Discovering Repetitive Code Changes in Python ML Systems

Malinda Dilhara
malinda.malwala@colorado.edu
University of Colorado
USA

Ameya Ketkar
ketkara@oregonstate.edu
Oregon State University
USA

Nikhith Sannidhi
nikhith.sannidhi@colorado.edu
University of Colorado
USA

Danny Dig
danny.dig@colorado.edu
University of Colorado
USA

ABSTRACT

Over the years, researchers capitalized on the repetitiveness of software changes to automate many software evolution tasks. Despite the extraordinary rise in popularity of Python-based ML systems, they do not benefit from these advances. Without knowing what are the repetitive changes that ML developers make, researchers, tool, and library designers miss opportunities for automation, and ML developers fail to learn and use common practices.

To fill the knowledge gap and advance the science and tooling in ML software evolution, we conducted the first and most fine-grained study on code change patterns in a diverse corpus of 1000 top-rated ML systems comprising 58 million SLOC. To conduct this study we reuse, adapt, and improve upon the state-of-the-art repetitive change mining techniques. Our novel tool, R-CPatMiner, mines over 4M commits and constructs 350K fine-grained change graphs and detects 28K change patterns. Using thematic analysis, we identified 22 pattern groups and we reveal 4 major trends of how ML developers change their code. We sent a survey to 650 ML developers to further shed light on these patterns and their applications. We present actionable, empirically-justified implications for four audiences: (i) researchers, (ii) tool builders and IDE designers, (iii) ML library vendors, and (iv) developers and educators.

ACM Reference Format:
Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2021. Discovering Repetitive Code Changes in Python ML Systems. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many software changes are repetitive by nature [6, 30, 49], thus forming change patterns. Like in traditional software systems, Machine Learning (ML) developers perform repetitive code changes too. For example, Listing 1 shows a common change where ML developers replaced a `for` loop that sums the list `elements` with `np.sum`, a highly optimized domain-specific abstraction provided by the library *NumPy* [56]. Since this change involves programming idioms [1, 73] at the sub-method level it is *fine-grained*. If this code change is repeated at multiple locations or in multiple commits, it is a *fine-grained code change* ***pattern***.

Listing 1: Commit c8b28432 in GitHub repository NifTK/NiftyNet: Replace `for` loop with NumPy `sum`

```
-for elem in elements:
-    result += elem
+result = np.sum(elements)
```

Over the years, researchers in the traditional software systems have provided many applications that rely upon the repetitiveness of changes: code completion in the IDEs [12, 31, 39, 51, 52], automated program repair [5, 8, 46], API recommendation [29, 51], type migration [37], library migration [2, 17, 22, 36, 75], code refactoring [18, 26], fine-grained understanding of software evolution [2, 23, 38, 50, 53, 70, 76]. Unfortunately, these are mostly available only for Java, and do not support Python and ML systems.

Researchers [10, 20, 34, 64] observed that Python dominates the ML ecosystem in both the company-driven and the community-driven ML software systems, yet the tooling is significantly behind [20, 82]. In order to advance the science and tooling for ML code development in Python, we need to understand how developers evolve and maintain ML systems. Previous researchers have focused on high-level software evolution tasks like identifying ML bugs [32, 34, 35], updating ML libraries [20], refactoring and technical debt of ML systems [68, 74], managing version control systems for data [7], and testing [9, 28, 33]. However, there is a lack of understanding of the repetitive fine-grained code change patterns that ML developers laboriously perform. *What are fine-grained changes performed in ML systems? Which ones are ML-specific? What kinds of automation do ML developers need?*

Without answers to such questions, researchers miss opportunities to improve the state-of-the-art in automation for software evolution in ML systems, tool builders do not invest resources where automation is most needed, language and library designers cannot make informed decisions when introducing new constructs, and ML developers fail to learn and use common practices.

In this paper, we conduct the first large-scale study and discover repetitive change patterns in Python-based ML systems. We employ both quantitative (mining repositories and thematic analysis) and qualitative methods (surveys) for answering these questions. Blending these methods has the advantage of the results being triangulated. The quantitative methods help us discover *what* fine-grained change patterns ML developers perform. The qualitative method helps us answer *why* these changes are performed, how they are performed, and how tool builders can improve ML developer productivity.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For the quantitative analysis, we use a large data set of 1000 Python ML projects from GitHub, comprising 58 million source lines of code (SLOC) at the latest revisions, 1.16 million mapped code change blocks, 1.5 million changed files, and 0.4 billion changed SLOCs. We extracted 28,308 fine grained code change patterns where 58% of them appear in multiple projects. We applied *thematic analysis* [11, 79] upon 2,500 most popular patterns from our dataset, and categorized them into 22 fine-grained change pattern themes that reveal 4 major trends.

Moreover, we designed and conducted a survey with 650 ML developers and we presented 1,235 patterns for their opinion. In the survey, 71% of the developers confirmed the need of automation for 22 pattern groups. Among these, we discovered four major trends: (1) *transform to Context managers* (e.g., disable or enable gradient calculation, swap ML training device, etc.), (2) *convert `for` loops to domain specific abstraction* (e.g., see Listing 1), (3) *update API usage* (e.g., migrate to `TensorFlow.log` from `log`, transform matrices), and (4) *use advanced language features* (e.g., transform to Python `list` comprehension).

The main challenge in conducting such large-scale, representative studies, is the lack of tools for mining non-Java repositories. To overcome this challenge we reuse, adapt, and extend the vast ecosystem of Java AST-level analysis tools [2, 23, 38, 50, 53, 70, 76] to Python. Most of these tools rely on techniques that are conceptually language-independent, i.e., they operate on intermediate representation of the code (e.g., AST nodes). Second, we observed that 72% of the Python AST node kinds identically overlap with those in Java (e.g., *While-Statment*, *Assignment-Statement*, etc.). Moreover, another 18% of Python AST node kinds also exist in Java with some differences (e.g., Python's `for` loop has multiple loop variables). Only 10% of the Python AST node kinds are unique to Python (e.g., `With` statement, `Generators`, etc.). Hence, one of our key ideas is to reuse the Java AST-level analysis tools to analyse 72% of the Python AST nodes and for the remaining 28% of AST nodes we either modify existing capabilities or add brand new ones.

We first developed a novel technique, JavaFyPy, to transform Python AST to a format that can be processed by Java AST-level mining tools. We used JavaFyPy to adapt to Python the state-of-the-art fine-grained change pattern mining tool, CPatMiner [53]. CPatMiner matches changed methods and their body statements across the commits and identifies fine-grained change patterns. Refactorings such as move, rename, and extract shift and obfuscate the code statements, and can no longer be matched, thus causing CPatMiner to miss several instances of patterns. To improve the accuracy of CPatMiner, we integrate it with the state-of-the-art refactoring mining technique- RefactoringMiner [76], that de-obfuscates the shifted code statements. Our novel tool R-CPatMiner performs *refactoring-aware*, *fine-grained* change pattern mining in the commit history of Python systems.

Our findings and tools have actionable implications for several audiences. Among others, they (i) advance our understanding of repetitive changes that the ML developers perform which helps the research community to improve the science and tools for ML software evolution, (ii) provide a rich infrastructure to automate and significantly extend the scope of existing studies on ML systems [34, 35, 68] (iii) help tool builders comprehend the ML developers' struggles and desire for automation, (iv) provide feedback to language and API designers when introducing new ML constructs, and (v) assist educators in teaching ML software evolution.

This paper makes the following contributions:

(1) To the best of our knowledge, we conducted the first and the largest study on fine-grained 28,308 code change patterns on ML systems. We identified code changes patterns. We applied *thematic analysis* on 2,500 most popular patterns and categorized them into 22 fine-grained change pattern themes that reveal 4 major trends.
(2) We designed and conducted a *survey* with 650 open-source ML developers to provide insights about the reasons motivating those changes, the current practices of applying those changes, and their recommendation for tool builders.
(3) We developed novel tools to collect fine-grained change patterns applied in the evolution history of Python-based ML systems. We applied these tools on 1000 open-source projects hosted on GitHub. We make the collected information and tooling publicly available for reuse [4] so that we enable further research.
(4) We present an *empirically-justified* set of *implications* of our findings from the perspective of four audiences: researchers, tool builders, language designers, and ML developers.

