022-10282-8

Towards a change taxonomy for machine learning pipelines

Empirical study of ML pipelines and forks related to academic publications

Aaditya Bhatia[1] · Ellis E. Eghan[2] · Manel Grichi[3] · William G. Cavanagh[4] · Zhen Ming (Jack) Jiang[5] · Bram Adams[1]

Accepted: 19 December 2022

Abstract

Machine Learning (ML) academic publications commonly provide open-source implementations on GitHub, allowing their audience to replicate, validate, or even extend the ML algorithms, data sets and metadata. However, thus far little is known about the degree of collaboration activity happening on such ML research repositories, in particular regarding (1) the degree to which such repositories receive contributions from forks, (2) the nature of such contributions (i.e., the types of changes), and (3) the nature of changes that are not contributed back to forks, which might represent missed opportunities. In this paper, we empirically study contributions to 1,346 ML research repositories and their 67,369 forks, both quantitatively and qualitatively, by building on Hindle et al.'s seminal taxonomy of code changes. We found that while ML research repositories are heavily forked, only 9% of the forks made modifications to the forked repository. 42% of the latter sent changes to the parent repositories, half of which (52%) were accepted by the parent repositories. Our qualitative analysis on 539 contributed and 378 local (fork-only) changes extends Hindle et al.'s taxonomy with two new top-level change categories related to ML (*Data* and *Dependency Management*), and 16 new sub-categories, including nine ML-specific ones (*input data, parameter tuning, pre-processing, training infrastructure, model structure, pipeline performance, sharing, validation infrastructure, and output data*). While the changes that are not contributed back by the forks mostly concern domain-specific features and local experimentation (e.g., *parameter tuning*), the origin repositories do miss out on a non-trivial 15.4% of *Documentation* changes, 13.6% of *Feature* changes and 11.4% of *Bug fix* changes.

Keywords Machine learning · Change taxonomy · GitHub collaborations · Contribution management

Communicated by: Lei Ma

✉ Aaditya Bhatia
aaditya.bhatia@queensu.ca

Extended author information available on the last page of the article.

Published online: 01 April 2023

1 Introduction

The notion of "Software Engineering for Machine Learning/Artificial Intelligence" (SE4ML/SE4AI) is becoming widespread in the software engineering community, with software engineering conferences featuring dedicated tracks and with dedicated venues appearing (RAISE, SEMLA, CAIN). The term "machine learning software" can mean different things depending on the context, spanning across a wide range of software projects from project-specific applications to third-party ML frameworks, to ML pipelines using such frameworks.

At one end of the spectrum, **Machine Learning (ML) frameworks** like Tensorflow[1] and PyTorch[2] provide generic implementations of ML classification, regression, recommendation, and clustering algorithms, for use in any possible domain. At another end of the spectrum, **end user applications** integrate models into their code base to make domain-specific predictions. Since those models are domain-specific, an infrastructure is needed to continuously ingest data, perform pre-processing, build, tune and evaluate ML models specific to a given domain (e.g., image recognition or fraud detection), by orchestrating scripts and ML framework tools that produce datasets, models and evaluation/execution metadata (Idowu et al. 2021). This infrastructure is called an **ML pipeline** (Amershi et al. 2019; Nahar et al. 2022), and forms the core of any organization's ML activities, catering to interdisciplinary teams of data scientists, data engineers, and developers (Sato et al. 2019). An important subset of ML pipelines is produced in the context of published academic papers (Fan et al. 2021). Typically, researchers would upload a preprint of their work on ArXiv, including a GitHub repository with the pipeline code and (potentially) a labeled data set to train the pipeline on. Alternatively, other researchers or open-source developers might open-source an implementation or dataset of such a paper. Such code and data allow the open-source community to leverage state-of-the-art ML research to develop applications and benefit from the publications' ideas. Popular online indexes like PapersWithCode[3] or ModelDepot[4] provide searchable lists of papers, and their associated artifacts like implementations and/or datasets. Given the popularity[5] of these ML pipeline open-source projects, the remainder of this paper focuses on this subset of ML pipeline projects.

While ML pipelines, including those related to an academic publication, typically are shared in the form of GitHub repositories[6], an important question is to what extent such projects benefit from the open-source collaboration models leveraged by traditional (non-AI) GitHub projects, as opposed to just being an "online backup" or a "replication package". The typical GitHub collaborative coding model would see the OSS community *fork* an ML research project (Zhou et al. 2020), make changes to the source code, and push those changes back to the original project using *Pull Requests* (PRs). The developers of the original project would then check such PRs and accept or reject the proposed changes. Accepted PRs would then lead to code changes being merged in the ML research repository, a

[1] https://www.tensorflow.org
[2] https://pytorch.org
[3] https://paperswithcode.com
[4] https://modeldepot.io/
[5] In August 2022, PapersWithCode indexed more than 77,640 academic AI publications along with their code bases.
[6] In the remainder of this paper, we use the term *"ML research repositories"* to identify such ML pipelines.

so-called *upstream change*. Oftentimes, community developers could also make changes for their own use that they would not contribute back, i.e., so-called *downstream changes*. GitHub's collaborative forking model has been known to improve the productivity of multifaceted software development and management tasks like making new features, handling issues, sharing knowledge, adding documentation, and managing upstream and downstream code (Zhou et al. 2020).

While OSS collaborations in non-ML software are extensively studied by researchers (Biazzini and Baudry 2014; Hu et al. 2016; Lima et al. 2014; Zhou et al. 2019, 2020), this is not the case in the context of ML. Certain assumptions of established software engineering activities like requirements engineering, software design, and quality assurance are no longer valid (Washizaki et al. 2019; Ozkaya 2020), while a typical ML development team no longer only features traditional developer and tester roles, but also data scientists and data engineers (Amershi et al. 2019). Such multi-disciplinary collaboration leads to a wide range of different artifacts other than source code that require proper versioning and traceability with the code (Idowu et al. 2021).

Among the emerging software engineering practices replacing existing models of (multi-disciplinary) collaboration in the context of SE4ML, the types of code changes made on typical ML pipelines need to be explored to capture the nature of such community-based changes and compare them to pre-ML types of changes. In particular, in 2008, Hindle et al. (2008) presented a seminal taxonomy of code changes for traditional (non-ML) software, which identified seven code change dimensions, along with 24 sub-categories of changes (Table 1). Despite being 13 years old at the time of writing this paper, the taxonomy is still authoritative today. However, the advent of the ML era within the collaborative development environment calls for the need to substantially revise this taxonomy.

Hence, this paper empirically studies changes in ML research repositories and their forks, using a mixed-methods approach. First, we quantitatively mine the community collaborations to 1,346 ML research repositories (containing implementations of 1,144 arXiv publications) obtained via ModelDepot's "Deep Search" engine to analyze the behavior involving their forks, i.e., how active is online collaboration on ML research repositories? We then perform a large-scale qualitative analysis of 539 upstream and 378 downstream changes, adapting Hindle et al.'s (2008) taxonomy of code changes. Notably, we address the following research questions:

- **RQ1: To what extent do ML research repositories form the basis of other contributors' work?**
  ML research repositories are heavily and transitively forked, yet overall only 9% of forks made modifications. 41.6% of the latter forks sent changes back to the parent ML research repositories (i.e., upstream changes), half of which (52%) were accepted by the parent repositories. The time taken to merge those pull requests is about four times faster than for NPM packages on GitHub (Dey and Mockus 2020) and 24 times faster than for GitHub projects in general (Gousios et al. 2014).
- **RQ2: What are the types of changes in ML research repositories?**
  Using the seminal code change taxonomy of Hindle et al. (2008) as a starting point, we identified two new categories of changes in ML research repositories, namely *Data*, and *Dependency Management*. Furthermore, we refined the taxonomy with 16 new sub-categories of changes. Nine of the sub-categories (i.e., *input data, parameter tuning, pre-processing, training infrastructure, model structure, pipeline performance, sharing, validation infrastructure, and output data*) are ML-specific, while seven (i.e., *add dependency, remove dependency, update dependency, file permissions, internal*

Table 1 Hindle et al.'s taxonomy of change types for traditional SE (Hindle et al. 2008)

Category	Sub-Category	Definition
Maintenance	Bug fix	Fixing bugs (e.g., adding exception control, conditional statements)
	Cross	Cross-cutting changes (e.g., logging)
	Maintenance	Performing activities during maintenance cycle other than fixing bugs
	Parameter list change	Updating in the parameters list
	Debug	Setting up debug, tracking process (e.g., printing variable values, execution times)
Meta-Program	Documentation	Changing the software documentation (e.g., read-me file, code comments)
	Build/Config	Changing build or work-space configuration files (e.g., setup.txt or .yml)
	Testing	Adding unit tests, bench-marking, changing test environment
	Internationalization	Adding language support other than English
Non-Functional Source Code Change	Refactor	Structural changes without changing the behavior (e.g., renaming variables, optimizing code)
	Clean up	Deleting code not used by the program (e.g., print statements, comments, unused imports)
	Indent	Adding proper indentation or formatting the code
	Token replace	Renaming tokens like variable or method names
Source Management	Merge	Merging commits or pull requests
	Source control	Managing repository files (e.g, adding files to git ignore)
	Versioning	Changing the software release version
	Branching	Creating a side development branch from the main branch
	External	Code submitted by developers who are not a part of the core team
Implementation	Feature	Adding new functional features
	Platform	Changing hardware or platform-specific code (e.g., changing GPU hardware acceleration, changing file access for a new platform)
Module Management	Add module	Adding modules/directories/files
	Move module	Moving modules/directories/files
	Remove module	Removing modules/directories/files
Legal	Licence	Changing copyright or authorship

documentation, add auto-generated code, and program-metadata) are more general sub-categories.

– **RQ3: How do downstream changes differ from upstream changes in ML research repositories?**
Manual comparison of changes contributed back by the forks to the origin ML repository (upstream changes) with changes not contributed back (downstream changes)

shows that downstream changes typically are domain-oriented and add *input/output data*, perform *parameter tuning*, add new functional *features*, and perform other non-functional changes like *indentation, refactoring* or *cleaning* up the source code. In contrast to this, upstream changes benefit the parent repository by *updating dependencies* or *fixing bugs* for the parent repository. Both downstream and upstream contributions add *documentation*, and *fix bugs*.

The remainder of the paper is organized as follows: Section 3 presents the data collection and design of our study. Section 2 summarises the related literature. Section 4 discusses the motivation, approach, and results for each of our research questions. Section 6 discusses threats to the validity of our study while Section 5 explains the implications of our findings. Finally, Section 7 concludes the paper.

2 Related Work

2.1 Code Change Classification

Prior work in code change taxonomies initiated in 1976 with Swanson et al.'s work on identifying changes during software maintenance in terms of *corrective, adaptive* and *perfective* (Swanson 1976). The goal of such taxonomies originated from the need to enhance software decision-making.

These changes were adopted as *extended-Swanson categories* by Hindle et al. (2008) in the latter seminal taxonomy of software changes in 2008. A detailed description of Hindle et al.'s taxonomy is provided in Table 1. Despite its important role, the taxonomy is in need of updates. For instance, software development has become more collaborative since 2008 due to platforms like GitHub, leading to additional change types that would need to be added to the taxonomy in Table 1. Furthermore, the types of changes required in an ML setting like ML pipelines could lead to further missing change types, which this paper aims to study. Hence, our qualitative study builds on Hindle et al.'s change taxonomy, extending it with two new high-level change categories and 15 new sub-categories of changes.

