MLCatchUp: Automated Update of Deprecated Machine-Learning APIs in Python

Stefanus A. Haryono*, Ferdian Thung*, David Lo*, Julia Lawall†, and Lingxiao Jiang*
*School of Information Systems, Singapore Management University, Singapore
†Inria, France
{stefanusah,ferdianthung,davidlo,lxjiang}@smu.edu.sg
Julia.Lawall@inria.fr

***Abstract*—Machine learning (ML) libraries are gaining vast popularity, especially in the Python programming language. Using the latest version of such libraries is recommended to ensure the best performance and security. When migrating to the latest version of a machine learning library, usages of deprecated APIs need to be updated, which is a time-consuming process. In this paper, we propose MLCatchUp, an automated API usage update tool for deprecated APIs of popular ML libraries written in Python. MLCatchUp automatically infers the required transformation to migrate usages of deprecated API through the differences between the deprecated and updated API signatures. MLCatchUp offers a readable transformation rule in the form of a domain specific language (DSL). We evaluate MLCatchUp using a dataset of 267 real-world Python code containing 551 usages of 68 distinct deprecated APIs, where MLCatchUp achieves 90.7% accuracy. A video demonstration of MLCatchUp is available at https://youtu.be/5NjOPNt5iaA.**

***Index Terms*—Python, Program Transformation, Automatic Update, Deprecated API**

I. Introduction

One factor behind the recent surge in popularity of machine learning (ML) is the availability of easy to use ML libraries, typically implemented in Python. However, similar to other libraries, Python ML libraries are also affected by API deprecations. The problem may even be more significant for Python ML libraries, as more popular libraries tend to see more evolution and thus have many deprecated APIs [1], [2]. Deprecation of APIs may occur with each update of the library. To use a new release of a library, updating usages of deprecated APIs is often required. However, the update can be cumbersome and time-consuming, and thus users may prefer to continue using the older version [3]–[7]. The use of the older version, however, may expose the program to lower performance and to security issues [3], [4]. Thus, a tool that supports the update of deprecated API usages is beneficial.

Several studies have attempted to automate updates process of deprecated API usages [8]–[11]. However, prior works mainly target Java programs, and no study has attempted to automate the update for Python deprecated API usages. Python has several unique features compared to Java: (1) Python is dynamically-typed, compared to Java which is statically-typed. In Python, variable types may change during code execution, and information regarding variable and function return types are typically unavailable. (2) Python has two types of parameters, i.e., positional and keyword parameters, while Java only supports positional parameter. Python also supports parameter default values.

In this paper, we present MLCatchUp, an automated update tool for usages of deprecated APIs of popular Python ML libraries. MLCatchUp performs type inference by leveraging static analysis and deep learning tools [12], [13]. MLCatchUp also supports both positional and keyword parameters. Moreover, MLCatchUp does not require any update example for its update, only requiring the deprecated and updated API signatures to infer the required migration. This is a demonstration paper accompanying our full research paper [14].

MLCatchUp takes as input a pair of deprecated and updated API signatures, and a file containing deprecated API usages to be updated. The deprecated and updated API signatures are used to infer the transformation operations needed to migrate the usage of the deprecated API. The transformation operations obtained from this inference process are expressed in a DSL to facilitate understanding the transformation performed. MLCatchUp automatically detects the specified deprecated API usages and applies the inferred transformation operations, updating the deprecated API usages within the code. MLCatchUp also leverages type inference, allowing it to determine variable types and function return types. To the best of our knowledge, we are the first to create an automated API usage update tool without update examples for Python.

We evaluated MLCatchUp using a test dataset of 267 real-world Python code files collected from Github public repositories. Each file contains one to five deprecated API usages, amounting to a total of 551 deprecated API usages. In this evaluation, MLCatchUp can detect 93.7% of the deprecated API usages without type inference. The addition of type inference improves MLCatchUp's deprecated API usage detection rate to 100%. MLCatchUp achieves an accuracy of 80.6% (i.e., it correctly updates 80.6% of the deprecated API usages). We conducted a qualitative analysis on the update result and found that there are context-dependent API migrations whose updates depend on the value of one or more arguments in the API. Thus, we enhance the capability of MLCatchUp to accept some constraints expressed using the DSL as additional user input. Using this additional input, MLCatchUp achieves an accuracy of 90.7%.

