IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

Impact Factor: 7.301

Volume 3, Issue 2, January 2023

Discovering Repetitive Code Changes in Python ML Systems

Dr. Manjunath HR[1], Ravindra Reddy , Yashvardhan SG[3], Raviraj[4], Sahana[5]

Assistant Professor, Department of Information Science and Engineering[1]

Students, Department of Information Science and Engineering[2 ,3,4,5]

Alvas's Institute of Engineering and Technology, Mijar, Moodbidre, Karnataka, India

Abstract: *Over the years, academics have automated numerous software evolution jobs by taking advantage of the repetitive nature of software modifications. Python-based machine learning systems have become extremely popular, yet they do not profit from these developments. without specifying are the frequent updates made by ML developers, the missed chances for automation by academics, tool and library designers, and the failure of ML developers to understand and adhere to basic coding practises. We conducted the first and most thorough study on code modification trends across a broad corpus of 1000 top-rated ML systems totaling 58 million SLOC in order to close the knowledge gap and enhance the science and tooling in ML software evolution. We repurpose, modify, and enhance cutting-edge repetitive change mining approaches to carry out this investigation. R-CPatMiner, our cutting-edge tool.*

CCS CONCEPTS, Software and its engineering, Software maintenance tools; Computing methodologies, Machine learning

Keywords: Refactoring, Repetition, Code changes, Machine learning, Python

I. INTRODUCTION

The recurring nature of many software updates leads to the formation of change patterns. Machine Learning (ML) developers also undertake repetitive code updates, just like in conventional software systems. Listing 1 illustrates a typical modification where ML

Developers used np.sum, a highly efficient domain-specific abstraction offered by the package NumPy in place of a for loop that sums the list members. This modification is fine-grained because it affects programming paradigms at the sub-method level. It is a fine-grained code change pattern if this modification is repeated in several places or in multiple contributions.

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)

For example, code completion in IDEs , automated programme repair [6 , 9 , 50] , API recommendation , type migration library migration , code refactoring , and fine-grained understanding of software evolution are all examples of applications that rely on the repetitiveness of changes. Sadly, most of these are only accessible for Java and do not support Python or machine learning systems.

1. As far as we are aware, we carried out the first and biggest analysis on 28,308 fine-grained code modification patterns on ML systems. We discovered trends in code alterations. We used thematic analysis to group the top 2,500 patterns into 22 themes that are specific to changing patterns, revealing four main trends.
2. To get insight into the causes driving those changes, the existing methods of implementing those changes, and their advice for tool designers, we created and administered a survey to 650 open-source ML developers.
3. To gather fine-grained change patterns used in the evolution history of Python-based ML systems, we created innovative methods.

We used these technologies on 1000 GitHub-hosted open-source projects. We release the tools and information we've gathered to the general public.

II. MOTIVATING EXAMPLE

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

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

The code modification in Listing 2 designates the hardware device for the TensorFlow operation in line 4 using tf.device(). Context Manager from the ML library is what tf.device() TensorFlow. The developer has combined this fine-grained code update with a Pull up Method refactoring, which transfers the TimeDelayNN's _init - model to the parent class _FERNeuralNet.

Is it common practise to specify hardware for TensorFlow operations? How often does this trend occur? Does this code change pattern require tool support for developers to recommend and automate it? If a comparable code change was made in the past of this project or any other project, we will see this instance of fine-grained code modification as a repeating pattern. Advanced methods to mine such fine-grained minerals have been proposed by researchers.

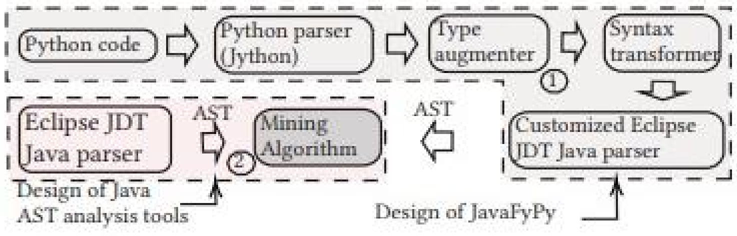


Figure 1: Design of JavaFyPy and existing AST analysis tools

Their methods only mine code change patterns for Java, and they do not take into consideration overlapping refactorings, the fine-grained code change patterns displayed in Listing 2.