Later work shifted direction from establishing taxonomies to automated classification of code changes in terms of activities defined by such change taxonomies. In Hindle et al. (2009) later publication, the authors automatically classify maintenance changes into *corrective, adaptive, perfective, feature addition*, and *non-functional improvement* categories using ML techniques. Yan et al. (2016) improved this approach of classifying code changes using a Discriminative Probability Latent Semantic Analysis (DPLSA) approach, which showed its benefits in multi-category classifications of code change activities during the evolution of software. Recently, in 2021, Ghadhab et al. (2021) further improvised the classification of code changes using BERT (Bidirectional Encoder Representations from Transformers) approach.

Code change taxonomies are used for a variety of purposes. In 2009, Benestad et al. (2009) performed a literature survey on publications that assessed the impact of individual code changes on the maintenance and evolution of software systems. Wu et al. (2011) extracted missing links between bugs and committed changes by creating an automated tool, *Relink*. Furthermore, Bissyandé et al. (2013) evaluated the efficacy of linking bug reports to code changes by benchmarking Relink against alternative bug-linking solutions. Cortés-Coy et al. (2014) used code changes to automatically generate commit messages. Faragó and Hegedũs (2014) studied code changes to understand the impact of change operations

(like add/update/delete) on ISO/IEC-9126 quality attributes of software. Software developers and researchers developing such tools may wish to inculcate our extended taxonomy of code changes to better support the development of ML systems.

2.2 Multi-Repository Software Development via Forking

Many researchers study collaborative development. For instance, Zhou et al. (2019) identified efficient practices for developers collaborating using forks. The authors build regression models to correlate efficient practices with respect to the behavior around forking. They found how the modularity of a code base and its contributions, as well as upfront management of which bugs require fixing by contributors, correlate with higher contribution volume and pull request acceptance.

Later, in 2020, in a follow-up work (Zhou et al. 2020), elucidated the perceptions around "hard forks" (forks that split development into a competing line of a new repository), against those of "social forks" (forks that create a public copy of the repository on a social website like BitBucket or GitHub). While hard forking traditionally has been considered a bad practice for developers and users (Fogel 2005), the authors found that the perceptions around hard forking have changed in modern times. Nowadays, hard forks emerge out of social forks, and are seen as a positive non-competitive alternative to the original repository. Constantino et al. (2020) identified the rationales, processes, and challenges behind collaborative activities on GitHub by conducting surveys. The authors found that GitHub collaborations contribute to software development, issue management, repository management, and documentation tasks.

Brisson et al. (2020) studied collaborations on GitHub projects by analyzing transitive forks, user statistics, pull requests, and issues. Furthermore, Biazzini and Baudry (2014) identified dispersion metrics for fork-induced code changes. Ren et al. (2018) developed a web UI for the management of forking-based collaborations with features like fork searching and tagging. Other research (Rahman and Roy 2014; Zhang et al. 2018) studies the nature of upstream contributions in the form of *Pull Requests* and identifies the nature of competing contributions.

However, none of this prior research studies the collaborative development of ML software. We build on the results from prior studies to compare the forking dynamics of ML research repositories.

2.3 Software Engineering for Machine Learning

ML systems are substantially different from traditional software systems and hence need dedicated research. For example, Washizaki et al. (2019) found that ML software engineering design patterns differ from those of non-ML software. Furthermore, in 2020, Ozkaya (2020) illustrated how the stochastic nature of ML changes the software development practices in ML. Overall, Martínez-Fernández et al. (2021) performed a literature review on Software Engineering for AI-based systems.

Recent widespread advances in ML have instigated researchers to study the maintenance activities and challenges in ML code. In 2019, Amershi et al. (2019) uncovered the challenges in managing ML software at Microsoft, in particular identifying the typical ML pipeline and corresponding software activities. Furthermore, Zhang et al. (2019) and Arpteg et al. (2018) studied the software challenges faced in deep learning applications. Sambasivan et al. (2021) identified data engineering challenges for which multiple roles of data engineers (like data collectors, annotators, ML developers, and data licensing teams) require

powerful data infrastructure in order to support machine learning processes. In the context of the data lifecycle used for ML, Polyzotis et al. (2018) illustrated the challenges faced at Google. O'Leary and Uchida (2020) also studied problems with creating ML pipelines from existing code at Google.

The work of Fan et al. (2021) is the closest to our paper. While the authors study a similar dataset as ours (i.e., 1,149 academic ML(AI) repositories referencing ArXiv publications), the authors focus on characterizing popular versus unpopular academic repositories in terms of the number of stars on GitHub, and analyzing factors correlations between the number of paper citations and GitHub repository metrics. However, our paper is the first to study the extent of actual OSS collaborations happening on ML research repositories (instead of paper activity based on those repositories), and to manually identify the nature of code changes performed on such ML pipeline projects.

3 Data Collection and Experiment Setup

Figure 1 presents an overview of our data collection procedure along with the design of our empirical study to address the research questions of the introduction. RQ1 quantitatively studies the OSS collaboration characteristics on ML research repositories, while RQ2 and RQ3 perform a qualitative analysis on the nature of changes performed during this collaboration.

3.1 Data Collection

ModelDepot was a popular online model store containing 1) a catalogue of pre-trained ML models, and 2) a GitHub search engine for ML model pipeline implementations called "Deep Search". The latter engine effectively was an index of GitHub projects related to ML, allowing to search the projects based on name, ML framework (e.g., Tensorflow),

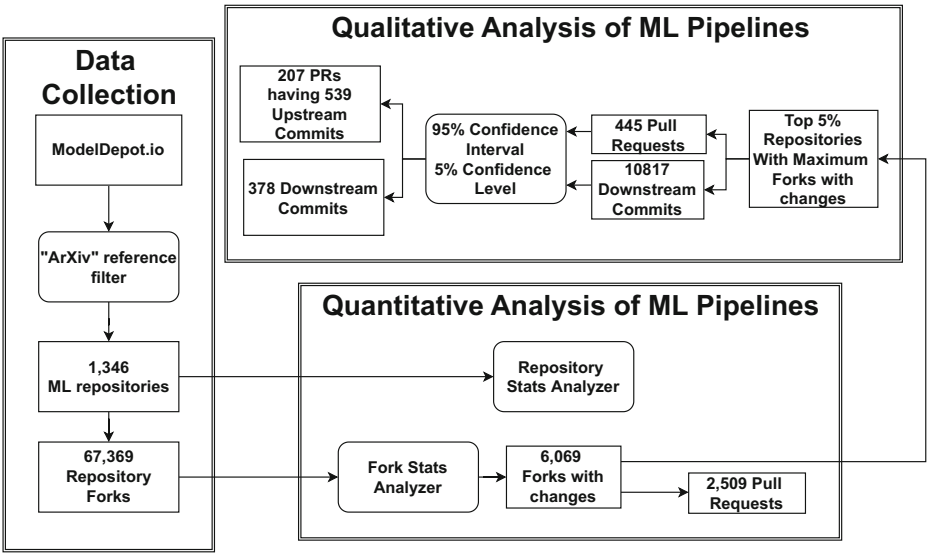

Fig. 1 Data collection and processing steps

programming language, and five model categories (i.e., "computer vision", "natural language", "reinforcement learning", "generative" and "audio"). At the time this study was conducted in summer 2019, ModelDepot indexed over 50,000 ML implementations.

While both ModelDepot and its major competitor at that time, PapersWithCode, indexed GitHub repositories, we selected ModelDepot for our study because it was the most popular and diverse at the time of crawling the data (i.e., 50,000 model implementations compared to 8,500 on paperswithcode.com[7]). The fact that ModelDepot did not require manual contributions to register new models, but leveraged its automated "Deep Search" engine to track new ML model repositories, was another reason why we opted for ModelDepot. To collect the ModelDepot data, we built a scraper (crawler) to mine ML projects in the "Computer Vision" and "Machine Learning" categories, which were the two most common categories of models[8]. After sorting by the search engine's "best match" feature, we then focused on the top 5,000 non-fork projects.

Since this paper focuses on the evolution of ML model pipelines produced by researchers or inspired by the work of researchers, we filtered the 5,000 crawled projects using string matching to check the readme files for the presence of the term "arxiv" (referring to an ArXiv URL of an academic paper). This yielded 1,346 ML research repositories as our dataset for this study. Within this dataset, 1,144 unique academic papers were referenced and 23% of these publications were referenced more than once. One of these publications, "Deep Residual for Image Recognition"[9] was referenced the most (46) by the repositories in our dataset.

To further verify the soundness of our dataset, we did a separate analysis to check the quality of the studied ArXiv publications. To do so, from our dataset of 1,346 ML research repositories, we used a 95% confidence interval and a 10% confidence interval to obtain a sample of 94 repositories. For each of the 94 repositories, we checked whether their referenced ArXiv papers:

- were peer-reviewed?
- involved authors from the industry?
- had any of the authors amongst the people contributing to the repository's development?

To check whether publications were peer-reviewed, we looked at whether the ArXiv paper was also published at another venue, by leveraging scholar.google.com. Next, to check whether the authors were from the industry, we looked at the email addresses and the designations of the authors presented in the publication. Finally, we check whether the authors contributed to the repository.

As such, we found that all repositories from our inspected sample of 94 repositories are implementations of academic ML research. The ArXiv paper(s) referenced in the repositories' README file clearly showed the intention to implement the published algorithms, not just mentioning the work as a comparison or reference. Only **30% of the repositories were developed by the referenced publications' authors themselves**, while the other other 70% were implemented by other researchers or by open-source developers. We observed that the latter repositories basically referenced the paper(s) they were implementing. Since those repositories still represent an implementation of an ML publication shared with the open-source community, all of these represent genuine ML pipeline research repositories that we focus on in our research.

[7] https://twitter.com/paperswithcode/status/1091315540092768257

[8] https://web.archive.org/web/20190404211946/https://modeldepot.io/search/results?q=

[9] https://arxiv.org/abs/1512.03385

For instance, a repository named darkflow[10] referenced two academic publications[11][12] to indicate that they implemented the image processing algorithm YOLO, proposed by the said research. Another repository, ActivityRecognition[13] presented multiple publications[14][15][16] within the "Reference Papers" section of its ReadMe, for the same reason. While both repositories have no indication that one of the publications' authors was a contributor, the repositories clearly indicate that they implemented the referenced academic research.

Furthermore, **81% of the repositories implemented peer-reviewed publications**, indicating that the implemented research is high quality. Finally, **59% of the repositories referenced publications involving authors from the industry**, indicating how repository development focuses on industry-relevant research.

3.2 Quantitative Analysis of Forking in ML Research Repositories (RQ1)

To analyze the dynamics of OSS collaboration through forks of ML research repositories (RQ1), we use the GitHub Search API[17] to analyze each of the 1,346 repositories in our dataset. We perform our quantitative analysis via seven metrics related to forking as described in Table 2. Three out of these metrics (the bolded ones in Table 2) are adapted from Brisson et al. (2020).

Given a large number of forks, Brisson et al. (2020) suggested that fork data is noisy. For this reason, we introduce the concept of `Forks_with_changes` to identify *forks with modifications*. Such `Forks_with_changes` contain at least one commit that does not occur in the parent repository (downstream changes), or contributed back at least one commit via a pull request (upstream changes).