2 MOTIVATING EXAMPLE

Listing 2: Specifies the device (CPU) for operations executed in the context and move method `_init_model` to parent class

```
1 class _FERNeuralNet():
2 +   def _init_model(self):
3 +       with tf.device('/cpu:0'):
4 +           B, H, T, _ = q.get_shape().as_list()
5 ...
6 class TimeDelayNN(_FERNeuralNet):
7 -   def _init_model(self):
8 -       B, H, T, _ = q.get_shape().as_list()
```

The code change shown in Listing 2 specifies the hardware device using `tf.device()` (line 3) for the *TensorFlow* operation in line 4. `tf.device()` is a Context Manager [58] from the ML library, *TensorFlow*. This is a fine-grained code change and the developer has interleaved this with a *Pull up Method* refactoring that pulls `_init_model` from `TimeDelayNN` into the parent class `_FERNeuralNet`.

Is specifying hardware device for *TensorFlow* operations a *pattern*? How frequent is this pattern? Do developers need tool support to recommend and automate this *code change pattern*? We consider this *fine-grained code change instance* a repeated *pattern* if a similar code change was performed in the history of this project or any other project. Researchers have proposed advanced techniques to mine such fine-grained change patterns from the commit histories [53, 54]. However, these techniques are inapplicable to mine the *fine-grained code change patterns* shown in Listing 2 because (1) their techniques mine code change patterns for Java, and (2) they do not account for overlapping refactorings.

Researchers [47, 48, 71] have shown that developers often interleave many programming activities such as bug fixes, feature additions, or other refactoring operations, and often these changes overlap [50] (as shown in Listing 2). Such overlapping changes and refactorings can easily obfuscate existing fine-grained change pattern mining tools [53, 54] because they do not account for these changes when matching code across the commit. For example, CPatMiner [53] does not match the method body of `_init_model` in the class `_FERNeuralNet` (lines 3–4) to the body of `_init_model`

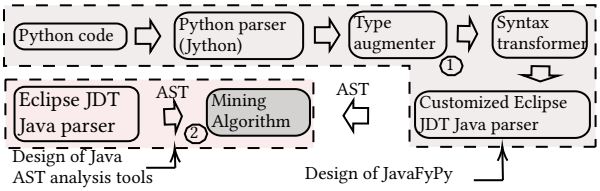


Figure 1: Design of JavaFyPy and existing AST analysis tools

in the class `TimeDelayNN` (line 7) as they are in different locations and different files. This lack of *refactoring awareness* is a serious limitation of existing pattern mining algorithms because they can miss several concrete instances of change patterns that are obfuscated by overlapping refactorings.

Re-implementing the existing Java AST mining tools for Python will require a significant amount of development effort. It is also neither feasible nor sustainable as researchers are continuously implementing new Java AST mining tools or improving existing tools. For this purpose, we propose JavaFyPy, a technique to adapt existing Java AST mining tools to Python that leverages the similarity between the Java and Python abstract syntax trees (AST). We use JavaFyPy to adapt the state-of-the-art fine-grained change pattern mining tool, CPatMiner [53], to Python. To make CPatMiner *refactoring aware*, we adapt the state-of-the-art Java refactoring inference tool, *RefactoringMiner* [76] (known as RMiner), to Python and integrate it with CPatMiner as R-CPatMiner. Particularly, the code-block mapping pairs (i.e., two versions of the same code-block in a method before and after the change) reported by RMiner are provided as input to CPatMiner. R-CPatMiner mines change patterns in Python software systems in a refactoring-aware manner.

3 TECHNIQUE

Most of the current code change mining tools (i.e. AST mining tools) are conceptually language-independent because they operate upon the abstract syntax trees (AST) only. However, their implementation is bound only to Java. To overcome this practical implementation limitation, we propose a very pragmatic solution - JavaFyPy, a technique that transforms the input Python program to an AST that can be processed by the mining algorithm of existing Java AST analysis tools. JavaFyPy will fast-track researchers and tool builders by making the AST-based mining tools implemented for Java programs applicable for Python programs. Thus, it will save several development-hours of work required for re-implementing these techniques. As shown in (1) in Figure 1, JavaFyPy takes a Python code as an input and produces an AST, that can be used in mining algorithms of Java AST analysis tools. To achieve this, JavaFyPy first transforms the Python code to AST and enriches the AST by augmenting type information. Then, the *Syntax transformer* maps the corresponding Java concrete syntax to the AST nodes. The Java parser (Eclipse JDT) uses it to produce the final AST. Eclipse JDT is the most popular Java parser used in AST mining research tools. Therefore, we selected Eclipse JDT as the parser that produces the final AST. This *enhanced and enriched* AST can be processed by the mining algorithms of Java AST analysis tools. Tool builders and researchers can use JavaFyPy, and extend their tools for Python.

3.1 Python code transformation

As shown in Figure 2 JavaFyPy first *parses* the input Python program to an AST. We define an AST as:

Definition 3.1. (AST) Let T be an AST. T has one root. Each node $N_i \in T$ has a parent (except the root node). Each node ($N_i \in T$) has a sequence of child nodes (denoted by C_{N_i}). Number of nodes in the sequence C_{N_i} is denoted by $Length_{C_{N_i}}$. Each node N_i is a specific syntax category known as AST node kind, $Kind_{N_i}$ = {Assignment Statement, For statement, Method Invocation ...}.

We leverage the syntactic similarity between Python and Java to adapt the Java AST analysis tools to Python. We thoroughly studied the Java and Python language specifications [57, 61] and *mapped* the Python AST node kinds to those in Java based on the description in the specifications.

Definition 3.2. (Mapped AST node) Let T_j be a Java AST and T_p be a Python AST. $N_j \in T_j, N_p \in T_p$. We state that N_j and N_p are mapped AST node kind, if N_j and N_p maintain a structural similarity. Mapped node of node N_p is denoted by $M(N_p) = N_j$.

We found three kinds of mappings namely, *Identical AST node*, *Nearly identical AST node*, and *Unique AST node.*

Definition 3.3. (Identical AST node) Let C_{N_j} be a sequence of child AST nodes of a parent Java node N_j and C_{N_p} be a sequence of child AST nodes of a Python node N_p. We state that N_j and N_p are identical AST nodes if (i) $M(N_p)$ is N_j, and (ii) $\forall N_i \in C_{N_p} : M(N_p) \in C_{N_j}$.

(1) ***Identical AST node*** (Definition 3.3) - 72% of the Python's AST node Kinds can be identically mapped to a Java's AST Node. For example, we mapped Python's If to Java's If Statement and mapped Python's Assign to Java's AssignmentStatement.

Definition 3.4. (Nearly Identical AST node) We state that N_j and N_p as nearly identical AST nodes, if N_j and N_p meets conditions: (i) $M(N_p)$ is N_j, and (ii) $\exists N_i \in C_{N_p} : M(N_i) \notin C_{N_j}$.

(2) ***Nearly identical AST node*** (Definition 3.4) - 18% of Python's AST Node kinds could be *partially* mapped to those of Java. For instance, both Python and Java provide the `for` construct to iterate over a collection, however unlike Java, Python allows to iterate over multiple variables (see the `for` loop in Figure 2), thus AST of Python `for` loop contains additional child AST node kinds.

Definition 3.5. (Unique AST node) Let N_p be a Python AST node. We state that N_p is unique to Python, if there is no mapped AST node in T_j. i.e., ($M(N_p) \notin T_j$).

(3) ***Unique AST node*** - 10% of the Python nodes had no Java counterpart. For instance, Java does not support `list` comprehensions or `yield` statement (as shown in Figure 2)

As we can observe, Java and Python syntax significantly overlaps. As shown in (2) in Figure 1, AST mining tools like CPatMiner and RMiner parse the input program to Eclipse JDT AST. To adapt their tools to Python with JavaFyPy, tool builders or researchers simply need to migrate their Java parser to our technique (JavaFyPy). After that, we can simply reuse tools' AST-based mining algorithms to analyse 72% of the *Identical AST node* kinds, and modify the current

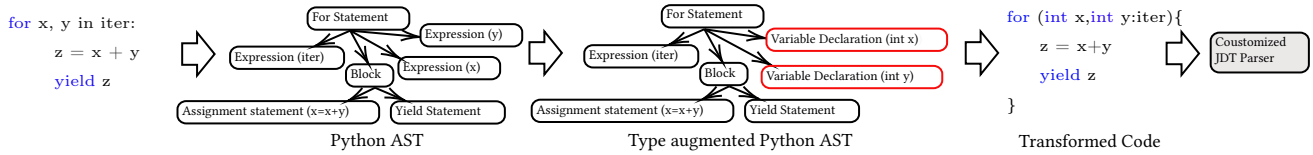


Figure 2: An example Code transformation performed by JavaFyPy

implementation to accommodate the 18% *Nearly identical AST node* kinds and add brand-new capabilities (often involving adding new *visitors*) for handling the 10% *Unique AST node* kinds. After the changes, the tools take Python code as an input and infer the results, thus adapting Java AST mining tools to Python.

