PyDriller: Python Framework for Mining Software Repositories

Davide Spadini
Delft University of Technology
Software Improvement Group
Delft, The Netherlands
d.spadini@sig.eu

Maurício Aniche
Delft University of Technology
Delft, The Netherlands
m.f.aniche@tudelft.nl

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

ABSTRACT

Software repositories contain historical and valuable information about the overall development of software systems. Mining software repositories (MSR) is nowadays considered one of the most interesting growing fields within software engineering. MSR focuses on extracting and analyzing data available in software repositories to uncover interesting, useful, and actionable information about the system. Even though MSR plays an important role in software engineering research, few tools have been created and made public to support developers in extracting information from Git repository. In this paper, we present Pydriller, a Python Framework that eases the process of mining Git. We compare our tool against the state-of-the-art Python Framework GitPython, demonstrating that Pydriller can achieve the same results with, on average, 50% less LOC and significantly lower complexity.

URL: https://github.com/ishepard/pydriller,
Materials: https://doi.org/10.5281/zenodo.1327363,
Pre-print: https://doi.org/10.5281/zenodo.1327411

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Mining Software Repositories, GitPython, Git, Python

ACM Reference Format:
Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264598

1 INTRODUCTION

Mining software repository (MSR) techniques allow researchers to analyze the information generated throughout the software development process, such as source code, version control systems metadata, and issue reports [5, 18, 22]. With such analysis, researches can empirically investigate, understand, and uncover useful and actionable insights for software engineering, such as understanding the impact of code smells [13–15], exploring how developers are doing code reviews [2, 4, 10, 21] and which testing practices they follow [20], predicting classes that are more prone to change/defects [3, 6, 16, 17], and identifying the core developers of a software team to transfer knowledge [12].

Among the different sources of information researchers can use, version control systems, such as Git, are among the most used ones. Indeed, version control systems provide researchers with precise information about the source code, its evolution, the developers of the software, and the commit messages (which explain the reasons for changing).

Nevertheless, extracting information from Git repositories is not trivial. Indeed, many frameworks can be used to interact with Git (depending on the preferred programming language), such as GitPython [1] for Python, or JGit for Java [8]. However, these tools are often difficult to use. One of the main reasons for such difficulty is that they encapsulate all the features from Git, hence, developers are forced to write long and complex implementations to extract even simple data from a Git repository.

In this paper, we present Pydriller, a Python framework that helps developers to mine software repositories. Pydriller provides developers with simple APIs to extract information from a Git repository, such as commits, developers, modifications, diffs, and source code. Moreover, as Pydriller is a framework, developers can further manipulate the extracted data and quickly export the results to their preferred formats (*e.g.*, CSV files and databases).

To evaluate the usefulness of our tool, we compare it with the state-of-the-art Python framework GitPython, in terms of implementation complexity, performance, and memory consumption. Our results show that Pydriller requires significantly fewer lines of code to perform the same task when compared to GitPython, with only a small drop in performance. Also, we asked six developers to perform tasks with both tools and found that all developers spend less time in learning and implementing tasks in Pydriller.

2 PYDRILLER

Pydriller is a wrapper around GitPython that eases the extraction of information from Git repositories. The most significant difference between the two tools is that GitPython offers many features (almost all the features of Git), while PyDriller offers only features that are important when performing MSR tasks, thus hiding the underlying complexity to the end user. In this section, we explain the design of Pydriller, as well as its main APIs.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264598

2.1 Domain Object

Commit. It contains all the information regarding the commit: the hash, the committer (name and email), the author (name and email), the message, the authored and committed dates, a list of its parents' hashes (a merge commit has two parents), and the list of modified files (see 'Modification' object below). Since loading the entire object is expensive and time consuming (*e.g.*, Pydriller needs to retrieve and parse the diff of the commit), objects are *lazy loaded*, i.e., are only computed when needed.

Modification. This object carries information regarding a file changed in a commit. A modification object has the following fields:

Old path: old path of the file (can be *None* if the file is added).
New path: new path of the file (can be *None* if the file is deleted).
Change type: type: 'Added', 'Deleted', 'Modified', or 'Renamed'.
Diff: diff of the file as Git presents it (starting with *@@ xx,xx @@*).
Source code: source of the file (can be *None* if the file is deleted).
Added: number of lines added.
Removed: number of lines removed.
Filename: The name of the file.

2.2 Architecture

RepositoryMining. This class is in charge of running the MSR study. The only required parameter of this class is the path to the Git repository to analyze. Based on the Git path, the framework will return the list of commits in the repository.