To identify forks with downstream changes, we first calculate for each fork F the set of commits S_F whose commit id does not occur in the parent repository. Since the resulting set S_F of fork commits could still contain commits that have been merged upstream through rebasing (changing their commit id), we then check for each commit in S_F to see if any commit in the parent repository has the same commit message subject, author name, and author date, since those metadata fields have been found to be stable during rebase (German et al. 2016). If so, we remove those commits from S_F, since they also exist in the parent repository. Since we may be missing cases where a fork had all of its commits merged as PRs, we then check the list of forks that submitted PRs using the GitHub Search API, and add such forks into S_F. If the resulting S_F is not empty, we consider each F in S_F to be a *changed fork*.

We used the `Star#` and `Fork#` metrics to measure the popularity of repositories, as indicated by Borges and Valente (2018). We computed the `First_Fork_time` and `Final_fork_time` metrics to indicate the temporal aspects of ML research repositories. The `First_Fork_time` indicates the speed of the OSS community in adapting an ML implementation, whereas the final fork time indicates the longevity of collaboration activities on ML pipeline code.

[10] https://github.com/thtrieu/darkflow
[11] https://arxiv.org/pdf/1506.02640.pdf
[12] https://arxiv.org/pdf/1612.08242.pdf
[13] https://github.com/mohammed-elkomy/two-stream-action-recognition
[14] https://arxiv.org/pdf/1406.2199.pdf
[15] https://arxiv.org/pdf/1604.07669.pdf
[16] https://arxiv.org/pdf/1507.02159.pdf
[17] https://docs.github.com/en/rest/reference/search

Table 2 Metrics used in RQ1

Metric	Definition
Star#	# users who starred the repository.
Fork#	# forks created from the repository.
Forks_with_changes#	# forks where the forked code base is modified.
PR#	# PRs sent to the parent repository. by its forks.
PR_Accept%	(# PRs merged into parent repository) / (PR#)
First_Fork_Time	# Days between creation of a repository, and its first fork.
Final_Fork_Time	# Days between creation of a repository, and its final fork.

The bolded metrics are adapted from Brisson et al. (2020)

In addition, we calculate for each repository the number of forks, both direct and transitive, as a measure of the value of these repositories for the OSS community in terms of collaborative potential. We call the direct forks of a repository *level-one transitivity*, while a transitivity of *level-two* indicates the transitive forks of the direct (level-one) forks, and so on. The higher the proportion of repositories with at least one *level-two* fork, the more collaboration seems to happen.

3.3 Qualitative Analysis of Change Types in Forks (RQ2/RQ3)

Since ML software has obtained a prominent place in software engineering, and the nature of the machine learning software lifecycle is substantially different from that of traditional software (Amershi et al. 2019), one would expect further change types to be added. ML practitioners use data pre-processing techniques, iteratively *tune* model hyperparameters for obtaining the most optimal ML model under the data science life cycle (Amershi et al. 2019; Nahar et al. 2022). Reusing ML software requires users to understand the rationale behind the ML implementations, instigating users to change their code for *documenting the internal working* of the ML code. All of this has led to a variety of types of artifacts other than source code that require changes as well (Idowu et al. 2021). In this study, we build on Hindle et al.'s taxonomy to identify the types of changes in ML research repositories by studying a statistically significant sample of 1) fork PRs merged by the parent ML research repository (i.e., **upstream changes**) and 2) commits within the forked repositories that were not submitted as PRs (i.e., **downstream changes**). We describe our qualitative analysis process below:

1. **Selection of repositories for sampling.**
   Since not all forked repositories made changes, let alone sent them as PRs upstream, we first select the repositories having the most active forks. To do so, we ordered the repositories by our metric `Forks_with_changes` and obtained the top 5 percentile, i.e., 23 repositories. Overall, these 23 repositories had 10,817 downstream commits and 445 PRs. From these, using a 95% confidence level and 5% confidence interval, we obtain a statistically representative sample of 1) 207 PRs containing 539 upstream commits (**upstream sample**[18]), and 2) 378 downstream commits (**Downstream Sample**).

[18] PRs on GitHub are not limited to just one commit.

Both samples were stratified, such that repositories with a higher proportion of `Forks_with_changes` had more data points in the samples. Since a PR can comprise multiple commits, we selected all the commits for the sample of 207 PRs and obtained 539 upstream commits. Similar to obtaining the stratified upstream sample, we used the proportion of `#Commits` with respect to the `Forks_with_changes` of each of the 23 parent repositories for creating the stratified downstream sample.

2. **Study Participants.** We used teams from two universities to manually classify the types of changes in upstream and downstream commits. `University-A` classified downstream changes while `University-B` classified upstream changes. Both teams included two or more grad students, and one faculty member, all having knowledge of ML and non-ML software design and development. Due to the large-scale nature of our study, we employed four coders in `team A` and three coders in `Team B`.
3. **Extending taxonomy of changes.** From the sample of 378 downstream commits, `Team-A` first performed a pilot study on an initial sample of 78 commits to validate the extent to which Hindle et al.'s (2008) taxonomy was able to classify code changes or required refinements. The 78 commits were distributed across the three coders of `Team-A` such that each commit had two coders and each coder had 26 commits in common with each of the other coders. The assignment of commits to each coder was anonymous.

   Each coder then individually coded their 52 (2×26) assigned commits, identifying all types of changes within the commits under study (more than one type of change could apply). In cases where the change (sub-)type could not be found using Hindle's change categories, the coders individually could create a new category. Once finished, the coders met online discussing only the new types of changes that they had identified, without considering the specific commits tagged with these new types. The proposed new types could be merged, renamed, or removed until a consensus was reached.

   `Team-A` then re-labeled their samples using the enhanced Hindle's taxonomy. Once done, the coding results were combined into a spreadsheet. In the first phase, each coder had to check the commits they were assigned that had conflicting coding results. This was done asynchronously by adding comments on the spreadsheet. If a coder was in accord with the other coder's interpretation, a disagreement was resolved; otherwise, it was left open. In a second phase, the remaining disagreement cases were then discussed in person by `Team-A`, possibly refining the taxonomy.

   With this final version of the taxonomy, `Team-A` started labeling the remainder of its samples, using the same style of assignment as for the initial 78 (i.e., anonymously sharing the same number of commits with each other coder). In parallel, `Team-B` was assigned the 207 PRs in a similar manner. While both teams could still make changes to the taxonomy during this coding, we observed saturation in the labels after tagging the initial set of 78 downstream commits, i.e., no new (sub-)categories were identified in the later part of labeling the 300 downstream samples and 539 upstream samples.
4. **Calculation of inter-rater agreement.** Once coding was finished, both teams individually used the spreadsheet-based and in-person resolution of disagreements used initially for the first 78 cases. To calculate inter-rater agreement, since each sample was rated by two participants, we use Krippendoff's Alpha (Krippendorff 2011) as our metric for the inter-rater agreement. This metric supports multi-label classification by multiple participants. A similar approach of obtaining inter-rater agreement in a multi-rater setting was used by Li et al. (2020) to manually tag logging data, and Salza et al. (2018) to classify mobile app updates.

Across the three coding activities (78 and 300 commits for `Team-A`, and 207 PRs for `Team-B`), the teams reported high agreements with a Krippendoff's α=98% on the sample of 378 downstream changes and a Krippendoff's α=92% for the sample of 539 upstream changes. These values of inter-rater agreements are high and reflect the statistical robustness of our data labeling results.

Given that the final change taxonomy spans 39 change sub-categories, and that we coded 378 downstream commits and 207 upstream PRs (containing 539 commits) across two teams of seven coders (and two universities), the resulting empirical study was non-trivial. For instance, in the downstream change[19] performed by fork "BoseAslCohort" for the project "Youtube-8m", the authors manually inspected changes for 27 changed files, which included 11,755 code additions. Overall, it took an estimated six man-months to finish the qualitative study.

4 Case Studies

4.1 RQ1: To What Extent do ML Research Repositories Form the Basis of Other Contributors' Work?

Motivation Currently, there is no empirical evidence regarding the extent to which 1) open-sourcing ML research code helps the OSS community in building new applications and 2) the OSS community contributes and helps maintain the original ML research implementations. In contrast, for non-ML software, prior research (Zhou et al. 2019; Biazzini and Baudry 2014; Brisson et al. 2020; Rahman and Roy 2014; Zhang et al. 2018) has studied the nature of multi-repository development and maintenance of OSS projects. Hence, in this RQ, we analyze the OSS development activities around research-based ML pipeline repositories.

Approach As discussed in Section 3, we extract 1,346 GitHub repositories having references to machine learning ArXiv publications, then use the GitHub Search API[20] to obtain the metrics identified in Table 2.

Results

4.1.1 Forks_With_Changes

Only 9% of forks of the ML research repositories have modifications to the forked source code (i.e., have `Forks_with_changes`) Overall, 82.5% of the 1,346 repositories had forks. Figure 2 shows the cumulative percentage of those repositories with a fork having at least one `Fork_with_changes`. Since 51.6% of the repositories do not have any fork modification (only non-changed forks), and the slope of the curve is gentle and linear until 90%, the percentage of ML research repositories with `Forks_with_changes` is low.

[19] https://github.com/BoseAslCohort/youtube-8m/commit/c1b01315bafc24e83248cd862a9324bb21d4d52d
[20] https://docs.github.com/en/rest/reference/repos

4.1.2 Popularity of ML Research Repositories

ML Research Repositories Have a High Median `star#` of 22 and `fork#` of 8. These numbers stand in stark comparison to the datasets used by prior research for non-ML repositories. We observed a median of zero stars and forks for the replication dataset provided by Brisson et al. (2020), consisting of 13,431 projects. The comparisons of `star#` and `forks#` for Brisson's dataset with our study are statistically significant with Wilcoxon Rank sum $p - value < 0.01$. **`Star#` and `fork#` are highly correlated (Spearman ρ=0.94)** as shown by the data distribution in Fig. 3. As indicated by the darker color at the low end of `star#` and `fork#` in Fig. 3, 20% of the repositories have less than five stars and five forks.

These results again contrast to the low correlation of 0.45 found by Brisson et al. on their non-ML dataset of 13,431 repositories, suggesting a much weaker connection between stars and forks. Several hypotheses might explain this contradiction, and require future work to be validated. For example, due to the current hype of AI technologies, ML repositories might be substantially more popular than non-ML repositories. It could also be that, due to the quick succession of new AI algorithms, the OSS community uses forks for the purpose of "bookmarking" or keeping copies of interesting ML research implementations (Kalliamvakou et al. 2014). One indication of the latter hypothesis could be the high percentage (91%) of forks without any code change (i.e., non-changed forks) that we found earlier.