Figure 2 shows an example of the code transformation steps (shown in Figure 1) that JavaFyPy performs automatically. The Python code snippet in Figure 2 contains all three AST node kinds: *Identical AST node* (`z = x + y`), *Nearly identical AST node* (`for` loop), and *Unique AST node* (`yield z`). The *Java parser* first constructs the AST of the code snippet, then the *Type Augmenter* augments the AST with type information by adding *Variable Declaration* nodes. This step is important because the Java-based AST mining tools like [53, 76] rely on the syntactic richness that the Java language offers. Unlike Python, Java programmers have to explicitly declare the types of variables, fields and methods. To add this syntactic richness to the input program, JavaFyPy augments the AST of the input program with type information (shown in Figure 2 as red nodes). We obtain this type information from PyType [27], the state-of-the-practice type inference tool for Python developed by Google, which is widely adapted by the Python community. As the last step, *Syntax Transformer* transforms the AST to code and passes it to our customized *Eclipse JDT* parser which we extended to parse *Nearly identical AST node* kinds and *Unique AST node* kinds.

Can JavaFyPy effectively transform all Identical, Nearly Identical, and Unique AST node kinds? We evaluated this empirically with 14 popular Python projects including *TensorFlow*, *PyTorch*, *Keras*, *NLTK*, *Scikit learn*, *Scipy*, and *NumPy* that comprise 23K Python files and 2.9M SLOC. We transformed all the Python files in the projects that consist of 12M Python AST nodes, and checked whether all AST nodes are successfully mapped and transformed to the output AST of JavaFyPy. This confirms that JavaFyPy can effectively transform any input Python program to an Eclipse JDT format.

3.2 Refactoring Aware Change Pattern Mining

3.2.1 ***Adapting CPatMiner.*** CPatMiner [53] is the state-of-the-art code change pattern mining tool that uses an efficient graph-based representation of code changes to mine previously unknown fine-grained changes from git commit history. It iterates changed methods in each commit and uses *Eclipse JDT Java parser* [25] to generate AST of Java source code. Then, its mining algorithm builds program-dependence graphs for each AST node independently and then merges the graphs to create one big graph, called **change graph**. CPatMiner builds *change graphs* for each changed method, and it represents the before and after a source code change that can be used to mine code change patterns. Since 72% of the Python AST node kinds overlap identically with those in Java, we reused most of the capabilities for building the change graphs. We added new capabilities in CPatMiner to create program-dependence graphs for *Unique AST nodes*, and modified the existing capabilities of *Nearly identical AST nodes*. Overall, we extended CPatMiner with 2% extra code lines due to the new or modified capabilities, and reused the rest of the code. While this ratio might be different when adapting other tools, it showcases the merit of JavaFyPy to reuse Java AST-based mining tools for Python.

3.2.2 ***Introducing Refactoring Awareness.*** As discussed in Section 2, CPatMiner [53] does not account for the overlapping refactorings applied in the commit. These refactorings move code blocks between methods or change the method signature, making it hard to match the changed code blocks. Thus missing the opportunities to build *change graphs* for the obfuscated changes. To overcome this, we made the CPatMiner refactorings aware by integrating it with RMiner [76]. We used JavaFyPy to adapt RMiner and use it to detect 18 refactoring kinds that move code blocks. RMiner uses AST-based statement matching algorithm to match classes, methods, and statements inside method bodies, thus helping us match moved code blocks. We consult the authors of RMiner and extend its statement matching algorithm to reason about the *Unique* and *Nearly identical* AST Node kinds. For example, Listing 3 shows a variable rename refactoring in List Comprehension, a Python *Unique AST node* kind of the project "Deepmedic" detected by Python-adapted RMiner.

Listing 3: Commit 8d4be555 in DeepMedic: Variable rename in List Comprehension detected by Python-adapted RMiner

```
- indices = [layerNum - 1 for layerNum in layers_norm]
+ indices = [layer_num - 1 for layer_num in layers_norm]
```

We use Python adapted RMiner to accurately match the moved code blocks. We extended CPatMiner to build change graphs for the code block pairs reported by RMiner. Hence, CPatMiner no longer misses obfuscated code-blocks that contain fine-grained changes. We developed the tool **R-CPATMiner**, to efficiently and accurately mines source code change patterns in the version histories of Python software systems, in a refactoring-aware manner.

4 RESEARCH METHODOLOGY

We first evaluate the effectiveness of the tools we developed (or adapted). Then, we use our reliable and validated tools[1], to explore the repetitive code changes applied in Python ML Systems. For this purpose, we answer three research questions:

RQ1. *What are the frequent code change patterns in ML code, and what patterns need automation?* To answer this research question, we triangulate complementary empirical methods, as shown in Figure 3. (i) We mined 1000 repositories using R-CPatMiner and extracted 28,308 patterns. (ii) We applied thematic analysis on 2,500 patterns. (iii) We sent a survey to 650 ML developers to seek their opinion on automating the identified code change patterns.

[1]We prefix all the adapted tool names with *Py* to disambiguate the tool names from their Java counterparts

RQ2. *How does the refactoring awareness improve the pattern mining over the baseline CPatMiner?* R-CPatMiner performs refactoring aware change pattern mining, thus improving baseline CPatMiner. Compared to CPatMiner does R-CPatMiner extract (i) more change graphs? (ii) more code change patterns? and (iii) more code instances per pattern?

RQ3. *What is the runtime performance of R-CPatMiner, PyCPatMiner, and PyRMiner?* To answer this, we compare the execution time of the Python adapted tools with their Java counterparts.

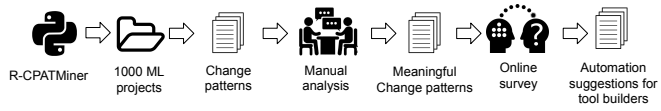


Figure 3: Schematic diagram of the research methodology to answer RQ3

4.1 Subject systems

Our corpus consists of 4,166,520 commits from 1000 large, mature, and diverse ML application systems, comprising 58M lines of source code and 150K Python files, used by other researchers [20] to understand the challenges of evolving ML systems. This corpus [20] is shown to be very diverse from the perspective of Python files, LOC, contributors, and commits. They vary widely in their domain and application, include a mix of frameworks, web utilities, databases, and robotics software systems that use ML. Further we added low-level ML libraries [9] such as Scipy, SpaCy, and high-level ML libraries [9] such as TensorFlow, Keras, PyTorch, Caffe, NLTK, and Theano to our subject systems. This ensures our dataset is representative and large enough to answer our research questions comprehensively.

4.2 Static Analysis of Source Code History

4.2.1 ***Change pattern identification:*** Running R-CPatMiner on the ML corpus extracted 28,308 unique code change patterns, where 58% of them have code change instances in multiple projects, 63% of them have been performed by multiple authors.

Since the mined patterns are numerous, we followed the best practices from Negara et al. [49]. They ordered the patterns along three dimensions - by frequency of the pattern (F), by the size of the pattern (S), and by F × S. Since the repetitive changes done by several developers and projects are stable [54] and have a higher chance of being automated, we also considered the number of projects and authors as extra two dimensions. Then we ordered the mined patterns along all five dimensions. Then, two of the authors who have more than three years of professional software development experience and extensive expertise in software evolution, manually investigated the top 500 patterns for each of the five dimensions and identified meaningful code patterns, i.e., the patterns that can be described as high-level program transformations.

Two authors of the paper manually analyzed each change pattern, to identify the high-level programming task performed in the change patterns. Following the best practices guidelines from the literature, the authors used negotiated agreement technique to achieve agreement [13, 79]. Two authors of the paper independently coded the change patterns carefully and assigned one or more descriptive phrases (i.e., codes) to the patterns. Both authors conducted the initial meeting after coding around 25% of the data (the suggested minimum size is 10% [13]). During the meeting, the authors carefully discussed the coding process of all the patterns. Also, they negotiated any disagreements between the assigned codes and the patterns that cannot be described as high-level program transformations. After 80% inter-coder agreement was achieved (recommended inter-coder agreement level ranges from 70% to more than 90% in the literature [13]), the two authors independently coded the remaining change patterns. This process identified all the patterns for which the two authors were able to agree upon the underlying meaning of the pattern. 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 meaning of the pattern. It also represents some level of patterned response or meaning within the data set [11]. 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. We identified four trends (themes) of patterns based on their structural similarity at the statement and expression level, namely (i) *transform to Context managers* (ii) *convert* `for` *loops to domain specific abstraction* (iii) *update API usage* and, (iv) *use advanced language features*

4.3 Qualitative Study

The most reliable way to understand the motivations and challenges associated with repetitive code changes is to ask the developers who performed them. To achieve this, we surveyed 650 developers who performed the identified change patterns.