Researchers have shown that programmers often interleave a variety of programming tasks, such as bug patches, feature additions, or other refactoring procedures, and that these changes frequently overlap . Due to the fact that these modifications are not taken into account when comparing code throughout the commit, such overlapping changes and refactorings can readily obscure existing fine-grained change pattern mining techniques . In CPatMiner , for instance, the method body of _init model in the class _FERNeuralNet and the body of _init model in the class TimeDelayNN are not identical.

It will take a large amount of work to adapt the current Java AST mining tools for Python. Additionally, it is not practical nor long-term because researchers are constantly

Introducing fresh Java AST mining tools or enhancing current ones. To do this, we suggestJavaFyPy, a method that takes use of the similarity between the Java and Python abstract syntax trees to convert existing Java AST mining tools to Python (AST). We convert the cutting-edge fine-grained change pattern mining programme, CPatMiner , to Python using JavaFyPy. We convert the cutting-edge Java refactoring inference tool, RefactoringMiner (also known as RMiner), to Python and combine it with CPatMiner as R-CPatMiner in order to make CPatMiner refactoring aware.

III. TECHNIQUE

Because they only work with abstract syntax trees (AST), the majority of the existing code change mining tools (also known as AST mining tools) are theoretically language-independent.

Nevertheless, its application is restricted to Java only. We provide a highly practical alternative

JavaFyPy - to get over this practical implementation constraint. This method converts the input Python programme into an AST that can be analysed by the mining algorithm of currently available Java AST analysis tools. By making the AST-based mining tools developed for Java programmes usable for Python systems, JavaFyPy will expedite research

and tool development. As a result, it will reduce the number of development hours needed to re-implement these strategies.
Python code is utilised as an input, and it generates an AST that may be used by Java AST analysis tools' mining methods. To do this, JavaFyPy converts the Python code to an AST first before enhancing it with type information. The Syntax converter then converts the relevant Java concrete syntax into AST nodes. It generates the finished AST using the Java parser (Eclipse JDT). The most often used Java parser in AST mining research tools is Eclipse JDT. As a result, we decided to use Eclipse JDT as the parser to create the final AST. The mining algorithms of Java AST analysis tools can process this improved and enriched AST. Researchers and tool developers can adapt their tools for Python by using JavaFyPy.

3.1 Python Code Transformation

Definition 3.1. (AST) Let T be an AST. T has one root. Each node Ni ∈ T has a parent (except the root node). Each node (Ni ∈ T) has a sequence of child nodes (denoted by CNi). Number of nodes in the sequence CNi is denoted by LengthCNi . Each node Ni is a specific syntax category known as AST node kind, KindNi = { 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 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 Tj be a Java AST and Tp be a Python AST. Nj∈Tj, Np ∈ Tp. We state that Nj and Np are mapped AST node kind, if Nj and Np maintain a structural similarity. Mapped node of node Np is denoted by M(Np) = Nj. 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 CNj be a sequence of child AST nodes of a parent Java node Nj and CNp be a sequence of child AST nodes of a Python node Np. We state that Nj and Np are identical AST nodes if (i) M(Np) is Nj, and (ii) ∀Ni ∈CNp : M(Np) ∈CNj .
Identical AST node (Definition 3.3) - 72 % of the Python's AST node Kinds can be identically mapped to a Java's AST Node. For example, we mapped Python's If to Java's If Statement and mapped Python's Assign to Java's AssignmentStatement.
Definition 3.4. (Nearly Identical AST node) We state that Nj and Np as nearly identical AST nodes, if Nj and Np meets conditions: (i) M(Np) is Nj, and (ii) ∃Ni ∈CNp : M(Ni) ∉CNj .
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 , thus AST of Python for loop contains additional child AST node kinds.
Definition 3.5. (Unique AST node) Let Np be a Python AST node. We state that Np is unique to Python, if there is no mapped AST node in Tj. i.e., (M(Np) ∉Tj).

