Understanding Software-2.0: A Study of Machine Learning library usage and evolution

MALINDA DILHARA, AMEYA KETKAR, and DANNY DIG, Oregon State University and University of Colorado

Enabled by a rich ecosystem of Machine Learning (ML) libraries, programming using *learned models*, i.e., Software-2.0, has gained substantial adoption. However, we do not know what challenges developers encounter when they use ML libraries. With this knowledge gap, researchers miss opportunities to contribute to new research directions, tool builders do not invest resources where automation is most needed, library designers cannot make informed decisions when releasing ML library versions, and developers fail to use common practices when using ML libraries.

We present the first large-scale quantitative and qualitative empirical study to shed light on how developers in Software-2.0 use ML libraries, and how this evolution affects their code. Particularly, using static analysis we perform a longitudinal study of 3,340 top-rated open-source projects with 46,110 contributors. To further understand the challenges of ML library evolution, we survey 109 developers who introduce and evolve ML libraries. Using this rich dataset we reveal several novel findings.

Among others, we found an increasing trend of using ML libraries: the ratio of new Python projects that use ML libraries increased from 2% in 2013 to 50% in 2018. We identify several usage patterns including: (i) 36% of the projects use multiple ML libraries to implement various stages of the ML workflows, (ii) developers update ML libraries more often than the *traditional libraries*, (iii) strict upgrades are the most popular for ML libraries among other update kinds, (iv) ML library updates often result in cascading library updates, and (v) ML libraries are often downgraded (22.04% of cases). We also observed unique challenges when evolving and maintaining Software-2.0 such as (i) binary incompatibility of trained ML models, and (ii) benchmarking ML models. Finally, we present actionable implications of our findings for researchers, tool builders, developers, educators, library vendors, and hardware vendors.

CCS Concepts: • **Software and its engineering** → *Software development techniques*; • **Computing methodologies** → *Machine learning*.

Additional Key Words and Phrases: Machine learning libraries, Empirical Studies, Software-2.0

1 INTRODUCTION

Software 2.0 was first coined by Karpathy [80] in 2017. Later on, in his inspirational keynote at FSE'2018, Erik Meijer [96] characterizes two radically different ways for developing software: *"**In Software 1.0**: Engineers formally specify their problems, carefully design algorithms, compose systems out of subsystems or decompose complex systems into smaller components. **In Software 2.0**: Engineers amass training data and simply feed it into an ML algorithm that will synthesize an approximation of the function whose partial extensional definition is that training data. Instead of code as the artifact of interest, in Software 2.0 it is all about the data where compilation of source code is replaced by training models with data".* In addition to functions that are synthesized by ML algorithms, Software-2.0 often contains many other functions which developers implement using Software-1.0 paradigm.

Software-2.0 has gained substantial popularity over the past years. Programming using learned models has completely reinvented domains such as self-driving cars [132], drug discovery [26], automatic language translation [102], financial fraud detection [119], malware detection [24], and even software engineering tools [57, 79, 87]. Similarly to how the availability of a rich ecosystem of reusable libraries has contributed to huge productivity gains during Software-1.0, the availability

Authors' address: Malinda Dilhara, malinda.malwala@colorado.edu; Ameya Ketkar, ketkara@oregonstate.edu; Danny Dig, danny.dig@colorado.edu, Oregon State University, Corvallis, OR, 97333, University of Colorado, Boulder, CO, 80301.

of libraries such as *TensorFlow*, *Scikit-Learn* and *Keras* is a key component [21, 108] for the growth of Software-2.0. Companies like Google, Facebook, Microsoft, and Amazon besides releasing and promoting their own ML frameworks, actively invest in open source ML libraries [21]. The availability of reusable ML libraries makes it easy to leverage state-of-the-art ML algorithms with minimal effort and therefore promotes the use of ML in software applications. Nowadays, leveraging ML libraries, developers can develop Software-2.0 applications from scratch, or they can transform Software-1.0 applications to Software-2.0 by retrofitting ML libraries.

Despite having lots of similarities with Software-1.0, Software-2.0 inherits additional challenges from ML libraries - *(i) Data-dependence:* ML algorithms infer rules that govern the behavior of systems from training data and produce ML models that comprise the learned rules. The accuracy of the decisions made by ML models depends on the training data [20]. *(ii) Dependence upon Pre-trained model:* Software-2.0 run trained ML models to make the decisions in production environments (e.g., identify an email as spam or not spam) [1, 108]. *(iii) Rapid evolution:* researchers discover new ML algorithms and continue to improve existing ML algorithms. Therefore, ML libraries that implement these algorithms change rapidly releasing new versions very frequently. *(iv) Requirement for optimized hardware:* ML libraries usually require optimised hardware such as GPU and TPU to efficiently train ML models. Due to these unique characteristics, the use of ML libraries deserves special considerations from researchers.

A substantial number of software engineering researchers extensively studied *traditional libraries* used in Software-1.0 [15, 16, 30, 35, 37, 38, 40, 47, 75, 129]. In contrast, there are very few studies about ML libraries used in Software-2.0. Islam et al. [69, 70] study StackOverflow posts and GitHub bug reports to identify ML-related bugs. Humbatova et al. [62] study GitHub issues and StackOverflow posts to create a taxonomy of bugs for Software-2.0 while Han et al. [56] study the same subject systems to discover the trends of discussion topics in Software-2.0. However, none of these answer fundamental questions about usage, evolution, challenges, practices, and tool support for using ML libraries in Software-2.0.

Despite some folkloric evidence about the widespread use of ML libraries, there is no systematic study about the use of ML libraries in Software-2.0. *How* do developers maintain and evolve ML libraries in Software-2.0? *What* challenges do developers face when using ML libraries? *What* are the tools the developers are currently using to evolve Software-2.0? *What* are the new opportunities to better assist Software-2.0 developers? We have very little quantitative and qualitative knowledge to answer these questions. This lack of knowledge negatively impacts five audiences:

(1) *Researchers* are unaware of the research gaps (i.e., the actual unsolved problems faced by the Software-2.0 developers) that arise with the special attributes of ML libraries, and thus miss opportunities to improve the current state of the art for Software-2.0.
(2) *Library vendors* are unaware of the challenges that their clients face and whether the clients use the library according to their design goals.
(3) *Hardware vendors* are left in the dark without knowing what to optimise on accelerators.
(4) *Tool builders* do not know how to tailor their tools to the actual needs and practices of the developers when maintaining and evolving ML libraries.
(5) *Developers* are not aware of the good and bad practices related to the use of ML libraries.

To fill in these crucial gaps, we employ complementary established research methodologies. In this work, we present the first large-scale static analysis on 3,340 open-source ML library client projects hosted on GitHub, containing 809,534 ML library constructs. Moreover, we study the evolution history of these projects longitudinally to discover trends in evolving ML libraries. Furthermore, to better understand the challenges that developers face when retrofitting ML libraries or adapting their code because of ML library evolution, we survey 109 avid users of ML libraries. Using this rich information, we answer these novel research questions:

RQ1: *What* is the trend in ML library usage? We found that between 2013 and 2018 the ratio of new Software-2.0 projects to other new projects in GitHub each year has increased from 1.75% to 49.63%. In the same period, the ratio of new ML library forks to all new forks added in GitHub each year has increased from 0.41% to 3.21%.

RQ2: *What* combinations of libraries do developers use? Among many of our findings, we observed that 40.10% of the Software-2.0 projects use at least two ML libraries to implement various stages of the ML development workflow.

RQ3: *How* do developers update ML library dependencies? Among others, we found that developers update versions of ML libraries more often than the other libraries used in the projects. ML library downgrades are more frequent compared to the other libraries in the projects, and 41.52% of ML library updates trigger cascading library updates.

RQ4: *What* challenges arise when updating ML libraries? We surveyed developers about the challenges of evolving Software-2.0 and found 7 common challenges, including binary incompatibility of trained ML models, re-selecting decision thresholds, benchmarking ML models, and supporting multiple versions.

RQ5: *What* help do developers seek for updating ML libraries? Our survey showed that 26.66% of the ML library updates have taken more than a week to complete the update process. Further, developers mostly use the tools that are designed for Software-1.0 when updating ML library versions while preferring *six* different kinds of new tool support to evolve Software-2.0.

RQ6: *What* challenges arise when retrofitting ML libraries to Software-1.0? We surveyed developers about why they retrofit ML libraries and what challenges they encountered. We found five reasons and eight challenges. Among others, we identified reasons such as augmenting functionalities with ML and replacing existing non-ML techniques with ML techniques. We identified challenges such as gathering data, managing data-pipeline, and adding ML to edge devices.

Based on these findings, we highlight several actionable implications: new tool building opportunities to better assist Software-2.0, opportunities to improve existing infrastructure according to the requirements of Software-2.0, common practices of maintaining and evolving Software-2.0, and blind spots in current Software-2.0 research.

This paper makes the following contributions:

(1) To the best of our knowledge, this is the first large-scale empirical study answering questions about the usage and evolution of Software-2.0 through analyzing the source code and its evolution.
(2) We propose a dependency update model for Python libraries that can be used to identify different kinds of Python library updates.
(3) We designed and conducted three surveys with 109 avid users of ML libraries to identify the challenges that developers face when maintaining and evolving Software-2.0.
(4) We make the collected data publicly available for further research and reuse [91].
(5) We present an empirically-justified set of implications of our findings from the perspective of five audiences: researchers, tool builders, ML library vendors, hardware vendors, and ML developers.

2 BACKGROUND

In this section, we discuss the impact of Python upon machine learning, package management system in Python, and the nine stages of the machine learning development workflow.

Lingua Franca for ML: Python is an interpreted, high-level, general-purpose programming language. It unites the power of general-purpose programming with the ease of use of domain-specific scripting languages. Python has become the *lingua franca* for machine learning [21] (the fastest growing field in computer science).

ML Libraries: ML comprises several kinds of learning algorithms such as *supervised learning*, *unsupervised learning*, *deep learning*, and *reinforcement learning*. To provide fast access to these algorithms, the open-source community has developed many open source Python ML libraries such as *Scikit-Learn* [108], *TensorFlow* [1], *Theano* [4], *Caffe* [73], *Keras* [27], and *PyTorch* [29]. These libraries provide ready-made algorithms that allow developers to easily apply ML-based solutions to business objectives. However, using these libraries pose unique challenges due to dependence upon data and pre-trained models, rapid evolution and a need for optimized hardware.

In addition to ML libraries, developers often use libraries that provide optimized numerical computations (e.g. `Numpy`), efficient data structures (e.g. `Pandas`), visualisation (e.g., `Matplotlib`), or other utilities. In the rest of the paper we refer to these libraries as ***traditional libraries***.

Because ML libraries bring unique challenges to Software-2.0, in the rest of the paper we focus on their usage and compare some of our results with the *traditional libraries*.

Package-management system: A package manager is a software tool that automates the process of installing, upgrading, and removing libraries. For example, Java developers maintain a `pom.xml` for using Maven [48] and Python developers maintain a `requirements.txt` [110] file to configure the *PIP* or *Conda* package management systems [113]. Python development tools (e.g., `Pycharm` IDE [72], CLI tools) create a runtime environment for executing Python programs based on this `requirement.txt` file. PEP [111] suggests a standard syntax for the `requirements.txt` to the Python community. An entry (e.g., `scikit-learn >= 1.0.4`) in the *requirements.txt* is composed of a library name (e.g., `scikit-learn`) and an optional version specifier. The version specifier (e.g., `>= 1.0.4`) is composed of a version clause (e.g., `>=`) and an identifier (e.g., `1.0.4`).

Machine learning workflow: Amershi et al. [8] describe nine stages of ML development workflow and group these stages into data and model oriented categories. The **data-oriented** stages are: (i) *Model requirements* (identifying the appropriate ML models and suitable features), (ii) *Data collection* (collecting datasets from available data pools or generating a new dataset), (iii) *Data cleaning* (removing inaccurate or noisy records from the dataset), (iv) *Data labeling* (assigning ground truth labels to each data record), and (v) *Feature engineering* (identifying informative features of a dataset for the model). The **model-oriented** stages are: (i) *Model training* (identifying the model and training it with the labeled dataset), (ii) *Model evaluation* (evaluating the trained model), (iii) *Model deployment* (deploying the model in software solutions), and (iv) *Model monitoring* (monitoring the model for possible errors).

3 RESEARCH-METHODOLOGY

In this study, we employ both quantitative and qualitative methods to answer `six` research questions. We first statically analysed a corpus 18,122 Python projects on GitHub, and performed a longitudinal study over the commit history of 3,340 projects that we identified as Software-2.0 applications. In addition, we conduct a qualitative study by surveying `477` developers who updated or introduced a ML library in a project from our corpus.

The quantitative analysis helps us understand how developers use ML libraries and how the usage evolves over the project's lifetime, while the qualitative study reveals the motivations and challenges associated with introducing ML libraries and updating them. The results of the quantitative study motivated the qualitative study, in turn the results of the qualitative study led to several research questions, that we can answer thoroughly. Blending the two methods has the advantage that the results can be triangulated [45].

Table 1. Usage of top ML libraries

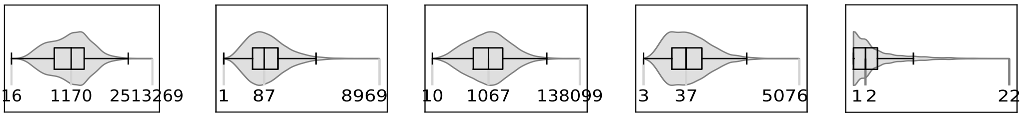
Library Name	Clients	Releases per year	Library Name	Clients	Releases per year
TensorFlow	**1711**	24.44	Inc-mxnet	91	4.48
Scikit-Learn	**1313**	10.67	TfLearn	40	5.95
Pytorch	**935**	9.15	Sonnet	16	13.14
Keras	**744**	11.34	CNTK	14	12.0
Theano	**342**	4.44	Paddle	8	10.72
Caffe	**224**	3.99	Neon	4	13.08
SpaCy	104	22.09	Pattern	0	1 [1]
NLTK	99	3.47			

[1] The library has only one release in 2013.

3.1 Subject Systems

3.1.1 ***Top rated Python Projects:*** Our corpus contains 18,122 large and popular Python projects hosted on GitHub. Our project selection process is inspired from Yu et al. [142], who studied the usage of Java annotations in practice. We retrieve all non-forked and non-archived Python repositories whose stargazers counts (that indicate popularity of projects [18, 19]) are larger than 50 as of *December 31, 2018*. To avoid inactive projects, we discard the ones that do not have a single event of activity within 6 months prior to our data collection date, *31 Dec 2018*. Next, following the guidance in Kalliamvakou et al. [78], we exclude repositories with less than 3 contributors to further focus on the active projects. This results 18,122 repositories. The studied repositories include most well-known Python projects in GitHub such as Seaborn, Bokeh and Gensim.

The projects cover a variety of domains like frameworks, web utilities, database, utilities, robotics, image processing, etc. Our data set is diverse with respect to the application domain, size, development ecosystem, contribution governance, and testing practices. This diversity is important to make sure that the collected data is representative. Section 3.1.1 depicts violin/box plots for the different metrics of the 18,122 projects, including the number of lines of code, Python files, programming languages and contributors. A range of dimensions of the metrics that are relevant for the generality of a research topic define the space of the research topic and improve the generalizability [101].



(a) LOC (b) Python files (c) Commits (d) Contributors (e) Languages

Fig. 1. Size metrics for projects in our corpus

3.1.2 ***ML Libraries:*** Braiek et al. [21] identified 7 popular company-driven and 8 popular community-driven Python-based high-level ML libraries. While previous researchers [56, 69, 70, 125, 144, 145] have particularly studied the usage of the six most popular libraries (*TensorFlow*, *Scikit-Learn*, *Keras*, *PyTorch*, *Caffe*, and *Theano*). We identified all the clients of these 15 libraries (identified by Braiek et al.) from our corpus, as shown in Table 1. We found that 95% of the projects in our corpus use the six popular libraries (identified by researchers [56, 69, 70, 125, 144]). In the rest of the paper, we analyse the clients of these six libraries and refer to them as"***ML libraries***".

3.2 Static Source Code Analysis

We analyze the latest version of each project in our corpus to identify the clients of ML libraries and the usage of these libraries.