4.3.1 ***Contacting the developers:*** We contacted the developers performing repetitive code changes that we considered worthy of further investigation by sending an email to the addresses provided in their GitHub account. The body of each email message was automatically generated by the application we developed, and included the following information:

- Introduction to the research team and the purpose of the study.
- A plot of number of repetitive changes done in the project.
- A link to the frequent repetitive changes done in the respective project so that the developer can use it as an educational resource.
- The following four questions for the developer:
  Q1. What are the reasons for performing above code changes?
  Q2. How often these code changes happen in ML codes?
  Q3. How often have you manually performed this kind of change?
  Q4. Would you like to have this change automated by a tool?

The first question aims at discovering actual motivations behind a code change as expressed by the developers themselves. The second question focuses on the frequency of performing the code change on ML codes, and it helps to examine the need for ML-specific IDEs tools. The last three questions aim at understanding whether developers trust and use tool support for performing the code changes. This is important, as there is relatively low IDE support for performing code changes in ML codes [20, 82]. A sample email is available on the companion website [4].

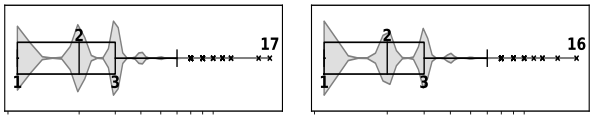
In total, we sent 650 emails to developers, out of which 97 responded, bringing us to a 15% response rate. This is significantly

higher than the usual response rate achieved in questionnaire-based software engineering surveys, which is around 5% [72].

5 RESULTS

5.1 Repetitive changes in ML systems (RQ1)

5.1.1 Characteristics of patterns mined by R-CPatMiner. We executed R-CPatMiner on our corpus described in Section 4.1 containing 1.5M changed source code files, comprising of over 490M lines of source code. For these changed files R-CPatMiner produced 349,406 change graphs with a total of 4.7M nodes. The tool extracted 28,308 unique code change patterns, where 63% and 58% of them are performed by multiple authors and in multiple projects, respectively. Figure 4 shows the *degree of sharing* of the patterns amongst developers and projects. We observed that 53% of the developers who performed the code change patterns share 100% of their change patterns with other developers, 79% share at least 50% of their patterns with others, and 91% share at least 10% of the patterns. Moreover, 36% of the projects share 100% of their patterns with other projects, 60% of them share at least 50% of their patterns with others, 91% of the projects share at least 10% of the patterns. This shows that R-CPatMiner extracts patterns that are pervasive amongst the developers and projects.



(a) Developers (b) Projects

Figure 4: *Degree of sharing* of patterns amongst developers and projects

5.1.2 Discovering pattern trends. Understanding code change patterns that the ML developers perform is important to advance science and tooling in ML software evolution. Our thematic analysis and developer surveys reveal 22 previously unknown repetitive change patterns groups that the developers ask for automation. Amongst these patterns, we identified four major trends based on their structural similarity (i.e., expression- and statement-level):

(1) *transform to Context managers*- 1237 instances
(2) *convert `for` loops to domain specific abstraction*- 239 instances
(3) *update API usage*- 166 instances
(4) *use advanced language features*- 415 instances

Next, we summarize and triangulate results obtained from source code mining, thematic analysis, and developer surveys.

Note: We use real-world code examples to describe frequent change patterns. The examples use identifiers *tf*, *np*, and *torch* as aliases of ML libraries *TensorFlow*, *NumPy*, and *PyTorch*. Due to space limitations, we provide few code examples. Our companion website [4] presents a curated repository of exemplars for each pattern, as well as *all* the instances for each pattern.

5.1.3 ***Trend 1 - Transform to Context managers:*** A Python Context manager is an abstraction for controlling the life-cycle for a code block. It declares the methods `__enter__` (initialization), and `__exit__` (finalization) which together define the desired runtime environment for the execution of a code block. The code block needs to be surrounded in a `with` statement [62] that invokes the Context manager. We observed 1,237 change instances belonging to eight pattern groups (P1–P8) where developers move code blocks into `with` statements and use *Context managers*.

Listing 4: Commit dfb7520c in Pytorch: Disable gradient

```
1 -input.grad.data.zero_()
2 +with torch.no_grad():
3 +    input.grad.zero_()
```

Listing 4 is an example of pattern P2 (Disable or enable gradient calculation). The survey respondent S21 said, *"when we do not need gradient computation in a DL network (using `Tensor.backward()`), it is important to disable the gradient calculation globally to reduce memory consumption and increase speed"*. The context manager `torch.no_grad()` from *PyTorch*, creates an execution environment for the code in line 3 and disables the gradient calculation. Like wise, the patterns (P2–P8) in Table 1 create new execution environments.

Listing 5: Commit 02ccf29b in tensorflow/datasets: Move Context managers that used to read data to with statement

```
1 -file_ = tf.gfile.GFile(label_path)
2 -dataset = csv.DictReader(file_, delimiter="")
3 +with tf.gfile.GFile(label_path) as file_:
4 +    dataset = csv.DictReader(file_, delimiter="")
```

We will now explain the most populous pattern group P1 (*Read, write, traverse data*) which moves an *already existing* Context manager within a `with` statement (see Listing 5).

The survey respondent S11 said, *"When we use Context managers in `with` statement, the required resources are allocated and released precisely."* Line 3 in Listing 5 uses the Context manager *`tf.gfile.GFile`* which handles I/O operations. The developers do not need to handle I/O operations such as file open (initialization) and file close (finalization) when they use the Context manager *inside* a `with` statement. However, if developers use the Context manager as a function call (see deleted line 1 in Listing 5), they need to handle all the initialization and finalization logic. Hence, if a Context manager is used as a function call: (i) API misuses often happen, and allocated resources will not be managed efficiently, (ii) developers also need to update all the initialisation and finalisation code when they upgrade the library version (if the APIs have changed). The respondent S13 said, *"I envision IDEs that automate moving Context managers to `with` statements."*

Table 1 tabulates the results for each major trend and pattern group, and shows survey responses for each pattern group. 90% of the survey respondents who performed Trend-1 changes confirmed that they move to `with` statements very often (VO) or often (O). All respondents perform the code transformation manually, and 74% of the respondents requested automation in their IDEs.

5.1.4 ***Trend 2 - Convert `for`-loops into domain specific abstraction:*** Listing 1 shows one such example where the developer uses `np.sum` from *NumPy* [56] instead of using `for` loop to compute the sum of elements in a `list`. Developers often perform this change to enhance the performance and code readability. Survey respondent S22 who performed pattern P9 said, *"Sometimes, Python `for` loop is a real performance killer. I want my IDEs to suggest the optimized APIs from ML libraries that I can use instead of loops"*. Moreover, as alternatives to `for` loops, developers use (i) List or

Table 1: Triangulating source code mining results with survey responses: 4 major trends, the pattern groups for each trend, and whether that pattern is specific to ML code (column ML). Column I shows the number of instances for each pattern. Column R shows number of survey respondents. Next columns indicate their responses to survey Q2 (How often these changes happen in ML code?) and Q3 (How often have you manually performed this change?), with frequency: Very Often (VO), Often (O), Rare (R), and Never (N). Q4 (Would you like to have this change automated by a tool?) response: Yes, No, Already Automated (AA).