Since MSR studies are highly customizable, to allow a researcher to customize the study, we expose a set of APIs , making it possible to set the dates in which Pydriller should start to analyze, as well as filtering only specific commits. The complete list of filters is the following:

Select starting point: *since* (after this date), *from commit* (after this commit hash), and *from tag* (after this commit tag)
Select ending point: *to* (up to this date), *to commit* (up to this commit hash), and *to tag* (up to this commit tag)
Select by commits: *single* (single hash of the commit), *only in branches* (only consider certain branches), *only in main branch* (only commits that belong to the main branch), *only no merge* (only commits that are not merge commits), and *only modifications with file types* (only commits in which at least one modification was done in that file type, *e.g.*, by specifying '.java', only commits with at least one Java file was modified are visited.)

In the following, we present some examples of how metrics can be customized and adapted to various MSR studies:

```
# Analyze single commit
RepositoryMining('path/to/the/repo',
    single='6411e3096dd2070438a17b225f4447')

# Since 8/10/2016
dt1 = datetime(2016, 10, 8)
RepositoryMining('path/to/the/repo', since=dt1)

# Between 2 dates
dt1 = datetime(2016, 10, 8, 17, 0, 0)
dt2 = datetime(2016, 10, 8, 17, 59, 0)
RepositoryMining('path/to/the/repo', since=dt1, to=dt2
    )

# Between tags
first_tag = 'tag1'
last_tag = 'tag2'
RepositoryMining('path/to/the/repo', from_tag=
    first_tag, to_tag=last_tag)

# Only commits in main branch
RepositoryMining('path/to/the/repo',
    only_in_main_branch=True)

# Only commits in main branch and no merges
RepositoryMining('path/to/the/repo',
    only_in_main_branch=True, only_no_merge=True)

# Only commits that modified a java file
RepositoryMining('path/to/the/repo',
    only_modifications_with_file_types=['.java'])
```

After the user configured the *RepositoryMining* class, thus specifying which commits to analyze, the user has only to call the **traverse_commit()** function that will return the desired list of commits. Internally, Pydriller obtains the list of all the commits, filters out the unnecessary ones, converts the commits in a domain object, and returns the list of resulting commits. This approach has the advantage that all the complexity is hidden from users.

Furthermore, if users need more than just visiting commits, we created a wrapper for the most common utilities of Git, for example checkout, reset, log, show a single commit. We also built APIs to help researchers in MSR studies, including:

Parse diff : The diff presented by Git is difficult to parse. With this API, given a diff, it returns a dictionary with the added and deleted lines. For both groups, the function returns a tuple, corresponding to 1) line number in the file and 2) actual line.
Get commits that last modified lines: This function applies SZZ [17]. Given a 'Commit' object as parameter, it returns the set of commits that changed last the lines modified in the files included in the commit. The algorithm works as follow (for every file in the commit): 1) obtain the diff, 2) obtain the list of deleted lines, and 3) blame the file and obtain the commits were those lines were changed last.

To facilitate the data analysis, Pydriller gracefully handles GitPython exceptions. For example, when retrieving the source code of non-UTF-8 files (*e.g.*, bytecodes), GitPython raises an exception, while PyDriller returns an empty string. Hence, Pydriller reduces the burden of handling several exceptions that a developer would have to do otherwise.

3 EVALUATION

To evaluate our tool, we compare Pydriller against the state-of-the-art Git framework for Python–GitPython. We select five different common MSR tasks that we encountered in our experience as researchers in the MSR field, and implement them using both frameworks. The tasks follow:

Task 1: Calculating complexity of the added lines for every commit. For the sake of simplicity, we define complexity as the number of *if* statements in the diff.
Task 2: Detecting bug inducing commits. We use SZZ [17] to retrieve the commits where the bug was introduced, as normally done in previous literature [14, 15, 20].
Task 3: Obtaining the list of commits that only modified Java files.
Task 4: Lines of code per source file over time.
Task 5: Day of the week developers fixed more bugs between two releases.

Table 1: Comparison between PyDriller and GitPython.

		PyDriller	GitPython	Total
Ex1	Time	00:14:59	00:13:25	+00:01:34
	Max Memory (MB)	169	148	+21
	LOC	21	54	-61%
	Complexity	7	15	-53%
Ex2	Time	00:01:14	00:01:00	+00:00:14
	Max Memory (MB)	–	–	–
	LOC	19	66	-71%
	Complexity	5	6	Similar
Ex3	Time	00:01:24	00:01:14	+00:00:10
	Max Memory (MB)	94	39	+55
	LOC	10	18	-44%
	Complexity	2	6	-67%
Ex4	Time	00:15:03	00:15:47	-00:00:44
	Max Memory (MB)	162	132	+30
	LOC	17	42	-60%
	Complexity	6	15	-60%
Ex5	Time	00:00:02	00:00:03	Similar
	Max Memory (MB)	–	–	–
	LOC	12	19	-37%
	Complexity	3	4	Similar

We run the five tasks (implemented in both PyDriller and GitPython) on 50 OSS projects, 25 belonging to the Eclipse Foundation and 25 to Apache. We selected the projects using GHTorrent [9], taking the 25 most starred projects of the two organizations. For the sake of simplicity, we only report the results of one project, Apache Hadoop. The results of the other 49 projects can be found in our on-line appendix [19].

