Discovering Repetitive Code Changes in Python ML Systems

Anonymous Author(s)

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 best coding 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 surveyed 97 ML developers to further shed light on these patterns and their applications, and we received a 15% response rate. We present actionable, empirically-justified implications for four audiences: (i) researchers, (ii) tool builders, (iii) ML library vendors, and (iv) developers and educators. *A full version of this work was published in the ICSE-2022 technical track, with artifacts available and reusable badges.*

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Refactoring, Repetition, Code changes, Machine learning, Python

1 INTRODUCTION

Many software changes are repetitive by nature [4, 23, 36], 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* [42]. Since this change involves programming idioms [1, 53] 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 [10, 24, 32, 38, 39], automated program repair [3, 6, 33], API recommendation [22, 38], type migration [30], library migration [2, 12, 17, 29, 55], code refactoring [13, 19], fine-grained understanding of software evolution [2, 18, 31, 37, 40, 50, 56]. Unfortunately, these are mostly available only for Java, and do not support Python and ML systems.

Researchers [8, 15, 27, 47] 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 [15, 58]. 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 [25, 27, 28], updating ML libraries [15], refactoring and technical debt of ML systems [49, 54], managing version control systems for data [5], and testing [7, 21, 26]. 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 Python ML systems? 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.

We employ both quantitative (mining repositories and thematic analysis) and qualitative methods (surveys) to answer the research questions. For the quantitative analysis, we use a large data set of 1000 ML projects from GitHub. We extracted 28,308 fine grained code change patterns where 58% of them appear in multiple projects. We applied *thematic analysis* [9, 57] 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 [2, 18, 31, 37, 40, 50, 56] 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 [40]. CPatMiner matches changed methods and their body statements across the commits and identifies fine-grained change patterns. Refactorings such as move, rename, and extract method, re-arrange and obfuscate the code statements, that are hard to match across the edit, leading CPatMiner to miss multiple occurrences of patterns. To improve the accuracy of CPatMiner, we integrate it with the state-of-the-art refactoring mining technique- RefactoringMiner [56], that de-obfuscates the re-arranged 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 [27, 28, 49], (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.
(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 class TimeDelayNN(_FERNeuralNet):
6 -    def _init_model(self):
7 -        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 [43] 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 automation support for this *code change pattern*? Researchers have proposed advanced techniques to mine such fine-grained change patterns from the commit histories [40, 41]. However, these techniques are inapplicable to mine the *fine-grained code change patterns* shown in Listing 2 because (i) their techniques mine code change patterns for Java, and (ii) they do not account for overlapping refactorings.

Researchers [34, 35, 51] have shown that developers often interleave many programming activities such as bug fixes, feature additions, or other refactoring operations, and often these changes overlap [37] (as shown in Listing 2). Such overlapping changes and refactorings can easily obfuscate existing fine-grained change pattern mining tools [40, 41] because they do not account for these changes when matching code across the commit. For example, CPatMiner [40] does not match the method body of `_init_model` in the class `_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 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 [40], to Python. To make CPatMiner *refactoring aware*, we adapt the state-of-the-art Java refactoring inference tool, *RefactoringMiner* [56] (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.

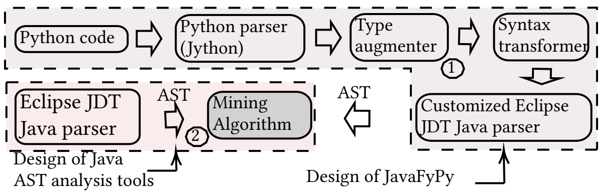

Figure 1: Design of JavaFyPy and existing AST analysis tools

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.

Figure 2 shows an example of the code transformation steps (shown in Figure 1) that JavaFyPy performs automatically. 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 [40, 56] 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 PyType [20], 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 Python AST nodes? 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 Python 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.

3.1 Refactoring Aware Change Pattern Mining

3.1.1 ***Adapting CPatMiner.*** CPatMiner [40] 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. We extended CPatMiner to Python using JavaFyPy as Py-CPatMiner. We prefix all the adapted tool names with *Py* to disambiguate the tool names from their Java counterparts.