3.2.1 ***Identifying Software-2.0 applications:*** To study the evolution of ML library usage in Software-2.0, it is important to identify projects that use ML library(s). We analyzed the major releases of these ML libraries and collected all the root package names. To determine if a project indeed uses a ML library, we check (i) if the project contains `import` statements pointing to the root packages from the ML library, and (ii) the project invokes at least one API method from the ML library. We identified a total of 3,340 projects that use the six popular ML libraries (Section 3.1.2). The complete list of projects is available in the companion website [1].

3.2.2 ***Analysing the API usage:*** We further analyze these 3,340 project to understand how developers use the APIs provided by the ML libraries, by identifying the method invocations and object references that are bound to these ML libraries. Python is dynamically typed, thus the required type binding information is available at run-time only. However, it is not feasible to run 3,340 to extract the type binding information. To overcome these challenges, tool builders have built tools such as Jedi[71] and `MyPy` [100] which perform static code analysis on Python programs and infer the types. MyPy requires the code to have these type annotations [50] (introduced with Python v3.5) to infer types. However, 93.88% projects in our corpus have not adopted type annotations, while `Jedi` only requires all the external dependencies and the entire source code to infer the binding information. `Jedi` is a popular (4000 GitHub stars), widely adopted (47,300 users) tool and used in previous studies[41, 44]. Therefore we use `Jedi` for inferring type binding information.

3.3 Tracking ML library introduction and update :

3.3.1 ***Dependency Model:*** Previous researchers [5, 30, 38, 40, 129, 129] studied library update and migration in context of Java, but studying it in context of Python is not straightforward. To study library updates, it is important to detect when a library was updated or replaced. Kula et al. [85] proposed a dependency model which works for Java, where developers can specify dependencies as one particular version of a library (e.g. `deeplearning4J == 0.7.0`).

The previously proposed dependency model [85] will not work for Python, since Python package managers allow developers to specify a set of versions of a library as dependencies (e.g. `tensorflow >= 1.0.4`) in a *requirements.txt* file. PEP 440 [51], the standard for Python version identification and dependency specification, specifies six supported version clauses, namely: (i) *compatible release* (~=), (ii) *version matching* (==), (iii) *version exclusion* (! =), (iv) *inclusive ordered comparison* (<=, >=), (v) *exclusive ordered comparison* (<, >), and (vi) *arbitrary equality* (===). An entry in *requirements.txt* without any version specifier (specify only the library name) will install the latest version of the library. Using these clauses, developers can express sets of individual versions or as a range of versions. We abstract the information needed to track library update into a dependency model that captures the versions of the library dependencies associated with each Python project.

Definition 1. ***Library Versions (LV):*** *Let* `L` *be a set of Python libraries and* `v` *be a valid version identifier. For library* `l ∈ L`, *a library version is a tuple* `LV = (l ,{v})`, *where* `{v}` *is a chronologically ordered set of version numbers of* `l`. *For example,* `(tensorflow, {1.0.0, 1.0.4, 2.0.0})`.

Definition 2. ***Software Project and Dependencies:*** *Let P be a set of Python projects, such that each project* `p ∈ P` *contains a* `requirements.txt`. *For each* `p` *at version* `i` *there exist a* `requirement.txt`

[1] https://serene-beach-16261.herokuapp.com

Table 2. Categories of Library version update

Category	**Rule** ($\forall\, v1 \in vs_1$ $\forall\, v2 \in vs_2$)	**Description**	**Example (from ⇒ to)**

Strict Upgrade	$v2 \notin vs_1$ & $v2 > v1$	Update to newer version(s)	`tensorflow==1.0.4 ⇒ tensorflow==1.0.5`
			`tensorflow <= 1.0.0 ⇒ tensorflow>1.0.4`

Strict Downgrade	$v2 \notin vs_1$ & $v2 < v1$	Update to older version(s)	`tensorflow==1.0.4 ⇒ tensorflow==1.0.3`
			`tensorflow > 1.0.4 ⇒ tensorflow==1.0.4`

Non-Strict Upgrade	Either $v2 \in vs_1$ or $v2 \notin vs_1$ & $v2 > v1$	Extend support for newer versions	`tensorflow==1.0.4 ⇒ tensorflow>=1.0.4`
Non-Strict Downgrade	Either $v2 \in vs_1$ or $v2 \notin vs_1$ & $v2 < v1$	Extend support for older versions	`tensorflow>=1.0.4 ⇒ tensorflow>=1.0.3`

Support more versions	$v1 \in vs_2$	Extend support for older and newer versions	`tensorflow>1.0.0 and tensorflow<2.0.0 ⇒`
			`tensorflow>0.9.0 and tensorflow<2.1.1`

Support fewer versions	$v2 \in vs_1$	Remove support for some versions	`tensorflow>1.0.0 ⇒`
			`tensorflow>1.0.0 and tensorflow!=1.0.6`

file RF_i which specifies {LV}, a set of library versions for p. *For example, project Texar* [2] *specifies dependencies -* `{(tensorflow, {1.10.0 , ... 1.15.x}), (sentencepiece, {0.1.8, ..., 0.1.85,})...}`.

Definition 3. ***Library Version Update:*** *We state that the project p ∈ P has changed the library version of library $l \in L$ from vs_1 to vs_2, if the requirement file at version* i*, RF_i contains dependency* $\mathbf{LV_i}$ *= ($\boldsymbol{l}$, $\mathbf{vs_1}$) and if* $\mathbf{LV_i}$ *is replaced by* $\mathbf{LV_j}$ *= ($\boldsymbol{l}$, $\mathbf{vs_2}$) in RF_j (at version* j*), where $i < j$ and vs_1 != vs_2.*

Definition 4. ***New Library Introduction:*** *We state that the project $p \in P$ introduces a new library $\boldsymbol{l} \in \boldsymbol{L}$, if the requirement file RF_i, ($\boldsymbol{l}$, _) ∉ RF_i and ($\boldsymbol{l}$, _) ∈ RF_j at version* j*, where $i < j$.*

3.3.2 ***Detecting Library Version Change:*** We analyse the source code history of each project p ∈ P. At each version i we extract the model *RF_i* by parsing the *requirements.txt* file. When versions are specified individually, we simply add the version to LV. If the versions are specified as a range, we obtain the subset of versions from all possible versions of that library (obtained from GitHub API), which satisfy the version specification clauses. At each commit, we check if a ML library's version is changed or a ML library is introduced. We categorise library version changes into six categories (i) strict upgrade (ii) strict downgrade (iii) non-strict upgrade (iv) non-strict downgrade (v) supporting more versions (vi) supporting fewer versions as shown in Table 2. These reported ML library version changes and new ML library introduction enable us to gain deeper insight into:

- evolution of ML library dependencies (RQ3) and the associated challenges (RQ4)
- retrofitting ML and the associated challenges (RQ6)

3.4 Association rule mining

We discover association rules in ML library combinations and cascade library updates (i.e., libraries that are updated together). This technique has been used in many fields in software engineering research such as API auto-completion [13], software error prediction [89], etc.

Association rules are in the form X ⇒ Y, where X is called the antecedent (if) and Y the consequent (then). To define the confidence of a mined rule, the technique use term called "Confidence" which is the number of transactions where X and Y co-occur divided by the total number of transactions in

[2] https://github.com/asyml/texar/blob/0704b3d4c93915b9a6f96b725e49ae20bf5d1e90/requirements.txt#L1

which X occurs. In our study, we use the FP-Growth [55] algorithm as our association rule mining algorithm [126]. The algorithm uses a set of transactions as its input and produces (a) frequent itemset, and (b) strong association rules. To define whether a particular itemset is frequent or not and generate the rules only from the frequent item set, support count is used. It is simply the number of transactions in which both X and Y occur.

$$confidence(X \Rightarrow Y) = \frac{frequency(X \cup Y)}{frequency(X)} \qquad support\_count(X \Rightarrow Y) = frequency(X \cup Y)$$

3.5 Qualitative Study

The most reliable way to understand the motivations and challenges associated with retrofitting and updating ML libraries is to ask the developers who performed it. To achieve this, we surveyed three groups of developers:

(1) **Group-A** *(Developers who* ***updated*** *ML library version):* These are developers of the commits where ML library version was updated. Surveying these developers reveals challenges and current practices.
(2) **Group-B** *(Developers who did* ***not update*** *ML library versions):* We selected these developers from the projects where none of the ML libraries were updated, although the developers have updated the *traditional libraries* in the projects. Surveying these developers reveals factors that prevent updates.
(3) **Group-C** *(Developers who* ***retrofitted*** *ML library):* These are developers who retrofitted the ML libraries in the project. Surveying these developers reveals challenges and motivations for retrofitting ML libraries.

Table 3. Survey questions for each group

Survey Group	Questions	Motivation	Response rate
Group A[1]	What changes were performed to update the ML library? Was the update difficult? How long it took to perform this update? Was the update performed manually or with tools ? Would automating the manual tasks be helpful for the Project?	Discover associated challenges, the time required for it, current tool usage and identify new opportunities to assist developers in this process.	60/264 (22.72%)
Group B[2]	What prevents the project from updating the ML library? What would help the Project to perform updates? Would you desire to have tools to perform these updates?	Discover the factors that prevent developers from updating their dependencies and identify new opportunities to assist them.	21/85 (24.70%)
Group C[3]	Why did you introduce the ML library in the project? What challenges were faced and how were they resolve?	Discover the reasons and challenges when adding ML libraries to a software solution	28/128 (21.88%)

[1] Developers who updated ML library version.
[2] Developers who did not update ML library versions.
[3] Developers who retrofitted ML library.

Our goal in designing the survey was to keep as short as possible, while still gathering all of the relevant information. In the Table 3 we summarize the survey questions of the three groups and our motivations for asking them.

3.5.1 ***Contacting the developers:*** We designed different survey questions for the three groups of developers mentioned above. To assign developers in Group-A and Group-C we first collected all the commits where a ML library version was updated or a ML library was retrofitted. To assign developers in Group-B, we first collected the projects where no ML libraries was updated but at least one other library was updated. From these projects, we identified the commits where the updates were performed. For all the aforementioned commits, we gathered the developer's name and email address, and then sent the respective survey questions as an email. In the body of the email, we include:

- A short message introducing the research team and explaining the purpose of our study.
- A timeline of library version updates and new library introductions of the project.
- A GitHub URLs to the commit, so that the developer can easily find it and inspect it.
- Questions for the participant.

The sample emails for the three categories are available in the companion website. In addition, to avoid spamming the same developers with multiple emails, we follow the best practices suggested by Mazinanian et al. [93]:

- If a developer is an author of more than one commits, we sent an email for the most recent event
- If a participant has not responded to our first email, we send a final reminder after a week

In total, we sent 477 emails to developers, out of which 109 responded, bringing us to a 23% response rate. This is significantly higher than the usual response rate achieved in questionnaire-based software engineering surveys, which is around 5% [123]. The category-wise response rate, shown in Table 3 highlights that each of the surveys had a response rate higher than 21%. Our survey design philosophy is to provide the developer with valuable information about the project along with the survey questions. For example rank of the project among the examined projects concerning the average density of ML library API calls per project file. Developers praised us for the information provided, for example, a developer said, *"I did not know the statistics that you write and they are really interesting! Can I publish them, for example on the MadTwinNet repo and/or social media?"*.

3.5.2 ***Thematic Analysis:*** Since all of our survey questions are open-ended questions, we received all of our responses in free-form text (i.e., emails). Therefore, a qualitative data analysis approach is needed to systematically extract useful information from the emails. Thematic analysis is "a method for identifying, analyzing, and reporting patterns (themes) within data" [31] and a widely-used qualitative approach that allows gaining a deeper understanding of the data [138].

We followed the steps required by thematic analysis as suggested by best practices guidelines [22]. Two authors of the paper independently read the responses carefully and assigned one or more descriptive phrases (i.e., codes) to each sentence. Both authors conducted the initial meeting after coding around 25% of the data (the suggested minimum size is 10% [25]). During the meeting, the authors carefully discussed the coding process and also negotiated any disagreements between the assigned codes. After 80% inter-coder agreement was achieved, the authors started assigning codes in the entire data set of the participants' responses. (recommended inter-coder agreement level is ranging from 70% to more than 90% in the literature [25]). After the coding finished, the authors held another meeting in order to finalize the codes and extract themes. Themes capture something important about the data in relation to the research question. It also represents some level of patterned response or meaning within the data set [22]. The two authors reviewed the initial themes against the data several times and refined their names and definitions until they both agreed that there were no further refinements possible.

4 RESULTS

In this section we present and discuss the results of our `six` research questions. We use qualitative and quantitative methods to answer each research question.

4.1 *RQ1*: What is the trend in ML library usage?

Over the last decade, the open source and company-driven communities have developed many machine learning libraries. These libraries have gained popularity with tens of thousands of users, thousands of stargazers and forks. In software, the primary units of reuse are functions, algorithms, libraries, and modules. Also, a software engineer can find the source code for a library (e.g. on GitHub), fork it, and easily make changes to the code, using the same skills they use to develop their own software. What is the trend of using the ML libraries? What is the trend of forking the ML libraries? Are developers increasing the usage of the ML library APIs overtime? Answering these research questions can motivate researchers to understand the challenges developers face when using and extending these libraries. It will also motivate library developers to make informed decisions when evolving their libraries. To understand the trend, we analyse all 18,122 projects in our corpus.

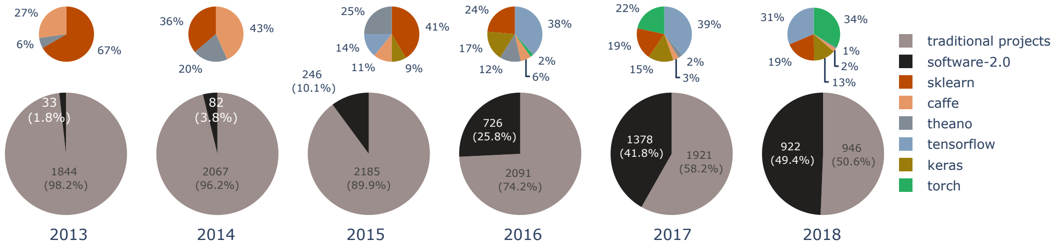


Fig. 2. The larger pie charts at the bottom show the distribution of Software-2.0 and Software-1.0 per year. The smaller pie charts at the top show the library-wise breakdown per each year.

4.1.1 ***Trend of creating ML projects:*** We group the projects in our corpus of 18,122 Python Projects, by the year in which they were created. For the creation date we consider the first commit date in Git log instead of project creation date given by GitHub search API. (to handle cases where projects moved from a different code management system like GitLab/BitBucket to GitHub). Figure 2 shows library usage distribution in each year. It further shows the ratio of new projects using ML libraries (Software-2.0) to the total number of projects created in each year. While in 2013 only 1.75% new projects used ML library(s), this ratio increased to 49.63% by 2018.

To conduct a more accurate and fine-grained trend analysis, we split the data further into monthly intervals and apply a statistical trend test on it. Particularly we used *Mann-Kendal* trend test; a non-parametric statistical test that examines the null-hypothesis that "there is no trend in the data". Obtaining small p-values (e.g., smaller than $\alpha = 0.05$) will lead to rejecting the null-hypothesis (i.e., there is a trend in the data). In this case, the test can also show the degree of the monotonicity of trend (i.e., the τ value). The τ value is ranging between $-1 < \tau < 1$, where -1 shows a perfectly decreasing trend, and 1 shows a perfectly increasing trend. While there are other ways to assess trends in the data (e.g., regression analysis), the *Mann-Kendall* test makes the fewest assumptions about the underlying data, making it suitable for our analysis. Indeed, this test has been used in previous software engineering studies, e.g., for examining the applicability of Lehman's Laws of Software Evolution [6]. Running this test on the dataset revealed that there is a strong positive trend in using ML libraries in the new Python projects created each month, i.e. p-value < 0.05 and

τ = 0.87. In addition, we used the *Sen's estimator* [122] to compute the slope of the trendline for the number of new projects using ML libraries each year for the entire epoch of six years. This estimator is basically the median slope of the lines drawn between all the pairs of points, which turned out to be 8×10^{-2}, which essentially shows the ratio of ML libraries introduced is increased in each month within the period we analyzed.

