PathMiner: A Library for Mining of Path-Based Representations of Code

Vladimir Kovalenko
Delft University of Technology
Delft, The Netherlands
v.v.kovalenko@tudelft.nl

Egor Bogomolov
JetBrains Research
Higher School of Economics
Saint Petersburg, Russia
eobogomolov@edu.hse.ru

Timofey Bryksin
JetBrains Research
Saint Petersburg State University
Saint Petersburg, Russia
t.bryksin@spbu.ru

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

***Abstract*—One recent, significant advance in modeling source code for machine learning algorithms has been the introduction of path-based representation – an approach consisting in representing a snippet of code as a collection of paths from its syntax tree. Such representation efficiently captures the structure of code, which, in turn, carries its semantics and other information. Building the path-based representation involves parsing the code and extracting the paths from its syntax tree; these steps build up to a substantial technical job. With no common reusable toolkit existing for this task, the burden of mining diverts the focus of researchers from the essential work and hinders newcomers in the field of machine learning on code.**

In this paper, we present PathMiner – an open-source library for mining path-based representations of code. PathMiner is fast, flexible, well-tested, and easily extensible to support input code in any common programming language. Preprint [https://doi.org/10.5281/zenodo.2595271]; released tool [https://doi.org/10.5281/zenodo.2595257].

I. Introduction

Recent achievements in methods of statistical learning have been lately making their way to the field of software engineering. Approaches based on machine learning have improved the state of the art in various software engineering contexts, such as program synthesis [1], [2], language modeling [3], [4], code summarization [5], [6], optimization [7], [8], code completion [9], and automatic code review [10].

The first step in applying machine learning algorithms to source code is to represent the subject code in a way that both captures aspects that are relevant to the task at hand and is digestible by the algorithms. Examples of such representations include a vector of tokens [3], a set of explicitly defined AST metrics [11], and a traversal sequence of the syntax tree [12].

The recently introduced *path-based representation* [13] is a particularly interesting modeling approach, because—even though it was initially introduced for prediction of program properties—it had since proven applicable to several tasks, such as learning embeddings of snippets of code [14] and mining of error-handling specifications [15].

The extraction of code representations requires substantial technical work, but, to date, we have no well-known and reusable tooling for this task. Conducting this technical work has several implications: (1) It diverts researchers' effort and focus away from the essential part of the work – designing the models and evaluating their applicability in practice, (2) it implies effort duplication within the research community, and (3) it poses a barrier for newcomers in the research area of machine learning on code. As a result, both the development and the adoption of machine learning on code are impaired.

In this paper, we present PathMiner – a library for mining of path-based representations of code. By providing a convenient API for retrieval of the path-based representations and efficient storage of the extracted data, PathMiner strives to provide value for both SE and ML communities by allowing the researchers to skip the time-consuming step of writing custom mining pipelines for the models that utilize the path-based representations of code.

PathMiner supports mining of code in Java as well as Python and is designed to be easily extensible to support other programming languages. We achieve this extensibility by providing a convenient extension point for parsers generated by ANTLR [16]. In addition, in the distribution of PathMiner we provide a Python library to read and process the output of PathMiner, and an example of usage of its output as a dataset for a machine learning task.

II. Path-based Representations

The idea of the path-based representation is to model a snippet of code as a collection of paths between the nodes in its syntax tree. In this section, we describe the related concepts.

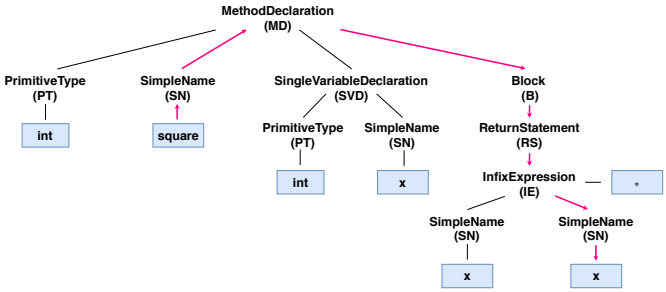
A. Abstract Syntax Tree

An abstract syntax tree (AST) is a tree-based structure that represents the syntactic structure of a program. ASTs do not represent the complete contents of a program's source code – some information (such as formatting of the code, parentheses, and the exact form of syntactic constructs) is omitted. Each node in the AST represents a syntactic unit of the program – such as a variable, an operation, or a logical operator. The children of the node represent the lower-level units associated with the current one.