4.1.3 Speed and Longevity of Forking

Forks on ML repositories appear as fast as the 11th day (median 11.5 days), while fork-based collaboration sustains a median of 2.6 years. Fig. 4 shows the distribution of the time of the first and the final (at the time of analysis) fork for each repository in our dataset. A median ML research repository receives its first forks on the **11th day** after creation date, while an ML research repository is forked till a median of **2.6 years** of the creation of the

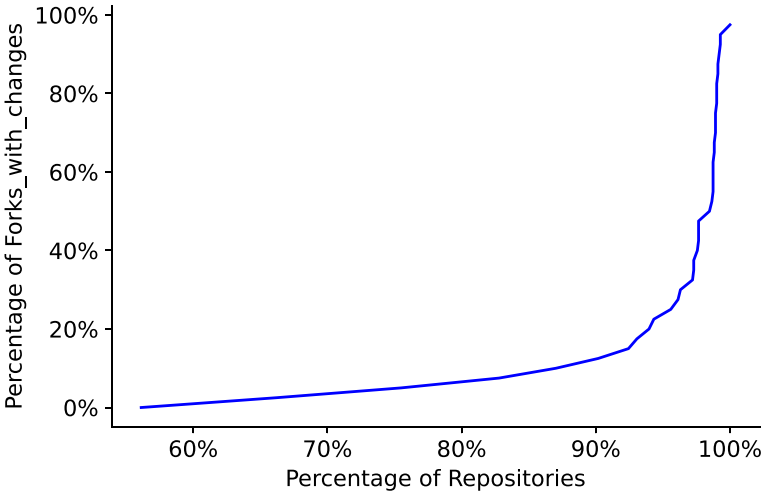

Fig. 2 Percentage of `Forks_with_changes` across studied repositories. 52% repositories do not have any modifications to the forked source code

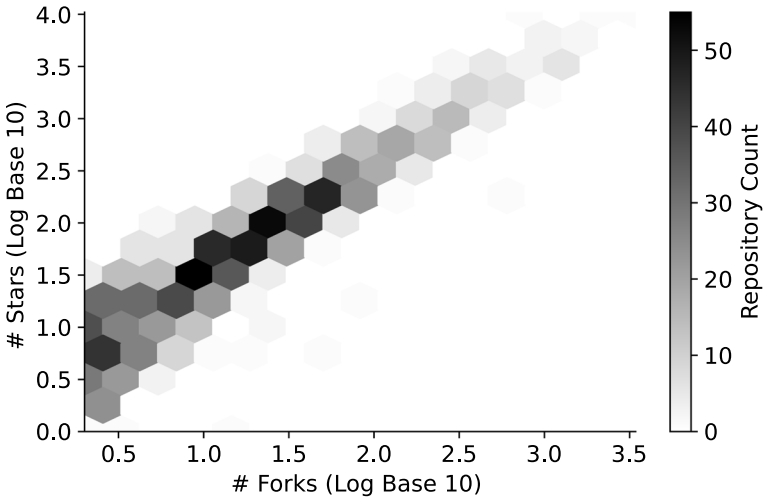

Fig. 3 Hexbin for `Star#` (logged) and `Fork#` (logged) indicates high correlation. Popularity can be indicated by either of the metrics

repository. Although the `final_fork_time` may be impacted by the time of our analysis, nevertheless, a value of 2.6 years definitely shows that the ML repositories are not just data dumps but can foster online collaboration.

4.1.4 Transitive Forking

Transitive forks (i.e., fork repositories with their own forks) are present in 20% of ML research repositories. We observed 67,369, 1,581, 44, and 7 cases of direct forks, level-two, level-three, and level-four forking transitivity for 1,110, 226, 28, and 3 ML

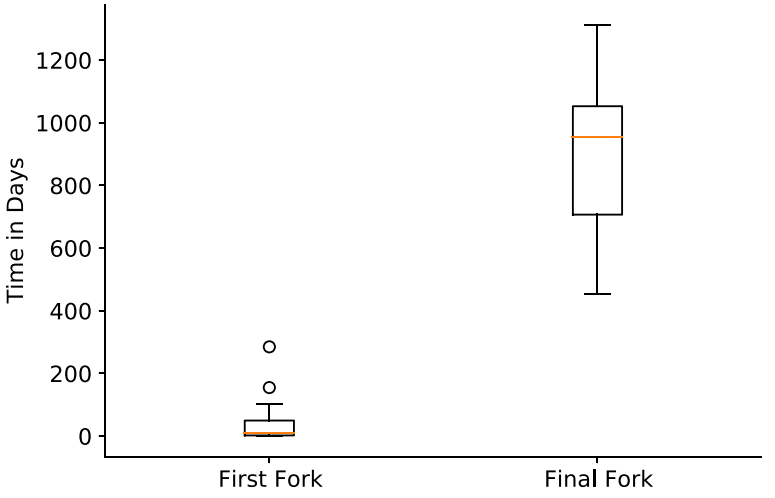

Fig. 4 The first forking time represents the speed of the open source community in adapting ML research repositories, whereas the final forking time represents the longevity of making contributions to the ML repositories

research repositories respectively. Hence, 226 out of 1,346 repositories, i.e., 20%, have at least one transitive fork (since the 28 with level-three forks are a subset of the 226, etc.). The ML research repository with the highest forking transitivity in our dataset includes the Autopilot-TensorFlow project[21], which is an implementation of self-driving car research[22]. This project has 281 direct forks, 10 level-two, 3 level-three, and 5 level-four transitive forks.

We compare our findings to the fork transitivity results reported by Brisson et al. (2020), who conducted an analysis of the March 2019 GHTorrent dataset and reported 12,171 level-one, 778 level-two, 84 level-three, 11 level-four, and 2 level-five forks. A χ^2 test of independence between these findings and ours yielded a $p - value < 0.01$, representing a statistically significant difference in data distribution between ML and non-ML repositories.

4.1.5 Upstream Contribution

In terms of upstream contributions to the ML research repository, 41.6% of `Forks_with_changes` send changes back to the original repositories. A total of 607 pull requests were submitted upstream by `Forks_with_changes`, out of which 316 were merged into the original repositories. This resulted in **52.1% acceptance.** This value is slightly lower than that of a recent study on NPM packages by Dey and Mockus (2020), who reported a PR acceptance rate of 60%.

27.5% of the upstream PRs were submitted on the same day as that of the creation of the fork. In particular, a fork takes a median of 22 hours to submit a PR. After receiving a PR, the parent ML research repository takes a median of seven hours to review the upstream changes before deciding on them, as shown by the violin plots in Fig. 5. This is approximately four times faster than the median PR acceptance times (27.7 hours) for NPM packages on GitHub (Dey and Mockus 2020). Another study performed on 1.9 million PRs on GitHub in 2013 by Gousios et al. (2014) reported a median of seven days to merge a PR.

Summary of RQ1

The OSS community forks the ML research as fast as a median of 11.5 days after the creation of the repository, and forking continues until a median of 2.6 years. ML research repositories are heavily and transitively forked, yet only 9% of the forks have modifications. Of those, 43% sent upstream changes back to the parent ML research repositories, with a 52% acceptance rate. PR merge times of ML research repositories are faster than the values reported by prior research for non-ML repositories.

4.2 RQ2: What are the Types of Changes in ML Research Repositories?

Motivation Since Hindle et al.'s (2008) taxonomy of changes focused on traditional software systems known in 2008, this research question performs a qualitative analysis to

[21] https://github.com/SullyChen/Autopilot-TensorFlow
[22] https://arxiv.org/pdf/1604.07316.pdf

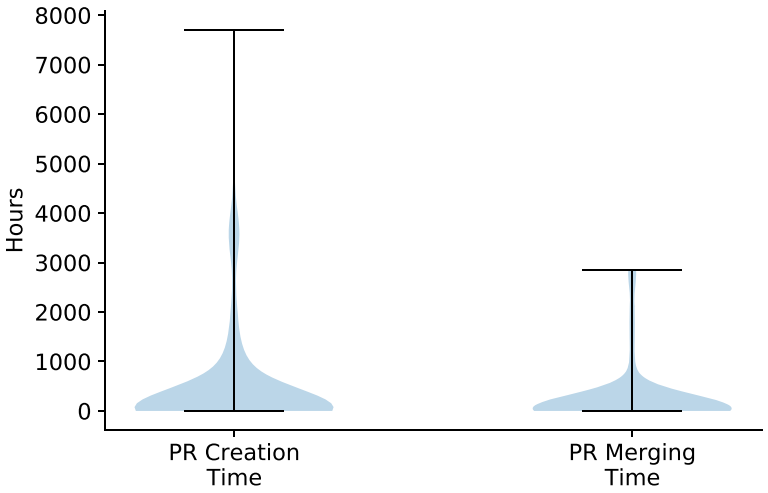

Fig. 5 Time spent in sending PRs (left) and merging the PRs into the upstream parent repository (right)

identify the types of changes made in ML research repositories, possibly extending Hindle's change taxonomy. Through this, we wish to help software practitioners in building and maintaining ML software, which not only involves code changes but also changes to many other kinds of artifacts (Idowu et al. 2021) (e.g., dataset and models). Furthermore, training/education teams need an understanding of the types of code changes to better equip students and novice developers in supporting ML applications.

Approach In **RQ1**, we observed that 52% of the PRs sent by `Forks_with_changes` are merged into the parent ML research repository. In this research question, we qualitatively analyze 1) the types of changes that were merged with the parent repositories, which we call **upstream changes**; and 2) the types of changes that were performed within the forked repositories, but not pushed to the parent repository, which we call **downstream changes**. Using the sampling and coding approach discussed in Section 3.3, the coders of `Team A` and `Team B` validated and enhanced Hindle et al.'s (2008) taxonomy. This section reports on the new change (sub-)categories identified in the analyzed code changes of ML repositories.

Results **Hindle et al.'s (2008) change taxonomy was extended with two new categories and 16 new sub-categories of changes.** Only one of the two new high-level change categories was ML-specific, i.e., *Data*, while the other one, i.e., *Dependency Management*, represents an update to the original taxonomy related to modern library dependency management activities (which may have been less relevant 13 years ago). A graphical summary of the extended taxonomy of change (sub-)categories is provided in Fig. 6. In the subsections below, we briefly describe and illustrate each new (sub-)category and how it complements the existing taxonomy:

4.2.1 Maintenance

Code changes performing software maintenance activities. We identified four new maintenance change sub-categories in the context of machine learning.

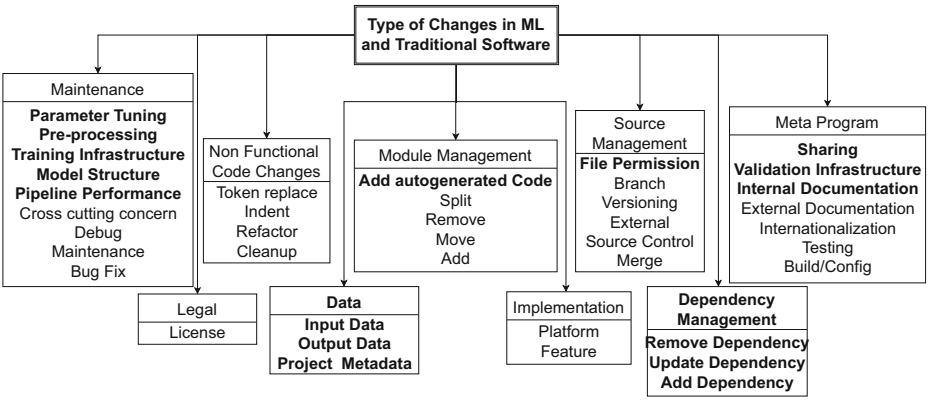

Fig. 6 Enhanced version of Hindle's change taxonomy. The bolded change (sub-)categories were identified by this study

– **Pre-processing**
Definition: Source code changes related to manipulation, cleaning, and filtering of data before feeding it to the model training or inference components of an ML pipeline.
Explanation: ML model training needs high-quality, clean data (Ng 2021) making data pre-processing a vital part of the ML pipeline (Amershi et al. 2019). Pre-processing changes internally mutate and clean the ingested data such that it can be consumed by the ML model, for example by changing the text-embeddings for NLP data. In contrast to the *Data category*, which deals with the ingestion or egestion of external data to/from the ML pipeline, pre-processing changes deal with internal data manipulation, and hence is a maintenance activity.
Notable Instances: In an image processing application (Pre-processing example 2018), pre-processing changes involve changing image color formats (Grayscale, RBG, BGR) before feeding the images to the ML model.