4.1.2 ***Trend of using ML library API within the projects***. We delve deeper into the trend of the usage of ML libraries within the project by performing a longitudinal analysis of ML API calls. For the last commit in each month of the project's history, we normalize the number of ML APIs used in the project by the project size (i.e., number of files). The normalization eliminates the adverse chance of observing an increasing (or decreasing) trend due to the increase (or decrease) of the amount of code committed to the repository [43].

To statistically assess the trend of using ML APIs, we used the *Mann-Kendal* trend test (Section 4.1.1). Applying this trend test revealed a p-value < 0.05 for 57% of the projects. This suggests that there is a significant increasing or decreasing trend in using ML APIs for these 1903 projects. Since the null hypothesis is not rejected for the other 43% projects, no conclusion can be made for them.

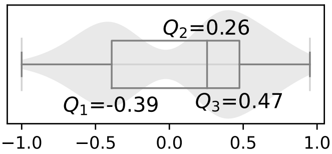


Fig. 3. ML API adoption trend as measured by τ value computed with the Mann-Kendal trend test

We further investigate if this trend is increasing or decreasing by studying the values of τ, for all the projects. Figure 3 shows that the median of τ in all projects having a significant trend in adopting ML APIs is 0.26 i.e., *a moderate, increasing trend*. We further found that 87.76% of these projects show an increasing trend in adopting ML APIs (i.e. $\tau > 0$). To compute the slope of the trendline for the number of ML APIs introduced per Python file per month in the commit history of all projects, we used the *Sen's estimator* [122]. The value turned out to be 2.37×10^{-1}. This shows that developers add 0.237 average number of ML APIs per Python file in each month.

4.1.3 ***Trend of ML library forks:*** Developers usually fork repositories to contribute to it by fixing bugs, adding new features and optimizations, or for keeping a copy of the source code [74]. For example, Intel maintain their own fork[66] of *TensorFlow*, *Caffe* and *Theano* where they add optimisations for Intel CPUs [65, 67, 68]. We found that *TensorFlow* is the most forked Python repository on GitHub with over 78k forks and other ML libraries like *Keras*, *Caffe* and *Scikit-Learn*

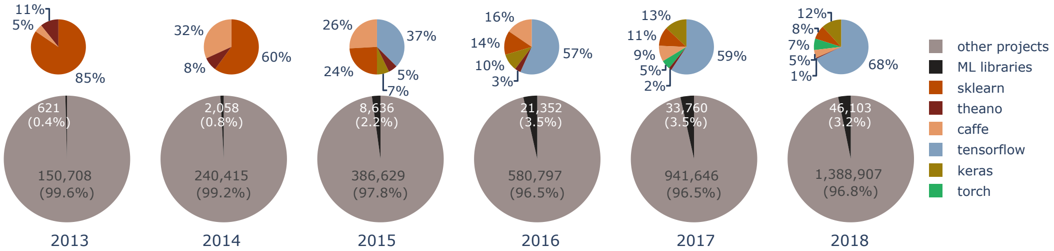


Fig. 4. The larger pie charts at the bottom show the distribution of the newly created forks of ML and non-ML projects per year. The smaller pie charts at the top show the library-wise breakdown per each year.

being forked more than 17 thousand times. Figure 4 shows the ratio of the number of new forks created from the ML libraries to the total number of new forks created on GitHub each year. While in 2013 only 0.41% of total forks created were created from the ML libraries, this increased to 3.21% in 2018. To conduct more fine-grained and accurate analysis upon the trends, we split the data further into monthly intervals and then apply statistical trend tests similar to Section 4.1.1. Running these tests on the dataset revealed that there is a strong positive trend in creating forks from ML libraries each month, i.e. p-value < 0.05 and τ= 0.52. Also the slope of the trendline using the *Sen's estimator* turns out to be 9×10^{-4}, which essentially shows the ratio of ML libraries forked increases month within the period we analyzed.

Observation 1: There is an increasing number of new projects using ML library(s) each year. ML library usage increases from 1.75% in 2013 to 49.63% in 2018. Furthermore, the amount of ML library API usage increases: in 50% of the projects in our corpus, developers add 0.237 ML APIs per Python file each month. We also observed an increasing number of forks of ML Library(s) each year, from 0.41% in 2013 to 3.21% in 2018.

4.2 *RQ2*: What combination of libraries do developers use?

With dozens of competing ML libraries that share the same objectives, developers can often feel overwhelmed when deciding which libraries best suit their requirements. They also need to invest significant time to become expert users of these libraries. How can they leverage the wisdom of the crowds and learn from others' success? Moreover, global companies like Intel, AMD, NVidia, Apple, Google, IBM [7, 10, 63, 66] are heavily investing in dedicated hardware accelerators to optimize ML workloads in real-time on the device. Without knowing the combinations of libraries that ML developers are using in the practice, these hardware and library vendors are left in the dark. How many ML libraries do developers use in their projects? What are the most common combinations of ML libraries used in practice? How does the ML development workflow affect the ML libraries used? Answering these questions will (i) motivate researchers to understand the challenge of using multiple ML libraries, (ii) open new opportunities for tool builders to assist developers (e.g., addressing library co-evolution), and (iii) help hardware and library vendors to make informed decisions.

4.2.1 ***Popular combinations of ML libraries:*** We performed static source code analysis as described in Section 3.2.1 to identify the ML libraries that are used in the projects. We found that 1,338 (40.10%) projects in our corpus use a combination of ML libraries (e.g., *TensorFlow* and *PyTorch*), while the remaining 59.90% projects use a single ML library. Table 4 shows that, while *TensorFlow* and *PyTorch* are the most popular libraries used by projects that depend upon a single ML library, *Scikit-Learn* and *TensorFlow* are the most commonly used libraries in the projects that use combinations of ML libraries.

We further analyse the combinations of ML libraries, by discovering association rules between the ML libraries used in combination (e.g. "Projects which used *TensorFlow* also use *X*"). Finding such rules can provide insight for hardware and ML library vendors to make informed decision for improving the current features or introducing new features in their products. We use the Frequent Pattern Growth (FP-Growth) algorithm [9] to identify the association rules (Section 3.4). Following the guidance as in previous works [13, 147], we choose thresholds that are not too low and not to high to ensure a trade-off between the number of rules and their relevance. We set the minimum support as 130 (~10% of data) and the confidence threshold as 0.5.

Table 5 shows all the mined association rules. *TensorFlow* and *Theano* provide symbolic model-building APIs where the developers can use them to build static computational graphs. For example,

Table 4. Library usage in projects

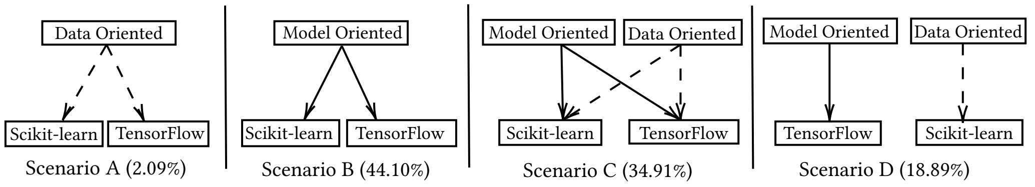
Single ML library projects (2,002 Projects)		Multiple ML library projects (1,338 Projects)	
Library	**Clients**	**Library**	**Clients**
TensorFlow	690	*Scikit-Learn* $\in L^1$	793
PyTorch	502	*TensorFlow* $\in L^1$	775
Scikit-Learn	405	*Keras* $\in L^1$	581
Keras	172	*PyTorch* $\in L^1$	334
Theano	138	*Theano* $\in L^1$	204
Caffe	95	*Caffe* $\in L^1$	126

[1] L is the set of ML libraries used by each project.

Table 5. Association rules

#	Rule (antecedent ⇒ consequent)
1	*PyTorch ⇒ TensorFlow*
2	*Theano ⇒ Scikit-Learn*
3	*PyTorch ⇒ Scikit-Learn*
4	*Keras ⇒ Scikit-Learn*
5	*TensorFlow ⇒ Scikit-Learn*
6	*Keras, Scikit-Learn ⇒ TensorFlow*
7	*Keras ⇒ TensorFlow*

the nodes `tf.add(x,y)` and `tf.subtract(x,y)` in *TensorFlow* can be used to build a computational graph to perform addition and subtraction. However, the actual computation happens when the created graph is transformed into `Session.run()`. In contrast, *PyTorch*, *Scikit-Learn*, and *Keras* follow the imperative programming paradigm where the compute happens at each line of the written code. This is quite similar to how a Python program is executed. The rules 1, 2, 5, and 6 (in Table 5) suggest that developers use a combination of libraries using different programming paradigms. Moreover, the rules 2, 3, 4, 5, and 6 (in Table 5) reveal that *Scikit-Learn* is often used with all the other five libraries. *Scikit-Learn* does not support GPU computing (hardware support), unlike the other libraries that support both CPU and GPU computing. The association rules reveal that the developers use libraries together that provide complementary programming paradigms, hardware support, and different strains of ML algorithms (e.g., supervised learning, unsupervised learning neural network and deep learning). Rule 7 (in Table 5) is not surprising as *Keras* is designed as a higher-level wrapper over *TensorFlow*.



Scenario A (2.09%)

Scenario B (44.10%)

Scenario C (34.91%)

Scenario D (18.89%)

Fig. 5. Four Scenarios for implementing Model-oriented (shown with full lines) and Data-oriented stages (shown with dashed lines) with ML libraries

4.2.2 ML library usage in ML development stages:

One possible reason for using a combination of ML libraries is that some libraries are better suited for certain stages of ML software development workflow (described in Section 2). For example, developers used `sklearn.impute` for the imputation of missing values in a dataset and `tensorflow.nn` for training a neural network.

We identify 1,002 projects from our corpus, that use a combination of ML libraries and contain a *requirements.txt*. We then analyze the source code of these projects using the type binding information obtained from `Jedi` (Section 3.2.2) and found 7,151 unique ML library methods used by these projects. The first two authors, assigned two labels for each ML library method - (i) data or model oriented and (ii) stage of the ML development workflow. We used a similar approach described under thematic analysis in Section 3.5.2 for the labeling process. The labels are assigned based on the description of the package (or the sub-package), in which the method is declared. We depended upon the description in the sub-package if the two authors could not come to a conclusion based on the description in the main package. We did not assign the two labels for the methods that (i) do not have a description in its package, or (ii) the two authors could not come to the same conclusion considering descriptions in all sub-packages, leaving 6,058 labeled APIs.

Our results reveal that (i) 44.10% projects use a combination of libraries for the model-oriented stages only (Scenario B in Figure 5), (ii) 2.09% projects use a combination of libraries for the data oriented stages only (Scenario A in Figure 5), (iii) 34.91% projects use a combination of ML libraries for both the data and model oriented stages (Scenario C in Figure 5), and (iv) 18.89% projects use different libraries for the data and model stage (Scenario D in Figure 5).

Table 6. Libraries used in ML development stages

Data Oriented Stages									Model Oriented Stages									
Data collection			Data cleaning			Feature engineering			Model Training					Model evaluation	Model Deployment		Occurrences	Scenario
sklearn	keras	torch	keras	sklearn	tensorflow	keras	sklearn	tensorflow	tensorflow	keras	torch	sklearn	caffe	sklearn	torch	tensorflow		
				✓			✓		✓								34	D
							✓		✓								27	D
		✓									✓	✓		✓	✓		31	C
			✓			✓			✓	✓							26	C
									✓	✓							23	B
									✓		✓						21	B
	✓			✓													5	A
✓	✓																4	A

Table 6 summarizes the top two ways of implementing ML development workflow using a combination of ML libraries in each scenario described in Figure 5 (We have given the results in our companion website [91]). For example, 31 projects use *PyTorch* for Data Collection and Model Deployment, and use the combination of *PyTorch* and *Scikit-Learn* for Model Training. We also observe that projects commonly use a combination of *Scikit-Learn* and *TensorFlow* for data and model oriented stages respectively.

Software engineering practice has found that modular designs with low coupling and high cohesion help create maintainable code in which it is easy to make isolated changes and improvements [52, 107]. Software-2.0 projects that use combination of ML libraries exclusively in stages (Scenario D in Figure 5) do not introduce dependence between the data and model oriented stages and therefore could be easier to evolve the models independently. In contrast, projects that use combination of libraries in multiple stages (Scenario A, B and C in Figure 5) potentially add dependence among different stages, thus increasing the complexity of maintaining and evolving the ML workflow stages independently.

Observation 2: 40.10% of the projects in our corpus use a combination of ML libraries. 34.91% of these projects share ML libraries between data and model oriented stages while 18.89% of the projects use ML libraries exclusively in both data and model oriented stages.

4.3 *RQ3*: How do developers update ML library dependencies?

ML libraries are rapidly evolving and releasing multiple versions each year to fix bugs, enhance performance or provide new features. How frequently do developers update their ML libraries in comparison to their *traditional libraries*? How frequently do developers upgrade or downgrade their ML dependencies? What are the other libraries that are updated along with ML libraries? Answering these questions will (i) highlight common practices for ML developers to follow, and (ii) provide new opportunities for tool builders to assist developers to update their ML dependencies.

From our corpus of 3,340 projects we retain all the projects that contain the *requirements.txt*, thus resulting in a set of 1,986 projects to answer this research question. We compare ML and traditional library version update frequencies in Section 4.3.1 using all 4,057 git commits that perform library updates. In these commits, we identified 1,055 commits in which at least one ML library version update was performed. We use them in Section 4.3.2 to understand library updates that trigger with ML library update. We further identified 8,389 library version updates from the git commits to understand the kinds of library version updates in Section 4.3.3.

4.3.1 ***Update frequencies in ML libraries vs traditional libraries:*** To understand how frequently projects update their libraries, we perform a longitudinal study in the commit history of all 1,986 projects that use *requirements.txt*. For each project in each category (i.e., ML vs traditional), we compute the *update ratio*, the ratio of the total number of library updates for that category to the total number of updates for the entire project (including all libraries across the categories). To achieve this, we identify all library version updates applied on projects using the dependency model explained in Section 3.3.1. Since the category of traditional libraries contains significantly many more libraries than the six ML libraries that we study, we need to normalize the aforementioned ratios with the number of libraries in each category.

Fig. 6. ML library update ratio vs *traditional library* update ratio
The box represents the interquartile range (IQR=$Q_3 - Q_1$) bounded by first and third quartiles and the central mark indicates the median. Anything outside $1.5 \times IQR$, shown with whiskers, is labeled as an outlier.

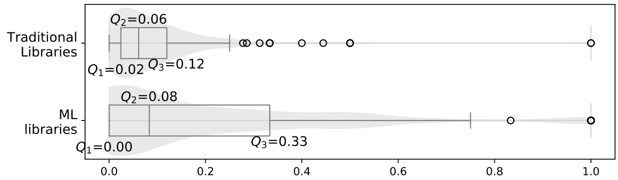


Table 7. Association rules for updates

#	Rule (antecedent ⇒ consequent)
1	*Scikit-Learn* ⇒ *Pandas*
2	*Scikit-Learn* ⇒ *NumPy*
3	Matplotlib, *Scikit-Learn* ⇒ *Scipy*
4	*Scikit-Learn* ⇒ *Scipy*
5	*Matplotlib, TensorFlow* ⇒ *NumPy*
6	*Theano, PyYAML* ⇒ *Keras*
7	*Tensorflow* ⇒ *Matplotlib*
8	*Matplotlib, NumPy* ⇒ *TensorFlow*
9	*Keras* ⇒ *TensorFlow*

Figure 6 shows the distributions of the ML and *traditional library* update ratio as a violin plot, which indicates that projects update their ML libraries more frequently than the *traditional libraries*. To assess if there is a statistically significant difference, we applied the *Wilcoxon Signed-Rank* test on the paired samples of ML library update ratio and *traditional library* update ratio for each project. The test rejected the null hypothesis that the ratio of *traditional library* update is more than the ratio of ML library at the significance level of 5% (p-value = 1.15×10^{-8}). We used the *Hodges-Lehman estimator* to quantify the difference between the ratios of library updates in the ML libraries and *traditional libraries*, as it is appropriate to be used with the Wilcoxon Singed-Rank test. The value turned out to be 0.0582, which is equal to the estimated median of the difference between the ratio of library updates in a sample taken from the ML libraries and a sample from the *traditional libraries*.