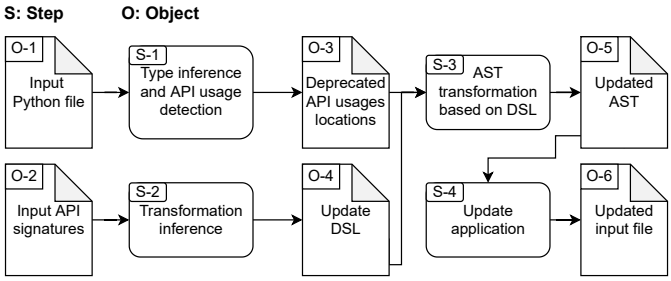

Fig. 1. MLCatchUp Architecture

II. Design and Implementation

A. MLCatchUp Architecture

The architecture of MLCatchUp is shown in Fig. 1. MLCatchUp has two main inputs: the input file to be updated (O-1) and the input API signatures for both a deprecated API and its replacement (O-2). The input file is parsed into an Abstract Syntax Tree (AST) using the built-in Python module.[1] Type inference is applied to the AST to determine the types of variables and method call return values. Then, MLCatchUp locates the deprecated API usages by comparing the fully qualified name of the method calls within the code to the deprecated API signature (S-1), resulting in the deprecated API usage locations (O-3).

MLCatchUp uses the input API signatures in the transformation inference process (S-2). The inferred transformations are in the form of DSL commands (O-4) which contain a series of operations that will be performed sequentially for the update (e.g., rename a method, rename a parameter, etc.). The DSL is then parsed into a list of operations to be executed by MLCatchUp. These operations are applied to the deprecated API usage locations (S-3), producing the updated AST (O-5).

The updated AST is then compared with the input file AST to check for code differences between the two. Any found code differences are then applied to the input Python file (S-4) by making only the necessary changes to the API usages without any modification to the original code comments and spacing. This results in the updated input file (O-6). We next further describe the main components of MLCatchUp.

B. MLCatchUp Transformation Operations

The transformation operations provided by MLCatchUp are based on an empirical study of 112 deprecated APIs and their replacements from the last two years of releases of `Scikit-learn`, `PyTorch`, and `Tensorflow` [14]. The transformation operations that are supported by MLCatchUp are: (1) rename a method; (2) rename a parameter; (3) remove a parameter; (4) convert a positional to a keyword parameter; (5) add a parameter; (6) change a parameter type; (7) add a constraint to a parameter value; and (8) remove API. Further details about the transformation operations are found in our full research paper [14].

C. MLCatchUp Transformation Inference

MLCatchUp automatically infers the required transformations for API usage update based on the difference between the deprecated and updated API signatures. We define an API signature as the fully qualified name of the API (i.e. the API's module and function name) and the list of parameters available for the API, where each parameter consists of the parameter name and an optional default value. Given a pair of deprecated and updated API signatures, MLCatchUp performs a sequence of comparisons, including a comparison of the API names, and comparisons of their parameter names, positions, types, and default values. These comparisons will produce the required transformation operations for the API update.

D. Additional Input Constraint

While MLCatchUp can automatically infer the required transformation from the deprecated and updated API signatures, it cannot infer the transformation constraints that are needed in context-dependent deprecated API updates (i.e., the update may differ according to the values of the deprecated API arguments), as such constraints are not provided in the API signature. To mitigate this problem, MLCatchUp accepts an additional input in the form of transformation constraints expressed in the DSL. This can be done by manually editing the inferred update DSL script (O-4) created by the transformation inference process, and adding the required constraints (See Section IV-B for a usage example).

E. Type Inference

MLCatchUp uses type inference to determine variable types and the return types of method calls. This type information is used to replace the variable or the method name used in an API invocation to determine the API's fully qualified name. The type inference is implemented using PEP (Python Enhancement Proposal) 484,[2] that provides type hints and annotations for Python code. For this purpose, MLCatchUp use Mypy [13], a static type checker for Python that uses the PEP 484 type hints. Mypy's APIs are used to analyze MLCatchUp's input code, collecting variable and method types by searching for type annotations from the imported library. The collected types are used to replace the variables and method calls during the deprecated API usage detection.

