Discovering Repetitive Code Changes in Python ML Systems

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

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

Ameya Ketkar*
ketkara@uber.com
Uber Technologies Inc.
USA

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

1 INTRODUCTION

Many software changes are repetitive by nature [7, 33, 53], 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 [61]. Since this change involves programming idioms [2, 80] 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

```python
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 [13, 34, 43, 55, 56], automated program repair [6, 9, 50], API recommendation [32, 55], type migration [40], library migration [3, 18, 24, 39, 84], code refactoring [19, 28], fine-grained understanding of software evolution [3, 25, 42, 54, 58, 77, 85]. Unfortunately, these are mostly available only for Java, and do not support Python and ML systems.

Researchers [11, 22, 37, 70] 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 [22, 91]. 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 [35, 37, 38], updating ML libraries [22], refactoring and technical debt of ML systems [75, 82], managing version control systems for data [8], and testing [10, 31, 36]. 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 best 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) to answer the research 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.

For the quantitative analysis, we use a large data set of 1000 ML projects from GitHub, comprising 58 million source lines of code (SLOC) at the latest revisions, 1.16 million mapped code 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 [12, 88] 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, in which we presented 1,235 patterns for their feedback and achieved a 15% response rate. 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), (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 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 [3, 25, 42, 54, 58, 77, 85] 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). However, these techniques are inapplicable to mine such fine-grained change patterns from the commit history of Python 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.

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. 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 so that we enable further research.

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

```python
class _FERNeuralNet():
    + def _init_model(self):
        + with tf.device('/cpu:0'):
            + B, H, T, _ = q.get_shape().as_list()
    ...

class TimeDelayNN(_FERNeuralNet):
    + def _init_model(self):
        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 [64] 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 [58, 59]. 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 [51, 52, 78] have shown that developers often interleave many programming activities such as bug fixes, feature additions, or other refactoring operations, and often these changes overlap [54] (as shown in Listing 2). Such overlapping changes and refactorings can easily obfuscate existing fine-grained change pattern mining tools [58, 59] because they do not account for these changes when matching code across the commit. For example, CPATMiner [58] does not match the method body of _FERNeuralNet (lines 3–4) to the body of _init_model 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 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 JAVAxFyPy, 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 JAVAxFyPy to adapt the state-of-the-art fine-grained change pattern mining tool, CPATMiner [58], to Python. To make CPATMiner refactoring aware, we adapt the state-of-the-art Java refactoring inference tool, RefactoringMiner [85] (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 - JAVAxFyPy, 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. JAVAxFyPy 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, JAVAxFyPy takes a

![Figure 1: Design of JAVAxFyPy and existing AST analysis tools](image)

Python code as an input and produces an AST, that can be used in mining algorithms of Java AST analysis tools. To achieve this, JAVAxFyPy 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 JAVAxFyPy, and extend their tools for Python.

3.1 Python code transformation

As shown in Figure 2 JAVAxFyPy 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 \(N_j\) in T has a parent (except the root node). Each node \((N_i, N_j)\) has a sequence of child nodes (denoted by \(C_{N_i}\)). Number of nodes in the sequence \(N_j\) is denoted by \(\text{Length}_{N_j}\). Each node \(N_j\) is a specific syntax category known as AST node kind, \(\text{Kind}_{N_j}\) = \{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 [62, 67] 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 \(T_p\) be a Java AST and \(T_p\) be a Python AST. \(N_j\) in \(T_p\) and \(N_p\) 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) \(\forall N_i \in C_{N_p} : M(N_i) \in C_{N_j}\).

(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 IfStatement 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 \(N_p\) meet conditions: (i) \(M(N_p)\) is \(N_j\), and (ii) \(\exists N_i \in C_{N_j} : M(N_i) \notin C_{N_j}\).