So far we found that developers update ML libraries more frequently than the traditional libraries. How do the ML library updates compare to traditional library updates in terms of complexity and developer effort? To answer these, we triangulate our quantitative findings with the qualitative analysis in Section 4.4 and Section 4.5.

4.3.2 ***Cascading library updates:*** To understand the impact of updating the ML library on the entire software system, we study all 1,055 identified commits where a ML library was updated. We found that in 41.52% of these commits, more than one library was updated. This shows that developers often update some other libraries along with a ML library. We call this *a cascading library update.* Researchers [76, 83] found that developers may take multiple commits to complete a single logical change, so researchers use a sliding window to study it. A sliding window can also introduce a large number of false positives. Therefore our results are a lower bound on the total number of the true cascading library updates performed.

Studying just the frequency of cascading library updates is not enough, because it does not identify the trigger of the cascading library updates. Association rule mining discovers rules that contain antecedents and consequences, e.g., when the library(s) X is/are updated, library(s) Y is/are also updated. We treat the set of libraries updated in each commit as a transaction, and apply the FP-Growth algorithm (minimum-support=10% of data, confidence=0.5) [55] to mine the rules. The mined rules shown in Table 7 reveal that antecedent of the rules (except rule 8) contain at least one ML library. Thus, developers need to be aware that updating one ML library triggers cascading library updates.

It is common for ML libraries to be built on top of *composite* libraries such as NumPy that provide numerical compute algorithms. These composite libraries are automatically downloaded once the developer installs the ML libraries into the development environment. Moreover, ML library clients often use other *supportive* libraries such as MatplotLib if they want to further process or visualize the results produced by ML libraries, or other libraries for unit testing, web development, etc.

The association shown in Table 7, reveal that developers often update the composite libraries alongside ML library updates. For example, the rules 2, 3, 4, 5, 6, 8, and 9 (Table 7) reveal that developers update *NumPy*, *Scipy*, or *PyYAML* with *Keras*, *TensorFlow*, *Theano* or *Scikit-Learn*. Similarly, the rules 6 and 9 (Table 7) further reveal that developers update *TensorFlow* or *Theano* with *Keras*. *TensorFlow* or *Theano* are used as composites of *Keras* and hence developers update them with *Keras*. We found that 60.41% of cascading library updates include at least one update of their composite libraries.

Fig. 7. Number of libraries in cascading library updates (except ML libraries and dependency libraries of ML libraries). The box represents the interquartile range (IQR=$Q_3 - Q_1$) bounded by first and third quartiles and the central mark indicates the median. Anything outside $1.5 \times IQR$, shown with whiskers, is labeled as an outlier.

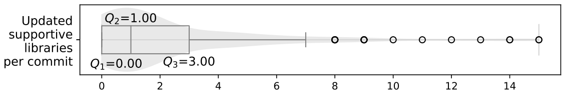


Table 8. Libraries in cascading updates

Library	Freq.	Library	Freq.
Matplotlib	106	certifi	17
Pandas	75	boto3	14
tqdm	31	botocore	13
requests	25	urllib3	13
pytest	23	tornado	11
opencv-python	19	seaborn	10

Furthermore, the rules 1, 3, 5, 7, and 8 (in Table 7) indicate that developers update *Matplotlib* or *Pandas* with *Scikit-Learn* or *TensorFlow*. *Matplotlib* and *Pandas* are supportive libraries for *Scikit-Learn* or *TensorFlow*. Understanding cascading library updates that involve supportive libraries is more challenging to study using associative rule mining because they do not have the same minimum-support (Section 3.4) levels as the composite libraries. To study the *supportive libraries*, we removed all the composite libraries from the cascading library updates. Figure 7 shows the distribution of the number of *supportive libraries* that are updated together with ML libraries. The median of the distribution shows that ML library updates usually contain 1 *supportive library*.

In the association rule generation, we use minimum-support count for the FP-Growth algorithm which is 10% of the dataset. Hence the algorithm only accounts for the libraries that appear above the minimum-support count for the rule generation. Therefore, the rules in Table 7 do not show the infrequent libraries that are updated with ML libraries. We use the FP-Growth algorithm on the aforementioned *supportive libraries* to generate frequently updated libraries and their frequencies instead of mining the association rules. Table 8 summarises the libraries that appear in more than ten cascading ML library updates. The complete list is available on the companion website [91]. We manually analysed the application domains of these *supportive libraries* and observed that libraries that use in *Web developments* (e.g., tornado, urllib3, certifi, and Requests), *Data Visualization* (e.g., Matplotlib and Seaborn), *Software Testing* (e.g., PyTest), and *Software Developments tools* (e.g., boto3, botocore) are usually updated together with ML libraries.

We observed that developers include both *composite* libraries and *supportive libraries* in cascading library updates. One possible reason for including composite libraries in cascading updates is that ML libraries depend on specific versions of composite libraries. For example, *Theano* 0.9.0 release[3] is only compatible with *NumPy* versions between 1.12 and 1.9.1. However, it is surprising that developers update supportive libraries alongside ML libraries.

4.3.3 ***Kinds of ML library updates:*** To understand the frequencies of version update kinds mentioned in Section 3.3.2, we studied 1,141 ML library updates and 7,248 *traditional library* updates. We considered all the version updates performed in the projects and categorized the updates of ML and *traditional libraries* according to the definitions in Table 2.

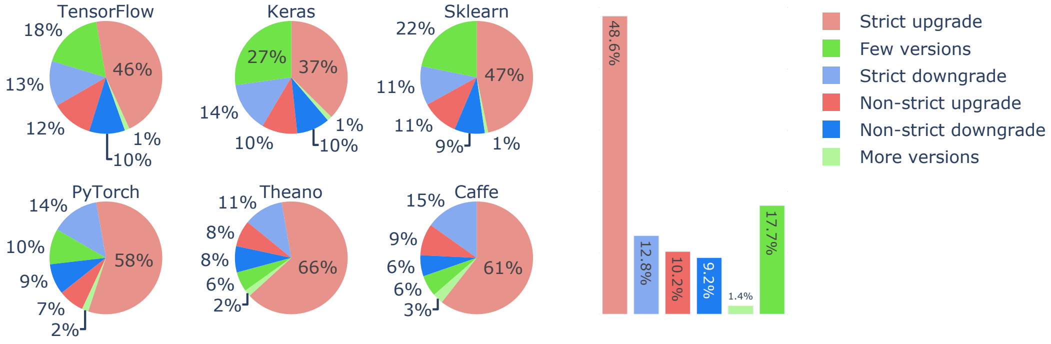


Fig. 8. The bar chart shows the distribution of cumulative update kinds for all ML libraries. The pie charts show the breakdown for each library.

From all the version updates for ML libraries, we identified (i) 48.62% strict upgrades, (ii) 12.81% strict downgrades, (iii) 10.23% non-strict upgrades, (iv) 9.23% non-strict downgrades, (v) 1.38% more version supports and (vi) 17.73% few version supports. Developers most frequently perform strict upgrades compared to non-strict upgrades. We also observed that in 22.04% cases developers performed library downgrades (strict and non-strict). We are interested to understand if developers downgrade libraries irrespective of whether it is ML or *traditional libraries*. We found that 10.90% of the *traditional library* updates were downgraded (strict and non-strict). This highlights that ML libraries are downgraded more compared to the *traditional libraries*. We further analysed git commits which have performed *traditional library* downgrades and observed 27.21% of the *traditional library* downgrades have performed with at least one ML library downgrade. It substantiates that 27.21% of *traditional library* library downgrades are done with ML library downgrades.

From our survey responses for Group-A, we try to understand the motivation for downgrading ML libraries. We found three motivations for performing library downgrade - (i) Unsatisfactory benchmarking results (e.g. developer S7 said, *"I downgraded the library after benchmarking the new ML model, and comparing it to the benchmark of the old ML model"*), (ii) Mismatched hardware instruction set (e.g. developer S8 said, *"I downgrade the TensorFlow because of this problem: Starting from 1.6 release, our pre-built binaries will use AVX instructions. This may break TF on older CPUs"*), and (iii) library version conflicts (e.g., developer S9 said, *"The time consuming part is to find the conflict libraries and set the version to a fixed version number, i.e. usually a downgrade"*).

[3] http://deeplearning.net/software/theano_versions/0.9.X/requirements.html

Observation 3: Developers update ML libraries more frequently than the traditional libraries. Moreover, 41.52% of ML library updates trigger cascading library updates and 60.41% of them update a composite library. We also observed that ML library downgrades are more frequent (22.04% of version updates) compared to *traditional libraries* (10.90% of version updates).

Table 9. Top challenges of ML library version updates

Challenge	Description	Percentage
Updating source code	Updating the source code for adapting to the breaking changes and deprecation.	51.85%
Retraining	(i) Pre-trained ML model's incompatibility across the ML library versions. (ii) Under-performing (in terms of precision and recall) decision thresholds of ML models. All these require developers to retrain the ML models.	40.74%
Library version conflicts	Mismatch of version usage due to direct and transitive dependencies.	30.86%
Understanding release documentation	Release documentation does not provide enough information about resolving breaking changes.	20.99%
Supporting multiple library versions	Supporting multiple versions of the ML libraries in the latest release of the software is tedious due to frequent API breaks and deprecations.	14.81%
Benchmarking new ML models	After updating the ML libraries developers retrain their models and compare the performance of it with the old model.	9.88%
Moving to Python 3	Libraries stop supporting older versions of Python. Therefore, clients are forced to update their Python version when updating the library.	7.41%

Survey group = Group A and B
Number of responses = 81

4.4 *RQ4*: What challenges arise when updating ML libraries?

The ML libraries continuously and rapidly release updates which fix bugs, enhance features and introduce new functionalities to compete with other ML libraries. For example, Table 1 highlights that *TensorFlow* released almost 24 versions each year. The respondent S19 said, *"It is easier to stay up to date with a more stable package like* `Matplotlib` *than it is with an evolving package like* `Scikit-Learn`*"*. Another respondent S18 said, *"ML libraries are dependent upon input data and pre-trained models, which makes it difficult to update them"*. Therefore, understanding the challenges for updating ML libraries that are dependent upon input data and pre-trained models will highlight blind spots in research and tool support for Software-2.0.

To gain a deeper insight into the associated challenges, we employ a quantitative and qualitative method (developer survey). We use 81 survey responses from groups A and B (Table 3), to identify the impediments to update ML dependencies and understand the reasons behind it. We then analyzed the source code to highlight how widespread are the problems identified from the developer surveys. Table 9 summarizes the challenges that are identified from the developer surveys. In some survey responses, developers expressed multiple challenges (thus the percentages do not add to 100%). In the following subsections, we provide a detailed explanation for the challenges along with real examples from source code and quotes from the developers.

4.4.1 ***Retraining****.*

(1) **Incompatibility of serialized ML models:** The model training stage of ML software development life cycle is resource-intensive [8]. Survey respondent S15 said, *"Library updates are almost always painful. For ML libraries like Scikit-Learn and TensorFlow, issues usually arrive from lack of binary compatibility of serialized models, requiring retraining"*. A case study done at Twitter [88] describes that the models are trained, serialized and then written to memory, to facilitate reuse. Our

survey respondents also confirmed this while describing a problem that arises for the serialized ML models with library version updates. When developers update their ML library dependencies, they are often unable to read the previously serialized model. Hence, they have to retrain the model which is time-consuming and tedious. Survey respondent S16 said, *"In my experience, a model saved in python 3.6 cannot be loaded in python 3.5, and same for python 3.5 -> python 3.6".* This further highlights the incompatibility of serialized model across Python versions.

A recent study done at Microsoft [8] describes the importance of tagging ML models with a provenance tag. This tag identifies the dataset that a model has been trained upon and the version of the model. Our survey responses revealed that apart from the training data, developers track the models based on library and language version. Developer S33 said, *"Currently, we have four separate pre-trained models, covering older/newer Scikit-Learn versions in both Python 2 and 3".* This opens new opportunities for researchers and tool builders to assist ML software developers, who continuously maintain, evolve and adapt the serialized models in addition to the source code.

(2) **Selecting decision thresholds:** Some ML models (like a SVM classifier for spam emails) work with decision thresholds. Developers conduct several experiments to select decision thresholds to get good trade-off on certain metrics, such as precision and recall. Sculley et al. [121] observed that changing the training data may force users to re-select a new threshold value from possible threshold values. Our survey respondents revealed that even updating the ML library may force them to re-select the threshold values. The process of re-selecting these values is tedious and time-consuming. This highlights a need for a technique that preemptively identifies when a version update might affect the decision thresholds and assist the clients in choosing the new values.

4.4.2 ***Benchmarking new ML models:*** "*In scikit-learn >= 0.20, they have changed the implementation of logistic regression and SVM, so the output of these two algorithms will be different (even if we fixed all the parameters)*", said developer R34. ML library developers release updates that introduce optimisations and enhancements. Usually the developers benchmark their models before and after the library update. They expect the same or improved performance and accuracy for the ML algorithms after a library update. If the update does not meets the expectations, they fall back upon the previous version of the library. Our respondents revealed that this process is very time-consuming. This constrains the developers to keep their ML library dependencies up-to-date.

```
try:
    from sklearn.metrics import calinski_harabasz_score as chs
except ImportError:
    from sklearn.metrics import calinski_harabaz_score as chs
```

Fig. 9. Supporting multiple versions of Scikit-Learn

4.4.3 ***Supporting multiple library versions:*** Respondent S19 said, *"We are currently supporting 7 versions of scikit-learn; 2 major versions, 0.20 and 0.21 and 5 minor versions."*. About 15% of the respondents reveal that they support multiple versions of the ML libraries in their project. This is challenging because developers have to implement workarounds for breaking changes and deprecation. The survey responses show that developers apply the *import statements with exceptions handling* strategy, to support multiple ML library versions with potential breaking changes and deprecation amongst them. For example, Figure 9 shows how the project *Yellowbrick*[4] applies the exception handling strategy to use the module `calinski_harabaz_score` in `Scikit-Learn 0.20.0` and the renamed module `calinski_harabasz_score` in `Scikit-Learn 0.20.2`. Likewise, developers can keep adding more `try-except` statements to handle other breaking changes and deprecation for the same module in the next version and support three library versions.

[4] https://github.com/Kautumn06/yellowbrick/commit/c525d1276d3d74d9e31fa7039906d3bd47f92600

We statically analyzed the source code of all 3,340 projects and found that 21.10% of the projects use this strategy. The respondent S23,*"Supporting multiple version adds some overhead, which increased code complexity."*. Our analysis shows that ML libraries like *TensorFlow* release an average of 24 versions each year (Table 1). Using the *import statements with exceptions handling* strategy to support multiple ML library versions would eventually increase the complexity of version updates. For example, if a project decides to support newer version(s) and stop supporting some older version(s) of a library, developers have to clean up the dead code that supported the older version(s) and add support for the new version(s).

Table 10. Number of Projects that use dependency libraries of ML libraries

TensorFlow			*Scikit-Learn*			*PyTorch*			*Keras*			*Theano*			*Caffe*		
1,632-Projects			1,204-Projects			842-Projects			753-Projects			342-Projects			221-Projects		
library	#projects		library	#project		library	#projects		library	#projects		library	#project s		library	#projects	
numpy	1521	▇	numpy	1105	▇	numpy	781	▇	numpy	638	▇	numpy	305	▇	numpy	197	▇
six	392	▇	scipy	644	▇	torchvision	467	▇	scipy	247	▇	scipy	178	▇	scipy	113	▇
h5py	130	▇	joblib	77	▇	PIL	311	▇	six	120	▇	six	59	▇	matplotlib	101	▇
absl	45	▇	-	-	-	six	98	▇	h5py	93	▇	-	-	-	PIL	93	▇
grpc	25	▇	-	-	-	-	-	-	yaml	46	▇	-	-	-	yaml	32	▇

4.4.4 ***Library version conflicts:*** Survey respondent S12 said, *"it's difficult to maintain the dependencies, because of the conflict between two library versions"*. Problems arise when using shared libraries on which several other libraries have dependencies but they depend on different and incompatible versions of the shared libraries [35]. For example, a project uses NumPy with TensorFlow and NumPy is a dependency library of TensorFlow. Therefore updating NumPy could lead to version conflicts with TensorFlow[5].