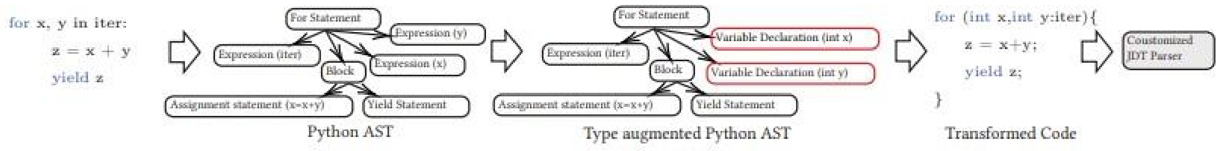


Figure 2: An example Code transformation performed by JavaFyPy

3.2 Refactoring Aware Change Pattern Mining

3.2.1 Adapting CPatMiner : The cutting-edge code change pattern mining application CPatMiner [58] mines previously unidentified fine-grained changes from the git commit history using an effective graph-based representation of code changes. It leverages the Eclipse JDT Java parser [27] to produce an AST of Java source code and iterates through modified methods in each commit. The mining algorithm then independently constructs program-dependence graphs for each AST node before merging the graphs to produce the change graph, a large graph. For each method that is updated, CPatMiner creates change graphs that show the before and after of a source code change and may be used to discover trends in code change. We utilised most of the features as 72% of the Python AST node types are identical to those in Java.

3.2.2 Introducing Refactoring Awareness

The overlapping refactorings that were used in the commit are not taken into consideration by CPatMiner . These refactorings make it challenging to match the modified code blocks because they shift code blocks between methods or alter the method signature. therefore passing up the chance to provide change graphs for the modifications that were obscured. To get around this, we integrated CPatMiner with RMiner in order to make the refactorings aware . We modified

RMiner with JavaFyPy and used it to find 18 different types of refactoring that shift code blocks. In order to match shifted code blocks, RMiner employs an AST-based statement matching algorithm to match classes, methods, and statements inside method bodies. In order to expand RMiner's statement matching technique and use it to justify the Unique and Nearly Identical AST Node types, we consulted the creators of RMiner.

IV. RESEARCH METHODOLOGY

To distinguish the tool names from their Java equivalents, we prefix all of the modified tool names with Py. We start by gauging how well the tools we created work (or adapted). Then, we examine the repetitious code modifications using our trusted and tested techniques. used in ML systems in Python. We respond to three research queries for this purpose:

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.

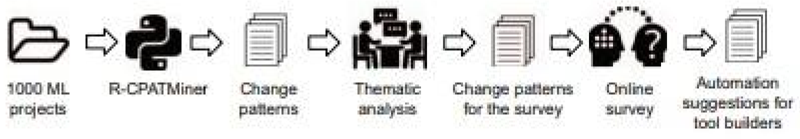


Figure 3: Schematic diagram of the research methodology to answer RQ3

4.1 Subject Systems

Our corpus consists of 4,166,520 commits from 1000 large, developed, and diverse machine learning application systems, totaling 58 million lines of source code and 150 thousand Python files, which have been used by other researchers to comprehend the difficulties associated with developing ML systems. According to this corpus's analysis of Python files, LOC, contributors, and commits, it is quite varied. They are made up of a variety of frameworks, online utilities, databases, and ML-using robotics software systems, and they differ greatly in both their domain and application. Additionally, we expanded our subject systems with low-level ML libraries like

Scipy and SpaCy and high-level ML libraries like Keras, PyTorch, Caffe, NLTK, and Theano. This guarantees that our dataset is enough in size and representative to fully address our research topics.

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.

We adhered to Negara et al. recommended procedures because there are many mined patterns. They arranged the patterns in three dimensions: by pattern frequency (F), by pattern size (S), and by F S. We also took into account the number of projects and authors as an additional two dimensions since repeated modifications made by several developers and projects are stable and are more likely to be automated. The mined patterns were then arranged along all five dimensions. The top 500 patterns for each of the five dimensions were then manually examined by two of the authors, who have more than three years of professional software development experience and in-depth knowledge of software evolution. These meaningful code patterns, or the "best practises," were then identified.

Each modification pattern was personally examined by the paper's two authors to determine the high-level programming job carried out in each pattern. The authors employed the negotiated agreement approach while adhering to the best practises recommendations from the literature.