Static Analysis						Survey Responses										
						Q_2				Q_3				Q_4		
Trend	**P**	**Pattern**	**ML**[1]	**I**	**R**	**VO**	**O**	**R**	**N**	**VO**	**O**	**R**	**N**	**Yes**	**No**	**AA**
Move to `with` statement and use Context managers	P1	Read, write, traverse data	×	467	6	67%	17%	-	17%	83%	-	17%	-	100%	-	-
	P2	Disable or enable gradient calculation	✓	92	5	100%	-	-	-	80%	-	20%	-	100%	-	-
	P3	Swap ML training devices	✓	21	3	100%	-	-	-	67%	33%	-	-	67%	33%	-
	P4	Change name and variable scopes in DL networks	✓	106	7	57%	43%	-	-	57%	29%	14%	-	57%	43%	-
	P5	Execute dependencies of a Tensorflow graphs	✓	57	3	100%	-	-	-	100%	-	-	-	33%	67%	-
	P6	Temporarily change configurations of libraries	×	59	2	-	50%	50%	-	100%	-	-	-	50%	50%	-
	P7	Transform to context managers in pytest	×	335	6	33%	50%	17%	-	67%	33%	-	-	67%	33%	-
	P8	Use context managers to open temporary directories	×	100	6	33%	50%	-	17%	67%	-	33%	-	83%	17%	-
Dissolve `for` loops, into domain specific abstractions	P9	Transform to optimized operations in NumPy	✓	179	10	100%	-	-	-	90%	-	10%	-	100%	-	-
	P10	Transform to functions in *list* or *dict*	×	24	3	67%	33%	-	-	33%	-	67%	-	33%	67%	-
	P11	Transforming to Python built in functions	×	15	2		100%	-	-	-	-	100%	-	100%	-	-
	P12	Transform functions in *String*	×	14	2	-	50%	50%	-	100%	-	-	-	100%	-	-
	P13	Transform to set operations	×	7	2	100%	-	-	-	50%	-	50%	-	100%	-	-
Update API usage, (212)	P14	Migrating to APIs ML libraries	✓	26	5	100%	-	-	-	100%	-	-	-	100%	-	-
	P15	Transform Matrix	✓	82	6	83%	17%	-	-	100%	-	-	-	33%	50%	-
	P16	Swap data visualization	✓	28	2	50%	50%	-	-	50%	-	50%	-	50%	50%	-
	P17	Composite ML APIs	✓	30	5	100%	-	-	-	100%	-	-	-	100%	-	-
	P18	Update Container	×	115	5	60%	-	40%	-	60%	-	40%	-	80%	20%	-
	P19	Update Type of Matrices	✓	31	4	100%	-	-	-	50%	-	50%	-	50%	50%	-
Use advanced Language Features	P20	Simplify conditional statement	×	24	3		33%	67%	-	67%	-	-	33%	33%	-	67
	P21	Migrate from Dict, Set, List constructors to literals	×	42	4		25%	75%	-	25%	-	-	75%	25%	-	75
	P22	Transform to Python List, Dict, or Set Comprehension	×	349	6		33%	67%	-	33%	-	-	67%	33%	-	67

[1] ML specific Patterns: i.e., patterns related to ML techniques. More than 80% developers confirmed they happen in ML very often or often

Dictionary operations (P10), (ii) Built in Python functions (P11), (iii) Python `String.join()` (P12), and (iv) Set operations `union` and `intersection` (P13). Python is an interpreted language, thus compiler level optimisations do not happen in Python. The respondent S24 said, *"For other programming languages, I might expect the compiler to optimize this type of loop, so yes, I would be interested in a suggestion by the IDE."* As shown in Table 1, 95% of the respondents who performed Trend 2 confirmed they do this very often (VO) or often (O) in ML code. All the respondents manually perform the change, and 89% of the respondents requested automation support in IDEs.

5.1.5 ***Trend 3 - Update API usage:*** Listing 6 shows an example API migration where the developer uses a readily-available `np.mean` instead of computing mean of the list `first_occ`. Survey respondent S35 said, *"NumPy offers efficient arrays and APIs for computational operations, tools that inspect the code and suggest NumPy APIs are very much needed."*

Listing 6: Commit 8592777b in inspirehep/magpie: Migrate API to NumPy

```
-return sum(first_occ) / len(first_occ)
+return np.mean(first_occ)
```

Matrix transformations (P15) such as `transpose`, `broadcast`, `squeeze`, and `unsqueeze` are frequent in ML projects. The respondent S31 said, *"bugs due to wrong matrix shapes are hard to detect and prevalent in ML systems. I like to have tools that identify these bugs and broadcast the matrices to correct shape"*. Chen et al. [14] observed developers change the bit size of matrices to get good trade-off between training time and accuracy of the predictions. Congruent to this, we also observed developers update the type of matrices, e.g., migrating int32 to int64 matrices (P19). Other patterns include swapping data visualization with Matplotlib[44](P16) between drawing all the plots in one figure vs using an individual figure for each plot. Another pattern is using composite ML APIs (P17). Developers often traverse datasets multiple times (which is inefficient). A more efficient solution is to apply a composite operation. In Listing 7, developers compute `dot` product on three matrices instead of applying a `np.multi_dot`.

Listing 7: Commit 180646fa in scikit-learn: Composite APIs

```
-denominator = np.dot(np.dot(W.T, W), H)
+denominator = np.linalg.multi_dot([W.T, W, H])
```

ML libraries offer several optimized containers (e.g., NumPy arrays, Tensors) for data processing. Updating containers, e.g., from Python's `List` to `NumPy.Array`, is another frequent change in ML systems (P18). Ketkar et al. [38] discovered that in Java code, these type migrations are more common than rename refactorings. Moreover, Table 1 shows that 85% of respondents who performed Trend 3, perform it very often (VO). All the respondents manually perform these changes, and 70% of respondents sought automation in IDEs.

5.1.6 ***Trend 4 - Use advanced language features:*** Python offers powerful features: (i) functions[59] such as `bool` and `isinstance` that can be used to simplify a conditional statement (P20), (ii) literals such as `[]`, `{}`, `()` to efficiently create containers instead of using constructors such as `list()`, `dict()`, `tuple()` (P21). (iii) Python comprehension [60] to make code concise and inline `for` loops (P22). Researchers [9, 41] observed ML code extensively operate on data, which results in expression that are longer and more complex than in traditional systems. Good software engineering principles [24, 43] require that developers change the code

to make it concise and readable. However, 69% of the survey respondents who performed Trend 4 changes confirmed they rarely perform this in their project. 62% of the developers use IDEs to perform this change.

5.2 Improvements caused by Refactoring Awareness (RQ2)

5.2.1 Impact of refactoring awareness. To answer this question, we executed the PyCPatMiner and R-CPatMiner on the study corpus and compared the results. We compared, number of change graphs, number of patterns, and distribution of code instances per pattern reported by both tools.

As shown in Table 3, R-CPatMiner processed 16% more changed methods, 0.1B more AST nodes than PyCPatMiner. PyCPatMiner builds one change graph for each mapped code block pair (i.e., before and after the changed method body). Therefore, R-CPatMiner produces **16%** more *change graphs*, thus confirming the valus of de-obfuscating change graphs that were previously obfuscated by refactoring. Then the R-CPatMiner mines all the generated *change graphs* and extracts repeated isomorphic sub-graphs as patterns. *CPATMiner* uses minimum frequencies of repeated subgraphs σ to be three adheres to the Rule of Three [67], a standard recurrence measure in pattern analysis. Therefore, all the patterns contain at least three code instances. We compared the number of patterns generated by the R-CPatMiner and PyCPatMiner and observed that R-CPatMiner captures **15%** more patterns than PyCPatMiner.

We also compared the distributions of the number of code instances per pattern in both PyCPatMiner and R-CPatMiner. To assess if there is a statistically significant difference in distributions of the number of code instances per pattern reported by PyCPatMiner and R-CPatMiner, we applied the *Wilcoxon Signed-Rank* test on the paired samples of number of code instances of each pattern. The test rejected the null hypothesis that the density of code instances of the pattern produced by PyCPatMiner is more than it is in R-CPatMiner at the significance level of 5% (p-value = 1.12×10^{-10}). We used the *Hodges-Lehman estimator* to quantify the difference between the two distributions, as it is appropriate to be used with the *Wilcoxon Singed-Rank* test. The value turned out to be 1, which is equal to the estimated median of the difference between the number of code instances per pattern from PyCPatMiner and R-CPatMiner. Therefore, R-CPatMiner extracts more code change instances per pattern than the PyCPatMiner.

5.2.2 Evaluating the precision of PyRMiner. It is important for PyRMiner to have a high precision as we use it to first match the refactored code blocks that we then pass to R-CPatMiner to build change graphs. We first identified 18 refactoring kinds that obfuscate fine-grained changes, i.e, the refactorings that change method signatures or shift the method bodies. First, we executed the PyRMiner on our study corpus and chose a statistically significant random sample of refactoring instances for each refactoring kind. Hence, using a t-test, we conclude with 95% confidence that the precision of the refactoring detection is only ± 5% for each refactoring kind, as shown in Table 2.

Two of the authors that have more than three years of software development experience and extensive expertise in software evolution manually validated the refactorings reported by PyRMiner.

