Discovering Repetitive Code Changes in ML Systems

Malinda Dilhara

malinda.malwala@colorado.edu
University of Colorado, Boulder, CO, USA

ABSTRACT

Similar to software evolution in other software systems, ML software systems evolve with many repetitive changes. Despite some research and tooling for repetitive code changes that exist in Java and other languages, there is a lack of such tools for Python. Given the significant rise of ML software development, and that many ML developers are not professionally trained developers, the lack of software evolution tools for ML code is even more critical. To bring the ML developers' toolset into the 21st century, we implemented an approach to adapt and reuse the vast ecosystem of Java static analysis tools for Python. Using this approach, we adapted two software evolution tools, *RefactoringMiner* and *CPATMiner*, to Python. With the tools, we conducted the first and most fine-grained study on code change patterns in 59 ML systems and surveyed 253 developers. We recommend empirically-justified, actionable opportunities for tool builders and release the tools for researchers.

CCS CONCEPTS

• Software and its engineering → Software maintenance tools.

KEYWORDS

Machine learning, Empirical analysis, Code change patterns.

ACM Reference Format:

Malinda Dilhara. 2021. Discovering Repetitive Code Changes in ML Systems. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*, August 23–28, 2021, Athens, Greece. ACM, New York, NY, USA, 3 pages. <https://doi.org/10.1145/3468264.3473493>

1 INTRODUCTION

Machine Learning (ML) has completely reinvented domains such as self-driving cars [20], malware detection [2], and even software engineering tools [5, 9, 10]. ML software systems, like other classical systems, are repetitive by nature [6, 12, 13]: developers perform repetitive code changes. While previous researchers have studied repetitive code changes in classical systems [12, 15], there is a gap of knowledge in repetitive changes in ML systems.

Tooling for Python and ML code development is significantly behind the level of support provided by IDEs in Java. Moreover, many ML developers are not professionally trained software engineers, thus they desperately need tool support. Furthermore, compared to classical systems, evolving ML software systems is further complicated by two extra dimensions: data and model [3, 17] evolution.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8562-6/21/08.

<https://doi.org/10.1145/3468264.3473493>

Thus, understanding and automating the code changes that the ML developers unknowingly repeat is critical.

What are the most frequent code change patterns in ML systems? What are the reasons for applying repetitive change patterns? To answer these research questions, we use complementary established research methodologies: static analysis and developer surveys. First, we present a novel technique for adapting Java-based static tools for Python. We used this technique to extend the state of the art refactoring detection tool- *RefactoringMiner* [21] and change pattern detection tool *CPATMiner*- [14] to Python. Using the two tools, we introduce a novel tool **R-CPATMiner** that performs *refactoring aware change pattern mining* in Python code. Second, we used R-CPATMiner to extract change patterns in 59 ML systems and surveyed the 253 developers to seek their opinion on automating the changes. To foster a community infrastructure, we released the extracted patterns for tools builders and other researchers [11].

2 RESEARCH METHODOLOGY

2.1 Subject Systems

Our corpus consists of 91,660 commits from 59 large, mature and diverse ML systems, used by other researchers [3, 7] to understand the challenges of evolving ML systems. This corpus is shown to be very diverse, from the perspective of Python files, LOC, contributors, and commits. This ensures our study is representative.

2.2 Static Analysis of Source Code History

2.2.1 Challenges. Python has become the *lingua franca* for ML [1]. Hence, developing tools that analyse Python source code is essential to study ML systems. Most of the static analysis tools for studying code evolution [14, 18, 21] have been designed for Java. Implementing these tools from scratch for Python requires a significant amount of effort. This is a serious limitation for researchers that venture into studying ML systems. Moreover, the availability of type information in Java is crucial for getting high accuracy for static analysis tools. Whereas with Python being a dynamically typed language, type information is available only at run time.

2.2.2 Novel Approach to Adapt Java Static Analysis Tools to Python.

Key idea - Java and Python have significant similarities at AST level. We observed that 74% of the Python AST node kinds overlap identically with those in Java (e.g., *WhileStatement*, *LambdaExpression*, etc.). Moreover, another 16% 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 static analysis tools to analyse 74% of the Python AST nodes and for the remaining 26% of AST nodes we either modify or add brand new capabilities.