Figure 1 features a simple code snippet and its AST. While omitting some information, an AST represents the essential structural information about the code, which, along with its own strict tree-based structure, makes it a good intermediate form of code representation in a wide variety of tasks, such as

```
int square(int x) {
  return x * x;
}
```

(a) An example code snippet



(b) The snippet's syntax tree. An example of a path is highlighted in pink.

Fig. 1. An example code snippet and its syntax tree

prediction of variable names [13], code summarization [17], and authorship attribution [11].

B. Abstract Syntax Tree Paths

An *AST path* is a sequence of connected vertices in the AST, logically representing a path from one vertex to another. While, in theory, a path can connect arbitrary nodes of the AST, existing approaches operate with paths between two leaves, which can be linked to concrete tokens in the code.

In practical contexts, a path is denoted by a sequence of types of its nodes and marks of traversal direction. Moreover, the concrete tokens denoting the start and the end vertex of the path are added to the representation, forming a *path-context*.

In Figure 1, we highlight an example of an AST path in pink. This path can be denoted as follows (using the abbreviated labels for node types and arrows for direction):

$$SN \uparrow MD \downarrow B \downarrow RS \downarrow IE \downarrow SN$$

Along with the tokens of the end nodes, it forms the following path-context:

$$(square, SN \uparrow MD \downarrow B \downarrow RS \downarrow IE \downarrow SN, x)$$

Formal definitions of path and path-context are available in existing literature [13], [14].

Semantically, a single AST path denotes a logical connection between two concrete elements of code. Representing the whole tree by a set of the contained paths allows to efficiently capture the semantics of code. This is demonstrated by high performance of path-based representations in tasks such as prediction of names for variables as well as methods, and prediction of variable types [13].

Mining of path-based representations involves parsing the code and extracting paths from its syntax tree. With the variety of programming languages, each requiring a specific toolkit for analysis, mining pipelines are generally not reusable. With PathMiner, by providing a reusable and extensible mining library, we strive to help the researchers to dedicate less effort to mining, thus allowing our community to focus on the essence of research problems.

III. PathMiner: An Overview

PathMiner is designed to do one thing: extracting path-based representations from code. The core of PathMiner is the `PathMiner` class, whose only method extracts all the paths meeting the height and width limitations from a provided AST.

Practically, it often makes sense to require the extracted paths not to exceed certain *width* (*i.e.,* the distance between the end nodes in the ordered list of AST leaves) and *height* (*i.e.,* the distance between the topmost node in the path and the lower of its end nodes). In fact, this step helps to reduce the amount of generated data and to only capture connections between the nearby code elements [14]. For this reason, the `PathMiner` class is instantiated with a `PathRetrievalSettings` object denoting the limits for path extraction.

ASTs for code in various languages are generated by various `Parser` implementations. The path extraction results are handled by `PathStorage` classes, which also produce the output. Listing 1 demonstrates an example of usage of the core components of PathMiner.

The value of PathMiner stands in reduction of development effort for implementation of the extraction of paths – instead of implementing the complete mining pipeline, the code written by the users of PathMiner should only provide integration of PathMiner into their own mining pipelines. Another strength of PathMiner is the ease in supporting arbitrary languages, which is achieved through integration with ANTLR (Section III-C1).

The distribution of PathMiner contains several usage examples (Section III-E). In the following, we describe PathMiner's inner workings in more detail.

```
val file = File("Example.java")
// instantiate the PathMiner
val miner = PathMiner(
    PathRetrievalSettings(maxHeight = 5, maxWidth = 5))
// generate the AST from the file
val astRoot = Java8Parser().parse(file.inputStream())
// retrieve the paths from the AST
val paths = miner.retrievePaths(astRoot)
// convert the paths into path-contexts and store
storage.store(paths.map { toPathContext(it) },
              entityId = file.path)
// produce the output in the specified folder
storage.save("out_examples/single_java_file")
```

Listing 1. An example of usage of PathMiner in Kotlin

A. An Overview Of The Internals

Converting a piece of code into its path-based representation includes several steps, which define the workflow and architecture of PathMiner. Figure 2 presents an overview of path extraction workflow and the key components of PathMiner.