To further enhance MLCatchUp's type inference functionality for libraries that do not provide type annotations, we also use deep learning-based type inference [12]. A neural model is trained using hierarchical neural network (HNN) with a dataset of 4,910 Python projects. From the Python source code files, type hints (i.e., information helpful in predicting types) are extracted. These type hints consist of information taken from the function (function name, function arguments) and the code context (return statement, function argument usages). To enable the model to learn from the type hints, the Word2Vec [15] word embedding is used. After the neural model is trained, the Triplet Loss function is used to create

[1] https://docs.python.org/3/library/ast.html

[2] https://www.python.org/dev/peps/pep-0484/

TABLE I
MLCATCHUP DETECTION RATE EVALUATION RESULTS

Evaluation	Detected	Ground Truth	Detection Rate
No Type Inference	516	551	93.7%
Type Inference	551	551	100.0%

TABLE II
MLCATCHUP UPDATE ACCURACY EVALUATION RESULTS

Experiment	Correct	Incorrect	Accuracy
Without added constraints	444	107	80.6%
With added constraints	500	51	90.7%

type clusters, where each cluster represents a similar type. These *type clusters* are used for type prediction. Given a method signature as an input, the model will extract the type hints and create the feature embedding. Then, KNN (k-nearest neighbor) search is used to determine which type is the closest to the given embedding. The predicted type is then used by MLCatchUp in the same way as the types inferred using Mypy.

III. EVALUATION

A. Dataset and Experiment Settings

We evaluated MLCatchUp on a dataset of 267 files containing 551 deprecated ML API usages in Python from GitHub repositories. They are comprised of 68 different deprecated APIs from `Scikit-learn`, `PyTorch`, `Tensorflow`, `Numpy`, and `Spacy`. For each deprecated API, we randomly collected at most five files. We consider two metrics in this evaluation: (1) *detection rate* (i.e., the number of detected deprecated API usages divided by the ground truth number of deprecated API usages) and (2) *update accuracy* (i.e., the number of correct updates produced by MLCatchUp compared to the ground truth number of deprecated API usages). We use automated and manual analysis to determine the correctness of the updated code. In the automated analysis, we check the validity of the updated code syntax by compiling it using the Python built-in py_compile module.[3] In the manual analysis, we employed the help of three Python programmers with four years of experience to label if the updated code is correct. These programmers are not authors of this paper and were given instructions and training prior to the labelling process. If labelling disagreement occurred, a discussion was conducted.

B. Evaluation Result

The *detection rate* experiment results are shown in Table I. Without type inference, MLCatchUp can detect 93.7% of the deprecated API usages. The addition of type inference improves the MLCatchUp detection rate to 100%. Table II shows the *update accuracy* experiment results. Without additional transformation constraints, MLCatchUp is able to update 443 out of 551 deprecated API usages successfully. The addition of transformation constraints enables MLCatchUp to update 481 out of 551 deprecated API usages, achieving 90.7% accuracy. We found that the failed updates occur in

[3]https://docs.python.org/3/library/py_compile.html

```
1  # main.py
2  import tensorflow.compat.v1 as tf
3  def classification_probe(features, labels, n_classes,
4                           logits, xe, labeled=None):
5    if labeled is not None:
6      xe = xe * tf.to_float(labeled)
7    acc = tf.reduce_mean(tf.to_float(tf.equal(
8        tf.argmax(logits, axis=1),labels)))
9    return xe, acc
```

Fig. 2. Deprecated `tensorflow.compat.v1.to_float` API usages example

deprecated API migrations with a 1:N API mapping, where a single deprecated API is replaced with multiple updated APIs. Currently, MLCatchUp only supports 1:1 mappings.

IV. USAGE EXAMPLE

MLCatchUp can be used by both library users and developers via a command line interface (CLI). To update usages of a deprecated API, a user needs to provide the deprecated and updated API signatures, along with the code that needs to be updated. MLCatchUp's update process consists of two steps:

1) **Transformation inference** is used to infer and output the required DSL script from the API signatures. MLCatchUp's CLI takes the command line argument `--infer` to specify the transformation inference mode, and `--output` to indicate the path for the output file to contain the produced DSL script. The format of this command is as follows:

```
python MLCatchUp.py --infer <deprecated_api_signature>
    <updated_api_signature> --output <output_filepath>
```

2) **Transformation application** is used to parse the DSL script and apply the transformation to an input file. MLCatchUp's CLI takes the command line arguments `--transform` to specify the transformation application mode, `--dsl` to indicate the path to the file containing the DSL script, `--input` to indicate the file to be updated, and `--output` to indicate the output file where the updated code is written. The format of this command is:

```
python MLCatchUp.py --transform --dsl
    <dsl_script_filepath> --input
    <deprecated_filepath> --output <output_filepath>
```