To come to an understanding . The change patterns were meticulously coded by the paper's two authors, who each gave each pattern one or more descriptors (i.e., codes). Following the coding of around 25 % of the data (the recommended minimum size is 10 %) , both authors conducted the initial meeting. The authors meticulously reviewed how each pattern was coded at the conference. They also discussed any discrepancies that could not be characterised as high-level programme transformations between the given codes and the patterns. After 80 % inter-coder agreement (recommended inter-coder) was attained.

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

Using the given email addresses, we sent an email to the developers making repeating code modifications that we thought warranted additional study.

inside their GitHub profile. The application we created automatically generated each email message's body, which contained the following data:

- Introduction to the research team and the purpose of the study.
- A plot of number of repetitive changes done in the project.
- A link to the frequent repetitive changes done in the respective project so that the developer can use it as an educational resource.

The following four questions for the developer:

- **Q1.** What are the reasons for performing 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 purpose of the first question is to ascertain the real drivers for a code change as stated by the developers themselves. The second query focuses on how frequently ML programmes are changed, which aids in determining if ML-specific IDE tools are required. The final three questions probe developers' faith in and utilisation of tool help when making code modifications. This is significant since ML programmes have limited IDE assistance for making code modifications . The associated website has a sample email .

We sent 650 emails to developers in all, and 97 of them replied, giving us a 15 % response rate. This is much more responses than are typically received using questionnaire-based software.

IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

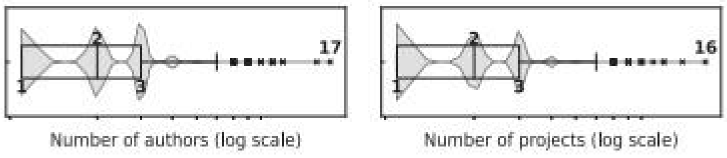
Impact Factor: 7.301

Volume 3, Issue 2, January 2023

V. RESULTS

5.1 Repeated Modifications to ML Systems (RQ1)

Features of the patterns discovered by R-CPatMiner. On the 1.5M modified source code files and over 490M lines of source code in our corpus mentioned in Section 4.1 , we ran R-CPatMiner. R-CPatMiner generated 349,406 change graphs with 4.7 M nodes for these modified files. The technique retrieved 28,308 distinct code modification patterns, 63% of which are conducted by several authors, and 58 % of which are performed in numerous projects. The degree of pattern sharing between developers and projects is seen in Figure 4. We found that 79% of developers that conducted code change patterns shared at least 50 % of their patterns with other developers, and 53% shared 100% of their change patterns with other developers.



(a) Developers (b) Projects

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

5.1.2 Finding Patterns and Trends

To progress science and tooling in the development of ML software, it is critical to comprehend the patterns of code modification that ML engineers use. In 22 previously unidentified recurrent change pattern groups, where the developers want for automation, our thematic analysis and developer surveys have found.

Amongst these patterns, we identified four major trends based on their structural similarity (i.e., expression- and statement-level):

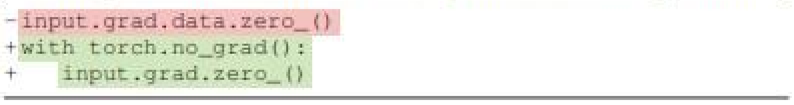
1. Transform to Context managers- 1237 instances.
2. Convert for loops to domain specific abstraction- 239 instances.
3. Update API usage- 166 instances.
4. Use advanced language features- 415 instances.

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

5.1.3 Trend 1: Context Managers Must Transform

An abstraction for managing the life cycle of a code block is a Python context manager. It specifies the __enter__ (initialization) and __exit__ (finalisation) methods, which when combined establish the intended runtime environment for a code block's execution. It is necessary to enclose the code block with a with statement [68] that calls the context manager. When developers insert code blocks into with statements and employ context managers, we saw 1,237 change occurrences that belonged to eight pattern groups (P1-P8 in Table 1).

Listing 4: Commit dfb7520c in Pytorch: Disable gradient



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

The pattern P2 (Disable or enable gradient computation) is seen in Listing 4. According to survey respondent S21, "it is crucial to deactivate the gradient calculation globally to minimise memory usage and boost speed when we do not need gradient computation in a DL network (using Tensor.backward())." In line 3 of the code, the context manager torch.no grad() from PyTorch sets an execution environment and turns off the gradient calculation. Similar to this, the patterns (P2-P8) in Table 1 produce different settings for execution.

