AI for Automated Code Updates

Salwa Alamir
J.P. Morgan AI Research
London, UK
salwa.alamir@jpmorgan.com

Petr Babkin
J.P. Morgan AI Research
New York, USA
petr.babkin@jpmorgan.com

Nacho Navarro
J.P. Morgan AI Research
Madrid, Spain
nacho.navarro@jpmorgan.com

Sameena Shah
J.P. Morgan AI Research
New York, USA
sameena.shah@jpmorgan.com

ABSTRACT

Most modern code bases extensively rely on external libraries to provide robust functionality out of the box. When these libraries are updated they can sometimes introduce breaking changes in the process, which require extensive developer maintenance. To mitigate this we propose to use artificial intelligence to parse the text of release notes to capture code deprecations in structured form. This, in turn, enables us to develop an IDE plugin that can automatically detect deprecated library usages in live code bases and even suggest recommended fixes. We evaluated our system on over 30 internal projects within J.P. Morgan.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; • **Computing methodologies → Artificial intelligence**.

KEYWORDS

artificial intelligence, software engineering, semantic parsing

ACM Reference Format:
Salwa Alamir, Petr Babkin, Nacho Navarro, and Sameena Shah. 2022. AI for Automated Code Updates. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3510457.3513073

1 INTRODUCTION

The use of dependencies in the form of libraries is extremely common, if not essential, in most software projects. Including these libraries can come at a cost, however, as code evolves over time for new features and changes, which require extensive developer maintenance and can therefore present a serious time constraint for developers. An example of a large codebase that inspired the research for this paper is a J.P. Morgan platform for pricing, trading, risk management, and analytics that has over 2,500 developers contributing code regularly to a shared code base comprised of over 10 million lines of code. The aim of this paper is to assess how an automated AI system can understand code dependencies and update code accordingly, thus freeing developers' time to spend on higher-value tasks. Our contributions include the following: (1) A fully-automated method of sourcing API deprecations for Python libraries by crawling release notes and using a novel transition-based parser. (2) An IDE plugin that can automatically detect and update deprecated references in a code base. (3) An evaluation of the tool on over 30 projects within J.P. Morgan Chase.

2 APPROACH AND METHODOLOGY

In order to source deprecations, we utilized Sphinx[2] to aid in crawling the release notes. Many Python libraries use Sphinx to automatically generate standardized API web documentation. We obtained the versions of 410 supported libraries by querying PyPI and managed to get release notes for 154 libraries using the Google Search API. We then scraped the deprecation section from each page, producing a collection of individual deprecation descriptions. Here is an example deprecation from our dataset, where highlighting of code entities is given by Sphinx' standardized HTML markup:

> *Deprecated parameters* `levels` *and* `codes` *in* `MultiIndex.copy()`. *Use the* `set_levels()` *and* `set_codes()` *methods instead.*

For a person reading this text, this is equivalent to the following replacements in the code:

```
MultiIndex.copy(levels) → MultiIndex.set_levels(levels)
MultiIndex.copy(codes) → MultiIndex.set_codes(codes)
```

The above interpretation achieves two objectives: 1) it pairs deprecated code references on the left hand side with their corresponding replacements on the right hand side; 2) it makes implied compositionality relationships among code references (such as classes, parameters and methods) explicit. We chose to model the above structure as a tree and rendered the problem of building such tree as a generalized form of transition-based parsing [1].

Due to space limitations, we defer the in-depth description of the inner workings of the parser to a forthcoming full paper. At a high level, this work mainly involved a) extending the standard transition system to handle issues such as constituency, non-projectivity and re-entrancy, building on prior work [2, 3], and b) designing informative features to capture the parser configuration as well as the linguistic structure of the sentence as to enable effective selection

[1]https://github.com/nirmalyaghosh/kaggle
[2]https://www.sphinx-doc.org/en/master/examples.html

ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513073

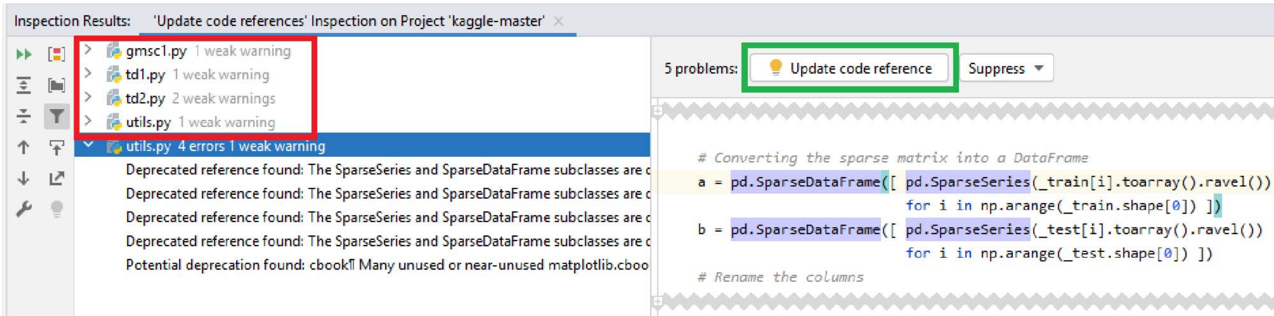

Figure 1: Example run of the plugin for a Kaggle project[1]. Boxed in red one can see the plugin detecting weak warnings (partial matches) or errors (full matches). In the green box the plugin offers a quick-fix button to automatically update the reference.

of parsing actions at inference time, using a standard supervised classifier with simple beam search.

The resulting tree is then passed as JSON to an IntelliJ plugin we built, in which a project's source code is inspected statically. The inspection algorithm goes through each reference in the code and attempts to match it to the specification provided by the deprecation tree. If the reference matches the specification exactly, for example, the argument `levels` is passed to the method `copy` in the namespace `MultiIndex`, then it is highlighted by the IDE as an "error", allowing for a fully automatic fix, if one is available. Otherwise, if the match is partial e.g., only the method name matched but the invoking namespace could not be resolved (due to Python's dynamic type system), the reference is highlighted as a "warning" — prompting a developer to take a closer look.

3 SYSTEM EVALUATION

We annotated gold trees for 426 deprecations from the release notes of popular data science libraries, such as pandas. As a baseline for evaluating the parser, we used a heuristic that splits all encountered code references into left and right hand side sets, relative of the word "deprecated". Since this approach is not compositional, we measured simple intersection-over-union of produced code tokens against the gold ones in each set. We ran the parser in cross-validation mode, measuring average weighted subtree overlap with gold deprecation trees. The parser outperformed the baseline by nearly two-fold in terms of the overall score (31.3 vs 16.9), on method deprecations (45.9 vs 21.8), and, most notably, on compositionally difficult parameter deprecations (15.3 vs 1.0), across all libraries.

To assess the plugin's usability in a realistic setting, we ran it on 33 internal repositories, using golden deprecation trees as input. In total, the plugin highlighted 342 potential deprecations from the following libraries: pandas, numpy, matplotlib and networkx. During qualitative analysis, we found many promising detections, especially for the exact matches. For instance, the reference `json_normalize()` moved from the module `pandas.io.json` to the top-level `pandas` in version 1.0, and the plugin was able not only to highlight this, but to automatically update the statements of the form `from pandas.io.json import json_normalize` to `from pandas import json_normalize`. On the other hand, partial matches proved much more ambiguous due to the lack of type information, ultimately resulting in a true positive rate of only about 20%. This validates our design choice of implementing partial matches as a developer prompt rather than an automatic fix. In terms of the overall distribution, partially matched deprecations accounted for the majority of all detections. Yet, we believe both types of detections provide valuable reduction in the number of code occurrences the developer has to examine.

4 CONCLUSION

We have presented a system to automate the challenging task of updating deprecated library usages in a Python project. The results of evaluating the underlying parsing technology on a labeled dataset, as well as the system as a whole on real J.P. Morgan projects are encouraging and open up avenues for future work. Having a system that can automatically build a knowledge base of code deprecations found in libraries and offer fixes is a valuable tool that can alleviate a project's maintenance in the long run and improve its quality.

Disclaimer. This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Coȧnd its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

REFERENCES

[1] Joakim Nivre. 2008. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics* 34, 4 (2008), 513–553. https://doi.org/10.1162/coli.07-056-R1-07-027

[2] David Vilares and Carlos Gómez-Rodríguez. 2018. A Transition-Based Algorithm for Unrestricted AMR Parsing. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 142–149. https://doi.org/10.18653/v1/N18-2023

[3] Xun Zhang, Yantao Du, Weiwei Sun, and Xiaojun Wan. 2016. Transition-Based Parsing for Deep Dependency Structures. *Computational Linguistics* 42, 3 (09 2016), 353–389. https://doi.org/10.1162/COLI_a_00252 arXiv:https://direct.mit.edu/coli/article-pdf/42/3/353/1806910/coli_a_00252.pdf