Parsing. The first step towards a path-based representation is to build an AST from code. All AST operations in PathMiner operate with `Node` – a simple interface representing a node of an AST. The implementations of `Node` are required to support a limited set of operations that are absolutely essential

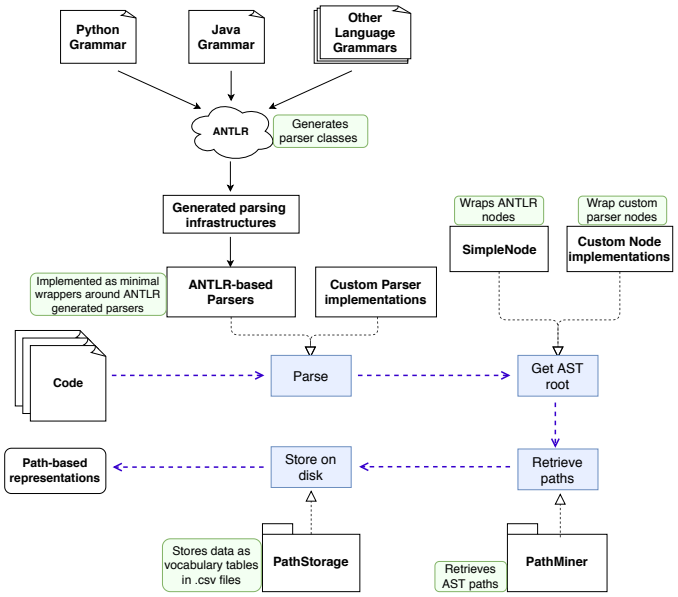


Fig. 2. An overview of PathMiner's workflow and components.

for path extraction, such as retrieval of node's type label and access to children and parent nodes, as well as storage of arbitrary metadata in the node object.

The code in the text form is converted to tree-shaped hierarchies of `Node` implementations by implementations of `Parser` – an interface defining a single method that takes an `InputStream` as an argument and returns an instance of a class implementing `Node`. The concrete implementations of `Node` and the corresponding `Parser` are defined by the programming language and the concrete underlying parser implementation that actually processes the source code. A parser can be implemented either from scratch, which is a hefty task, or, in a simpler way, as an adapter of an external parsing library.PathMiner supports two practical ways to implement a `Parser`, thus supporting a new programming language: (1) conversion from an existing standalone library with parsing capabilities (see `GumTreeJavaParser` class, which wraps the AST representation created by GumTree [18] into `Node`, for an example) and (2) generation of a parser from a predefined grammar with ANTLR (see Section III-C1).

Extracting paths. Extracting the paths from the tree is the task of the `PathWorker` class. Given a `Node` object representing the root node of an AST and an object that defines limitations on width and height of the paths, `PathWorker` traverses the tree bottom-up starting from the leaves, memorizing sequences of nodes encountered during the traversal up to the given node. For each non-leaf node, `PathWorker` matches the sequences from each of the node's children and collides the pairs of these sequences to complete paths. Matching of the sequences is performed in a way that ensures that all generated paths are trivial, *i.e.,* each path features a node at most once, and that the paths do not exceed the limits on height and width.

Output. Upon extraction, paths are passed to an implementation of `PathStorage` interface. These entities take care of storing the paths and producing the output. An AST of even a simple program may potentially yield thousands of paths, if the limits are not strict enough. Due to a potentially high amount of produced data, the default implementation of `PathStorage` (*i.e., VocabularyPathStorage*) uses a deduplication technique to ensure an efficient usage of the memory. The output format (Section III-D) is also designed to avoid storing duplicate information in the output.