– **Parameter Tuning**
Definition: Changes made to hard-coded (hyper-)parameter values for tweaking the performance or functionality of an ML pipeline.
Explanation: A machine learning pipeline consists of many different phases (e.g., data preprocessing, feature extraction, model training, and validation) that, together, aim to generate models with the best fit possible. Each of these phases (Amershi et al. 2019) involves choosing values for various thresholds, model hyper-parameters, and other configuration options, many of which have hard-coded values. Hence, ML pipeline developers often find themselves tweaking these hard-coded values while building the model or preprocessing the data (i.e., parameter tuning). Even in the case of model hyper-parameters, which are optimized during the model training process, their initial (hard-coded) value or range often has to be chosen well for quicker convergence during training. In contrast to *model structural* changes, which change the model building code at a structural level, parameter changes are performed at the variable (value) level.
Notable Instances: The model hyper-parameter variable, *weights_regularizer* in project `youtube-8m` (Parameter tuning example 2017), was changed from $1e^{-5}$ to $1e^{-8}$. In another image processing application (Parameter tuning example 2 2017), adding

a sliding window variable for pre-processing of video frames is a parameter tuning change.

– **Model Structure**
Definition: Structural change to the source code responsible for training the machine learning model.
Explanation: Model structural changes involve changing the code encompassing the structure of deep learning models or the various functions or modules that manipulate the model during training, such as adding functions for dropout layers, loss functions, or regularizers for a model class. These changes are different from *parameter tuning* changes since they are at a structural level rather than the variable level.
Notable Instances: In file `train_val.py` of `tf-faster-rcnn` project[23], the model structure was changed to accommodate features related to im0age and mask height and width, which manifested in numerous structural changes to the model building process (Model structure example 2018).

– **Training Infrastructure**
Definition: Pipeline-level changes performed for training the model.
Explanation: Amershi et al. (2019) identified the canonical components of a typical ML pipeline, such as data wrangling, feature engineering, and model training. Making changes in one component (for instance, a different dataset schema) may manifest in other pipeline components (e.g., data pre-processing, feature engineering, model training, etc.). As such, training infrastructure changes correspond to any pipeline-level changes to the logic driving the ML model training phases. This is similar to how Hindle et al. (2008)'s original Build/Config change sub-category focuses on the logic driving the compilation (build) process, in contrast to changes to the actual source code (in our case: changes to, for example, the training scripts themselves).
Notable Instances: While adding a new demo in a semantic segmentation project (Training infrastructure example 2017), 12 files pertaining to the ML pipeline were changed. This was accompanied by a new training driver script[24] for the new demo data. Clearly, the model building pipeline had to be updated at multiple places to accommodate this new data.

– **Pipeline Performance**
Definition: Any change pertaining to the run-time efficiency of the ML pipeline.
Explanation: ML operations are computationally expensive and time-consuming. Iteratively retraining ML pipelines to find optimal values for model hyper-parameters and data cleaning configurations exacerbates performance needs. Hence, this sub-category of code changes involves any changes improving the run-time efficiency of the ML pipeline.
Notable Instances: Re-writing specific ML operations related to Principle Components Analysis (PCA) in Tensorflow in a project (Pipeline Performance example 2018) enabled higher computation efficiency. In particular, CPU utilization dropped from 5,600% to 240%.

[23] https://github.com/shikorab/tf-faster-rcnn
[24] https://github.com/TSchattschneider/PointCNN/commit/1827a79b2ede15007a06d327d95f10bc0753420

4.2.2 Meta Program

As identified by Hindle et al., Meta Program[25] changes update the metadata of the program (i.e., data required by the project, but not the source code). For instance, makefiles, and readme (*external-documentation*) files. We identified three new change sub-categories.

– **Sharing**
Definition: Changes in the way the source code of ML projects are presented or deployed to enable better collaboration between different roles involved in an ML project.
Explanation: In the modern collaborative development era, projects are shared with other developers and end users (Zhou et al. 2020). In the case of ML pipelines, such changes involve converting python scripts into Jupyter notebooks better suited for understanding and working with complex ML operations (Bloice and Holzinger 2016); or sharing the dependency environment via docker containers, enabling others to quickly deploy and run experiments on their infrastructure.
Notable Instances: A docker file was created for the project Neural-style (Sharing example 2016). Another project changed the demo jupyter notebooks files (Sharing example 2018) to disseminate the developed ML project and its parameters.

– **Validation Infrastructure**
Definition: Changes made to the ML model validation component of an ML pipeline (Amershi et al. 2019).
Explanation: Validation changes involve changes to any modules or components responsible for driving the evaluation of a trained ML model's (accuracy) performance, possibly comparing to the performance of prior trained models or earlier iterations of the trained model. This kind of change is similar to *Training Infrastructure* change, but focuses on the validation infrastructure instead of the training infrastructure.
Notable Instances: The file `evaluate3.py` was added in an image processing project (Validation example 2017) to evaluate the model's performance by comparing the predicted labels (annotations on images) against the true labels.

– **Internal documentation**
Definition: Changes that explain the internal workings of the ML code to developers.
Explanation: Internal documentation changes clarify the fine-grained implementation of the code, with developers and data scientists as the intended audience. Such changes not only add code comments but could also add log statements to the code, for example, a succession of `print` statements, to better comprehend the workings of ML pipeline operations. *Internal documentation* changes differ from *external documentation*, since the latter explicitly document a project for end-users, typically using README files or API documentation.
We introduce *internal documentation* as an augmentation to Hindle et al.'s (2008) *Documentation* sub-category, as they did not provide any distinction between internal and external documentation.
Notable Instances: In a FasterRCNN project (Internal documentation example-1 2017), ambiguous internal documentation about an internal flag variable using `DEFINE_bool`

[25] Note that Hindle et al.'s meta program change category is unrelated to the field of Metaprogramming (https://en.wikipedia.org/wiki/Metaprogramming).

was rectified. In another project (Internal documentation example-2 2017), the grammar of the comments that explain the internal working of the code was fixed.

4.2.3 Module Management

As identified by prior research, module management changes the way files are named and organized into source code modules. In addition to Hindle et al.'s sub-categories (i.e., *add*, *rename* and *delete* module), we identified a new change sub-category, *Adding auto-generated code*.

- **Adding auto-generated code**
  Definition: Adding new files to the project that are generated automatically by external tools, alternative IDEs, or varying environment configurations.
  Notable Instances: A commit involving 8,491 added lines of code and 3,813 deleted lines of code across three C files (Adding auto-generated files example 2018) corresponded to a re-generated C implementation of the Non-Maximum Suppression (NMS) algorithm, typically used for selecting the best bounding boxes of objects in an image. Since pure Python implementations of this algorithm are not scalable, data scientists tend to use the Cython dialect of Python, which allows generating efficient C code.

4.2.4 Data Category [NEW]

Any change to the infrastructure that handles ingestion/egestion of domain-specific data (e.g., for training or testing) required by an ML pipeline, or to the metadata of said data (e.g., directory paths). Note that this category does not involve committing actual data files, since Git repositories are not the right place to store large-scale data.

- **Input Data**
  Definition: Code changes to the logic responsible for loading data or ingesting external data into an ML pipeline.
  Explanation: ML pipelines need to deal with a variety of data storage platforms (e.g., CSV files, SQL database, Kafka, data lakes) to obtain domain-specific input data. Hence, this sub-category of changes relates to the logic of dealing with such data platforms and the data schemas of ingested data.
  Notable Instances: File `extract_tfrecords_main.py` in project, `Youtube-8m` (Input data example 2017), added functionality to load external video frames data and feed it in the right data format to the ML pipeline.

- **Output Data**
  Definition: Changing the way the output data of the ML program is stored.
  Explanation: Output data changes pertain to the way the results/output of the ML pipeline's are saved to the file system. Such changes may be needed to improve the integration of a model or its prediction results into an end-user application (e.g., UI applications or dashboards), or in other pipelines.
  Notable Instances: The faster-rcnn demo program was changed to save its output to an image file (Output data example 2018).

- **Project Metadata**
  Definition: Changing the metadata of all data files an ML project manages.

Explanation: ML pipelines contain a variety of metadata about the input and output data that they ingest/egest, such as paths of base directories or specific data files, license information of said data, etc. Hence, project metadata changes include adding, updating or deleting such metadata. This does not include changes to the actual data (*pre-processing*) or the infrastructure used to ingest/egest such data (*Input Data*/*Output Data*), only to the project metadata.
Notable Instances: Project directories for loading various model artifacts like model graphs and pretrained models were updated in a facenet implementation (Project data example 2017).

4.2.5 Source Management

Hindle et al. described *Source management* as changes performed due to the way a version control system is being used by a project. Along with the five sub-categories identified by Hindle et al. (2008), we identified one new sub-category.

– **Changing file permissions**
Definition: Changes adding, updating, or removing file permissions (like executability of a script).
Explanation: Traditional programs and ML operations are often run on shared high-performance computers (typically Unix-based servers). Managing file permissions is essential for assigning ownership of files while dealing with multiple users, thereby enforcing security.
Notable Instances: The file start.sh was given 775 permissions (Change file permission example 2017) since it's previous 664 permission did not allow the script to be executed by the owner of the file or its Unix user group.

4.2.6 Dependency Management [NEW]

While we identified this new category related to handling third-party dependencies (e.g., libraries or packages of a Linux distribution) on code changes of the studied ML pipelines, the management of such dependencies is common across both ML and non-ML projects (Decan et al. 2019; Pashchenko et al. 2020; Mukherjee et al. 2021).

– **Add Dependency**
Definition: Adopting a new third-party dependency in the source code.
Explanation: Adoption of a new third-party dependency typically requires adding the name and version of the dependency to a configuration file, as well as adding import statements to various files in the source code, in order to declare the dependency to compilers or interpreters.
Notable Instances: Addition of new import statements like
"`from tensorflow.python.lib.io import file_io`"
(Adding/removing dependency example 2019).

– **Remove Dependency**
Definition: Stopping the adoption of a third-party dependency.
Explanation: Removing an unused import statement from a source code file, or even removing the actual third-party dependency from the list of dependencies of a file.

Notable Instances: Removal of unused import statements (Adding/removing dependency example 2019).

– **Update Dependency**
*Definition:*After the adoption of a dependency, changes might be needed to the metadata of the dependency.
Explanation: This change category involves updating the metadata of a dependency, for example, to keep the dependency compatible with the code base, or vice versa. This typically includes updating the dependency version.
Notable Instances: Change of the `cloudml-gpu` runtime version from "1.0" to "1.8" (Update dependency example 2018).

4.2.7 Mapping the Updated Change Taxonomy to Amershi's ML Pipeline Architecture

In Fig. 7, we provide an association between the nine new ML-specific categories of code changes identified in this research to the ML pipeline architecture of Amershi et al. (2019). We notice that most (6) of the identified change categories apply to the *data cleaning* and *model training* phases, followed by the *feature engineering* (4) and *model evaluation* (3) phases. On the other hand, none of the change types map to the initial phases of *model requirements*, and *data collection*, since those involve tasks performed by management and data engineers, respectively. Similarly, the end phases, namely, *model deployment* and *model monitoring*, are geared towards third-party applications where the trained model is integrated and deployed by MLOps engineers into (amongst others) dashboards, UI applications, and back-end servers for prediction.