Our Pipeline Implementation - As shown in Figure 1, we generate AST of the Python code using *Jython* [8] which is a widely-used tool to execute Python code in Java. Then, we augment the

Figure 1: Approach of adapting Java static analysis tools to Python

AST with type information using a widely-used Python type inference tool *PyType* [4] developed by Google. Next, we pretty-print the migrated AST and parse it using our *customized eclipse JDT parser* for which we added a special ability to parse Python-specific language constructs. This step is necessary to generate the indexes of AST nodes such as line numbers, length, and start character index. To ensure the robustness of our whole pipeline, we tested it on 12M Python AST nodes from 14 popular Python projects such as *TensorFlow*, *PyTorch*, *Keras*, *Matplotlib*, and *NumPy*.

2.3 Refactoring Aware Change Pattern Mining

CPatMiner [14] is the state of the art tool that uses an efficient graph-based representation to mine previously unknown fine-grained changes from git commit history. It builds program-dependence graphs for each AST node kind independently and then merges the graphs to create a big change graph that represents the before and after a source code change. Since 74% of the Python AST node kinds overlap identically with those in Java, we were able to reuse most of the capabilities for building the graphs and extend *CPATMiner* to Python only with 2% extra code using our technique (Section 2.2.2).

Accurately detecting a source code change pattern is not straightforward, as these changes can get obfuscated by refactorings in a commit, where methods and classes containing the code change get moved, renamed, or removed. The *CPATMiner* proposed by Nguyen et al. [14] is not aware of refactorings, thus it can miss several true instances of change patterns that are obfuscated by refactorings. To overcome this, using our approach (Section 2.2.2), we adapt the Java refactoring inference tool *RefactoringMiner* [21] to Python. Moreover, we manually validated 357 refactorings from 18 refactoring kinds. The Python-adapted *RefactoringMiner* achieves an average precision of 97.34% which is close to the precision of the original *RefactoringMiner*. Because of space constraints, we present the detailed validation results on our companion website [11].

As a novel contribution, we introduce *Refactoring-aware change pattern mining* by merging our Python-adapted *RefactoringMiner* and *CPATMiner*. We developed a novel tool **R-CPATMINER**, which efficiently and accurately mines source code change patterns in the version histories of Python software systems. To compare the improvement of R-CPATMINER over the baseline *CPATMiner*, we mined change patterns using both the Python-adapted *CPATMiner* and R-CPATMINER. The R-CPATMINER extracts 12% more change graphs compared to *CPATMiner* which showcases the importance of refining source code changes using refactorings.

2.4 RESULTS

We ran the **R-CPATMiner** on 59 projects. The tool extracted 3,513 unique code change patterns, where 63% of them have code change instances from at least two projects. Since the mined patterns are numerous, we followed the best-practices from Negara et al. [12]: we ordered the mined patterns along three dimensions: by frequency of the pattern (F), by size of the pattern (S), and by $F \times S$. Then, we manually investigated the top 100 patterns for each dimension

and identified meaningful code patterns, i.e., the patterns that can be described as high-level program transformations. Then, we surveyed 253 developers who performed the identified patterns and asked three questions i) What are the reasons for performing that code change, ii) How often does the change appear in ML codes?, and iii) Would you like to have the change automated by a tool? We describe below the top three automation requests from the developers. All other patterns are available on our website [11].

Move to *with* statement: When developers want to manage resources efficiently, they use Python *with* statement.

```
sess = tf.Session()
sess.run()
```

Identified change instances= 81

```
with tf.Session() as sess:
    sess.run()
```

tf.Session typically owns physical resources (e.g., GPUs, network connections) when we use it to build *TensorFlow* computational graphs. Hence, developers move to *with* statement that automatically releases resources when the *with* block terminates.