We are interested to find how widely this problem can happen in SOFTWARE-2.0. Therefore, we identified all the dependency libraries of the ML libraries and analysed `import` statements of all our 3,340 projects to find the use of these libraries. Table 10 summarizes results with the number of projects which use each dependency of ML libraries. Due to space limitations, we summarised only the top five libraries and listed all the results on our companion website. From our static analysis, we identified that 95.37% of the projects directly depend on transitive dependencies generated by ML libraries. This substantiates how vulnerable the ML library clients are to the dependency version conflicts and how the use of a combination of ML libraries will further amplify the issue. Further, In Table 10, we observed that ML library clients and library vendors mostly share *NumPy* and making version constraints for *NumPy* will initiate more version conflicts in ML library clients.

4.4.5 ***Updating source code.***

(1) **Change of computing architecture:** The survey respondent S21 said *"Tensorflow changed from using* `sess.run` *to* `tf.Estimator`*, which changes "how" we can train on using multiple GPU. (tower gradient -> mirrored strategy)"*. These changes require rewriting the code that demands significant developer effort to finish ML library update process.

(2) **Python-specific source incompatibilities :** Apart from the catalog of API changes proposed by previous researchers [30, 38, 40] for Java, Python showcases different kinds of changes such as default value update, keyword rename, keyword type change and transform parameter from keyword to positional. Islam et al.[69] observed that developers search for software breaks that

[5] https://github.com/tensorflow/tensorflow/issues/21939

happened due to keyword renames between library versions. However, researchers have not delved into extending the catalog of the API changes beyond Java, for languages like Python.

(3) **Deprecation:** The respondent S19 said, *"Deprecated functionality is the most work especially when there isn't a 1-to-1 replacement"*. Often the deprecation documentation suggests none or multiple alternative API. Since the *deprecate-replace-remove* [39] cycle is not always followed when deprecating ML APIs, handling such deprecation is challenging because developers have to (i) find alternative APIs for the deprecated APIs, (ii) replace a deprecated API with multiple APIs (iii) implement workaround for the deprecated APIs or (iv) add another ML library that provides the deprecated capability.

(4) **Library splitting:** Apart from the above changes, ML libraries are also subjected to splitting. Sometimes ML library developers create new library by separating a portion of the ML library. Respondent S27 said,*"Scikit Learn announced that Python2 support will be dropped and therefore library* `six` *will not be part of sklearn.externals anymore"*. This will force the developers to update the separated APIs and maintain another separate dependency along with the original library.

4.4.6 ***Python 2 to Python 3:*** *"There's also the Python 3-only gap imposed by the latest version of Scikit-Learn - we can't upgrade to that until our package stops supporting Python 2"*, said respondent S27. Python introduced backward incompatibility when they released Python 3. All ML libraries adopted Python 3 and some of them have stopped supporting Python 2. For example, *TensorFlow* release note[6] states, *"TensorFlow 2.1 will be the last TF release supporting Python 2"*. If the clients want to use the latest version of these ML libraries, they have to adopt Python 3. However, this could lead to many problems, especially if the client depends upon a library that does not support beyond Python version 2. Malloy et al. [92] also confirm that developers have not been willing to make a full transition to Python 3 from Python 2.

Observation 4: ML library update process consists of unique challenges such as "retraining", "benchmarking new ML models" and "updating decision thresholds" due to data/model dependency. Our quantitative analysis shows ML libraries are highly vulnerable to the challenges "supporting multiple library versions" and "dependency hell".

4.5 *RQ5*: What help do developers seek for updating ML libraries?

"TensorFlow often has backward-incompatible changes which make it difficult to know if the code in our project will continue to run as expected. Also, in general, we've had issues with new versions of packages breaking the build so not updating the version made it easier to ensure everything works", said a survey respondent. Continuous evolution and maintenance of software are the essential factors for a healthy codebase. Therefore, it is important for researchers to identify and provide solutions to the pain points developers face when updating ML libraries. What tools do developers use to update ML libraries? What are the gaps in the current tooling? How can the current tooling be extended to better assist developers to update their ML libraries? This will open new opportunities for tool builders and researchers, to assist developers when performing ML library update.

We use answers from developer Group-A (Table 3) to identify the current tool usages and development effort for updating ML libraries in Section 4.5.1. Further, we use answers from both group A and B to spot gaps in the current tooling and ideas for some new tool assistance specifically for updating ML libraries in Section 4.5.3.

[6] https://github.com/tensorflow/tensorflow/releases/tag/v2.1.0-rc0?linkId=78227050

Table 11. Tools used

Suggestion	Percentage
Continuous integration	38.33%
Version control system	31.67%
Automated test coverage generation	13.33%
Static Analysis tools	11.67%
Containers	6.67%
API update tool	3.33%
Manually	48.33%

Survey group = Group A
Number of responses = 60

Table 12. Development effort

Effort	Percentage
Less than a day	45.00%
A day	5.00%
Few days but less than week	13.33%
A week	18.33%
Few weeks but less than month	3.33%
Months	5.00%
I don't know	10.00%

Survey group = Group A
Number of responses = 60

4.5.1 ***Tools used by developers:*** The survey results reveal that 48.33% of the developers do all the required changes manually while 43.33% of the developers use tool support to update ML libraries. We summarise the observed tools in Table 11. The rest of this subsection explains how ML developers use these tools in the ML library update process.

(1) **Version Control System (VCS):** Survey respondents reveal that ML models are checked for fairness, performance, or accuracy after the update, prior to releasing it into production. Respondent S47 said, *"if we are not happy with the update, git version control helps to roll things back if needed"*. As highlighted in Section 4.4.1, developers of Software-2.0 version data and ML models along with the source code. Moreover, a recent online article [53] discusses the limitation of the current version control system (e.g., Git) to assist in ML-specific problems like (i) storing and versioning large files (ii) versioning data, models and source code such that multiple experiments can co-exist. Tools like DVC[7] and LFS[8] try to solve this problem. Surprisingly, none of of our respondents mentioned these tools, highlighting the lack of awareness in the community.

(2) **Static analysis tools**: To identify undesirable behavior of a program that might lead to vulnerabilities, developers use static analysis tools after a ML library update. Respondent S60 said, *"if there is more static analysis in the form of Python linters and Mypy, the source code is more robust against error. So mypy usage just builds trust"*. Tools like `Jedi`, `Mypy` [100] are static type checkers for Python that aim to combine the benefits of dynamic typing and static typing. `MyPy` performs type checking on program that have type annotations (introduced in Python-v3.5 in 2015). Moreover, linters flag the presence of suspicious constructs, stylistic errors, code smells or bugs, thus enforcing a standard quality of the source code.

(3) **Continuous Integration:** Continuous integration (CI) automates the compilation, building, and testing of software. Hilton et al. [60] attribute the proliferation of CI for identifying bugs, testing across multiple platforms, or enforcing a workflow. Respondent S55 said, *"Most of our dependency version issues after a library update have been caught using CI with AppVeyor and Travis"*. Our survey shows that 38.33% of the projects use a CI system to identify errors or bugs in the program after a ML library update. However, Software-2.0 systems change on three axes: the source code, the model, and the data. Therefore, Software-2.0 systems are harder to develop, harder to deploy, harder to test and harder to explain than Software-1.0. Companies like ThoughtWorks[9] are working on adapting CI systems that are designed for Software-1.0 to better suit Software-2.0. Our survey respondents still rely on CI designed for Software-1.0 such as AppVeyor and Travis. Therefore, it

[7] https://dvc.org
[8] https://git-lfs.github.com
[9] https://www.thoughtworks.com/radar/techniques/continuous-delivery-for-machine-learning-cd4ml

Table 13. Requested tool capabilities

Suggested tool	Percentage	Suggested tool	Percentage
Dependency analysis/management tool	34.43%	Change summarizing tool	16.39%
Source Code update tool	31.15%	Impact analysis tool	8.20%
Model coverage detection tool	24.59%	Do not like to have any tool	6.56%
ML Model benchmarking tool	18.03%		

Survey group = Group A and B
Number of responses = 81

highlights opportunities for these open source CI systems to evolve and adapt to mitigate challenges for SOFTWARE-2.0.

(4) **Containers:** Developers use containers (Docker) to implement clean independent environments for builds. Respondent S59 said, *"We use Docker which I feel was very helpful in library update process. Dockerfile clearly tells the instructions to follow and we can easily experiment with changes"*. Our survey shows that 6.67% of the projects use a container tool. Applications like `Docker` were originally designed to support SOFTWARE-1.0. New open source tools like MLflow[10] and Pachyderm[11] are designed to specifically assist developers containerize SOFTWARE-2.0, so that developers can manage and track multiple experiments with different models or data. However, none of our respondents mentioned these tools that are specifically designed to support SOFTWARE-2.0.

(5) **Source Code Update:** Respondents revealed two *API update* tools that they use to perform ML library API updates - (i) *Dependabord* [36] reports the available new version updates and creates pull requests with API update changes (ii) *TensorFlow upgrade tool* adapts the *TensorFlow* clients to the API update. For example respondent S44 said, *"This update consisted mainly of renaming functions and prefixing them with* `tensorflow.compat.v1` *and were done mostly automatically with the tf_upgrade_v2 tool shipped with TensorFlow. This tool had a bunch of bugs at the time though so it did not work perfectly"*. Section 4.4.5 highlights the source code adaptations performed by developers when updating the ML libraries, such as *changing computing architecture* or *Python-specific source incompatibilities*. However, it remains unknown how representative are the source code adaptations performed by these tools in the real world.

4.5.2 ***Library update effort:*** We surveyed developers to understand the estimated effort for ML library updates and Table 12 summarizes the results. It shows that 39.99% of respondents take more than a day to update ML library versions while 26.66% of respondents take a week or more to update the ML library versions.

4.5.3 ***Tools suggested by developers:*** Table 13 summaries the new tool ideas revealed by survey respondents. In the following section, we provide a detailed explanation for the tool suggestion along with quotes from the developers.

(1) **Testing tools**

Model Coverage: Survey respondent S88 said, *"Normal software update requires only to test source code, but ML projects need a test for the dataset. A little bit of change in data will result in a crash. Therefore, knowing the tested ML model and code coverage from the testing dataset would be helpful."*. Depending on how representative the testing data is of the real-world data, the tested coverage of the model could be incomplete and the model may not be a true representation of real world application. Therefore, the concept of code coverage (i.e., a metric that describes the degree to which the source code of a program is executed when a particular test suite is executed) of SOFTWARE-1.0

[10] https://www.mlflow.org/docs/latest/projects.html

[11] https://www.pachyderm.com/platform/#community

should be extended to *model coverage* in Software-2.0. Researchers [109, 132] suggest test dataset that gets full or high neurons activation of a deep learning model can be considered as a good dataset for testing. Building end user tools that implement these techniques and integrating them to CI of Software-2.0 will help developers when deciding to ship the ML model to the production. For an example, developers in Software-1.0 say *"We can't go into production with less than 87% test coverage*[12] *"* whereas developers in Software-2.0 can say *"We can't go into production with less than 87% model coverage"*.
***ML Model benchmarking tool*:** ML developers benchmark the Software-2.0 before and after a ML library update as described in Section 4.4.2. Respondents requested tool support that can benchmark the software and identify good trade-off on the metrics such as precision, recall, performance, energy consumption, model bias, etc. It will help in the decision process when deciding to ship an updated ML model into production.

(2) **Update management tools**
Impact analysis tool: Survey respondent S77 said, *"An automatic task that updates the ML libraries and analyse the codebase to check for run time crashes due to ML model changes would be a significant removal of overhead"*. Tonella [134] et al. studied the impact of source code changes on the software. Ren et al. [114] propose a technique that detects mapping between the updated test cases and the updated source code. However the current impact analysis techniques should be extended for Software-2.0, to identify the impact of ML model change (introduced by an update) upon the source code.
Dependency analysis/management tool: Survey results reveal that developers frequently face challenges due to dependency conflicts. Moreover the quantitative analysis Section 4.4.4 highlights that 95.37% ML projects are vulnerable to dependency conflicts. Survey respondents want to have automated support for (i) reasons about version conflicts considering all direct and transitive dependencies, and (ii) recommends the possible compatible library version combinations. Ying et al. [141] propose a tool that identifies version conflicts by analysing the *requirements.txt*. However, it does not recommend possible library version combinations that can resolve the conflicts considering runtime errors.
Change summarising tool: Before updating ML library from one version to another version, developers need to understand all the changes between the two library versions. Since the frequency of releasing ML library versions is high, number of versions between source and destination versions will also be high. Therefore, understanding release documentation/code changes in a series of releases are identified as a painful task. Respondent S79 said,*"What helps me when updating is a clear changelog in text and a git diff compared to the foregoing releases"*. Foo et al. [47] present a tool to automatically check if a library update between any two library versions, introduces an API incompatibility for the software. While highlighting the applicability of these tools for Software-2.0, we observed that apart from the API incompatibilities, users expect a detailed description with a summary of features, bug fixes and non-resolved bugs between any two versions.

4.5.4 ***Discussion:*** Section 4.4 describes a plethora of challenges that developers face when developing Software-2.0 and Section 4.3.1 highlights that developers update ML libraries more frequently compared to the *traditional libraries* in the projects. Moreover, we found in Section 4.5.2, that (39.99%) of the survey respondents required more than a day's effort to update the ML library. Section 4.5 highlights that the tools (e.g. VCS, static analyzers) that developers use are inept for Software-2.0. Therefore, all of these observations together highlight the requirements of adapting existing tools and creating new tools to better assist Software-2.0 developers.

[12] https://martinfowler.com/bliki/TestCoverage.html

Observation 5: We observed that 39.99% of the respondents take more than a day to complete a library update process. Respondents use the tools that are designed to fulfill the requirements of Software-1.0 while they are requesting specific tools to evolve Software-2.0.

4.6 *RQ6*: What challenges arise when retrofitting ML libraries?

There is an abundance of Software-1.0 systems that can be improved by transforming to Software-2.0. For example, detecting anomalies in old banking systems, making mobile applications more personalised, suggesting items in old retail systems, etc. Some of these software systems have been developed before ML gained widespread adoption. While the new software applications already leverage the advantages of ML, we would like older software applications to take advantage of these opportunities. What motivates developers to retrofit ML into their existing systems? What are the challenges of retrofitting ML to an old software system? Answering these questions will motivate Software-1.0 developers to integrate ML techniques into existing software. It will also highlight blind spots in current research and tool development while opening new opportunities for researchers and tool builders.

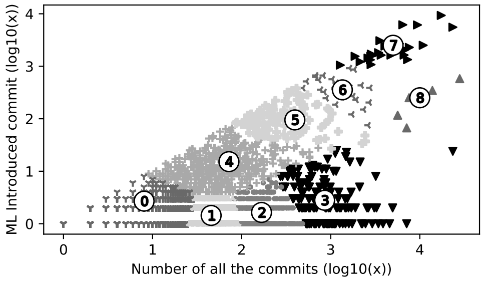


Fig. 10. ML retrofitting commits

Cluster	Contacted Developers	Total Responses	Retrofitted?
⑧	6	2	2 (100%)
⑦	22	4	4 (100%)
⑥	25	7	6 (86%)
⑤	75	15	12 (80%)

Table 14. Survey responses in each cluster

4.6.1 *Project selection:*

We first collect a set of projects that retrofit ML. For this purpose, we first identify projects that introduce ML significantly later in the project life cycle. Figure 10 shows a scatter plot, where each point in the plot corresponds to the point in the life-cycle of a project when ML was introduced. The x-axis indicates the total number of commits in the version history of the project (N_t) and the y-axis indicates the total number of commits before introducing ML in the version history of the project (N_r). We then apply the *K-Means* clustering technique to group projects that have introduced ML at similar points in their lifetime, as shown in Figure 10.

The value of *"k"* (i.e., number of clusters) in *K-Means* needs to be manually provided. To choose the optimal value of *"k"*, we use *Silhouette analysis* [118]. For each data point, we compute the *Silhouette coefficient (s)*, where -1 < s <+1. It measures how similar a point is to its own cluster compared to other clusters. Hence large s values are preferred. We found that the mean s of all data points was close to 1 for k = 8, 9, 10, and 11. We then manually analyzed the clusters at these values of k, and concluded that k *=9* was the most optimal value because the cluster sizes were uniform and the mean of all clusters was the largest.