A. Scenario I - Basic Transformation

Consider the `tensorflow.compat.v1.to_float` deprecated API usages given in lines 6 and 7 of Fig. 2. The deprecated API needs to be updated into an invocation of `tensorflow.cast` with an addition of a `dtype=tensorflow.float32` parameter. The command used for the transformation inference process is:

```
python MLCatchUp.py --infer tensorflow.compat.v1.to_float()
    tensorflow.cast(dtype=tensorflow.float32)
    --output out.dsl
```

The command instructs MLCatchUp to infer the required transformation for migrating a deprecated `tensorflow.compat.v1.to_float()` API usage. MLCatchUp detects that the two API signatures have different method names and an addition of API parameter in the updated API signature, thus a `rename_method` and an `add_parameter` operations are added. The added parameter value for the

```
1   # main.py
2   + from tensorflow import cast
3     import tensorflow.compat.v1 as tf
4     def classification_probe(features, labels, n_classes,
5                              logits, xe, labeled=None):
6       if labeled is not None:
7   -     xe = xe * tf.to_float(labeled)
8   +     xe = (xe * cast(labeled, dtype=tensorflow.float32))
9   -   acc = tf.reduce_mean(tf.to_float(tf.equal(
10  -       tf.argmax(logits, axis=1),labels)))
11  +   acc = tf.reduce_mean(cast(tf.equal(
12  +       tf.argmax(logits, axis=1),labels),
13  +       dtype=tensorflow.float32))
14      return xe, acc
```

Fig. 3. Update result for deprecated `tensorflow.compat.v1.to_float` API usages migration

`add_parameter` operation is obtained from the parameter of the updated API signature. The DSL script is as follows:

```
add_parameter dtype with_value tensorflow.float32 for
    tensorflow.compat.v1.to_float
rename_method tensorflow.compat.v1.to_float to
    tensorflow.cast
```

We use the DSL file to instruct MLCatchUp to transform the code shown in Fig. 2; we use the following command:

```
python MLCatchUp.py --transform --dsl out.dsl
    --input main.py --output updated.py
```

This command instructs MLCatchUp to apply the transformation specified in the DSL. The DSL script is parsed into its corresponding operations, comprising a `rename_method` operation and an `add_parameter` operation. Both operations are applied to each instance of a `tensorflow.compat.v1.to_float` API invocation, migrating them to use the `tensorflow.cast` API with the addition of a `dtype=tensorflow.float32` parameter. Fig. 3 highlights the code differences between the input and updated code. In line 2, an import statement is added for the `tensorflow.cast` updated API. In lines 9-10 and 11-15, the deprecated API usages are updated to use the newly imported `tensorflow.cast` API invocation with the addition of the `dtype=tensorflow.float32` parameter.

B. Scenario II - Context-dependent API Migration

As MLCatchUp is unable to automatically infer the required transformation constraints for context-dependent deprecated API migration, the user needs to manually add this information into the inferred DSL script. Consider the migration of usages of the deprecated API `sklearn.model_selection.KFold` that needs to be updated with an `add_parameter` operation only if the value of `random_state` argument is not `None`. The input for the transformation inference process is as follows:

```
python MLCatchUp.py --infer sklearn.model_selection.KFold()
    sklearn.model_selection.KFold(shuffle=True)
    --output out.dsl
```

This transformation inference results in this DSL script:

```
add_parameter shuffle with_value True for
    sklearn.model_selection.KFold
```

While the above script provides the desired transformation operation, the required transformation constraint is not present. We can fix it by adding a constraint to the inferred DSL script:

```
add_parameter shuffle with_value True for sklearn.
    model_selection.KFold if random_state has_value != None
```