Summary of RQ2

Hindle et al.'s (2008) taxonomy of software code changes had to be extended with two high-level change categories (ML-specific *Data*, and generic *Dependency management*). We also extended the taxonomy by identifying 16 new sub-categories of changes, nine of which (i.e., *input data, parameter tuning, pre-processing, training infrastructure, model structure, pipeline performance, sharing, validation infrastructure, and output data*) are ML-specific.

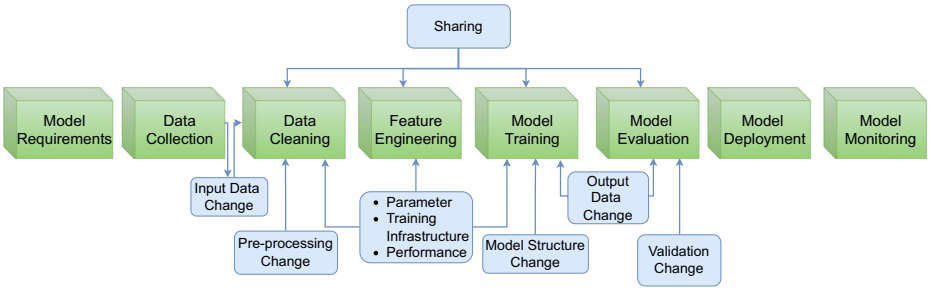

Fig. 7 ML-specific sub-categories mapped to Amershi et al.'s (2019) ML pipeline architecture

4.3 RQ3: How do Downstream Changes Differ From Upstream Changes in ML Research Repositories?

Motivation In **RQ1**, we observed that 41.6% of `Forks_with_changes` submit upstream contributions back to the ML research repositories. Since this means that contributions by more than half of the forks were never sent upstream, it is interesting to understand the nature of such contributions, i.e., what did the ML community need in addition to the original development in the parent repository (merged PRs), and what did the authors of the ML repository miss out on (i.e., code changes not contributed back)? In particular, the upstream changes studied in this paper help to identify the missing aspects of the original ML parent repository contributed back by the OSS community. Conversely, understanding the downstream changes studied in this paper helps us determine to what extent essential features or contributions have been missed.

Approach This RQ uses the sample of 378 downstream changes and 539 upstream changes labeled with high inter-rater agreements in **RQ2**, but this time to analyze the prevalence of each change sub-category of the taxonomy in Fig. 6. In particular, we compute the percentage of upstream and downstream changes for each (sub-)category, then compare our findings between downstream and upstream changes. These results are summarized in Fig. 8.

Results **For both upstream and downstream commits, heavy changes occur in *Documentation* (Meta program), *Bug fixes* and *Model training* (Maintenance); and adding new *features* (Implementation).** Figure 8 shows substantial peaks in the *Maintenance* and *Meta program* categories. We attribute such results to the nature of the data science life cycle, where ML pipelines require substantial *maintenance* activities during experimentation with and tweaking of data and models. ML tasks include multiple iterations of updating data *pre-processing,* tuning *parameters,* updating *model building* code, and improving *pipeline performance*.

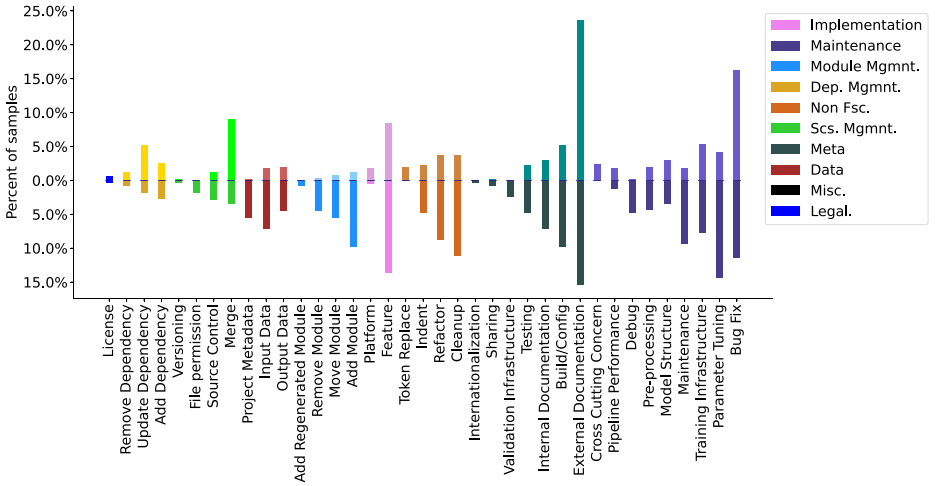

Fig. 8 Percentage of change sub-categories present in the 378 samples of Downstream commits and 539 samples of Upstream commits. Values above $y = 0.0\%$ (plotted using a lighter color palette) indicate the percentage of upstream changes containing a specific change sub-category, whereas values below $y = 0.0\%$ (darker color palette) indicate the same for downstream changes

Maintenance changes are much more prevalent in downstream commits than in upstream commits. This is visible through the higher percentages of *non functional* and *data* changes in downstream commits. We attribute this imbalance to downstream users adapting ML research for their domain-specific tasks, rather than improving the upstream repository for generic usage. In order to improve personal understanding of the ML code, downstream users added changes from the new change sub-category, *internal documentation*, such as added `print` statements. In contrast, we found no cases of *internal documentation* in upstream commits.

We also observed nine cases of our new sub-category *change evaluation* for downstream changes, while none for upstream commits. Intuitively, downstream developers needed scripts to train and test models (potentially after making some other changes) against their domain-specific data. Conversely, there were 45 cases of our new change sub-category, *Dependency Management*, in upstream commits. Such changes *add, update* or *remove* ML library dependencies. Finally, accepted PRs were merged into either a project's main branch or alternative branches, which led to more instances of *merging* changes (Source Management) in upstream commits than in downstream commits.

Noticeable Instances of the Most Popular Pre-AI Change Categories In the results of RQ3, we notice that three of the top four most occurring change types across upstream and downstream commits, i.e., Bug Fix, Documentation, and Feature, belong to Hindle's pre-AI taxonomy. Hence, here we provide some examples of those change sub-categories in the context of ML-based pipeline projects.

1. *External Documentation:* In the project *TensorBox*, the README.md file was updated to present information about how to manually download external dependencies (e.g., CUDA version) and how to set up and configure the corresponding runtime environment (External documentation example 2017). This was a downstream change.
2. *Bug Fix:* In the project *tf-faster-rcnn*, a bug in the `test_rpn` function was fixed in a downstream change as indicated by its commit message (Bug fix example 1 2019). In another project, *Kitti-Seg*, the order of height and width parameters was incorrectly swapped, as indicated by the commit message for the upstream change (Bug fix example 2 2017).
3. *Feature:* In a downstream commit for project *PointCNN* (Feature example 2018), new features were added allowing to set GPU flags and the CUDA path, and to create project directories for saving the model, if not yet existing.

Summary of RQ3

Both upstream and downstream contributions to ML research repository add *documentation*, *fix bugs* and *add features*. Downstream developers change *input/output data*, perform *parameter tuning*, add new functional *features*, and perform other non-functional changes like *indentation*, *refactoring*, or *cleaning up* the source code. Such changes are domain oriented. On the other hand, upstream changes benefit the parent repository by *updating dependencies* or *fixing bugs* for the parent repository.

5 Implications

In this section, we discuss the implications of our findings for software researchers, ML practitioners, the OSS ML community, toolsmith engineers, and ML educators.

5.1 Implications for Researchers

We extended Hindle et al.'s taxonomy of code changes (Hindle et al. 2008) with two new categories and 16 new sub-categories of changes. **Researchers can use our extended taxonomy to obtain a holistic picture of software changes in ML pipelines**. In particular, nine (*input data, parameter tuning, pre-processing, training infrastructure, model structure, pipeline performance, sharing, validation infrastructure, and output data*) of the ML-related categories of code changes indicate a need for revising and adapting existing best practices towards the needs of software engineering for ML systems. This is only exacerbated by the prevalence of the *internal* and *external documentation* change sub-categories, indicating difficulties for developers to comprehend complex ML code and keep track of hefty ML pipelines.

Our work also updates existing code-change dimensions towards modern SE paradigms in SE4AI. Even though some code change categories identified by Hindle et al. still apply in the context of ML pipelines, we were able to better understand their applicability within our context of ML pipelines. For instance, studying software bugs has been an important focus of the software engineering community for decades. With the advent of ML, recently many studies (Chen et al. 2022; Tizpaz-Niari et al. 2020; Cheng et al. 2018; Dwarakanath et al. 2018) started focusing on bugs in the machine learning domain. However, thus far the scope of these types of ML studies is limited to machine learning frameworks, while bugs in the different phases and components of actual ML pipelines or even end-user ML applications are not yet explored in depth. For example, initial studies found that the data wrangling phase introduces a variety of pipeline-level (Amershi et al. 2019) challenges, including pipeline-level bugs.

As another example, we split the "documentation" category of Hindle's change taxonomy into "internal" and "external" documentation, since our analysis of ML pipeline code changes made it especially apparent that both cater to a different audience in the modern SE paradigm of collaborative development. In particular, the (external) API-level or application-level documentation is aimed toward black-box (re)use of a given project, while the fined-grained (internal) documentation instead explains the inner working and state of processing of the code to people interested in changing, or at least better understanding it. Such a distinction may not have been that obvious 13 years ago (Hindle et al. 2008).

5.2 Implications for Toolsmiths

An updated taxonomy of code changes can help toolsmiths in adapting and innovating software engineering tools. As mentioned in Section 2, code change data is used for a variety of purposes such as extraction of missing traceability links (Wu et al. 2011), auto-generation of commit messages (Cortés-Coy et al. 2014), and analysis of quality impact (Faragó and Hegedũs 2014). At the same time, current development environments and tools used by developers need to be modernized as well.

Our observed instances of code changes spanning across the *Maintenance, Dependency Management, Source management*, and *Data* domains imply a need for toolsmiths to *better support ML engineering teams in handling requirements, managing data dependencies,*

configurations, training ML models. For example, given that many developers add comments to the code to better understand the ML logic, code bases might get polluted. The boom of Jupyter Notebooks (Granger and Pérez 2021) for data science only provides a workaround to this problem (Wang et al. 2020), which might not scale to real-life ML practices of large systems. Hence, perhaps less invasive annotation or other functionality is required in future IDEs.

As another example, code changes play an important role in tracking bugs (Zhao et al. 2017; Kim et al. 2011; Shivaji et al. 2012). An updated taxonomy of code changes may enable a more effective automated classification of bug reports or code changes submitted for code review. For example, most automated change classification techniques focus on a limited number of possible change types. Our results could also help fault localization and defect prediction researchers improve their models.

5.3 Implications for Educators

Software educators may wish to update their curricula to revise future training of (ML) software engineers. Moreover, novice software developers need training on practices related to *Dependency management*, to be better equipped to use and support ML frameworks. Overall, ML practitioners need to be aware of the change taxonomy to anticipate future changes that occur in ML software.

Particularly, Fig. 7 provides a map for educators of the different ML-related change (sub-)types to expect while providing a travel guide for education and training teams. Since ML-based organizations tend to have a distributed team with overlapping roles ranging from data developers, data scientists, statisticians, DevOps engineers, to software developers, software teams may wish to leverage such a map for a clearer understanding of the roles and responsibilities w.r.t. the nature of development changes performed by a specific role.