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

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

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

Static Analysis						Survey Responses										
						Q_2				Q_3				Q_4		
Trend	P	Pattern	ML[1]	I	R	VO	O	R	N	VO	O	R	N	Yes	No	AA
Move to `with` statement and use Context managers	P1	Read, write, traverse data	×	467	6	67%	17%	-	17%	83%	-	17%	-	100%	-	-
	P2	Disable or enable gradient calculation	✓	92	5	100%	-	-	-	80%	-	20%	-	100%	-	-
	P3	Swap ML training devices	✓	21	3	100%	-	-	-	67%	33%	-	-	67%	33%	-
	P4	Change name and variable scopes in DL networks	✓	106	7	57%	43%	-	-	57%	29%	14%	-	57%	43%	-
	P5	Execute dependencies of a Tensorflow graphs	✓	57	3	100%	-	-	-	100%	-	-	-	33%	67%	-
	P6	Temporarily change configurations of libraries	×	59	2	-	50%	50%	-	100%	-	-	-	50%	50%	-
	P7	Transform to context managers in pytest	×	335	6	33%	50%	17%	-	67%	33%	-	-	67%	33%	-
	P8	Use context managers to open temporary directories	×	100	6	33%	50%	-	17%	67%	-	33%	-	83%	17%	-
Dissolve `for` loops, into domain specific abstractions	P9	Transform to optimized operations in NumPy	✓	179	10	100%	-	-	-	90%	-	10%	-	100%	-	-
	P10	Transform to operations in *List* or *Dictionary*	×	24	3	67%	33%	-	-	33%	-	67%	-	33%	67%	-
	P11	Transforming to Python built in functions	×	15	2		100%	-	-	-	-	100%	-	100%	-	-
	P12	Transform functions in *String*	×	14	2	-	50%	50%	-	100%	-	-	-	100%	-	-
	P13	Transform to set operations	×	7	2	100%	-	-	-	50%	-	50%	-	100%	-	-
Update API usage, (212)	P14	Migrating to APIs ML libraries	✓	26	5	100%	-	-	-	100%	-	-	-	100%	-	-
	P15	Transform Matrix	✓	82	6	83%	17%	-	-	100%	-	-	-	33%	50%	-
	P16	Change data visualization	✓	28	2	50%	50%	-	-	50%	-	50%	-	50%	50%	-
	P17	Composite ML APIs	✓	22	5	100%	-	-	-	100%	-	-	-	100%	-	-
	P18	Update Container	×	115	5	60%	-	40%	-	60%	-	40%	-	80%	20%	-
	P19	Update Type of Matrices	✓	16	4	100%	-	-	-	50%	-	50%	-	50%	50%	-
Use advanced Language Features	P20	Simplify conditional statement	×	24	3		33%	67%	-	67%	-	-	33%	33%	-	67%
	P21	Migrate from Dict, Set, List constructors to literals	×	42	4		25%	75%	-	25%	-	-	75%	25%	-	75%
	P22	Transform to Python List, Dict, or Set Comprehension	×	349	6		33%	67%	-	33%	-	-	67%	33%	-	67%

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

update all the initialisation and finalisation code when they update the library versions (if the APIs have changed). The respondent S13 said, “I envision IDEs that automate moving Context managers to with statements.” Table 1 tabulates the results for each major trend and pattern group, and shows survey responses for each pattern group. 90% of the survey respondents who performed Trend-1 changes confirmed that they move to with statements very often (VO) or often (O). All respondents perform the code transformation manually, and 74% of the respondents requested automation in their IDEs.

5.2 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.

Table 3 demonstrates that compared to PyCPatMiner, R-CPatMiner processed 16 % more modified methods and 0.1 B more AST nodes. For each pair of mapped code blocks (i.e., before and after the modified method body), PyCPatMiner creates a separate change graph. The value of de-obfuscating change graphs that have been previously obfuscated by refactoring is therefore confirmed because R-CPatMiner generates 16 % more change graphs. The R-CPatMiner then extracts repeating isomorphic sub-graphs as patterns from each of the resulting change graphs.

CPATMiner follows the Rule of Three] , a commonly used recurrence measure in pattern analysis, by setting the minimum frequencies of repeated subgraphs to be three. As a result, each pattern has at least three instances of the code.