It can be observed from Figure 10 that projects in clusters 0, 1, 2, 3, and 4 have a very short version history or introduce ML very early in their lifetime (within the first 100 commits). This means that the projects in these clusters adopted ML at their inception and did not retrofit ML. Therefore we discard these projects from our further analysis to understand the challenges around retrofitting ML libraries.

We contacted 128 developers who retrofitted the ML into the projects in clusters 5, 6, 7, & 8. One of our survey questions to these developers, confirms if the developer agreed that the project was mature before adopting ML. Table 14 shows that 85.71% of the developers we contacted, agreed that their project retrofitted ML. We then studied the motivations and challenges for retrofitting ML by analysing the answers from these developers.

4.6.2 ***Motivations for retrofitting ML libraries:*** To find out the motivations why developers retrofit ML libraries, we surveyed the developers in Group-C (Table 3). We found the following scenarios that motivate developers to retrofit using a ML library.

(1) **Add some brand new functionality:** We found that 39.29% of the survey respondents in Group-C started using ML algorithms when they added brand new functionality to their applications. For example, to emulate *Instagram* filters, project `Pixelhouse`[13] (an animation library) used *TensorFlow* to learn a neural network that approximates the functionality of the filter. Similarly, the project `Artemis`[14] (a framework for organising experiments) introduced hyperparameter search using *Scikit-Learn.*

(2) **Replace existing non-ML techniques with ML:** We found that 21.43% of the survey respondents migrated from deterministic rule-based computation to probabilistic computation that relies on ML algorithms implemented by the libraries. These ML algorithms are totally data-driven, and their models are trained on historic data which is labeled by a human domain expert. Developers continuously adapt to the changing trends and improve the models using feature engineering, changing the algorithm, or parameter tweaking. For example, the developer of an automatic music generation project `Musegen`[15] said, *"We don't want to use any rule-based models, hand-craft grammar or presets to compose songs. Therefore, we removed them and decided to use ML/DL to let computers find and learn patterns ... Finally, we believe learning patterns from big data would produce more creative results than merely writing rules".*

(3) **Augment existing functionality:** We observed that 14.29% of survey respondents enhanced existing functionality using ML techniques. For example, to enhance the audio experience for its users, `AutoEq`[16] (an automatic audio equalizer) used *TensorFlow* to optimize the parameters of the biquad filter. Similarly, *Samacharbot2* (a Reddit bot for summarizing news) improved the reading experience for its users by scraping news articles using ML-based text summarization techniques.

(4) **Replacing a ML algorithm with ML library:** We found that 7.14% of survey respondents replace their own implementation of ML algorithms with the one provided by third-party libraries. Developer S92 said, *"When I found the project Pykrige*[17]*, it was a natural fit for the Sklearn framework. Newer advanced algorithms could be easily derived from the existing algorithms in the Pykrige package that the Pykrige maintainers were not aware of at that time".*

(5) **Use mathematical or data packages from ML library:** We found that 3.57% of survey respondents did not use any ML algorithm from the library. Instead they use other capabilities such as (i) data and (ii) statistical functions. For example, the aircraft design toolbox (*SUAVE*[18]) uses the `sklearn.gaussian_process` to obtain a Gaussian distribution.

[13] https://github.com/thoppe/pixelhouse/commit/850966b3ae74627d76d30bc1189729294d4e07ed

[14] https://github.com/QUVA-Lab/artemis/commit/fa686a7efcd33f720ad9f63a0e71de48b1

[15] https://github.com/salu133445/musegan/commit/be3e348276e6e97eab15b994b888523fbcb0a5

[16] https://github.com/jaakkopasanen/AutoEq/commit/f121419df9b026cdd206252b8ead06a763216008

[17] https://github.com/GeoStat-Framework/PyKrige/commit/07af4a54260d602b6ba3eda92b20b928e9c9b57a

[18] https://github.com/suavecode/SUAVE/commit/96093098f49ff3e4d12f4082c8c564b11a13a14f

Table 15. Challenges faced when retrofitting ML

Challenge	Percentage
Data gathering and labeling	39.29%
Refactoring the code	28.57%
Inadequate documentation & developer training	21.43%
Retrofitting ML to edge devices	17.86%
Updating data types	14.29%
Managing data pipeline	10.71%
New dependency conflicts	10.71%
Interacting with other languages	7.14%

Survey group = Group C
Number of responses = 28

4.6.3 ***Challenges when retrofitting ML libraries:*** Transforming an established software system to depend on data and pre-trained models could be challenging. We surveyed developers from group-C (Table 3) to understand these challenges. Table 15 reports how widespread are these challenges for our survey respondents.

Due to space limitations, we provide a detailed explanation for five of these challenges faced by developers along with quotes from the developers.

(1) **Data gathering and labeling:** *"For the ML part, creating the training dataset was the most time consuming. It needed to listen to hours of previous episodes of music and label parts manually"*, said respondent S89. Data gathering and labeling could be hard because of (i) scarcity of domain experts to label the dataset or (ii) inaccessibility of the data (e.g., not having access to client data). Amershi et al. [8] also observed that data gathering and labeling are costly and time-consuming.

(2) **Documentation and developer training:** *"A challenge was the initial learning required to understand how it works, especially the math involved in it"*, said developer S90. Understanding ML documentation for any developer (who lacks expertise in mathematics or statistics) could be hard due to the underlying mathematical concepts in ML. To fill this gap, it is important to simplify and summarize the documentation of ML libraries so that all users can better understand it.

(3) **Retrofitting ML to edge devices:** *"I wanted to do advanced image processing on low-performance, low-power, non-GPU equipped edge devices"*, said developer S94. Retrofitting ML to applications on mobile devices is challenging due to limited processing power and limited battery life [95]. Quantization is a possible software solution to increase the performance of ML algorithms on such devices. However, a developer said, *"I tried to apply quantization to speedup but it sacrifices accuracy instead of speeding up the inference"*.

(4) **Updating data types:** *"Main challenges were to develop a suitable framework of classes to hold the classification and clustering code, that would interface with the existing data structures while maintaining backward compatibility"*, said developer S93. Since ML libraries use optimised data types (e.g. `numpy.array`, `pandas.DataFrame`), the projects adopting the ML libraries also have to adopt these types. However updating the project to use these types might require significant source code changes and could potentially break existing clients.

(5) **Managing data pipelines:** *"We had to build up a data pipeline [33] for each ML algorithm. We could reuse some parts of previous pipelines but most parts were algorithm-specific"*, said developer S91. Selecting an appropriate ML model, requires experimenting with multiple libraries and models. To conduct these experiments, developers create multiple data pipelines to perform *Data collection*, *Model training* and *Model evaluation*. While tools like *DVC* [128] or *MLFlow* [34] facilitate multiple experiment management, our survey respondents highlighted the cost of integrating these tools into the workflow of mature projects.

Observation 6: We found five reasons and eight challenges for retrofitting ML libraries. Among others, we identified reasons such as add brand new functionality (39.29% of respondents), replace existing non-ML techniques with ML (21.43% of respondents). Among others, we identified challenges such as data gathering/labeling (39.29% of respondents), retrofitting ML to edge devices (17.86% of respondents), and managing data-pipeline (10.71% of respondents).

5 IMPLICATIONS

Using our findings we offer an empirically justified set of practical implications for researchers, tool builders, library vendors, hardware vendors and software developers.

5.1 Researchers

R1. Call for more research on Software-2.0 Libraries (RQ1): Our results reveal that the ratio of new Python projects that used a ML library increased from 2% in 2013 to 50% in 2018. This increasing trend highlights the prevalence of Software-2.0 in open source Python software systems. However, current research has focused on usage of libraries in Software-1.0: evolution and maintenance [30, 38, 40, 75, 137], development practices [60], and analysing library usage specific to languages such as C# [106], Java [135]. Our dataset [91] contains a set of 3,340 Python projects that use a single or a combination of ML libraries and the 1,211 GitHub commits where these projects updated or adopted ML library(s). We hope that the research community would use our rich information as a starting point to investigate software evolution and development practices specific to Software-2.0.

R2. New Techniques to analyze and support multi-library projects (RQ2): We observed that 40.10% of the projects use multiple ML libraries at once. Moreover, we found that developers prefer to use multiple libraries to implement specific stages of the ML development workflow (e.g., *Scikit-Learn* for *Data collection*, and *TensorFlow* for *Model training*). Therefore, these multi-library environments pose several unique challenges for developers: the evolution of multi-library systems, partial and multiple library migrations, communication incompatibilities between libraries, inconsistent support for hardware accelerators, etc. These challenges offer a plethora of problems that the research community can address. We provide examples of such multi-library environments in our dataset that contains 1,338 multi-library projects.

R3. New techniques for solving ML model incompatibility (RQ4): When updating ML library version, 32 developers (out of 81) revealed that they retrained their ML models after performing a ML library update due to the binary incompatibility of serialized ML models that was trained on previous library versions. While Software-1.0 works based on source code of the program, Software-2.0 works based on source code of the program and trained ML models. This highlights new research opportunity for researchers to extend the previous research on source code incompatibilities [30, 38, 40, 120, 139] to (i) understand the ML model incompatibility or (ii) explore the interplay between source code and ML model incompatibilities.

R4. Catalog breaking changes for Software-2.0 (RQ4 & RQ5): We found that that 52% of the respondents (out of 81) adapt the source code when updating a ML library. Our results highlight Python-specific (e.g., keyword removing, keyword reordering, convert keyword parameter to positional parameter,etc) or ML library specific breaking changes (e.g., change of computing architecture). Software-1.0 researchers [30, 38, 40] have focused on studying Java applications and created a catalog of the most common breaking changes. Further, Cossette et al. [30] categorize this catalog as fully, partially, or non automatable and Dietrich et al. [38] categorize the breaking changes as binary (and/or) source incompatible. Our dataset [91] contains the commits where a ML library was updated, which can be used to extend the catalog and explore breaking changes for Software-2.0.

R5. **Extend version control systems for Software-2.0 (RQ5):** Our results shows that 32% of respondents use VCS (e.g., GitHub) in the library update process. Software-1.0 researchers have previously studied how the VCS affects the granularity of software changes [23, 77]. However, current VCS systems have limitations when versioning large data files or ML models. ML models are versioned in binary formats, so they are stored as large binary objects (which can waste a lot of disk space) and users cannot understand model changes. Researchers need to extend the current VCS tools to address unique challenges of Software-2.0, study how the proposed extensions impact the evolution of Software-2.0, and study the effectiveness of recently introduced ML specific VCS like DVC [128].

5.2 Tool Builders

T1. **Update ML libraries (RQ3):** We observed several differences on how Software-2.0 developers use ML libraries: (i) they upgrade/downgrade ML libraries more often than *traditional libraries*, (ii) strict upgrades are the most popular among other update kinds (see Table 2), (iii) ML library upgrades/downgrades often result in cascading library updates (see Table 7), (iv) developers often downgrade ML libraries (22.04% of updates). The current research [37, 85] and tooling [28, 47, 86] for library updates in statically-typed languages work from one specific version to another. This highlights blind spots in the current tooling that does not account for the strict and non-strict nature of library updates in Python. Moreover, current techniques that support only upgrades should also support downgrades. To help advance the current tooling, we release 1,055 GitHub commits with ML library (cascading) updates, categorized by update kind. Tool builders [42, 46, 90, 97] can mine our dataset to learn from our mined exemplars.

T2. **Tools for evolving trained ML models (RQ5):** From our survey with 81 developers who performed a ML library update we found that developers need several tools specific for Software-2.0. (i) *Tools for reporting model coverage* (24.59% of respondents) compute coverage of ML model based on the testing data. (ii) *Tools for benchmarking of ML models* (18.03% of respondents) compute and compare metrics such as precision, recall, performance, energy consumption, model bias, with ML models trained on other library versions. (iii) *Impact analysis tools* (8.20% of respondents) examine the impact due to code *and* ML model changes.

T3. **Tools for type-related changes (RQ6)**: We found that when developers retrofit ML libraries they start using optimized datatypes (e.g., `numpy.array`, `pandas.DataFrame`). Our survey also reveals that developers experiment with multiple libraries to find a good trade-off between accuracy and performance. This highlights the need for tools that perform type-related changes (e.g., type migration or library migration). The previous tools [81, 133, 137] that perform type-related changes heavily rely on the type-binding information provided by the compiler. Since in dynamically typed languages (like Python) type binding information is available only at runtime, the type-related change tools need to be extended with type inference techniques.

T4. **Static analysis tools (RQ5):** Our respondents said that static type checkers like *MyPy* and linters (that catch stylistic errors or suspicious constructs) build developer trust. However, *MyPy* requires type annotations to type check the program. To reduce the development effort of manually adding type annotations, Hellendoorn et al. [57] proposed a type annotation inference tool for JavaScript based on deep-learning. Extending this technique for Python and integrating it with *MyPy* will be useful for Software-2.0 developers. Moreover, tool builders can extend tools that identify and eliminate bugs or code smells (e.g., *Error prone* [2] or *JDeodorant* [136]) to identify and repair ML-specific bug patterns proposed by Islam et al. [69, 70].

5.3 Library vendors

L1. Manage increasing number of forks (RQ1): Our results highlight that there is an increasing number of forks of ML Library(s) each year: 0.41% of all newly created forks in 2013 in GitHub were for our studied ML Libraries, and this percentage grew to 3.21% in 2018. We also found that *TensorFlow* is the most forked Python repository on Github with over 78K forks. Similarly, other ML libraries like Keras, Caffe and Scikit-Learn have been forked around 18K times. Zhou et al. [146] observed that forking results in (i) community fragmentation and competition, (ii) lack of synchronization across the forks and the upstream, and (iii) disengagement of valuable contributors from the main project. This highlights a need for library vendors and tool builders to invest resources for mitigating the challenges faced by the ever growing number of forks.

L2. Improve API documentation (RQ6): Developers who lack expertise in mathematics or statistics find it hard to understand the documentation of ML libraries. This is confirmed by 21.43% of respondents from our survey. To fill this gap, library developers should use novel techniques [3, 98] that simplify and summarize documentation of ML libraries.

L3. Add API support for more languages (RQ6): Our survey results show that the unavailability of ML library APIs across all major languages is a significant challenge. While *TensorFlow* supports a wide variety of languages (like C++, Python, Java, C, R, Go and Swift), other ML libraries only support a subset of Python, Java, and C++. The unavailability of ML library APIs across all major languages is an impediment for projects that are trying to retrofit ML. While providing support across all languages is not feasible, library vendors could invest resources to develop bindings for their libraries for multiple languages.

5.4 Hardware vendors

H1. Optimise ML library combinations (RQ2): We observe that 40.10% of the projects use multi ML libraries at once. This highlights a blind spot for hardware manufacturers who are optimising their hardware for one specific library [7, 10, 63, 66] (e.g., *Intel* is optimising its CPUs for TensorFlow and Caffe, *Apple* is optimising TensorFlow, Keras, and Caffe). Our dataset contains popular projects that use multi-libraries. We reveal the most frequently used combinations and also identify the features that developers use in various stages of the ML workflow (Table 6). When designing hardware accelerators for ML tasks, hardware vendors can use our dataset to gain insights into representative patterns of computations in multi-library environments. This helps them prioritize what to optimize on their hardware.

H2. Optimise edge devices (RQ6): We observe that 18% of respondents retrofit ML in applications for edge devices such as mobile and smartwatches. This is challenging because edge devices have limited processing power and energy for running ML algorithms. Our respondents call for hardware optimizations that allow processing on edge devices while maintaining the accuracy of the output.

5.5 Software developers and educators

S1. Rich educational resources (RQ2 and RQ3): Developers learn and educators teach new programming constructs through examples. Robillard et al. [117] studied the challenges of learning APIs and concluded that one of the important factors is the lack of usage examples. Using our dataset of 809,534 total number of ML constructs that we mined in our corpus, developers and educators can learn from real-world examples (e.g., selecting proper values for hyperparameters in ML library APIs is not easy [17]. Developers can use our dataset to learn the values from other codes). Moreover, we also release our data set with 1,055 commits that contain ML library API updates.