(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., \(M(N_p) \notin T_j\).
As we can observe, Java and Python syntax significantly overlaps. As shown 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 analyze 72% of the Identical AST node kinds, and modify the current 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 \((\text{for loop})\), and Unique AST node \((\text{yield} \ z)\). The Java parser first constructs the AST of the code snippet, then the Type Augmenter augments the AST with type information in a JavaFyPy manner by adding Variable Declaration nodes. This step is important because the Java-based AST mining tools 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 the type information from JType [30], the state-of-the-practice type inference tool for Python developed by Google, which is widely adopted 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 checked whether all AST nodes, i.e., Identical AST nodes. Nearly identical AST nodes and Unique AST nodes were successfully mapped and transformed to the output AST of JavaFyPy. We achieved this by transforming all of the Python files in the projects, which had 12M Python AST nodes. This confirms that JavaFyPy can effectively transform any input Python program to an Eclipse JDT AST format.

As shown 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 analyze 72% of the Identical AST node kinds, and modify the current 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: An example Code transformation performed by JavaFyPy

3.2 Refactoring Aware Change Pattern Mining
3.2.1 Adapting CPatMiner. CPatMiner [58] 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 over changed methods in each commit and uses Eclipse JDT Java parser [27] to generate an 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 [58] 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 [85]. 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

```
1 - indices = [layerNum - 1 for layerNum in layers_norm]
2 + 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 prefix all the adapted tool names with Py to disambiguate the tool names from their Java counterparts. We first evaluate the effectiveness of the tools we developed (or adapted). Then, we use our reliable and validated tools, 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.

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.

Since the mined patterns are numerous, we followed the best practices from Negara et al. [53]. They ordered the patterns along three dimensions - by frequency of the pattern (F), by the size of the pattern (S), and by $F \times S$. Since the repetitive changes done by several developers and projects are stable [59] 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 [14, 88]. 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% [14]). 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 [14]), 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 [12]. 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.

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 [22] to understand the challenges of evolving ML systems. This corpus [22] 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 [10] such as Scipy, SpaCy, and high-level ML libraries [10] such as 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.
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 the code changes above?
Q2. How often do 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 IDE 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 [22, 91]. A sample email is available on the companion website [21].

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% [79].

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.

![Number of authors (log scale)](a) Developers

![Number of projects (log scale)](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 where 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 if, np, and torch as aliases of ML libraries TensorFlow, NumPy, and PyTorch. Due to space limitations, we provide few code examples. Our companion website [21] 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 [68] that invokes the Context manager. We observed 1,237 change instances belonging to eight pattern groups (P1–P8 in Table 1) where developers move code blocks into with statements and use Context managers.

Listing 4: Commit dfb7520c in Pytorch: Disable gradient