3.1.2 ***Introducing Refactoring Awareness.*** As discussed in Section 2, CPatMiner [40] does not account for the overlapping refactorings applied in the commit. To overcome this, we made the CPatMiner refactorings aware by integrating it with RMiner [56]. We used JavaFyPy to adapt RMiner and use it to detect 18 refactoring kinds that move code blocks. 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 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 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. To highlight the improvement caused by R-CPATMiner, we compared, number of change graphs, and number of patterns reported by both R-CPATMiner and PyCPatMiner. R-CPatMiner processed 16% more changed methods, 0.1B more AST nodes than PyCPatMiner. R-CPatMiner produces **16%** more *change graphs*, **15%** more *change patterns*, thus confirming the value of de-obfuscating change graphs that were previously obfuscated by refactoring.

4 RESEARCH METHODOLOGY

What are the frequent code change patterns in ML code, and what patterns need automation? To answer this research questions, 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.

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 [15] to understand the challenges of evolving ML systems. This corpus [15] 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.

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. [36]. 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

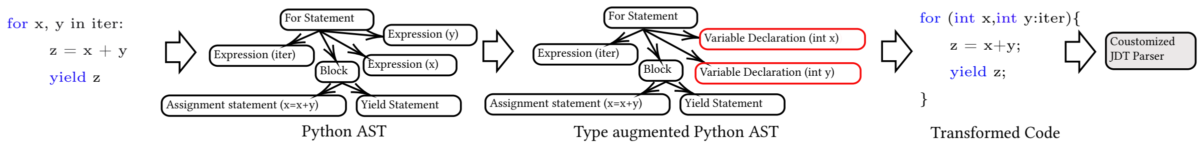

Figure 2: An example Code transformation performed by JavaFyPy

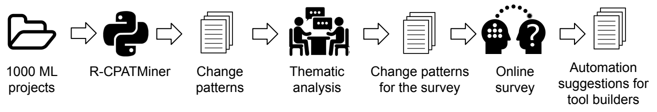

Figure 3: Flow diagram of the RQ3's research methodology

and projects are stable [41] 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. Following the best practices guidelines from the literature, the authors used negotiated agreement technique to provide themes for the patterns [11, 57]. We identified four trends (themes) of patterns based on their structural similarity at the statement 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. We asked:

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?

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% [52].

4.4 Results

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. 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.

4.5 Discovering pattern trends

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*- 8 patterns, (2) *convert `for` loops to domain specific abstraction*- 5 patterns, (3) *update API usage*- 6 patterns, (4) *use advanced language features*- 3 patterns. Next, we summarize and triangulate results obtained from source code mining, thematic analysis, and developer surveys. We describe only one pattern per trend below. Our companion website [14] presents all the patterns and a curated repository of exemplars for each pattern, as well as and our ICSE-2022 technical paper [16] explains each pattern in detail.

4.5.1 ***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 [46] that invokes the Context manager. We observed 1,237 change instances belonging to eight pattern groups where developers move code blocks into `with` statements and use *Context managers*.

Listing 3: Commit dfb7520c in Pytorch: Disable gradient

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

Listing 3 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. 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.

4.5.2 ***Trend 2 - Convert `for-loops` into domain specific abstraction:*** Listing 1 shows one such example where the developer uses `np.sum` from *NumPy* [42] 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"*. 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

4.5.3 ***Trend 3 - Update API usage:*** Listing 4 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."* 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.