PathMiner features multiple extension points to ensure its applicability to a wide range of path extraction tasks. Section III-C provides details on possible extension scenarios.

B. Technologies In Use

PathMiner is implemented in Kotlin [19], but its output is designated for use in machine learning pipelines, which are commonly implemented in Python. While implementing PathMiner in Python would help ensure easier interoperability, implementation of PathMiner in a statically typed language was our prerequisite to ensure better code quality, null-safety, maintainability, and ease of debugging. To compensate for the extra effort of using the output of PathMiner in Python pipelines, the distribution of PathMiner includes a Python library to handle its output format, as well as an example of its usage for a machine learning task.

PathMiner depends on GumTree for one of the parser implementations, on the ANTLR runtime for generated parsers, and (only in compile time) on JUnit for unit testing.

C. Extensibility

PathMiner includes several extension points, which help ensure its applicability in a wide range of contexts.

1) Custom language support: One can easily extend PathMiner to process code in languages that are not supported out of the box. We achieve this by integrating with ANTLR [16].

```
class PythonParser : Parser<SimpleNode> {
    override fun parse(content: InputStream): SimpleNode? {
        // Instantiate the lexer and parser generated by ANTLR
        val lexer = Py3Lexer(ANTLRInputStream(content))
        val tokens = CommonTokenStream(lexer)
        val parser = Py3Parser(tokens)
        // Retrieve the root node from the tree by ANTLR
        val context = parser.file_input()
        // Convert the tree to PathMiner format
        return convertAntlrTree(context, Py3Parser.ruleNames)
    }
}
```

Listing 2. A complete implementation of Python support for PathMiner

Given a grammar in a predefined format, ANTLR generates a lexer and a parser for the language described by the grammar. Resulting classes transform the source code input into a syntax tree. With many existing grammars for ANTLR [20], PathMiner supports a wide variety of languages. In fact, any language supported by ANTLR 4 can be supported by PathMiner through the implementation of a simple wrapper around the lexer and parser classes generated by ANTLR. Implementing the wrapper is necessary due to variation in methods to retrieve the root node of the parse tree across

various generated ANTLR parsers. However, the coding effort is very minimal: Listing 2 features a complete example of an implementation of support for a new language in PathMiner, containing only 4 lines of meaningful code.

2) AST splitting: Parsers in PathMiner generate a single AST for the whole input – the basic unit of input is a file. Depending on the task, one can operate with smaller units of code, such as method definitions. To support possible scenarios of use for extracting paths from smaller units of code, PathMiner features the `TreeSplitter` interface. Its implementations extract finer units from the tree, returning a collection of AST nodes, each representing such unit.

```
interface TreeSplitter<T : Node> {
    fun split(root: T): Collection<T>
}
```

Due to variability of AST node types across languages and corresponding parsers, the implementations of `TreeSplitter` are not reusable across languages and have to be implemented individually. Our distribution of PathMiner includes an example implementation (`GumTreeMethodSplitter`) that extracts method definition nodes from the ASTs generated by `GumTreeJavaParser`.

3) Output: Storage of extracted path-contexts and generation of output are handled by the ancestors of the `PathStorage` class. PathMiner includes a default implementation (`VocabularyPathStorage`) that handles the storage and output in a memory-efficient manner. However, mining tasks that require alternative representations of the data can be supported by implementing a custom `PathStorage`.

D. Output format

By default, PathMiner uses a custom output format, introduced by the authors of the path-based representation [14]. The format is designed to present the data in numerical form, while storing it in a memory-efficient manner. Figure 3 presents an overview of the format, which is based on vocabulary tables. The format exploits the fact that, due to the structured nature of code and limited number of unique node types and tokens, many identifiers and paths are likely to occur more than once over mining tasks of significant size. The storage algorithm associates every token, node type, and path with a unique identifier, thus avoiding storing duplicate data. Each vocabulary table is stored on disk in a separate `.csv` file.

E. Distribution And Usage Examples

The distribution of PathMiner [21] includes several examples of its usage. The examples of path retrieval tasks, implemented in Kotlin and Java, are located in the `examples` package.