We compare the tools under different metrics: lines of code (LOC) and complexity (McCabe complexity [11]) of both implementations, as well as their memory consumption, and execution time. Table 1 shows the results. For all the exercises, both in PyDriller and GitPython, the number of lines that are not a core functionality (for example the constructor) is three. We keep this number as it is always the same for all the exercises and for both tools.

Regarding execution time, PyDriller is generally slower than GitPython. This decrease in speed is expected, given that PyDriller is a wrapper built on top of the python framework. However, the difference is small: In the most expensive tasks (Ex1 and Ex4), in which the tools have to analyze the diff or source code of every file in 20,000 commits, PyDriller is only 1:34 minutes slower in the first case. In the other task, PyDriller is 44 seconds faster than GitPython. Nevertheless, both tools take less than 16 minutes to analyze the entire history of Apache Hadoop (avg. 22 commits per second). As for memory consumption, the tools behave similarly: In some cases, the used memory is less than 50MB. In the most memory consuming task (number 1), the used memory was 169MB.

The large difference between both tools is in terms of LOC and complexity of the implementation. For the former, we see that using PyDriller results (on average) in writing 50% less lines of code than using GitPython. The biggest difference is in the task 2, where the tool had to retrieve the bug inducing commits using the SZZ algorithm: This problem was solved in 19 LOC using PyDriller, while 66 LOC with GitPython (70% difference).

Table 2: Time spent by the participants of the experiment in solving tasks 3 and 4 together.

Participant	Time (minutes) with PyDriller	Time (minutes) with GitPython	Total
P1	45	80	-44%
P2	23	45	-49%
P3	13	20	-35%
P4	19	26	-27%
P5	44	46	–
P6	17	30	-43%

We also observe that the complexity of the code written for PyDriller is significantly lower than for GitPyhon. Table 1 shows that, on average, the code for PyDriller is 60% less complex. This is especially the case in tasks that have to deal with retrieving the diff or source code of the modified files; indeed, obtaining this information in PyDriller is just 1 API call, while GitPython requires many lines of code and exceptions handling.

4 EVALUATION WITH DEVELOPERS

To further evaluate our tool, we invited six developers to perform the same two tasks using both PyDriller and GitPython, and to note the time they took to solve the problems, as well as their personal opinions on both tools. All developers had experience in developing with Python and on performing MSR studies, but they had never used PyDriller nor GitPython before.

We asked the participants to solve tasks 3 and 4. We chose these tasks because they are simpler than the first two (to keep the experiment short) and do not require participants to have notions on how to identify bug fixing commits (Ex5). The setting of the experiment is the following:

- Participants should implement both tasks, first with PyDriller, then with GitPython. Since understanding how to solve the tasks does require some additional time, we asked the participants to start with PyDriller. This choice clearly penalizes our tool, as participants will have a better intuition about the tasks when doing the task in GitPython. However, we believe that PyDriller is simpler to use, and that the difference between the two tools will still be significant.
- Participants should take notes about the time it takes them to implement the tasks. We ask participants to also include the time spent reading the documentation of the two tools, since understanding how to use the tool is part of the experiment.
- After having implemented both tasks, we ask to the participants to elaborate on the different advantages and disadvantages between both tools.

The result of the experiment is shown in Table 2. Five out of six participants spent significantly less time to solve the problems (27% less in the worst case, 49% less in the best case). P5, instead, solved both problems in the same amount of time: the participant did not know how to solve the second task and, since he started with PyDriller, this translated in more time in the first part. When he understood how to solve it, he moved to GitPython already knowing the solution.

All participants agreed that PyDriller was easier to use than GitPython [P_{1-6}]. P_6 said: *"I thought PyDriller was a lot more intuitive than using GitPython. GitPython works exactly like Git, so it isn't very well suited when trying to gain insights about the repository."*

Similarly, P_1 affirmed that, using PyDriller, he was able to achieve the same result with simpler and shorter code, and that he will continue to use PyDriller in his next MSR studies. P_2 added that GitPython is useful when one has to simulate Git commands in Python, but it can be overcomplicated when the goal is to perform MSR studies, for which PyDriller is more appropriate, because it hides this complexity from its users.