```python
Listing 4: Commit dfb7520c in Pytorch: Disable gradient
```

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 TensorFlow), 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

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

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 [1]. Hence, if a Context manager is used as a function call: (i) API misuse often happen, and allocated resources might not be managed efficiently [74, 83], (ii) developers need to
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).

<table>
<thead>
<tr>
<th>Trend</th>
<th>P</th>
<th>Pattern</th>
<th>ML</th>
<th>I</th>
<th>R</th>
<th>VO</th>
<th>O</th>
<th>N</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Yes</th>
<th>No</th>
<th>AA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move with statement</td>
<td>P1</td>
<td>Read, write, traverse data</td>
<td></td>
<td></td>
<td>167</td>
<td>6</td>
<td>17%</td>
<td>-</td>
<td>17%</td>
<td></td>
<td>83%</td>
<td>17%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>Disable or enable gradient calculation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>Swap ML training devices</td>
<td></td>
<td></td>
<td>92</td>
<td>5</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td></td>
<td>80%</td>
<td>20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context managers</td>
<td>P4</td>
<td>Change name and variable scopes in DL networks</td>
<td></td>
<td></td>
<td>106</td>
<td>7</td>
<td>57%</td>
<td>43%</td>
<td>-</td>
<td>57%</td>
<td>29%</td>
<td>14%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>Execute dependencies of a Tensorflow graphs</td>
<td></td>
<td></td>
<td>57</td>
<td>3</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td></td>
<td>100%</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>Temporarily change configurations of libraries</td>
<td></td>
<td></td>
<td>59</td>
<td>2</td>
<td>50%</td>
<td>50%</td>
<td>-</td>
<td></td>
<td>50%</td>
<td>50%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>Transform to context managers in pytest</td>
<td></td>
<td></td>
<td>355</td>
<td>6</td>
<td>33%</td>
<td>50%</td>
<td>17%</td>
<td>67%</td>
<td>33%</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>Use context managers to open temporary directories</td>
<td></td>
<td></td>
<td>106</td>
<td>6</td>
<td>33%</td>
<td>50%</td>
<td>17%</td>
<td>67%</td>
<td>33%</td>
<td>-</td>
<td>83%</td>
<td>17%</td>
<td></td>
</tr>
<tr>
<td>Dissolve for loops,</td>
<td>P9</td>
<td>Transform to optimized operations in NumPy</td>
<td></td>
<td></td>
<td>179</td>
<td>10</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td></td>
<td>90%</td>
<td>10%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>into domain specific</td>
<td>P10</td>
<td>Transform to operations in List or Dictionary</td>
<td></td>
<td></td>
<td>24</td>
<td>3</td>
<td>67%</td>
<td>33%</td>
<td>-</td>
<td></td>
<td>33%</td>
<td>67%</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>abstractions</td>
<td>P11</td>
<td>Transforming to Python built in functions</td>
<td></td>
<td></td>
<td>15</td>
<td>2</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P12</td>
<td>Transform functions in String</td>
<td></td>
<td></td>
<td>14</td>
<td>2</td>
<td>50%</td>
<td>50%</td>
<td>-</td>
<td></td>
<td>100%</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P13</td>
<td>Transform to set operations</td>
<td></td>
<td></td>
<td>7</td>
<td>2</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td></td>
<td>50%</td>
<td>50%</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update API usage, (212)</td>
<td>P14</td>
<td>Migrating to APIs ML libraries</td>
<td></td>
<td></td>
<td>26</td>
<td>5</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td></td>
<td>100%</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P15</td>
<td>Transform Matrix</td>
<td></td>
<td></td>
<td>82</td>
<td>6</td>
<td>83%</td>
<td>17%</td>
<td>-</td>
<td></td>
<td>100%</td>
<td>-</td>
<td>33%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>P16</td>
<td>Change data visualization</td>
<td></td>
<td></td>
<td>28</td>
<td>2</td>
<td>50%</td>
<td>50%</td>
<td>-</td>
<td></td>
<td>50%</td>
<td>50%</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P17</td>
<td>Composite ML APIs</td>
<td></td>
<td></td>
<td>22</td>
<td>5</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td></td>
<td>100%</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P18</td>
<td>Update Container</td>
<td></td>
<td></td>
<td>115</td>
<td>5</td>
<td>60%</td>
<td>40%</td>
<td>-</td>
<td></td>
<td>60%</td>
<td>40%</td>
<td>-</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>P19</td>
<td>Update Type of Matrices</td>
<td></td>
<td></td>
<td>16</td>
<td>4</td>
<td>100%</td>
<td>-</td>
<td>-</td>
<td></td>
<td>50%</td>
<td>50%</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use advanced Language</td>
<td>P20</td>
<td>Simplify conditional statement</td>
<td></td>
<td></td>
<td>24</td>
<td>3</td>
<td>33%</td>
<td>67%</td>
<td>-</td>
<td>67%</td>
<td>-</td>
<td>33%</td>
<td>33%</td>
<td></td>
</tr>
<tr>
<td>P21</td>
<td>Migrate from Dict, Set, List constructors to literals</td>
<td></td>
<td></td>
<td>42</td>
<td>4</td>
<td>25%</td>
<td>75%</td>
<td>-</td>
<td></td>
<td>25%</td>
<td>75%</td>
<td>-</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td>Features</td>
<td>P22</td>
<td>Transform to Python List, Dict, or Set Comprehension</td>
<td></td>
<td></td>
<td>349</td>
<td>6</td>
<td>33%</td>
<td>67%</td>
<td>-</td>
<td></td>
<td>67%</td>
<td>33%</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>439</td>
<td>6</td>
<td>33%</td>
<td>67%</td>
<td>-</td>
<td></td>
<td>67%</td>
<td>33%</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

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` [61] 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 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.

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

1. return sum(first_occ) / len(first_occ)
2. 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. [15] 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 observed developers update the type of matrices, e.g., migrating to an int64 matrix (P19) as shown in Listing 7.

**Listing 7: Commit 8f9cabbf in stellargraph/stellargraph: Update matrix type**

```python
1. np.hstack((A.row[:, None], A.col[:, None]))
2. np.hstack((A.row[:, None], A.col[:, None]))

```

Other patterns include changing data visualization with Matplotlib [48][P16], e.g., between drawing all the plots in one figure vs using an individual figure for each plot. Another pattern is using complex ML APIs (P17). Developers often traverse datasets multiple times (which is inefficient). A more efficient solution is to apply a composite operation. In Listing 8, developers compute dot product on three matrices instead of applying a np.multi_dot.

**Listing 8: Commit 180646fa in scikit-learn: Composite APIs**

```python
1. denominator = np.dot(np.dot(W.T, W), H)
2. 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. [42] 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 developers sought automation in IDEs.