Refactoring For loops: Refactoring computations that use *for loop* is prevalent in ML software due to various reasons such as speed, code quality, and readability. Developers prefer several alternatives such as Python comprehension, and optimized APIs implemented in ML libraries [16]. A developer said, “I removed *For loop* and use *np.sum()* to increase the speed.”

```
for i in range(0, len(cnt)):
    final += cnt[i]
```

Identified change instances= 41

```
final = np.sum(cnt)
```

Move Tensors to CPU: This is a frequent transformation in which developers move Tensors from GPU to CPU. A developer said, “*NumPy* does not support GPU. Hence, I copied the tensor to CPU before converting it to *NumPy*”

```
delta = forward(x).\
    detach().numpy()
```

Identified change instances= 6

```
delta = forward(x).\
    detach().cpu().numpy()
```

3 RELATED WORK

The closest related work is by Tang et al. [19]. 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. [12] and Nguyen et al. [15] study code change patterns in classical software systems whereas we study ML systems and provide suggestions for automation.

4 CONCLUSION AND FUTURE WORKS

We introduce a novel tool, **R-CPATMiner** that performs refactoring-aware change pattern mining in Python. Using the tool, we extracted change patterns in ML systems and surveyed 253 developers to seek their opinion on automation. As future work, we will scale the analysis to more projects and provide automation for some of the most frequent changes.

REFERENCES

- [1] Housseem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The Open-Closed Principle of Modern Machine Learning Frameworks. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) (MSR '18). Association for Computing Machinery, New York, NY, USA, 353–363. <https://doi.org/10.1145/3196398.3196445>
- [2] Haipeng Cai. 2020. Assessing and Improving Malware Detection Sustainability through App Evolution Studies. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 8 (March 2020), 28 pages. <https://doi.org/10.1145/3371924>
- [3] 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.* (2021).
- [4] Google. 2021. *PyType*. <https://github.com/google/pytype> Accessed: 2021-03-31.
- [5] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 152–162. <https://doi.org/10.1145/3236024.3236051>
- [6] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) (ICSE '12). IEEE Press, 837–847.
- [7] Hadhemi Jebnoun, Housseem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. 2020. The Scent of Deep Learning Code: An Empirical Study. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) (MSR '20). Association for Computing Machinery, New York, NY, USA, 420–430. <https://doi.org/10.1145/3379597.3387479>
- [8] Jython. 2021. *Jython*. <https://www.jython.org> Accessed: 2021-03-31.
- [9] Ritu Kapur and Balwinder Sodhi. 2020. A Defect Estimator for Source Code: Linking Defect Reports with Programming Constructs Usage Metrics. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 12 (April 2020), 35 pages. <https://doi.org/10.1145/3384517>
- [10] Yangguang Li, Zhen Ming (Jack) Jiang, Heng Li, Ahmed E. Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. Predicting Node Failures in an Ultra-Large-Scale Cloud Computing Platform: An AIOps Solution. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 13 (April 2020), 24 pages. <https://doi.org/10.1145/3385187>
- [11] Malinda. 2021. *An Empirical Study of Frequent Code Change Patterns In Machine Learning Systems*. <https://mledits.github.io> Accessed: 2021-05-05.
- [12] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining Fine-Grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) (ICSE 2014). Association for Computing Machinery, New York, NY, USA, 803–813. <https://doi.org/10.1145/2568225.2568317>
- [13] 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. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) (FSE 2016). Association for Computing Machinery, New York, NY, USA, 511–522. <https://doi.org/10.1145/2950290.2950333>
- [14] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 819–830. <https://doi.org/10.1109/ICSE.2019.00089>
- [15] 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 *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) (ASE'13). IEEE Press, 180–190. <https://doi.org/10.1109/ASE.2013.6693078>
- [16] numpy. 2021. *NumPy*. Accessed: 2021-05-05.
- [17] 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. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) (NIPS'15). MIT Press, 2503–2511.
- [18] D. Silva, J. Silva, G. De Souza Santos, R. Terra, and M. O. Valente. 5555. RefDiff 2.0: A Multi-language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 01 (jan 5555), 1–1. <https://doi.org/10.1109/TSE.2020.2968072>
- [19] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *International Conference on Software Engineering* (Madrid, United States) (ICSE '21). ACM/IEEE. <https://doi.org/10.1109/ICSE43902.2021.00033>
- [20] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) (ICSE '18). Association for Computing Machinery, New York, NY, USA, 303–314. <https://doi.org/10.1145/3180155.3180220>
- [21] N. Tsantalis, A. Ketkar, and D. Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 1–1. <https://doi.org/10.1109/TSE.2020.3007722>