When we examined the amount of patterns produced by R-CPatMiner and PyCPatMiner, we found that R-CPatMiner captures 15% more patterns.

Additionally, we examined PyCPatMiner and R-distributions CPatMiner's of the number of code instances per pattern. We used the Wilcoxon Signed-Rank test to determine if there is a statistically significant difference between the distributions of the number of code occurrences per pattern given by PyCPatMiner and R-CPatMiner.

test the number of code occurrences for each pattern on the matched samples. At a significance level of 5 %, the test rejected the null hypothesis that PyCPatMiner produces a pattern with a higher density of code instances than R-CPatMiner does (p-value = 1.12 1010). Since the Hodges-Lehman estimator is acceptable for use with the Wilcoxon Singed-Rank test, we utilised it to calculate the difference between the two distributions. The value ended up being.

5.2.2 Evaluating PyRMiner's accuracy PyRMiner's

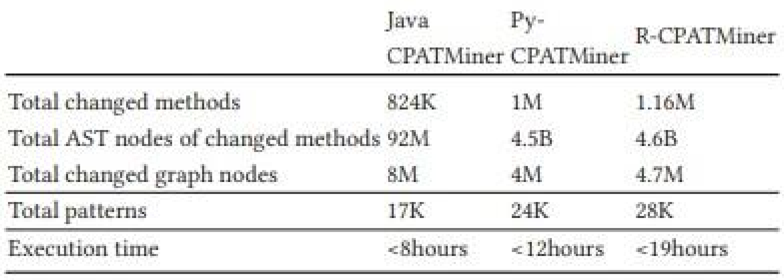
Accuracy is crucial since we utilise it to first match the refactored code blocks before passing them to R-CPatMiner. construct change graphs. We initially identified 18 refactoring types, i.e., refactorings that alter method signatures or move the method bodies, that conceal fine-grained modifications. Prior to selecting a statistically significant random sample of refactoring instances for each refactoring kind, we first ran the PyRMiner on our research corpus. Since the precision of the refactoring detection is only 5% for each type of refactoring, we may conclude with 95% confidence using a t-test, as shown in Table 2.

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

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

The customers are convinced that the customised tools can be executed on their systems (even on a consumer laptop) in a reasonable period of time thanks to a quick execution time. It will make it possible to build larger datasets of repeated code changes that can be used to train learning-based code recommendation systems, support the validity of empirical research, and support creative applications of code change pattern mining at commit time. As a result, we assessed the adaptable tools' execution times and contrasted them with their Java equivalents.

Table 3: Analysed data set and execution time.



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

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

We used a large corpus and the Python versions of RMiner, CPatMiner, and R-CPatMiner to measure the tools' execution times and compare them to the Java versions. On the same system, we ran each tool independently under the following conditions: MacOS 10.14.6, Java 13.0.1 64, 32 GB DDR4 RAM, and an Intel Core i9 processor running at 2.90 GHz. The type inference tool PyType's execution time is the first thing we track . (version 2020.10.08). PyType is independent of JavaFy.

VI. 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 , automated program repair , API recommendation, type migration , library migration , and automated refactoring. Using our rich and diverse dataset of 28 K change patterns instances and our JavaFyPy, researchers can bring the same benefits to Python ML systems.

R2. Foundations to study Python ML Software Evolution (RQ1, RQ2, RQ3).
Although Python ML systems are widely used, their evolution and maintenance chores are the least automated and investigated as a result of the lack of tool support for ML systems . Infrastructure has already been developed by researchers to investigate several facets of software evolution. For instance, refactorings and type modifications are mined by RMiner and RefDiff, API migrations are mined by MigrationMiner and APIMigrator , and fine-grained repetitive code changes in Java are mined by CPATMiner and CodingTracker .

R3. Improve current ML system research and tools (RQ1, RQ3):
Our infrastructure may be utilised to further current ML system research. For instance, Humbatova et al. and Islam et al. provide a taxonomy of defects in ML systems based on manual examination of StackOverFlow postings. Tang et al. introduce a taxonomy of refactoring types conducted in Java ML systems. These studies use a limited dataset of changes and a lot of manual analysis to create different taxonomies. They may greatly enhance or further validate their taxonomies using our far bigger dataset (28K change patterns). Researchers also demonstrated the possibility of using ML for code completion. However, they either use noisy or little data to train the ML models.