5.1.6 **Trend 4 - Use advanced language features:** Python offers powerful features: (i) functions [65] 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) (see Listing 9). (iii) Python comprehension [66] to make code concise and inline for loops (P22). Researchers [10, 45] observed ML code extensively operates on data, which results in expressions that are longer and more complex than in traditional systems. Good software engineering principles [26, 47] 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, and 30% of developers sought automated help.

**Listing 9: Commit 15d7634d in RasaHQ/rasa: Use set literals instead of set constructor**

```python
1. set(utils.module_path_from_instance(p)
   2. for p in agent.policy_ensemble.policies)
   3. for p in agent.policy_ensemble.policies)
```

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 value 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 [73], 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 Signed-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.

<table>
<thead>
<tr>
<th>Refactoring Kind</th>
<th>Precision (±TP)</th>
<th>Refactoring Kind</th>
<th>Precision (±TP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename Method</td>
<td>96.32% (183)</td>
<td>Move And Rename class</td>
<td>96.67% (116)</td>
</tr>
<tr>
<td>Move Method</td>
<td>96.3% (136)</td>
<td>Move class</td>
<td>100% (160)</td>
</tr>
<tr>
<td>Pull Up Method</td>
<td>86.46% (83)</td>
<td>Extract Class</td>
<td>98.82% (84)</td>
</tr>
<tr>
<td>Push Down Method</td>
<td>89.36% (84)</td>
<td>Extract Subclass</td>
<td>100% (45)</td>
</tr>
<tr>
<td>Extract Supercall</td>
<td>95.24% (40)</td>
<td>Parameterize Variable</td>
<td>87.23% (82)</td>
</tr>
<tr>
<td>Split Parameter</td>
<td>91.66% (22)</td>
<td>Move &amp; Rename Method</td>
<td>88.08% (133)</td>
</tr>
<tr>
<td>Rename Class</td>
<td>99.2% (124)</td>
<td>Remove Parameter</td>
<td>97.56% (146)</td>
</tr>
<tr>
<td>Rename Parameter</td>
<td>98.11% (104)</td>
<td>Rename Parameter</td>
<td>95.9% (146)</td>
</tr>
<tr>
<td>Add Parameter</td>
<td>100% (192)</td>
<td>Merge Parameter</td>
<td>91.0% (51)</td>
</tr>
</tbody>
</table>

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.
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 JAVAfPy to adapt Java AST-analysis tools to Python. We release all the validated refactoring instances on our companion website [21]. 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 to compute recall of tools [42, 85]. To the best of our knowledge, there is no such unbiased refactoring oracle for Python. 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)

A fast execution time persuades the users that the adapted tools can be run on their systems (even on a consumer laptop) in an acceptable amount of time. It will enable the creation of bigger datasets of repetitive code changes to strengthen the validity of empirical research or train learning-based code recommendation systems, as well as enable innovative uses of code change pattern mining at commit time. Therefore, we evaluated the execution times of the adapted tools and compared them with the Java counterparts.

Table 3: Analyzed data set and execution time.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>PyCPATMiner</th>
<th>PyRMINER</th>
<th>R-CPATMiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total changed methods</td>
<td>824K</td>
<td>1M</td>
<td>1.16M</td>
<td></td>
</tr>
<tr>
<td>Total AST nodes</td>
<td>92M</td>
<td>4.5B</td>
<td>4.6B</td>
<td></td>
</tr>
<tr>
<td>Total changed graph</td>
<td>8M</td>
<td>4M</td>
<td>4.7M</td>
<td></td>
</tr>
<tr>
<td>Total patterns</td>
<td>17K</td>
<td>24K</td>
<td>28K</td>
<td></td>
</tr>
<tr>
<td>Execution time</td>
<td>&lt;8hours</td>
<td>&lt;12hours</td>
<td>&lt;19hours</td>
<td></td>
</tr>
</tbody>
</table>

1 The data of the Java-CPATMiner is obtained from its original paper [58].

To measure the execution time of the tools, we executed the Python adapted RMining, 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, and Java 13.0.1 ×64.