Table 2: Precision of PyRMiner per refactoring kind

Refactoring Kind	Precision (#TP)	Refactoring Kind	Precision (#TP)
Rename Method	96.32% (183)	Move And Rename class	96.67% (116)
Move Method	96.3% (156)	Move class	100% (160)
Pull Up Method	86.46% (83)	Extract Class	98.82% (84)
Push Down Method	89.36% (84)	Extract Subclass	100% (45)
Extract Superclass	95.24% (40)	Parameterize Variable	87.23% (82)
Split Parameter	91.66% (22)	Move & Rename Method	88.08% (133)
Rename Class	99.2% (124)	Remove Parameter	97.56% (160)
Reorder Parameter	98.11% (104)	Rename Parameter	93.59% (146)
Add Parameter	100% (192)	Merge Parameter	91.07% (51)

Most cases were straightforward and thus were validated individually, but both authors inspected some challenging cases to reach an agreement. In total, we validated 2,062 unique refactoring instances, out of which 1,965 were true positives and 97 were false positives. This achieves an average precision of 95%, which is close to the precision of the original Java-RMiner (99.6%). This also shows the effectiveness of JavaFyPy to adapt Java AST-analysis tools to Python. We release all the validated refactoring instances on our companion website [4]. To the best of our knowledge, this is the largest to date Python data-set of validated refactoring instances.

Recall indicates the proportion of actual refactorings identified by PyRMiner. Java researchers use previously formed unbiased oracles or use multiple tools that produce the same results to build an oracle and compute recall of tools [38, 76]. To date, there is no such unbiased refactoring oracle for Python or tools that can detect the Python refactoring kinds given in Table 2. Considering the complexity of building an oracle and our main focus (to retrieve correct code-block mappings to make PyCPatMiner refactoring-aware), we leave computing recall as future work. However, we observed R-CPatMiner detects 16% more change-graphs (i.e., mapped code blocks) and 15% more patterns than the PyCPatMiner, which strongly indicates that the PyRMiner has a satisfactory recall.

5.3 Runtime performance of R-CPatMiner, PyCPatMiner, and PyRMiner (RQ3)

To measure the execution time of the tools, we executed the Python adapted RMiner, CPatMiner, and R-CPatMiner on a large corpus and compared the execution time with the Java version of the tools. We executed each tool separately on the same machine with the following specifications: Intel Core i9 CPU @ 2.90GHz, 32 GB DDR4 memory, 1 TB SSD, macOS 10.14.6 OS, and Java 13.0.1 ×64.

First, we record the running time of the *type inference* tool, *PyType* [27] (version 2020.10.08). *PyType* is decoupled from JavaFyPy. Therefore, we computed the execution time of *PyType* separately. *PyType* took on mean 360ms and 61ms on median for type inference of all changed file in a commit. We pushed the inferred type

Table 3: Analysed data set and execution time.

	Java CPATMiner	Py-CPATMiner	R-CPATMiner
Total changed methods	824K	1M	1.16M
Total AST nodes of changed methods	92M	4.5B	4.6B
Total changed graph nodes	8M	4M	4.7M
Total patterns	17K	24K	28K
Execution time	<8hours	<12hours	<19hours

[1] The data of the Java-CPATMiner is obtained from its original paper [53]

information of all the changed files in all the studied projects' commits to a repository [63] in Github. Similar to *Typeshed* [77], a type repository of Python library APIs that the library clients use for type annotations, researchers can fast-track their analysis by simply reusing this inferred type information .

To record running time of PyRMiner, we followed the steps used by Tsantalis et al. [76] for computing the running time of their Java-RMiner. We recorded the time required for parsing the source code of the commit(and its parent), and the time required for detect refactorings. Our analysis shows that PyRMiner takes *55ms* on median and *296.32ms* on mean to process a Python commit. Tsantalis et al. [76] found that the Java version of RMiner takes *44ms* on median and *253ms* on mean to process one Java commit. Therefore, PyRMiner takes reasonable time overhead for the additional processing (i.e., AST transformation), and will not impact the primary goal of RMiner, i.e., create larger refactoring datasets to strengthen the validity of empirical studies or enable novel applications of refactoring mining [76].

Table 3 shows the size of processed data and the execution time of the tools, Java-CPatMiner, PyCPatMiner, and R-CPatMiner. The running time of the PyCPatMiner and R-CPatMiner is less than 12 hours and 19 hours respectively, whereas Java-CPatMiner takes less than 8 hours to mine patterns. However, the Java and Python corpus is diverse (see Table 3). Hence, it is hard to make a fair comparison. Nguyen et al. [53] state that the primary goal of the CPatMiner is to mine the corpus weekly to build a database of patterns. We believe that R-CPatMiner and PyCPatMiner have reasonable execution times for achieving the same goal.

6 IMPLICATIONS

We present actionable, empirically-justified implications for four audiences: (i) researchers, (ii) tool builders and IDE designers, (iii) ML library vendors, and (iv) developers and educators.

6.1 Researchers

R1. Exploit applications of change repetitiveness to Python ML software (RQ1, RQ3). In the past, researchers exploited the repetitiveness of changes in Java systems through: code completion [12, 31, 39, 51, 52], automated program repair [5, 8, 46], API recommendation [29, 51], type migration [37], library migration [2, 17, 22, 36, 75], and automated refactoring [18, 26]. Using our rich and diverse dataset of 28K change patterns instances and our JavaFyPy, researchers can bring the same benefits to Python ML systems.

R2. Foundations to study Python ML Software Evolution (RQ1, RQ2, RQ3). Despite the widespread use of Python ML systems, its evolution and maintenance tasks are the least automated and the least studied due to the unavailability of tool support to study ML systems [20]. Previously, researchers have built infrastructure to study many aspects of software evolution. For example, *RMiner* [76] and *RefDiff* [70] mine refactorings, *TypeFactMiner* [38] mines type changes, *MigrationMiner* [2] and *APIMigrator* [23] mine API migrations, *CPATMiner* [53] and CodingTracker [50] mine fine-grained repetitive code changes in Java. This rich infrastructure allows researchers to study Java software systems, and there are hundreds of published papers that are built upon this infrastructure. Unfortunately, they miss the whole ecosystem of Python ML codebases. Researchers can use our data and toolset to study whether previously-held beliefs from traditional software are still valid for ML systems or whether we need to design new tools and workflows (e.g, CI tools, version-control systems, code smells, technical debt, etc.) that are specific to ML.

R3. Enhance existing research and tools for ML systems (RQ1, RQ3): Our infrastructure can be used to enhance the existing research on ML systems. For example, Tang et al. [74] introduce a taxonomy of refactoring kinds performed in Java ML systems, Humbatova et al. [32], Islam et al. [34, 35] introduce a taxonomy of bugs in ML systems based on manual analysis of *StackOverFlow* posts. These studies perform extensive manual analysis to build various taxonomies using a smaller dataset of changes. With our significantly larger dataset (of 28K change patterns), they can significantly extend or further validate their taxonomies. Moreover, researchers showed the potential of leveraging ML techniques for code completion [15, 29]. However, they train the ML models either on a noisy or small dataset which could reduce the accuracy of the recommendations. Researchers can use our tools/dataset to train their models on a large, curated dataset and improve their accuracy.

R4. Build novel applications for ML developers (RQ1). Researchers can also use our tools/dataset to build novel applications. Braiek and Khomh [9] observed ML libraries are core components of ML systems and are frequently evolving, which causes developer frustration [20, 83]. A tutoring system can suggest to developers which constructs to use in order to modernize their ML code. For example, a tutoring system could recommend changing the code to use ML library APIs instead of `for` loops, or removing redundant matrix operations (or other changes from RQ1:Table 1). Moreover, a tutor system can recommend novice programmers how to use advanced language features (see Trend 4 in RQ1:Table 1).

R5. Revisit existing studies and tools for making them refactoring-aware (RQ2). There exist a plethora of research tools for mining software repositories, and hundreds of researchers used these tools to conduct empirical studies. Given that refactorings obfuscate program elements during software evolution, we showed that making a state of the art tool such as CPatMiner refactoring aware increases its overall effectiveness by as much as 15%. We call the research community to adapt similar methods and to revisit previous results obtained with tools that were not refactoring-aware.

6.2 Tool Builders and IDE Designers

T1. New inspirations for tool development (RQ1). To help tool builders invest resources where automation is most needed, in Table 1 we present 22 patterns along with the ML developers request for automation. Moving to `with` statements and using Context managers is the most prevalent change pattern among the analysed patterns (See RQ1, P1-P8). In the survey, 74% of respondents suggest tools that inspect deep learning codebases and recommend using `with` statements to (i) turn on or off gradient calculations (P2), (ii) specify hardware type (P3), (iii) change variable scopes (P4), and (iv) execute dependencies (P5). Respondents further suggested tools to (i) move Context manager invocations to `with` statements, and (ii) detect misuses of Context managers.