5.4 Implications for OSS Community

Organizations and/or individuals wishing to open-source their ML Repositories should have realistic assumptions. While our findings show that organizations and researchers do not necessarily "dump" their ML research implementations on GitHub, but receive and merge open-source contributions, this is not guaranteed. For one, only 9% of forks are `Forks_with_changes`, of which 41.6% send contributions upstream via a pull request, about half of which (52.1%) are accepted into the parent repository (see **RQ1**).

Two lessons can be learned from this. On the one hand, the ML research repositories are missing out on almost 60% of forks having contributions that are never sent back upstream. Even the 41.6% of forks that do contribute might not contribute back all contributions they have made. While it is OK for changes like *parameter tuning* not to be contributed back, the "lost" contributions of forks also include 16% of new bug fixes, 13% of new features for the ML pipeline, etc. Future work should look into why those were never sent back.

On the other hand, of those contributions that were propagated back, only half were merged. Future work should consider the reasons for rejection of this work, i.e., to what extent was rejection based on the quality of the contribution versus the contribution being too tied to the contributor's own use case, or even versus the responsiveness of the ML repository owners. Whichever the outcome, and similar to traditional open-source development, receiving many high-volume contributions requires effort (Rahman and Roy 2014). Researchers can gain leverage from our updated taxonomy to pay special attention to ML pipeline components that are updated while maintaining ML code.

6 Threats to Validity

Threats to Internal Validity Qualitative studies can be subject to researcher bias. To minimize this, we used multiple participants (i.e., two teams with four people in Team-A, and three people in Team-B) for the manual coding of changes. Both teams had in-depth knowledge of software development, as well as SE4ML. Furthermore, the teams pair-wise labeled each sample and achieved high inter-rater agreements of Krippendoff's $\alpha = 98\%$ for Team-A and $\alpha = 92\%$ for Team-B.

Threats to Construct Validity In Section 3, we mine ML pipeline repositories implementing algorithms published in ArXiv papers. For this, we check whether a repository cites an ArXiv research paper in its README file. Analysis of a sample of 94 such repositories in Section 1 showed that all repositories citing an ML ArXiv publication are influenced by the research and can be termed as a "ML research repository", irrespective of whether the repository is created by the authors of the ML research publication (30%) or by external members of the community (70%).

Threats to External Validity For answering our **RQ2**, we analyzed both the types of changes made by PRs merged into the upstream parent repository and by downstream changes performed within the forks but never sent upstream. However, we do not study changes rejected by the upstream repository. While future work should analyze such cases, we feel confident about the completeness of our taxonomy, as we reached saturation in obtaining new labels within the initial 78 samples of downstream commits. No new categories were found in the later part of 300 downstream or any of the 539 upstream commits.

Moreover, we focused on the repositories implementing image processing or machine learning in ModelDepot, since they were the most popular on Modeldepot and cover a wide range of popular ML application domains. Future work should focus on other domains like NLP and Audio Processing.

Finally, we sampled our data for qualitative analysis only from the 23 repositories that were at the top five percentile of *Forks_with_changes*. We put such a filter to select repositories with the maximum amount of activity in terms of downstream commits and pull requests which may thereby manifest as upstream commits. While we need "popular or active trends" to study rich and meaningful data that has low noise, nonetheless, this poses a threat to external validity as is also indicated by prior research (Kalliamvakou et al. 2014; Santos et al. 2015; Bird et al. 2009).

Threats to Reliability Validity These threats take into account the replication of our study. After our data collection process from ModelDepot finished and the analysis was well underway, the website was shut down. However, ModelDepot only pointed to the repositories hosted on GitHub. To mitigate this threat, we provide[26] our lists of 1,346 repositories, along with the ArXiv papers cited by these repositories. We also provide a snapshot of the mined forking data at the time of our analysis for our quantitative investigation of **RQ1**. The replication data for qualitative analysis of **RQ2, RQ3** consists of the labeled sample of upstream and downstream changes to ensure the reproducibility of our study.

[26] https://github.com/SAILResearch/suppmaterial-22-aaditya-ml_change_taxonomy

7 Conclusion

Open-source community developers use and refine OSS repositories. Although prior studies have investigated the nature of open source contributions in non-ML software, one can imagine the nature of such community changes, as well as the way in which developers collaborate, to be different for Machine Learning projects. Hence, this paper studies the forking dynamics and the types of changes performed in 67,369 forks of 1,346 ML pipeline projects related to research publications.

We found that, while most forks (91%) do not modify an ML research repository after forking it, 41.6% of the forks with modifications contribute valuable changes to the parent ML research repository, with a 52.1% acceptance rate. We performed an extensive qualitative study that identified the types of changes in ML software. We identified one new top-level change category, *Data*, in the context of ML, and one more generic category (*Dependency management*). Along with this, we extend the taxonomy of changes by adding 15 new sub-categories, including nine ML-specific ones (*input data, output data, program data, sharing, change evaluation, parameter tuning, pipeline performance, pre-processing, model training*) and seven generic ones (i.e. *adding dependency, removing dependency, updating dependency, file permissions, internal-documentation, adding auto-generated code* and *project metadata*).

Our results aim to help software practitioners in having a better understanding of ML changes that can be leveraged while training new developers, and to support building and maintaining ML software. Furthermore, future work should look deeper into the reasons why potentially valid *Documentation*, *Feature* and *Bug fix* changes were not contributed back upstream.

Acknowledgements We thank Greg Wilson for providing insightful ideas and comments for this work. We also thank Boyuan Chen, Minke Xiu, Javier Rosales and Wanqing Li for their contributions to the analysis and feedback on this work.

Funding This research is partially supported by the NSERC grants RGPIN-2019-06014 and RGPAS-2019-00075.

Data Availability All the scripts along with the mined data are provided in the replication package[27]

Declarations

Ethics approval No ethics approval was required for this paper.

Conflict of interests/Competing interests The authors have no relevant financial or non-financial interests to disclose.

References

Adding auto-generated files example (2018). https://github.com/alorozco53/text-detection-ctpn/commit/f90326f68522f3af3e4cdf5688138685de66bace

Adding/removing dependency example (2019). https://github.com/google/youtube-8m/commit/09774db80a515b667a91b14fe21a6134f3856c7a

[27] https://github.com/SAILResearch/suppmaterial-22-aaditya-ml_change_taxonomy

Amershi S, Begel A, Bird C, DeLine R, Gall H, Kamar E, Nagappan N, Nushi B, Zimmermann T (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), pp 291–300

Arpteg A, Brinne B, Crnkovic-Friis L, Bosch J (2018) Software engineering challenges of deep learning. In: 2018 44th Euromicro conference on software engineering and advanced applications (SEAA). IEEE, pp 50–59

Benestad HC, Anda B, Arisholm E (2009) Understanding software maintenance and evolution by analyzing individual changes: a literature review. J Softw Maint Evol Res Pract 21(6):349–378

Biazzini M, Baudry B (2014) may the fork be with you: novel metrics to analyze collaboration on github. In: Proceedings of the 5th international workshop on emerging trends in software metrics, pp 37–43

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced? bias in bug-fix datasets. In: Proceedings of the 7th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp 121–130

Bissyandé TF, Thung F, Wang S, Lo D, Jiang L, Ré veillère L (2013) Empirical evaluation of bug linking. In: 2013 17th European conference on software maintenance and Reengineering, pp 89–98

Bloice MD, Holzinger A (2016) A tutorial on machine learning and data science tools with python. Machine Learning for Health Informatics, pp 435–480

Borges H, Valente MT (2018) What's in a github star? understanding repository starring practices in a social coding platform. J Syst Softw 146:112–129

Brisson S, Noei E, Lyons K (2020) We are family: analyzing communication in github software repositories and their forks. In: 2020 IEEE 27th international conference on software analysis Evolution and Reengineering (SANER). IEEE, pp 59–69

Bug fix example 1 (2019). https://github.com/piaosonglin1985/tf-faster-rcnn/commit/8e60b9dc92390f1bfb8cf6e62d93bcabbc123c4a

Bug fix example 2 (2017) https://github.com/MarvinTeichmann/KittiSeg/commit/ec6b5ccb6f30ac6591d03fa a2fa0bf8b1fdbf3ef

Change file permission example (2017). https://api.github.com/repos/CodeRecipeJYP/fast-style-transfer/commits/7027a3843fa3d793697da5ba188887629a4d69eb

Chen Z, Zhang JM, Sarro F, Harman M (2022) Maat: a novel ensemble approach to addressing fairness and performance bugs for 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). ACM Press

Cheng D, Cao C, Xu C, Ma X (2018) Manifesting bugs in machine learning code: An explorative study with mutation testing. In: 2018 IEEE international conference on software quality, reliability and security (QRS). IEEE, pp 313–324

Constantino K, Zhou S, Souza M, Figueiredo E, Kästner C (2020) Understanding collaborative software development: an interview study. In: Proceedings of the 15th international conference on global software engineering, pp 55–65

Cortés-Coy LF, Linares-Vásquez M, Aponte J, Poshyvanyk D (2014) On automatically generating commit messages via summarization of source code changes. In: 2014 IEEE 14th international working conference on source code analysis and manipulation, pp 275–284

Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir Softw Eng 24(1):381–416

Dey T, Mockus A (2020) Which pull requests get accepted and why? a study of popular npm packages, arXiv:2003.01153

Dwarakanath A, Ahuja M, Sikand S, Rao RM, Bose RJC, Dubash N, Podder S (2018) Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 118–128

External documentation example (2017). https://github.com/Raochuan89/TensorBox/commit/aeb45e8fdc100f74aa8cf2fa85b1324483a1fff1

Fan Y, Xia X, Lo D, Hassan AE, Li S (2021) What makes a popular academic AI repository? Empir Softw Eng 26(1):1–35

Faragó C, Hegedűs P (2014) R Ferenc, The impact of version control operations on the quality change of the source code. In: International conference on computational science and its applications. Springer, pp 353–369

Feature example (2018). https://github.com/tch/PointCNN/commit/891f3e04b44805b066865aeef1275ac6f217c58f

Fogel K (2005) Producing open source software: How to run a successful free software project. O'Reilly Media, Inc.,