First, we record the running time of the type inference tool, PyType [30] (version 2020.10.08). PyType is decoupled from JAVAfPy. Therefore, we computed the execution time of PyType separately. PyType took on mean 360ms and 61ms on median for type inference of all changed files in a commit. We pushed the inferred type information of all the changed files in all the studied projects' commits to a repository [69] in Github. Similar to Typeshed [86], 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. [85] 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. [85] 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 [85].

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. [58] 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 of Python ML software (RQ1, RQ3). In the past, researchers exploited the repetitiveness of changes in Java systems through: code completion [13, 34, 43, 55, 56], automated program repair [6, 9, 50], API recommendation [32, 55], type migration [40], library migration [3, 18, 24, 39, 84], and automated refactoring [19, 28]. Using our rich and diverse dataset of 28K change patterns instances and our JAVAfPy, 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 [22]. Previously, researchers have built infrastructure to study many aspects of software evolution. For example, RMINER [85] and RefDiff [77] mine refactorings, TypeFactMiner [42] mines type changes, MigrationMiner [3] and API Migrator [25] mine API migrations, CPATMiner [58] and CodingTracker [54] 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.
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. [82] introduce a taxonomy of refactoring kinds performed in Java ML systems, Humbatova et al. [35], Islam et al. [37, 38] 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 [16, 32]. 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 [10] observed ML libraries are core components of ML systems and are frequently evolving, which causes developer frustration [22, 92]. 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, Pt-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 [11, 22]. Library developers deprecate APIs, introduce more efficient alternative APIs (e.g., Table 1 : P18), and split ML libraries [22]. Our findings, the accompanying dataset [21], 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 [23, 49].

6.4 Software Developers and Educators

S1. Rich educational resource (RQ1) Developers learn and educators teach new programming constructs through examples. Robillard et al. [71] 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 [21].

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 [30], a mature tool developed by Google. Thousands of projects at Google and other places rely upon PyType to keep their codes well-typed [30]. 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 [14, 88] 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 [21].

8 RELATED WORK

8.0.1 Studies on evolution of ML software systems. Researchers have studied repetitive tasks of ML systems from many aspects.
Researchers reimplemented certain Java AST mining tools for Python from scratch. PYREF [5] is a Python reimplement of Java RMiner [85] that identifies 9 kinds of Python refactoring. In contrast, our PyRMiner does not require reimplementing from scratch, and it finds all legitimate Java refactoring in Python. Moreover, we manually validated 18 kinds of refactoring, as shown in Table 2. Golubev et al. [29] developed PythonCHANGEsMiner, a tool that uses fgPDG [58] to mine Python repetitive code changes involving function calls. In contrast, we present JavaFyPy, a technique for adapting Java AST mining tools to Python without having to rewrite them from the ground up. We adapted CPATMiner [58] and RMiner [85] to Python using JavaFyPy, and subsequently created R-CPATMiner, a refactoring-aware code change pattern miner for Python systems.

Researchers proposed several techniques that infer specific kinds of code change patterns. For example, Revisar [72], GetAFix [6], and DeepDelta [50] infer repeated bug fixes and compilation errors from commit histories. LibSync [60], MEDITOR [89], and A3 [44] infer the adaptations required to perform library migration. Ketkar et al. [41] developed TCIInfer a technique to accurately infer rewrite rules for type changes from a project’s version history. They also proposed INTELLlTC [81], a configurable IDE refactoring plugin that automates these inferred type changes. 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 [21] 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, Rahul Khanna, Bob Banfield, Julia Romero, Carla Pomian, Dorin Pomian, students from CSCI 7000-08-Spring 2022 at CU Boulder, CUPVL 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 grants CCF-1553741, CNS-1941898, and the Industry-University Cooperative Research Center on Pervasive Personalized Intelligence.