To ease further use of the vocabulary-based output of PathMiner in real-world machine learning tasks, we include a small Python library that reads the output and wraps it into wrapper classes for fast access to tokens, paths, node types and path-contexts. Data in this form can later be used with any machine learning framework.

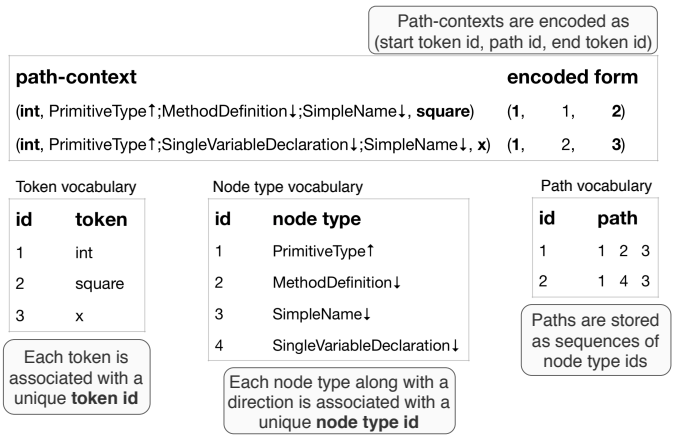


Fig. 3. An example of vocabulary encoding for two path-contexts

To demonstrate the usage of the Python library,[1] we also provide an example of use of the path-based representations for an actual machine learning task. We implement a linear classifier that is trained to distinguish between two possible projects of origin for a file based on its path-based representation. When evaluated on two projects of several thousand Java files each, the classifier achieves over 99% accuracy, precision, and recall after a few minutes of training.

While we only provide the classifier as an example of usage of PathMiner's output for a new task, high accuracy achieved with a very simple model suggests that the range of applications for path-based representations stretches beyond existing research. This point highlights the potential value of PathMiner for the research community.

IV. Quality and performance

The critical components of PathMiner—path extraction pipeline and parsers—are covered by unit tests, which are run on every change by a continuous integration system. To ensure quality and readability of PathMiner's sources, we employed code reviews during the development process: All of the code in PathMiner was reviewed by, at least, a second developer.

On a mid-range developer machine, PathMiner is capable of processing 300-500 Java files per second, with the most of CPU time being spent on I/O and parsing. This suggests that PathMiner is efficient and is not going to be a bottleneck in existing pipelines. Our own experience with using PathMiner for ongoing research projects confirms its high performance.

V. Acknowledgements

Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation – SNF Project PP00P2_170529.

[1]The Python library and its usage example are located in the `/py_example` folder in the distribution. The folder contains a `README.md` file with instructions to run the example.

REFERENCES

[1] X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst, "Program synthesis from natural language using recurrent neural networks," *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

[2] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2017, pp. 440–450.

[3] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 311–322.

[4] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[5] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 200–210.

[6] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 2073–2083.

[7] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 489–504.

[8] R. R. Bunel, A. Desmaison, P. K. Mudigonda, P. Kohli, and P. Torr, "Adaptive neural compilation," in *Advances in Neural Information Processing Systems*, 2016, pp. 1444–1452.

[9] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI'14: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, pp. 419–428.

[10] A. Gupta and N. Sundaresan, "Intelligent code reviews using deep learning," 2018.

[11] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security), Washington, DC*, 2015.

[12] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, "Source code authorship attribution using long short-term memory based networks," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 65–82.

[13] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 404–419.

[14] ——, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[15] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to error-handling specification mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 423–433.

[16] "ANTLR," https://antlr.org/, accessed: 2019-01-16.

[17] A. T. Ying and M. P. Robillard, "Code fragment summarization," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 655–658.

[18] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[19] "Kotlin programming language," https://kotlinlang.org/, accessed: 2019-01-21.

[20] "antlr/grammars-v4: Grammars written for antlr v4; expectation that the grammars are free of actions." https://github.com/antlr/grammars-v4, accessed: 2019-01-21.

[21] "pathminer," https://github.com/vovak/astminer/releases/tag/pathminer, accessed: 2019-03-15.