Listing 4: Commit 8592777b in inspirehep/magpie: Migrate API to NumPy

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

4.5.4 ***Trend 4 - Use advanced language features:*** Python offers powerful features: (i) functions [44] such as `bool` and `isinstance` that can be used to simplify a conditional statement, (ii) literals such as `[]`, `{}`, `()` to efficiently create containers instead of using constructors such as `list()`, `dict()`, `tuple()` (see Listing 5). (iii) Python comprehension [45] to make code concise and inline `for` loops. 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. 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 5: Commit 15d7634d in RasaHQ/rasa: Use set literals instead of set constructor

```
-set(utils.module_path_from_instance(p)
    ↪ for p in agent.policy_ensemble.policies)
+{utils.module_path_from_instance(p)
    ↪ for p in agent.policy_ensemble.policies}
```

5 IMPLICATIONS

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

5.1 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 [10, 24, 32, 38, 39], automated program repair [3, 6, 33], API recommendation [22, 38], type migration [30], library migration [2, 12, 17, 29, 55], and automated refactoring [13, 19]. For this purpose, researchers have built infrastructure to study many aspects of software evolution. For example, *RMiner* [56] and *RefDiff* [50] mine refactorings, *TypeFactMiner* [31] mines type changes, *MigrationMiner* [2] and *APIMigrator* [18] mine API migrations, *CPATMiner* [40] and CodingTracker [37] mine fine-grained repetitive code changes in Java. Adapting the above tools to Python using JavaFyPy, together with a dataset of 28K change patterns, researchers can provide Python ML systems with the same benefits as traditional systems.

5.2 Tool Builders and IDE Designers

T1. New inspirations for tool development (RQ1). To help tool builders invest resources where automation is most needed, 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. 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, (iii) change variable scopes, and (iv) execute dependencies. Respondents further suggested tools to (i) move Context manager calls to `with` statements, and (ii) detect misuses of Context managers.

5.3 Software Developers and Educators

S1. Rich educational resource (RQ1) Developers learn and educators teach new programming constructs through examples. Robillard et al. [48] 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 [14].

6 RELATED WORK

The closest related work is by Tang et al. [54]. The authors manually studied 327 code patches in 26 ML projects and described 14 new ML-specific refactoring categories. In contrast, we used an automated approach to longitudinally mine the source code to extract not only refactorings but also all the frequently repetitive code changes. Negara et al. [36] and Nguyen et al. [41] study code change patterns in classical software systems whereas we study Python based ML systems and provide suggestions for automation.

7 CONCLUSION

We introduce a novel tool, **R-CPATMiner** that performs refactoring-aware change pattern mining in Python. Using the tool, we extracted code change patterns in ML systems and surveyed 97 developers to seek their opinion on automation. We released rich datasets and tools that have empirically-justified, actionable implications for researchers, tool builders, ML developers, and educators. Our framework can be used to discover change patterns in any Python code base. In this paper, we apply it to a corpus of ML codebases, so some of the patterns are unique to ML, while others are best practices that can be found in any domain.

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. Migration-Miner: 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] 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

[4] 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

[5] 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

[6] 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)*. ACM, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

[7] 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

[8] 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

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

[10] 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

[11] 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

[12] 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

[13] 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

[14] Malinda Dilhara. 2022. *Discovering Repetitive Code Changes in Python-based ML Systems.* https://mlcodepatterns.github.io Accessed: 2022-02-07.

[15] 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

[16] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering Repetitive Code Changes in Python ML Systems. In *International Conference on Software Engineering (ICSE'22). To appear.*

[17] 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

[18] 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

[19] 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

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

[21] 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)*. ACM, New York, NY, USA, 851–862. https://doi.org/10.1145/3368089.3409754

[22] 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

[23] 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

[24] 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

[25] 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)*. ACM, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[26] 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)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/3460319.3464825

[27] 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

[28] 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)*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

[29] 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)*. ACM, New York, NY, USA, 154–164. https://doi.org/10.1145/2901739.2901769

[30] 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

[31] 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

[32] 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.

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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.* ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2950290.2950333

[39] 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.

[40] 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

[41] 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

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

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

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

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

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

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

[48] 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

[49] 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.

[50] 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

[51] 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

[52] 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.

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

[54] 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

[55] 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

[56] 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

[57] 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

[58] 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).* ACM, New York, NY, USA, 1159–1170. https://doi.org/10.1145/3377811.3380362