S2. Awareness about ML-specific development tools (RQ5): We observed that developers use VCS and CI systems that are specifically designed for Software-1.0 development processes. Even though there exists tools like *DVC* [128], *Pachyderm* [64], and *MLFlow* [34] that are specifically designed for versioning and integrating Software-2.0, these tools are still not popular among developers. We encourage developers to use these novel tools for Software-2.0.

6 THREATS TO VALIDITY

6.1 Internal validity

Did we skew the accuracy of our results with how we collected and analyzed information? The validity of our results primarily depends on how accurately we detected the usage of ML library APIs in the analyzed projects. Because of the dynamic nature of Python, the type binding information (which is required for precise static analysis) is only available at runtime, thus it can be hard to disambiguate similar constructs. We mitigated this threat by building our static analysis tools upon state of the art tools: (i) Jedi [71] (for extracting ML library APIs), and (ii) Python standard AST parser (for parsing the source code). Jedi is a widely used tool for Python static analysis and provides type information. It has 47,300 clients, 4,000 GitHub stars, and it is also used in previous studies [41, 44]. To further increase the trust in our tools, we carefully tested our data extraction and statistics generation tools with unit and end-to-end testing.

The findings in Section 4.3 (i.e., library update kinds and cascading library updates) depend on accuracy of extracting library updates. Unlike other package managers like Maven or Gradle, Python package manager (PIP) does not enforce that projects specify their library versions. Therefore, extracting library version information of Python projects is not straightforward. We mitigated this threat by studying the change history of *requirements.txt* which is the file the developers [49], IDEs [72], and CLI tools [112] use to specify library versions of Python projects. We also used the state-of-the art PyDriller [124] (for mining git history) to conduct longitudinal studies.

The manual coding of the open-ended responses in the survey and API labels according to API description may have introduced subjective bias in the results. To remove the bias that can happen when filtering out cases with an evident motivation, we sought the agreement of the first two authors of the paper. In addition, we achieved an inter-coder agreement of 80% in assigning codes to the survey responses and API labels, before labeling the entire dataset.

6.2 External Validity

Do our results generalize? Out of the top 18,122 Python projects on GitHub, we studied in depth *all* projects which use at least one ML library API (this yielded 3,340 projects). They account for a wide range of application domains, making the results of our study generalizable to other projects in similar domains. However, we only analysed Software-2.0 projects in Python. Developers in other languages (e.g. `C++`, `R` or `Scala`) could have different practices and challenges when maintaining and evolving Software-2.0. However, Python has become the *lingua franca* for many Software-2.0 applications [21, 99].

Moreover, a study of proprietary code-bases might also reveal different results. However, when we surveyed contributors to the projects that we studied in our corpus, 15.03% of our respondents had emails from domains from global IT companies.

6.3 Verifiability

Can others replicate our results? We provide all the necessary details about our study to help others replicate. In particular, we released publicly [91] all studied programs, extracted data, and Python scripts that we used for statistical analysis. The survey questions and emails are available in the companion website.

7 RELATED WORK

We group the related work into two areas: (i) studies on Software-2.0, (ii) studies on Software-1.0 libraries.

7.1 Studies on Software-2.0

There have been several studies on various aspects of Software-2.0. Scully et al. [121] highlight the rapid accumulation of technical debt in Software-2.0. They explore several Software-2.0 specific risk factors including boundary erosion, entanglement, hidden feedback loops, undeclared consumers, and data dependencies. Tang et al. [127] studied 327 code patches in Software-2.0 systems, identified 14 new refactorings and 7 new technical debt categories of Software-2.0 systems. Amershi et al. [8] conducted a case study at Microsoft and identified common workflows and practices for developing Software-2.0. Braiek et al. [21] observed an unprecedented growth of ML libraries. Further, they observed that contributions from both academics and companies fuel the ML ecosystem. Mcintosh et al. [95] studied the power consumption of ML algorithms on mobile devices and revealed that many ML algorithm implementations consume more power to train than to evaluate.

Researchers have conducted several studies based on the StackOverflow posts and bug reports of ML libraries. Abdul et al. [14] observed that the most frequently discussed topics of Software-2.0 are ML algorithms, classification, and training data. Zhang et al. [144] revealed that the top three most frequently asked questions in StackOverflow are program crashes, model migration, and implementation questions. They further observed that the main root causes for many Software-2.0 problems are: API misuse, incorrect hyperparameter selection, GPU computation, static graph computation, and limited debugging support. Islam et al. [69, 70], Sun et al. [125] and Thung et al. [130] categorize StackOverflow posts and GitHub bugs and identify fix patterns for the bugs of Software-2.0. Humbatova et al. [62] introduce a large taxonomy for Software-2.0 bugs using StackOverflow posts and bugs of TensorFlow, Keras, and PyTorch.

Even though there have been many studies on Software-2.0, none of them study the challenges of maintaining and evolving Software-2.0. Using complementary quantitative and qualitative methods, we answer six broad research questions using 3,340 Software-2.0 projects and 109 survey responses. We identify (i) common practices of maintaining and evolving Software-2.0 (see implications R2, T1, & H1 in Section 5.1), (ii) new tool building opportunities to better assist Software-2.0 (see implications T1, T2, R2, & R3 Section 5.1), (iii) opportunities to improve existing infrastructure according to the requirements of Software-2.0 (see implications R4, R5, T3, & T4 in Section 5.1), and (iv) blind spots in current Software-2.0 research (see implications L1, L2, L3, H1, H2, S1 & S2 in Section 5.1).

7.2 Studies on Software-1.0 libraries

7.2.1 ***Studies on software ecosystems:*** Researchers have studied software ecosystems to reveal challenges and common practices. Hora et al. [61] studied how client projects react to the changes of the Pharo ecosystem. They found that some clients want time to discover and apply the API changes while the majority of clients do not act upon the changes at all. Robbes et al. [116] analysed Squeak and Pharo software ecosystems and observe a ripple effect that arises due to API deprecations. McDonnell et al. [94] observed fast evolving APIs are more popular than the slow evolving APIs among the clients in Android ecosystem. Bavota [16] observed an exponential increase of inter-dependencies among projects in the Apache ecosystem. Similarly, with the previous studies, we identify challenges of maintenance and evolution, but in contrast to previous work, we focus on the novel Software-2.0 ecosystem.

7.2.2 ***Studies on library updates:*** Researchers have extensively studied API changes of libraries in Software-1.0 systems. Dig et al. [40] studied two versions of five libraries, classified API changes from the perspective of refactoring and discussed how refactorings lead to breaking API changes in client codes. Cossette et al. [30] explored how representative are the current library update techniques, and classified API changes into fully, partially, and non-automatable. Dietrich et al. [38] classified the Java compile-time and link-time incompatibilities in the Qualitas corpus and reported that such incompatibilities exist but rarely affect client programs. In our research we observed that developers perform API changes that are unique to Software-2.0. Some of the Python-specific API changes involve changes in the name and types of keywords and default values. Also some API changes are ML library-specific and involve changes in the software architecture (e.g., from programming with computational graphs to an imperative style). Our implication R4 in Section 5.1 provides more examples and highlights the need to extend API change research for Software-2.0.

Researches have studied developer reactions to library updates. Kula et al. [84, 85] studied a large corpus of library updates. They observed that updating a dependency is not a common practice for many developers and most systems keep their libraries outdated. Bovita et al. [15] monitored Apache projects for 14 years and found that clients are more likely to update libraries when the library release includes major changes. In contrast to library updates in Software-1.0, in our quantitative analysis we discovered that Software-2.0 developers update ML libraries more frequently than *traditional libraries* (see Section 4.3.1).

Researchers have studied dependency networks and library version conflicts. Nguyen et al. [105] studied the role of dependency network in the prediction of post-release failures. Daniel et al. [54] analysed projects of R ecosystem and observed proportionality between the number of dependencies and its popularity. Decan et al. [35] studied seven library ecosystems and Kikas et al. [82] studied three library ecosystems. They observed that dependency networks and transitive dependencies of projects grow with time. Artho et al. [11] grouped library conflicts into five main categories while Ying et al. [141] found an automated approach to spot dependency conflicts of Python libraries. In this paper, we observed that 95.37% of Software-2.0 systems have transitive dependencies and have complex versioning requirements. This highlights that Software-2.0 might be even more vulnerable to dependency conflicts (see Section 4.4.4).

As discussed above, researchers have studied library updates in Software-1.0 systems from different perspectives. For example, source code adaptation, version conflict, dependency networks, etc. In our quantitative and qualitative analysis, we uncovered that ML library update process shares the same challenges with Software-1.0. Moreover, Software-2.0 processes need to address unique challenges (e.g., retrain ML models, reselect decision thresholds, benchmark ML models, etc. – see implications R2, R3, R5 in Section 5.1 and T2 in Section 5.2).

7.2.3 ***Tools used in library update process:*** Researchers have developed tools to assist developers when maintaining and evolving libraries. API recommendation tools developed for statically-typed languages [59, 103, 104, 115, 131] reduce developer burden when selecting the most appropriate APIs in libraries. D'Souza [44] extended the API recommendation tools for dynamically-typed languages (i.e., Python). Asaduzzaman et al. [12] and Zhan et al. [143] introduced parameter auto-completion with their new tools. Researchers [32, 58, 86, 140] also automated the *source code adaptations* that need to be performed due to library updates.

In this paper we discovered unique challenges (e.g., a high percentage of downgrades, six kinds of updates, cascading library upgrades) for Software-2.0 that require previous tools to be extended (see implications T1 & T3 & T4 in Section 5.2). Moreover, we present several new criteria (e.g., model coverage, model benchmarking, and model impact analysis) for a new generation of tools that address the workflow of Software-2.0 (see implications T2 in Section 5.2).

8 CONCLUSIONS

Developers use learned models to build Software-2.0 systems. This novel paradigm gained substantial popularity over the past years. A major contributor to the rapid growth of ML are the ML libraries as they simplify the huge complexity of implementing Software-2.0 systems. However, the rate of growth will slow without understanding the practices and challenges of maintaining and evolving ML libraries in Software-2.0 systems. In this paper we use complementary empirical methods (mining 3,340 software repositories containing over 809,534 ML constructs, and conducting surveys with 109 avid users of ML libraries) to answer six broad research questions. Some of our key findings are:

(1) Software-2.0 projects that use ML Libraries are rapidly increasing. This is not a fad, but it is an established trend.
(2) Developers use multiple ML libraries to implement ML development workflows.
(3) ML library updates result in cascading library updates.
(4) ML library updates pose more challenges (e.g, binary incompatibility of pre-trained ML models, ML models benchmarking, etc.) than source code adaptations.
(5) Software-2.0 developers use tools from Software-1.0 (e.g., TravisCI, AppVeyor) that are not suitable for their new needs. We discovered several new tools (e.g., reporting model coverage, benchmarking ML models, etc.) for evolving Software-2.0 systems.
(6) Retrofitting ML libraries for mature projects is challenging. We uncover *eight* different barriers including inaccessible data, not-enough processing/battery power in edge devices, and inadequate developer training/documentation.

We hope that this paper serves as a call to action to address unique challenges when maintaining and evolving Software-2.0 systems. Our goal is to inspire a symbiotic ecosystem where researchers, tool builders, library vendors, and hardware vendors work together to assist developers towards creating better Software-2.0 systems.

9 ACKNOWLEDGEMENTS

We thank Grady Booch, Foutse Khomh, Ellick Chan, Johirul Islam, Minsuk Kahng, Rahul Khanna, and the anonymous reviewers for their insightful and constructive feedback for improving the paper. This research was partially funded through the NSF grant CCF-1553741, and by the PPI Center at CU Boulder.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM '12)*. IEEE Computer Society, USA, 14–23. https://doi.org/10.1109/SCAM.2012.28

[3] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 1199–1210. https://doi.org/10.1109/ICSE.2019.00122

[4] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. (2016).

[5] Hussein Alrubaye, Deema AlShoaibi, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. How Does API Migration Impact Software Quality and Comprehension? An Empirical Study. (Jul 2019). arXiv:cs.SE/1907.07724

[6] Theodoros Amanatidis and Alexander Chatzigeorgiou. 2016. Studying the Evolution of PHP Web Applications. *Inf. Softw. Technol.* 72, C (April 2016), 48–67. https://doi.org/10.1016/j.infsof.2015.11.009

[7] AMD. 2020. Deep Learning Solutions. https://www.amd.com/en/graphics/servers-radeon-instinct-deep-learning. Accessed: 2020-02-01.

[8] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19).* IEEE Press, Piscataway, NJ, USA, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[9] Apache. 2019. Frequent Pattern Mining. https://spark.apache.org/docs/latest/ml-frequent-pattern-mining.html#fp-growth. Accessed: 2020-01-07.

[10] Apple. 2020. Machine Learning. https://developer.apple.com/machine-learning/. Accessed: 2020-02-01.

[11] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Why Do Software Packages Conflict?. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12).* IEEE Press, 141–150. https://doi.org/10.1109/MSR.2012.6224274

[12] Muhammad Asaduzzaman, Chanchal K. Roy, Samiul Monir, and Kevin A. Schneider. 2015. Exploring API Method Parameter Recommendations. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15).* IEEE Computer Society, USA, 271–280. https://doi.org/10.1109/ICSM.2015.7332473

[13] Shams Azad, Peter C. Rigby, and Latifa Guerrouj. 2017. Generating API Call Rules from Version History and Stack Overflow Posts. *ACM Trans. Softw. Eng. Methodol.* 25, 4, Article Article 29 (Jan. 2017), 22 pages. https://doi.org/10.1145/2990497

[14] Abdul Ali Bangash, Hareem Sahar, Shaiful Chowdhury, Alexander William Wong, Abram Hindle, and Karim Ali. 2019. What Do Developers Know About Machine Learning: A Study of ML Discussions on StackOverflow. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19).* IEEE Press, Piscataway, NJ, USA, 260–264. https://doi.org/10.1109/MSR.2019.00052

[15] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache Community Upgrades Dependencies: An Evolutionary Study. *Empirical Softw. Engg.* 20, 5 (Oct. 2015), 1275–1317. https://doi.org/10.1007/s10664-014-9325-9

[16] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-Dependencies in a Software Ecosystem: The Case of Apache. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13).* IEEE Computer Society, USA, 280–289. https://doi.org/10.1109/ICSM.2013.39

[17] James Bergstra, Dan Yamins, and David D Cox. 2013. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. Citeseer.

[18] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 334–344. https://doi.org/10.1109/ICSME.2016.31

[19] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016

[20] Houssem Ben Braiek and Foutse Khomh. 2020. On testing machine learning programs. *Journal of Systems and Software* 164 (2020), 110542. https://doi.org/10.1016/j.jss.2020.110542

[21] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The Open-Closed Principle of Modern Machine Learning Frameworks. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18).* Association for Computing Machinery, New York, NY, USA, 353–363. https://doi.org/10.1145/3196398.3196445

[22] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. Qualitative research in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101.

[23] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. 2014. How Do Centralized and Distributed Version Control Systems Impact Software Changes?. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014).* Association for Computing Machinery, New York, NY, USA, 322–333. https://doi.org/10.1145/2568225.2568322

[24] Haipeng Cai. 2020. Assessing and Improving Malware Detection Sustainability through App Evolution Studies. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 8 (March 2020), 28 pages. https://doi.org/10.1145/3371924

[25] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. 2013. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research* 42, 3 (2013), 294–320. https://doi.org/10.1177/0049124113500475

[26] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. 2018. The rise of deep learning in drug discovery. *Drug discovery today* 23, 6 (2018), 1241–1250. https://doi.org/10.1016/j.drudis.2018.01.039

[27] François Chollet et al. 2018. Keras: The python deep learning library. *Astrophysics Source Code Library* (2018).

[28] Kingsum Chow and David Notkin. 1996. Semi-Automatic Update of Applications in Response to Library Changes. In *Proceedings of the 1996 International Conference on Software Maintenance (ICSM '96)*. IEEE Computer Society, USA, 359.

[29] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: a modular machine learning software library*. Technical Report. Idiap.