6.3 ML Library vendors

L1. Understand ML Library Usage (RQ1). ML library vendors continuously improve libraries and introduce new ML libraries at an unprecedented rate [10, 20]. Library developers deprecate APIs, introduce more efficient alternative APIs (e.g., Table 1 : P18), and split ML libraries [20]. Our findings, the accompanying dataset [4], and the tools we developed can help ML library vendors to understand what APIs are most commonly used, misused, and underused, and how the developers adapt to new APIs. Thus, they can make informed, empirically-justified decisions to improve features [21, 45].

6.4 Software Developers and Educators

S1. Rich educational resource (RQ1) Developers learn and educators teach new programming constructs through examples. Robillard et al. [65] studied the challenges of learning APIs and concluded that one of the important factors is the lack of usage examples. Using our dataset of 28K code change patterns that we mined in our corpus, developers and educators can learn from real-world code transformations (e.g., transforming to `multi_dot`). We provide 22 empirically justified code change patterns that improve ML code from many aspects, including speed, code quality, and readability. ML developers can absorb these changes to their code and improve the code. We released this through an educational resource [4].

7 THREATS TO VALIDITY

Internal Validity: *Can we trust the results produced by tools?* The findings of our study depend on the accuracy of our tools to mine code change patterns in a refactoring aware manner. We rely on *type inference* for augmenting the AST with rich type information. Since *Type inference* deduces the types of elements by statically analysing the program, it may not accurately detect them (compared to the run time). This can effect the quality of mappings reported by R-CPatMiner and the statements matched by RMiner. To mitigate this threat, we use *PyType* [27], a mature tool developed by Google. Thousands of projects at Google and other places rely upon *PyType* to keep their codes well-typed [27]. Moreover, we validate JavaFyPy's effectiveness at transforming a variety of syntactic variations upon 12M AST nodes from 14 popular projects. Our manual validations shows that PyRMiner reports refactorings with high precision (95%). We also manually validated 2,500 most popular patterns produced by R-CPatMiner.

To avoid the *experimental bias*, we followed the best practices [13, 79] for applying thematic analysis by achieving 80% inter coder agreement when labelling the patterns.

External Validity: *Do our results generalize?* We studied 1000 projects from a wide range of application domains, making the study results generalizable to other open-source projects in similar domains. However, a study of proprietary code bases might reveal other trends. Nevertheless, we make our tools available so that others can use them to mine patterns in proprietary codebases. Moreover, R-CPatMiner reports numerous patterns, we manually analyzed a subset of them; a complete investigation is not practical. To mitigate this, we ranked the code change patterns in five dimensions and manually validated the top ones.

Verifiability: *Can others replicate our results?* To ensure replicability, we make the tools and the data publicly available [4].

8 RELATED WORK

8.0.1 ***Studies on ML software systems.*** Researchers have studied repetitive tasks of ML systems from many aspects. Humbatova et al. [32] and Islam et al. [34, 35] introduce a taxonomy of ML-related bugs and bug fix patterns, Shen et al. [69] study bugs in Deep Learning (DL) compilers, and Yan et al. [81] study numerical bugs in DL systems. Zhang et al. [82] observe DL program failures at Microsoft and highlight the need for DL-specific tools to fix DL bugs. Lwakatare et al. [42], Zhang et al. [83] classify common challenges that ML developers face when maintaining and evolving ML systems. Wan et al. [78] found anti-patterns and misused APIs in ML libraries. Amershi et al. [3] perform a field study at Microsoft and propose best practices to address common challenges specific to engineering ML systems. In contrast, our focus is on understanding fine-grained repeated code changes that the ML developers perform and their motivations.

The closest related work is by Tang et al. [74]. The authors study 327 git patches from 26 *Java* ML systems and present a taxonomy of refactoring kinds that occurred in ML systems. In contrast, we quantitatively and qualitatively studied fine-grained code change patterns in a significantly larger corpus: 1000 *Python*-based ML systems comprising 4,166,520 git patches. Our findings include both refactorings (e.g., P1, P9) and other semantics-modifying code changes (e.g., P2, P6). Moreover, we cross-validated our findings by surveying 97 ML developers. We make our tools available to further enhance the science and tooling for evolving Python ML systems.

8.0.2 ***Studies on repetitive code changes.*** Researchers have conducted many studies on repetitive code changes. Nguyen et al. [53, 54] use a graph-based algorithm to mine fine-grained code changes at commit level. They conduct a large-scale study on the repetitiveness of code changes in Java software evolution and show that repetitiveness is common in small granularity (number of lines), and it drops exponentially as the granularity increases. Researchers have also studied repetitiveness from the vantage point of higher-level maintenance and evolution tasks, (i) Dig et al. [19], Cossette [16] study incompatibilities between API versions, (ii) Teyton et al. [75] mine the library migrations trends and observe how frequently, when, and why they are performed, and (iii) Ketkar et al. [37] conduct a large-scale study on type changes in Java systems and reveal that type changes are more frequent than renaming. All of these studies focus on repetitiveness in traditional systems and do not reveal the kinds of repetitive changes the developers perform in ML software. In contrast, we study the practices of fine-grained code changes in ML systems and found four trends of fine-grained changes. 71% of the surveyed developers requested automation support for the identified trends in their IDEs.

Researchers proposed several techniques that infer *specific kinds* of code change patterns. For example, Revisar [66], GetAFix [5], and DeepDelta [46] infer repeated bug fixes and compilation errors from commit histories. LibSync [55], MEditor [80], and A3 [40] infer the adaptations required to perform library migration. In contrast, we discover *previously unknown* patterns in ML systems that involve adapting ML libraries (E.g. P9- `for` loop to *NumPy*)

9 CONCLUSIONS

This paper presents the first and the largest study of fine-grained code change patterns in Python-based ML software systems. To provide unique insights, we use complementary empirical methods: (i) mining 1000 software repositories containing over 58 million LOC, (ii) using thematic analysis to identify the groups and trends, and (iii) surveying 97 ML developers. To conduct this study and advance the science and tooling in Python ML software evolution, we designed a novel technique, JavaFyPy, to reuse, adapt and improve upon the Java state-of-the-art AST mining tools. We introduce a novel tool R-CPatMiner that performs refactoring-aware change pattern mining in the version history of Python projects. We present 22 code change pattern groups in four trends, where 10 of them are specific to ML. In the developer survey, 71% of the respondents requested these patterns automated in their IDEs. The results and the tools presented in this study have actionable implications for researchers, tool builders, library designers, ML developers, and educators. We hope that this paper and our readily available dataset and tools [4] catalyzes the community to advance the science and tooling for the evolution of Python-based ML systems.

10 ACKNOWLEDGEMENTS

We would like to thank Ellick Chan, Julia Romero, CUPLV group at CU Boulder, 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, CNS-1941898, and by the PPI Center at CU Boulder.

REFERENCES

[1] Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In *FSE 2014* (Hong Kong, China). Association for Computing Machinery, New York, NY, USA, 472–483. https://doi.org/10.1145/2635868.2635901

[2] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. MigrationMiner: An Automated Detection Tool of Third-Party Java Library Migration at the Method Level. In *ICSME 2019*. 414–417. https://doi.org/10.1109/ICSME.2019.00072

[3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *ICSE* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, Piscataway, NJ, USA, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[4] Anonymous Authors. 2021. *Discovering Repetitive Code Changes in Python-based ML Systems*. https://mlcodepatterns.github.io Accessed: 2021-05-05.

[5] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

[6] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *FSE 2014* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 306–317. https://doi.org/10.1145/2635868.2635898

[7] Amine Barrak, Ellis E. Eghan, and Bram Adams. 2021. On the Co-evolution of ML Pipelines and Source Code - Empirical Study of DVC Projects. In *SANER 2021*. 422–433. https://doi.org/10.1109/SANER50967.2021.00046

[8] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *ESEC/FSE 2019* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

[9] 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

[10] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The Open-Closed Principle of Modern Machine Learning Frameworks. In *MSR '18* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 353–363. https://doi.org/10.1145/3196398.3196445

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

[12] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from Examples to Improve Code Completion Systems. In *ESEC/FSE '09* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 213–222. https://doi.org/10.1145/1595696.1595728

[13] 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

[14] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A Comprehensive Study on Challenges in Deploying Deep Learning Based Software. In *FSE* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 750–762. https://doi.org/10.1145/3368089.3409759

[15] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *TSE 2019* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2940179

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

[17] Barthélémy Dagenais and Martin P. Robillard. 2011. Recommending Adaptive Changes for Framework Evolution. *ACM Trans. Softw. Eng. Methodol.* 20, 4, Article 19 (Sept. 2011), 35 pages. https://doi.org/10.1145/2000799.2000805

[18] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *ECOOP'06* (Nantes, France) *(ECOOP'06)*. Springer-Verlag, Berlin, Heidelberg, 404–428. https://doi.org/10.1007/11785477_24

[19] 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.