V. Related Work

Several works have attempted to automate the update of deprecated API usages. Diff-CatchUp [16] is a classic work that automatically detects API changes in the library, infers possible replacements, and identifies possible examples on how to migrate the changed APIs from the library's code base. Diff-CatchUp however does not produce a transformation script or come with a transformation engine that can automatically update occurrences of broken API usages in a client code base. Rather, developers need to manually perform the transformations based on the examples and information produced by Diff-CatchUp. A4 [8] automatically assists Android API migrations by learning API migration patterns from code examples. AppEvolve [9] automates API-usage update for Android, creating patches from before- and after- update code examples. CocciEvolve [10] infers a readable transformation in the form of the semantic patch language (SmPL) [17] using only a single after-update code example. NEAT [11] generates transformation rules for deprecated Android API usage migrations without any update example. REFAZER [18] synthesizes program transformations for Python code by inductive programming using input-output examples. T2R [19] performs automatic type migration for Java code to help fix API-breaking changes in evolving codebases. No work has been done on the automated migration of Python deprecated APIs. MLCatchUp offers automated migration of Python deprecated API usages without any update example. MLCatchUp does not require the source code of the APIs' library, as it only needs the deprecated and updated API signatures to infer the required update. MLCatchUp also addresses Python's unique features such as dynamic-typing through type-inference using a combination of static analysis and deep learning.

VI. Conclusion and Future Work

In this paper, we present MLCatchUp, a tool that automates the update of usages of deprecated APIs of popular Python ML libraries. MLCatchUp automatically infers the required transformation from the differences between the deprecated and updated API signatures, and presents the operations in the form of a DSL. MLCatchUp is built with type inference, allowing it to determine variables and function return types. We have evaluated the performance of MLCatchUp and demonstrated that it achieves 100% deprecated API usages *detection rate* with type inference. We have also evaluated the *update accuracy* of MLCatchUp, where it achieves an accuracy of 80.6% without user-provided transformation constraints, and an accuracy of 90.7% with the addition of such constraints. We promote MLCatchUp as a tool that can be used by both library users and developers to automate the process of updating deprecated API usages. The implementation of MLCatchUp is available at https://github.com/soarsmu/MLCatchUp.

Acknowledgement. This research is supported by the Singapore National Research Foundation (award number: NRF2016-NRF-ANR003) and the ANR ITrans project.

REFERENCES

[1] L. Xavier, A. Brito, A. Hora, and M. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," 02 2017, pp. 138–147.

[2] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated Android APIs," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*. ACM, 2018, pp. 254–264.

[3] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated Python library APIs are (not) handled," in *FSE*, 2020.

[4] R. Kula, D. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, p. 384–417, Feb. 2018.

[5] J. Han, S. Deng, D. Lo, C. Zhi, J. Yin, and X. Xia, "An empirical study of the dependency networks of deep learning libraries," in *ICSME*, 2020.

[6] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. Bacchelli, "Why are features deprecated? an investigation into the motivation behind deprecation," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 13–24.

[7] A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs," 10 2016, pp. 400–410.

[8] M. Lamothe, W. Shang, and T.-H. Chen, "A4: Automatically assisting Android API migrations using code examples," *ArXiv*, vol. abs/1812.04894, 2018.

[9] M. Fazzini, Q. Xin, and A. Orso, "Automated API-usage update for Android apps," in *ISSTA*. ACM, 2019, pp. 204–215.

[10] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automatic Android deprecated-API usage update by learning from single updated example," in *IEEE International Conference on Program Comprehension*, 2020.

[11] F. Thung, H. Kang, L. Jiang, and D. Lo, "Towards generating transformation rules without examples for Android API replacement," 09 2019, pp. 213–217.

[12] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, "Type4py: Deep similarity learning-based type inference for python," *arXiv preprint arXiv:2101.04470*, 2021.

[13] J. Lehtosalo, G. v. Rossum, I. Levkivskyi, M. J. Sullivan, D. Fisher, G. Price, M. Lee, N. Seyfer, R. Barton, S. Ilinskiy, E. Smith, and J. Zijlstra, "Mypy - optional static typing for python," https://github.com/charlespwd/project-title, 2017.

[14] S. A. Haryono, F. Thung, D. Lo, J. Lawall, and L. Jiang, "Characterization and automatic update of deprecated machine-learning API usages," 2020.

[15] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, 10 2013.

[16] Z. Xing and E. Stroulia, "API-evolution support with Diff-CatchUp," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[17] Y. Padioleau, J. Lawall, and G. Muller, "SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 47–62, 01 2007.

[18] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *ICSE*. IEEE Press, 2017, pp. 404–415.

[19] A. Ketkar, A. Mesbah, D. Mazinanian, D. Dig, and E. Aftandilian, "Type migration in ultra-large-scale codebases," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1142–1153.