[30] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article Article 55, 11 pages. https://doi.org/10.1145/2393596.2393661

[31] Daniela S Cruzes and Tore Dybå. 2011. Research synthesis in software engineering: A tertiary study. *Information and Software Technology* 53, 5 (2011), 440–455. https://doi.org/10.1016/j.infsof.2011.01.004

[32] Barthelemy Dagenais and Martin P. Robillard. 2009. SemDiff: Analysis and Recommendation Support for API Evolution. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 599–602. https://doi.org/10.1109/ICSE.2009.5070565

[33] Sato Danilo, Wider Arif, and Windheuser Christoph. 2020. Continuous Delivery for Machine Learning. https://martinfowler.com/articles/cd4ml.html. Accessed: 2020-03-26.

[34] Databricks. 2020. Open source platform for managing the end-to-end machine learning lifecycle. https://www.mlflow.org/docs/latest/projects.html. Accessed: 2020-03-26.

[35] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *Empirical Softw. Engg.* 24, 1 (Feb. 2019), 381–416. https://doi.org/10.1007/s10664-017-9589-y

[36] Dependabot. 2019. Dependabot. https://dependabot.com. Accessed: 13 Jun. 2019.

[37] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2187–2200. https://doi.org/10.1145/3133956.3134059

[38] J. Dietrich, K. Jezek, and P. Brada. 2014. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE Computer Society, Washington, DC, USA, 64–73. https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[39] Danny Dig and Ralph Johnson. 2005. The Role of Refactorings in API Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE Computer Society, USA, 389–398. https://doi.org/10.1109/ICSM.2005.90

[40] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring: Research Articles. *J. Softw. Maint. Evol.* 18, 2 (March 2006), 83–107. https://doi.org/10.1002/smr.328

[41] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for Machine Learning Programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3211346.3211349

[42] Georg Dotzler, Marius Kamp, Patrick Kreutzer, and Michael Philippsen. 2017. More Accurate Recommendations for Method-Level Changes. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 798–808. https://doi.org/10.1145/3106237.3106276

[43] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 779–790. https://doi.org/10.1145/2568225.2568295

[44] Andrea Renika D'Souza, Di Yang, and Cristina V Lopes. 2016. Collective Intelligence for Smarter API Recommendations in Python. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, IEEE Press, Piscataway, NJ, USA, 51–60. https://doi.org/10.1109/SCAM.2016.22

[45] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*. Springer, 285–311.

[46] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 204–215. https://doi.org/10.1145/3293882.3330571

[47] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient Static Checking of Library Updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 791–796. https://doi.org/10.1145/3236024.3275535

[48] Apache Software Foundation. 2019. Introduction to the POM. https://maven.apache.org/guides/introduction/introduction-to-the-pom.html. Accessed: 2019-12-26.

[49] Python Foundation. 2019. Requirements file format. https://pip.pypa.io/en/stable/reference/pip_install/. Accessed: 13 Jun. 2019.

[50] Python Software Foundation. 2019. Python Type Hints. https://docs.python.org/3/library/typing.html. Accessed: 13 Oct. 2019.

[51] Python Software Foundation. 2019. Version Identification and Dependency Specification. https://www.python.org/dev/peps/pep-0440/. Accessed: 2019-08-23.

[52] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[53] Martin Fowler. 2019. Continuous Delivery for Machine Learning. https://martinfowler.com/articles/cd4ml.html. Accessed: 2020-02-14.

[54] Daniel M. German, Bram Adams, and Ahmed E. Hassan. 2013. The Evolution of the R Software Ecosystem. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE Computer Society, USA, 243–252. https://doi.org/10.1109/CSMR.2013.33

[55] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/342009.335372

[56] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. 2020. What do Programmers Discuss about Deep Learning Frameworks. *Empirical Software Engineering* (2020). https://doi.org/10.1007/s10664-020-09819-6

[57] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[58] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 274–283. https://doi.org/10.1145/1062455.1062512

[59] Rosco Hill and Joe Rideout. 2004. Automatic Method Completion. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*. IEEE Computer Society, USA, 228–235. https://doi.org/10.1109/ASE.2004.1342740

[60] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

[61] Andre Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Tulio Valente. 2015. How Do Developers React to API Evolution? The Pharo Ecosystem Case. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, Washington, DC, USA, 251–260. https://doi.org/10.1109/ICSM.2015.7332471

[62] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. (2020), 11. https://doi.org/10.1145/3377811.3380395

[63] IBM. 2020. Get started with PyTorch. https://developer.ibm.com/articles/cc-get-started-pytorch/. Accessed: 2020-02-01.

[64] Pachyderm Inc. 2019. Pachyderm. https://www.pachyderm.com/platform/#community. Accessed: 2020-02-14.

[65] Intel. 2019. Intel Caffe. https://github.com/intel/caffe. Accessed: 2019-11-19.

[66] Intel. 2019. Intel Optimization for TensorFlow*. https://software.intel.com/en-us/frameworks/tensorflow. Accessed: 2020-01-03.

[67] Intel. 2019. Intel TensorFlow. https://software.intel.com/en-us/frameworks/tensorflow. Accessed: 2019-11-19.

[68] Intel. 2019. Intel Theano. https://github.com/intel/Theano. Accessed: 2019-11-19.

[69] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery,

New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[70] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 11. https://doi.org/10.1145/1122445.1122456

[71] JEDI. 2019. Jedi - an awesome autocompletion/static analysis library for Python. https://jedi.readthedocs.io/en/latest/. Accessed: 2019-08-23.

[72] Jetbrains. 2019. Managing Dependencies. https://www.jetbrains.com/help/pycharm/managing-dependencies.html. Accessed: 13 Jun. 2019.

[73] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[74] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. 2017. Why and how developers fork what from whom in GitHub. *Empirical Software Engineering* 22, 1 (01 Feb 2017), 547–578. https://doi.org/10.1007/s10664-016-9436-6

[75] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging Library Migrations: A Case Study for the Apache Software Foundation Projects. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. Association for Computing Machinery, New York, NY, USA, 154–164. https://doi.org/10.1145/2901739.2901769

[76] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. 2006. Mining Sequences of Changed-Files from Version Histories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. Association for Computing Machinery, New York, NY, USA, 47–53. https://doi.org/10.1145/1137983.1137996

[77] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M. German. 2015. Open Source-Style Collaborative Development Practices in Commercial Projects Using GitHub. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 574–585.

[78] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2016. An In-depth Study of the Promises and Perils of Mining GitHub. *Empirical Softw. Engg.* 21, 5 (Oct. 2016), 2035–2071. https://doi.org/10.1007/s10664-015-9393-5

[79] Ritu Kapur and Balwinder Sodhi. 2020. A Defect Estimator for Source Code: Linking Defect Reports with Programming Constructs Usage Metrics. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 12 (April 2020), 35 pages. https://doi.org/10.1145/3384517

[80] Andrej Karpathy. 2017. Software 2.0. https://medium.com/@karpathy/software-2-0-a64152b37c35. Accessed: 2020-05-05.

[81] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. 2019. Type Migration in Ultra-Large-Scale Codebases. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 1142–1153. https://doi.org/10.1109/ICSE.2019.00117

[82] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, 102–112. https://doi.org/10.1109/MSR.2017.55

[83] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/1985793.1985815

[84] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, and Katsuro Inoue. 2015. Trusting a library: A study of the latency to adopt the latest maven release. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 520–524. https://doi.org/10.1109/SANER.2015.7081869

[85] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[86] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. 2015. SWIN: Towards Type-Safe Java Program Adaptation between APIs. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation (PEPM '15)*. Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/2678015.2682534

[87] Yangguang Li, Zhen Ming (Jack) Jiang, Heng Li, Ahmed E. Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. Predicting Node Failures in an Ultra-Large-Scale Cloud Computing Platform: An AIOps Solution. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 13 (April 2020), 24 pages. https://doi.org/10.1145/3385187

[88] Jimmy Lin and Alek Kolcz. 2012. Large-scale Machine Learning at Twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 793–804. https://doi.org/10.1145/2213836.2213958

[89] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 296–305. https://doi.org/10.1145/1095430.1081754

[90] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. 2017. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*. Springer, 229–246. 10.1007/978-3-319-66399-9_13

[91] Malinda. 2020. Study of Machine Learning Library Usage. https://serene-beach-16261.herokuapp.com. Accessed: 2020-01-14.

[92] Brian A Malloy and James F Power. 2017. Quantifying the transition from Python 2 to 3: an empirical study of Python applications. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 314–323. https://doi.org/10.1109/ESEM.2017.45

[93] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang.* 1, OOPSLA, Article Article 85 (Oct. 2017), 31 pages. https://doi.org/10.1145/3133909

[94] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, USA, 70–79. https://doi.org/10.1109/ICSM.2013.18

[95] Andrea Mcintosh, Safwat Hassan, and Abram Hindle. 2019. What Can Android Mobile App Developers Do about the Energy Consumption of Machine Learning? *Empirical Softw. Engg.* 24, 2 (April 2019), 562–601. https://doi.org/10.1007/s10664-018-9629-2

[96] Erik Meijer. 2018. Behind Every Great Deep Learning Framework is an Even Greater Programming Languages Concept (Keynote). In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3236024.3280855

[97] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 502–511.

[98] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2016. ARENA: an approach for the automated generation of release notes. *IEEE Transactions on Software Engineering* 43, 2 (2016), 106–127. https://doi.org/10.1109/TSE.2016.2591536

[99] Andreas C Müller, Sarah Guido, et al. 2016. *Introduction to machine learning with Python: a guide for data scientists.* " O'Reilly Media, Inc.".

[100] Mypy. 2019. mypy. http://mypy-lang.org/. Accessed: 2019-08-23.

[101] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 466–476. https://doi.org/10.1145/2491411.2491415

[102] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan. 2019. Speech Recognition Using Deep Neural Networks: A Systematic Review. *IEEE Access* 7 (2019), 19143–19165. https://doi.org/10.1109/ACCESS.2019.2896880

[103] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2012. GraPacc: A Graph-Based Pattern-Oriented, Context-Sensitive Code Completion Tool. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 1407–1410.

[104] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 69–79. https://doi.org/10.1109/ICSE.2012.6227205

[105] Thanh H. D. Nguyen, Bram Adams, and Ahmed E. Hassan. 2010. Studying the Impact of Dependency Network Measures on Software Quality. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/ICSM.2010.5609560

[106] Semih Okur and Danny Dig. 2012. How Do Developers Use Parallel Libraries?. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article Article 54, 11 pages. https://doi.org/10.1145/2393596.2393660

[107] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

[108] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[109] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for

Computing Machinery, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

[110] PIP. 2019. Requirements. https://pip.readthedocs.io/en/1.1/requirements.html. Accessed: 2019-08-23.

[111] Python. 2019. PEP Purpose and Guidelines. https://www.python.org/dev/peps/pep-0001/. Accessed: 13 Jun. 2019.

[112] Python. 2019. PIP User Guide. https://pip.pypa.io/en/stable/user_guide/. Accessed: 2019-08-23.

[113] Realpython. 2019. what is pip. https://realpython.com/what-is-pip/. Accessed: 2019-08-23.

[114] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 432–448. https://doi.org/10.1145/1028976.1029012

[115] R. Robbes and M. Lanza. 2008. How Program History Can Improve Code Completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, USA, 317–326. https://doi.org/10.1109/ASE.2008.42

[116] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 56, 11 pages. https://doi.org/10.1145/2393596.2393662

[117] Martin P. Robillard and Robert Deline. 2011. A Field Study of API Learning Obstacles. *Empirical Softw. Engg.* 16, 6 (Dec. 2011), 703–732. https://doi.org/10.1007/s10664-010-9150-8

[118] Peter Rousseeuw. 1987. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *J. Comput. Appl. Math.* 20, 1 (Nov. 1987), 53–65. https://doi.org/10.1016/0377-0427(87)90125-7

[119] A. Roy, J. Sun, R. Mahoney, L. Alonzi, S. Adams, and P. Beling. 2018. Deep learning detecting fraud in credit card transactions. In *2018 Systems and Information Engineering Design Symposium (SIEDS)*. IEEE Press, Piscataway, NJ, USA, 129–134. https://doi.org/10.1109/SIEDS.2018.8374722

[120] Thorsten Schäfer, Jan Jonas, and Mira Mezini. 2008. Mining Framework Usage Changes from Instantiation Code. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 471–480. https://doi.org/10.1145/1368088.1368153

[121] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 2503–2511.

[122] Pranab Kumar Sen. 1968. Estimates of the Regression Coefficient Based on Kendall's Tau. *J. Amer. Statist. Assoc.* 63, 324 (1968), 1379–1389.

[123] Janice Singer, Susan E Sim, and Timothy C Lethbridge. 2008. Software engineering data collection for field studies. In *Guide to Advanced Empirical Software Engineering*. Springer, 9–34.

[124] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[125] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 2017. An empirical study on real bugs for machine learning programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, IEEE Press, Piscataway, NJ, USA, 348–357. https://doi.org/10.1109/APSEC.2017.41

[126] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2005. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[127] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *International Conference on Software Engineering (ICSE '21)*. ACM/IEEE. To appear.

[128] DVC Team. 2020. Open-source Version Control System for Machine Learning Projects. https://dvc.org. Accessed: 2020-03-26.

[129] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A Study of Library Migrations in Java. *J. Softw. Evol. Process* 26, 11 (Nov. 2014), 1030–1052. https://doi.org/10.1002/smr.1660

[130] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE '12)*. IEEE Computer Society, USA, 271–280. https://doi.org/10.1109/ISSRE.2012.22

[131] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. 2013. Automatic Recommendation of API Methods from Feature Requests. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, 290–300. https://doi.org/10.1109/ASE.2013.6693088

[132] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

[133] Frank Tip, Robert M. Fuhrer, Adam Kieundefinedun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring Using Type Constraints. *ACM Trans. Program. Lang. Syst.* 33, 3, Article Article 9 (May 2011), 47 pages. https://doi.org/10.1145/1961204.1961205

[134] Paolo Tonella. 2003. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. *IEEE Trans. Softw. Eng.* 29, 6 (June 2003), 495–509. https://doi.org/10.1109/TSE.2003.1205178

[135] Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor. 2011. Are Java Programmers Transitioning to Multicore? A Large Scale Study of Java FLOSS. In *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11 (SPLASH '11 Workshops)*. Association for Computing Machinery, New York, NY, USA, 123–128. https://doi.org/10.1145/2095050.2095072

[136] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering (CSMR '08)*. IEEE Computer Society, USA, 329–331. https://doi.org/10.1109/CSMR.2008.4493342

[137] Victor L. Winter and Azamat Mametjanov. 2007. Generative Programming Techniques for Java Library Migration. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE '07)*. Association for Computing Machinery, New York, NY, USA, 185–196. https://doi.org/10.1145/1289971.1290001

[138] Claes Wohlin and Aybüke Aurum. 2015. Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering* 20, 6 (2015), 1427–1455.

[139] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: A Hybrid Approach to Identify Framework Evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1806799.1806848

[140] Zhenchang Xing and Eleni Stroulia. 2007. API-Evolution Support with Diff-CatchUp. *IEEE Trans. Softw. Eng.* 33, 12 (Dec. 2007), 818–836. https://doi.org/10.1109/TSE.2007.70747

[141] Wang Ying, Wen Ming, Liu Yepang, Wang Yibo, Li Zhenming, Wang Chao, Yu Hai, Cheung Shing-Chi, Xu Chang, and Zhu Zhiliang. 2020. Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem. In *In Proceedings of 42nd International Conference on Software Engineering (ICSE '20)*. ACM, New York, NY, USA, 11. https://doi.org/10.1145/3377811.3380426

[142] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. 2019. Characterizing the Usage, Evolution and Impact of Java Annotations in Practice. *IEEE Transactions on Software Engineering* (2019), 1–1. https://hal.inria.fr/hal-02091516

[143] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. 2012. Automatic Parameter Recommendation for Practical API Usage. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 826–836.

[144] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, IEEE Press, Piscataway, NJ, USA, 104–115. https://doi.org/10.1109/ISSRE.2019.00020

[145] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[146] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub. In *In Proceedings of 42nd International Conference on Software Engineering (ICSE '20)*. ACM, New York, NY, USA, 11. https://doi.org/10.1145/3377811.3380426

[147] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. 2005. Mining Version Histories to Guide Software Changes. *IEEE Trans. Softw. Eng.* 31, 6 (June 2005), 429–445. https://doi.org/10.1109/TSE.2005.72