R4: Create innovative software for ML developers (RQ1).
Our resources can be used by researchers to create fresh apps. According to Braiek and Khomh , ML libraries are essential parts of ML systems and are constantly changing, which makes developing annoyance. A tutoring system can advise developers on the best constructions to utilise while updating their ML code. For instance, a tutoring system may advise modifying the code to substitute ML library APIs for for loops or to eliminate unnecessary matrix operations. Additionally, a tutor system may advise inexperienced programmers on how to use sophisticated language features .

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.

VII. THREATS TO VALIDITY

Internal Validity: Can we rely on the outcomes that tools produce? The accuracy of our tools to mine code change patterns in a refactoring aware manner is a prerequisite for the study's conclusions. We depend on type inference to add extensive type information to the AST.

Type inference, which determines the types of elements by statically analysing the programme, may not be as accurate at spotting them as run time. This may have an impact on the accuracy of the mappings that R-CPatMiner reports and the statements that RMiner matches.

We make use of PyType , a sophisticated tool created by Google, to lessen this vulnerability. PyType is used by thousands of projects at Google and other locations to maintain correctly typed programmes .

7.1 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 .

VIII. RELATED WORK

8. 1 Studies on Evolution of ML Software Systems

Researchers have studied repetitive tasks of ML systems from many aspects.A taxonomy of ML-related defects and bug repair patterns is introduced by Humbatova et al. and Islam et al. , respectively. Both Yan et al. and Shen et al. examine numerical flaws in Deep Learning (DL) systems. DL programme is observed by Zhang et al.

Microsoft's mistakes and the demand for DL-specific solutions to address DL problems. Both Lwakatare et al. and Zhang et al. categorise typical difficulties faced by ML developers when maintaining and advancing ML systems. In ML libraries, Wan et al. discovered anti-patterns and improperly utilised APIs. To speed up AutoML, Nguyen et al. compile a database of DL models taken from GitHub and use them as default models.

8.2 Research on Frequent Code Modifications

Numerous studies have been done by researchers on repeated code modifications. To mine fine-grained code, Nguyen et al. employ a graph-based method.

at the commit level, changes. They carry out a thorough investigation on the repetitiveness of code changes in Java software evolution and find that it is prevalent at low granularities (number of lines), and that it decreases exponentially as granularity grows. Researchers have also examined repetitiveness from the perspective of higher-level maintenance and evolution tasks. For example, I Dig and Johnson , (ii) Teyton et al. mine library migration trends to determine how frequently, when, and why they occur, and (iii) Ketkar et al. carry out a significant study on API compatibility between versions.

IX. CONCLUSION

The research of fine-grained code modification trends in Python-based ML software systems is presented in this work for the first time and in its entirety. We employ complementing empirical techniques to offer distinctive insights:

1000 software repositories totaling more than 58 million LOC were mined, groups and trends were identified using thematic analysis, and 97 machine learning developers were polled. We developed a unique method called JavaFyPy to reuse, adapt, and enhance the Java state-of-the-art AST mining tools in order to carry out this work and advance the science and technology in Python ML software evolution. We provide R-CPatMiner, an unique tool that mines change patterns in Python projects' version histories while taking refactoring into account. We provide 22 pattern groups of code changes in four trends, with 10 of them being ML-specific.

X. 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, CUPLV group at CU Boulder, and the anonymous reviewers for their insightful and constructive feedback for improving the paper. This research was partially funded through the NSF grants CCF-1553741, CNS-1941898, and the Industry-University Cooperative Research Center on Pervasive Personalized Intelligence.

REFERENCES

[1]. Carol V. Alexandru, José J. Merchante, SebastianoPanichella, Sebastian Proksch, Harald C. Gall, and Gregorio Robles. 2018. On the Usage of Pythonic Idioms. In ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Boston, MA, USA) (Onward! 2018). ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3276954.3276960

[2]. MiltiadisAllamanis 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

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

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

[5]. Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, NaoyasuUbayashi, Gabriele Bavota, and Michele Lanza. 2021. PYREF: Refactoring Detection in Python Projects. In 2021 IEEE 21st International Working Conference on Source Code Analysis and Man