[20] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 55 (July 2021), 42 pages. https://doi.org/10.1145/3453478

[21] 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 *ICSE* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 779–790. https://doi.org/10.1145/2568225.2568295

[22] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-Usage Update for Android Apps. In *ISSTA 2019* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 204–215. https://doi.org/10.1145/3293882.3330571

[23] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2020. APIMigrator: An API-Usage Migration Tool for Android Apps. In *MOBILESoft '20* (Seoul, Republic of Korea) *(MOBILESoft '20)*. Association for Computing Machinery, New York, NY, USA, 77–80. https://doi.org/10.1145/3387905.3388608

[24] Michael Feathers. 2004. *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. Prentice Hall Professional.

[25] Eclipse foundation. 2021. *JDT Core Component*. Eclipse. https://www.eclipse.org/jdt/core/#JDT_CORE Accessed: 2021-03-31.

[26] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. 2013. LAMBDAFICATOR: From Imperative to Functional Programming through Automated Refactoring. In *ICSE* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 1287–1290. https://doi.org/10.1109/ICSE.2013.6606699

[27] Google. 2021. *PyType*. https://github.com/google/pytype Accessed: 2021-03-31.

[28] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. 2020. Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?. In *FSE* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 851–862. https://doi.org/10.1145/3368089.3409754

[29] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. 2021. PyART: Python API Recommendation in Real-Time. In *ICSE 2021*. 1634–1645. https://doi.org/10.1109/ICSE43902.2021.00145

[30] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the Naturalness of Software. *Commun. ACM* 59, 5 (April 2016), 122–131. https://doi.org/10.1145/2902362

[31] Reid Holmes, Robert J. Walker, and Gail C. Murphy. 2006. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Trans. Softw. Eng.* 32, 12 (Dec. 2006), 952–970. https://doi.org/10.1109/TSE.2006.117

[32] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *ICSE '20* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[33] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepCrime: Mutation Testing of Deep Learning Systems Based on Real Faults. In *ISSTA-2021* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/3460319.3464825

[34] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *ESEC/FSE 2019* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[35] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *ICSE '20* (Seoul, Republic of Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

[36] 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 *MSR '16* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 154–164. https://doi.org/10.1145/2901739.2901769

[37] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. 2019. Type Migration in Ultra-Large-Scale Codebases. In *ICSE '19* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1142–1153. https://doi.org/10.1109/ICSE.2019.00117

[38] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. Understanding Type Changes in Java. In *FSE* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 629–641. https://doi.org/10.1145/3368089.3409725

[39] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *ICSE* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 802–811.

[40] M. Lamothe, W. Shang, and T. Chen. 2020. A3: Assisting Android API Migrations Using Code Examples. *TSE 2020* 01 (apr 2020), 1–1. https://doi.org/10.1109/TSE.2020.2988396

[41] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2020. Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks. In *ICSE* (Seoul, South Korea) *(ICSE-SEIS '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3377815.3381377

[42] Lucy Ellen Lwakatare, Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. 2019. A Taxonomy of Software Engineering Challenges for Machine Learning Systems: An Empirical Investigation. In *Agile Processes in Software Engineering and Extreme Programming*, Philippe Kruchten, Steven Fraser, and François Coallier (Eds.). Springer International Publishing, Cham, 227–243.

[43] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[44] Matplotlib. 2021. *Matplotlib*. https://matplotlib.org Accessed: 2021-05-05.

[45] 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 85 (Oct. 2017), 31 pages. https://doi.org/10.1145/3133909

[46] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. In *ESEC/FSE*. 925–936. https://doi.org/10.1145/3338906.3340455

[47] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *ICSE '09 (ICSE '09)*. Association for Computing Machinery, New York, NY, USA, 287–297. https://doi.org/10.1109/ICSE.2009.5070529

[48] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP'13* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 552–576. https://doi.org/10.1007/978-3-642-39038-8_23

[49] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining Fine-Grained Code Changes to Detect Unknown Change Patterns. In *ICSE* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 803–813. https://doi.org/10.1145/2568225.2568317

[50] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. 2012. Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?. In *ECOOP'12* (Beijing, China) *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 79–103. https://doi.org/10.1007/978-3-642-31057-7_5

[51] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. *API Code Recommendation Using Statistical Learning from Fine-Grained Changes*. Association for Computing Machinery, New York, NY, USA, 511–522. https://doi.org/10.1145/2950290.2950333

[52] 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 *ICSE* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 69–79.

[53] H. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton. 2019. Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns. In *ICSE 2019*. IEEE Computer Society, Los Alamitos, CA, USA, 819–830. https://doi.org/10.1109/ICSE.2019.00089

[54] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2013. A Study of Repetitiveness of Code Changes in Software Evolution. In *ASE '13* (Silicon Valley, CA, USA) *(ASE'13)*. IEEE Press, 180–190. https://doi.org/10.1109/ASE.2013.6693078

[55] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-Based Approach to API Usage Adaptation. *SIGPLAN Not.* 45, 10 (Oct. 2010), 302–321. https://doi.org/10.1145/1932682.1869486

[56] numpy. 2021. *NumPy*. https://numpy.org Accessed: 2021-05-05.

[57] Oracle. 2021. *Java SE specification*. Oracle. https://docs.oracle.com/javase/specs/ Accessed: 2021-03-31.

[58] Python. 2021. *Context Manager*. https://docs.python.org/3/reference/datamodel.html#context-managers Accessed: 2021-03-31.

[59] Python. 2021. *Functions*. Python. https://docs.python.org/3/library/functions.html Accessed: 2021-03-31.

[60] Python. 2021. *list-comprehensions*. https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions Accessed: 2021-05-05.

[61] Python. 2021. *Python AST*. Python. https://docs.python.org/3/library/ast.html Accessed: 2021-03-31.

[62] Python. 2021. *With Statement*. Python. https://docs.python.org/3/reference/compound_stmts.html#the-with-statement Accessed: 2021-03-31.

[63] PythonTypeInformation. 2021. *PythonTypeInformation*. GitHub. https://github.com/mlcodepatterns/PythonTypeInformation Accessed: 2021-05-05.

[64] Sebastian Raschka and Vahid Mirjalili. 2017. Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow.

[65] 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

[66] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. (2018). http://arxiv.org/abs/1803.03806

[67] RuleOfThree. 2021. *RuleOfThree*. wikic2. http://wiki.c2.com/?RuleOfThree Accessed: 2021-05-05.

[68] 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.

[69] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *FSE* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

[70] D. Silva, J. Silva, G. De Souza Santos, R. Terra, and M. O. Valente. 5555. RefDiff 2.0: A Multi-language Refactoring Detection Tool. *TSE 2020* 01 (jan 5555), 1–1. https://doi.org/10.1109/TSE.2020.2968072

[71] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *FSE* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[72] 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.

[73] Brett Slatkin. 2019. *Effective python: 90 specific ways to write better python*. Addison-Wesley Professional.

[74] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *ICSE 2021*. IEEE Computer Society, Los Alamitos, CA, USA, 238–250. https://doi.org/10.1109/ICSE43902.2021.00033

[75] 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

[76] N. Tsantalis, A. Ketkar, and D. Dig. 5555. RefactoringMiner 2.0. *TSE 2020* 01 (jul 5555), 1–1. https://doi.org/10.1109/TSE.2020.3007722

[77] TypeShed. 2021. *TypeShed*. https://github.com/python/typeshed Accessed: 2021-09-03.

[78] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu. 2021. Are Machine Learning Cloud APIs Used Correctly?. In *ICSE 2021*. IEEE Computer Society, Los Alamitos, CA, USA, 125–137. https://doi.org/10.1109/ICSE43902.2021.00024

[79] David Wicks. 2017. The coding manual for qualitative researchers. *Qualitative research in organizations and management: an international journal* (2017). https://doi.org/10.1108/QROM-08-2016-1408

[80] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, Piscataway, NJ, USA, 335–346. https://doi.org/10.1109/ICPC.2019.00052

[81] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing Numerical Bugs in Deep Learning via Gradient Back-Propagation. In *FSE* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 627–638.

[82] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *ICSE* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1159–1170. https://doi.org/10.1145/3377811.3380362

[83] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *ISSRE*. IEEE Computer Society, Los Alamitos, CA, USA, 104–115. https://doi.org/10.1109/ISSRE.2019.00020