German DM, Adams B, Hassan AE (2016) Continuously mining distributed version control systems: an empirical study of how linux uses git. Empir. Softw. Eng. 21(1):260–299
Ghadhab L, Jenhani I, Mkaouer MW, Messaoud MB (2021) Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model, vol 135
Gousios G, Pinzger M, Deursen AV (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering, pp 345–355
Granger B, Pérez F (2021) Jupyter: thinking and storytelling with code and data Authorea Preprints
Hindle A, German DM, Godfrey MW, Holt RC (2009) Automatic classication of large changes into maintenance categories. In: 2009 IEEE 17th International Conference on Program Comprehension. IEEE, pp 30–39
Hindle D, German M, Holt R (2008) What do large commits tell us? a taxonomical study of large commits. In: Proceedings of the 2008 international working conference on mining software repositories, ser. MSR '08. New York, NY, USA: association for computing machinery, pp 99–108. [Online]. Available:. https://doi.org/10.1145/1370750.1370773
Hu Y, Zhang J, Bai X, Yu S, Yang Z (2016) Influence analysis of github repositories. SpringerPlus 5(1):1–19
Idowu S, Strüber D, Berger T (2021) Asset management in machine learning: a survey. In: 2021 IEEE/ACM 43rd international conference on software engineering: software engineering in practice (ICSE-SEIP), pp 51–60
Input data example (2017). https://github.com/google/youtube-8m/commit/4619056162f466293d99e0c595112f8d0f3427fe2
Internal documentation example-1 (2017). https://github.com/google/youtube-8m/commit/3439e33d81df8cd906987ee5889ebc937186114a
Internal documentation example-2 (2017). https://github.com/CharlesShang/FastMaskRCNN/commit/0d8ddfaa55dbd3d553b79aed34f40662c46aa45f
Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th working conference on mining software repositories, ser. MSR 2014. New York, NY, USA: association for computing machinery, pp 92–101. [Online]. Available: https://doi.org/10.1145/2597073.2597074
Kim M, Cai D, Kim S (2011) An empirical investigation into the role of api-level refactorings during software evolution. In: Proceedings of the 33rd international conference on software engineering, pp 151–160
Krippendorff K (2011) Computing krippendorff's alpha-reliability
Li H, Shang W, Adams B, Sayagh M, Hassan AE (2020) A qualitative study of the benefits and costs of logging from developers' perspectives. IEEE Transactions on Software Engineering
Lima A, Rossi L, Musolesi M (2014) Coding together at scale: Github as a collaborative social network. In: Eighth international AAAI conference on weblogs and social media
Martínez-Fernández S, Bogner J, Franch X, Oriol M, Siebert J, Trendowicz A, Vollmer AM, Wagner S (2021) Software engineering for ai-based systems, a survey, arXiv:2105.01984
Model structure example (2018). https://github.com/shikorab/tf-faster-rcnn/commit/327778b2c4f297b307ff0de552d2bfc47278e290
Mukherjee S, Almanza A, Rubio-González C (2021) Fixing dependency errors for python build reproducibility. In: Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis, pp 439–451
Nahar N, Zhou S, Lewis G, Kästner C (2022) Collaboration challenges in building ml-enabled systems: communication, documentation, engineering, and process. In: 2022 IEEE/ACM 44th international conference on software engineering (ICSE)
Ng A (2021) Mlops: from model-centric to data-centric ai
O'Leary K, Uchida M (2020) Common problems with creating machine learning pipelines from existing code
Output data example (2018). https://github.com/Mappy/tf-faster-rcnn/commit/51e0889fbdcd4c48f31def4c1cb05a5a4db04671
Ozkaya I (2020) What is really different in engineering ai-enabled systems? IEEE Softw 37(4):3–6
Parameter tuning example (2017). https://github.com/google/youtube-8m/commit/0e526caace96d3cf6f068f9ffba998b4
Parameter tuning example 2 (2017). https://github.com/DeepLabCut/DeepLabCut/commit/6568c2ba6facf5d90b2c39af7b0f024a40f2b15f
Pashchenko I, Vu D-L, Massacci F (2020) A qualitative study of dependency management and its security implications. In: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security, pp 1513–1531
Pipeline Performance example (2018). https://github.com/google/youtube-8m/pull/69

Polyzotis N, Roy S, Whang SE, Zinkevich M (2018) Data lifecycle challenges in production machine learning: a survey. ACM SIGMOD Rec 47(2):17–28
Pre-processing example (2018). https://github.com/lancele/Semantic-Segmentation-Suite/commit/d50b5c812392614fc2bdaf269921beb1f7086f63
Project data example (2017). https://github.com/Bruceeeee/facenet/commit/d9e6213cd8286334000ddf75529eba3662cef38a#diff-dbc5c3b9f46e69236207956b34904d0dea62ff866d442e97bb397ff49a03a86b
Rahman MM, Roy CK (2014) An insight into the pull requests of github. In: Proceedings of the 11th working conference on mining software repositories, pp 364–367
Ren L, Zhou S, Kä stner C (2018) Poster: forks insight: Providing an overview of github forks. In: 2018 IEEE/ACM 40th international conference on software engineering: companion (ICSE-Companion), pp 179–180
Salza P, Palomba F, Di Nucci D, D'Uva C, De Lucia A, Ferrucci F (2018) Do developers update third-party libraries in mobile apps? In: Proceedings of the 26th conference on program comprehension, pp 255–265
Sambasivan N, Kapania S, Highfill H, Akrong D, Paritosh P, Aroyo LM (2021) Everyone wants to do the model work, not the data work: data cascades in high-stakes ai. In: Proceedings of the 2021 CHI conference on human factors in computing systems, pp 1–15
Santos JAM, Santos AR, Mendonç a MG (2015) Investigating bias in the search phase of software engineering secondary studies. In: CIbSE, pp 488
Sato D, Wider A, Windheuser C (2019) Continuous delivery for machine learning. https://martinfowler.com/articles/cd4ml.html#DeploymentPipelines
Sharing example (2016). https://github.com/anishathalye/neural-style/pull/40
Sharing example (2018). https://github.com/jerichooconnell/tf_unet/commit/60b67bb964d19dd4a4677f7557dc738838a116e9
Shivaji S, Whitehead EJ, Akella R, Kim S (2012) Reducing features to improve code change-based bug prediction. IEEE Trans Softw Eng 39(4):552–569
Swanson EB (1976) The dimensions of maintenance. In: Proceedings of the 2nd international conference on Software engineering, pp 492–497
Tizpaz-Niari S, Černỳ P, Trivedi A (2020) Detecting and understanding real-world differential performance bugs in machine learning libraries. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp 189–199
Training infrastructure example (2017). https://github.com/IAC-Team/SemSeg/commit/efbfffbd202cccbd54fca1125ed6de41b5df2f90
Update dependency example (2018). https://github.com/google/youtube-8m/commit/72f42cd938d3cf4f928614a5fcdca237489e7c92
Validation example (2017). https://github.com/bethesirius/TensorBox/commit/1eb41e944494e721f3c4b1a5d287af99f4035a42
Wang J, Li L, Zeller A (2020) Better code, better sharing: on the need of analyzing jupyter notebooks. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering: new ideas and emerging results, pp 53–56
Washizaki H, Uchida H, Khomh F, Gué héneuc Y-G (2019) Studying software engineering patterns for designing machine learning systems. In: 2019 10th International workshop on empirical software engineering in practice (IWESEP). IEEE, pp 49–495
Wu R, Zhang H, Kim S, Cheung S-C (2011) Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th european conference on foundations of software engineering, pp 15–25
Yan M, Fu Y, Zhang X, Yang D, Xu L, Kymer JD (2016) Automatically classifying software changes via discriminative topic model: supporting multi-category and cross-project. J Syst Softw 113:296–308
Zhang X, Chen Y, Gu Y, Zou W, Xie X, Jia X, Xuan J (2018) How do multiple pull requests change the same code: a study of competing pull requests in github. In: 2018 IEEE international conference on software maintenance and evolution (ICSME).IEEE, pp 228–239
Zhang T, Gao C, Ma L, Lyu M, Kim M (2019) An empirical study of common challenges in developing deep learning applications. In: 2019 IEEE 30th international symposium on software reliability engineering (ISSRE). IEEE, pp 104–115
Zhao Y, Leung H, Yang Y, Zhou Y, Xu B (2017) Towards an understanding of change types in bug fixing code. Inf Softw Technol 86:37–53
Zhou S, Vasilescu B, Kä stner C (2020) How has forking changed in the last 20 years? a study of hard forks on github. In: 2020 IEEE/ACM 42nd international conference on software engineering (ICSE). IEEE, pp 445–456
Zhou S, Vasilescu B, Kastner C (2019) What the fork: a study of inefficient and efficient forking practices in social coding. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 350–361

Aaditya Bhatia is currently a Ph.D. student in the Software Analytics and Intelligence Lab (SAIL) at School of Computing, Queen's University. His research interests include AI pipeline software analytics, management and understanding of AI quality issues.

Ellis E. Eghan holds a Ph.D. in Computer Science from Concordia University (Montreal, Canada) and is currently a Lecturer at the University of Cape Coast (UCC), where he teaches courses in software engineering, artificial intelligence, data science, and research methods. His main research interests revolve around software quality, semantic web technologies, knowledge graphs and explainable AI (XAI), and software dependency management. His current research focuses on using semantic modeling and analysis of data to support a holistic and explainable assessment of software quality and trustworthiness. His work has been published in top software engineering conferences and journals such as JSS, IEEE Transactions on Reliability, ICSME, and ICSE. He has also served as a reviewer in several reputable peer-reviewed journals and conferences (e.g., JSS, IST journal).

Manel Grichi is currently a lead data science at Vibrosys. She was a Postdoctoral Fellow at the Computer Research Institute of Montreal. She received her Ph.D. in 2020 from the Department of Computer Science and Software Engineering at Polytechnique Montreal, Quebec, Canada. She also has a bachelor's degree (Licence degree) and a software engineering's degree from the University of Tunisia in 2011 and 2014 respectively.

William G. Cavanagh graduated in Software Engineering and is currently pursuing a Master's degree in ML at the University of Montréal. He is passionate about clean code, data processes and statistical methods for learning. More information at: https://williamglazer.github.io/website/.

Zhen Ming (Jack) Jiang is an associate professor at the Department of Electrical Engineering and Computer Science, York University in Toronto, Canada. His research interests lie within software engineering and computer systems, with special interests in software performance engineering, software analytics, source code analysis, software architectural recovery, software visualizations, and debugging and monitoring of distributed systems. Some of his research results are already adopted and used in practice daily. He is the cofounder of the annually held International Workshop on Load Testing and Benchmarking of Software Systems (LTB). He received several Best Paper Awards including ICST 2016, ICSE 2015 (SEIP track), ICSE 2013, and WCRE 2011. He is the receipt of 2020 NSERC Discovery Accelerator Supplements (DAS) and 2021 CSCAN/InfoCAN Outstanding Early Career Computer Science Researcher Award. More information at: http://www.cse.yorku.ca/~zmjiang.

Bram Adams is a full professor at Queen's University (until June 2020: Polytechnique Montreal). He obtained his PhD in 2008 at Ghent University's GH-SEL lab (Belgium). His research interests include software release engineering, mining software repositories, and the role of human affect in software engineering. His work has been published at premier software engineering venues such as ICSE, FSE, MSR, ICSME, EMSE and TSE, and received the 2021 Mining Software Repositories Foundational Contribution Award. In addition to co-organizing the RELENG International Workshop on Release Engineering from 2013 to 2015 (and the 1st/2nd IEEE Software Special Issue on Release Engineering), he co-organized the SEMLA, SoHEAL, PLATE, ACP4IS, MUD and MISS workshops, and the MSR Vision 2020 Summer School. He has been PC co-chair of SCAM 2013, SANER 2015, ICSME 2016 and MSR 2019, and ICSE 2023 software analytics area co-chair. He is a Senior IEEE Member.

Affiliations

Aaditya Bhatia[1] · Ellis E. Eghan[2] · Manel Grichi[3] · William G. Cavanagh[4] · Zhen Ming (Jack) Jiang[5] · Bram Adams[1]

Ellis E. Eghan
elliseghan@gmail.com

Manel Grichi
grichimanel@gmail.com

William G. Cavanagh
william.glazer-cavanagh@polymtl.ca

Zhen Ming (Jack) Jiang
zmjiang@cse.yorku.ca

Bram Adams
bram.adams@queensu.ca

1 Queen's University, Kingston, ON, Canada
2 University of Cape Coast, Cape Coast, Ghana
3 VibroSyst Inc., Montreal, Canada
4 Polytechnique Montreal, Montreal, Canada
5 York University, Toronto, Canada