5 RELATED TOOLS

In this section we compare PyDriller against two of the most recent and used MSR tools.

Boa [7]: Boa is a domain-specific language and infrastructure that eases MSR. The main difference between PyDriller and Boa is that, while the former can be run on every project, Boa can only be used on their snapshots of GitHub or SourceForge, which currently are 3 and 6 years old. Furthermore, PyDriller is written in Python. Hence, it has all the flexibility of a the programming language and can be used together with other frameworks. Boa, on the other hand, has its own DSL, and can not be used with other (external) libraries. Furthermore, Boa currently includes only the history and source code of Java projects, while PyDriller can be used to analyze repositories of any programming language.

GHTorrent [9]: GHTorrent is a scalable, queriable, offline mirror of data present on GitHub. The main difference between PyDriller and GHTorrent is that, while the former retrieves all the information regarding a commit (*e.g.*, what files changed, diffs, and source code), the latter focuses on GitHub's social data, such as pull requests, issues, and users. However, GHTorrent does not offer the possibility of navigating through the commits or analyzing the project's source code over time (which is a feature of PyDriller).

6 CONCLUSION

In this paper, we presented PyDriller, a Python framework that helps developers on mining software repositories. We showed that with PyDriller, developers can easily extract information from any Git repository, such as commits, developers, modifications, diffs, and source codes, since PyDriller releases simple APIs to help researchers and practitioners performing MSR.

We evaluated PyDriller on 5 exercises, comparing it against GitPython. The evaluation showed that using PyDriller results in writing (on average) half the code, and 60% less complex. Furthermore, we asked 6 developers to solve two exercises using our tool, and they all agreed that PyDriller helped them in solving the problems in less time with less code.

The first version of PyDriller has been released on 9th April 2018, and since then it has been downloaded approximatively 1,000 times per month (as computed through "Pypinfo" [1] and Google BigQuery). We plan to keep improving Pydriller's performance as well as to perform more user studies with the goal of understanding even better what MSR researchers require in their studies.

[1] https://github.com/ofek/pypinfo

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 642954. A. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] [n. d.]. GitPython. https://github.com/gitpython-developers/GitPython.
[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proc. of the 35th International Conference on Software Engineering*. 712–721.
[3] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. 2010. Are popular classes more defect prone?. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 59–73.
[4] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix?. In *Proc. of the 11th working conference on mining software repositories*. ACM, 202–211.
[5] K. K. Chaturvedi, V. B. Sing, and P. Singh. 2013. Tools in Mining Software Repositories. In *2013 13th International Conference on Computational Science and Its Applications*. 89–98.
[6] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the Impact of Design Flaws on Software Defects. In *Proc. of the 10th International Conference on Quality Software*. 23–31.
[7] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proc. of the 35th Int'l Conference on Software Engineering*. 422–431.
[8] Eclipse Foundation. [n. d.]. JGit. https://www.eclipse.org/jgit/.
[9] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proc. of the 10th Working Conference on Mining Software Repositories*. 233–236.
[10] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this?: Evaluating code contributions with language models. In *Proc. of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 157–167.
[11] T J McCabe. 1976. A Complexity Measure.
[12] Audris Mockus, Roy T Fielding, and James Herbsleb. 2000. A case study of open source software development: the Apache server. In *Proc. of the 22nd international conference on Software engineering*. Acm, 263–272.
[13] Steffen M. Olbrich, Daniela Cruzes, and Dag I. K. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. 1–10.
[14] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *Proc. of the 30th International Conference on Software Maintenance and Evolution*. 101–110.
[15] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. The Scent of a Smell: An Extensive Comparison between Textual and Structural Smells. *IEEE Transactions on Software Engineering* (2017).
[16] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2018. Re-evaluating Method-Level Bug Prediction. In *Proc. of the 25th International Conference on Software Analysis, Evolution, and Reengineering*. 592–601.
[17] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proc. of the 2nd International Workshop on Mining Software Repositories*.
[18] Francisco Zigmund Sokol, Mauricio Finavaro Aniche, and Marco Aurélio Gerosa. 2013. MetricMiner: Supporting researchers in mining software repositories. *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013* (2013), 142–146.
[19] Davide Spadini. 2017. PyDriller Dataset. https://doi.org/10.5281/zenodo.1327363
[20] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. 2018. When Testing Meets Code Review: Why and How Developers Review Tests. In *Proc. of the 40th International Conference on Software Engineering*. 677–687.
[21] Patanamon Thongtanunam, Shane Mcintosh, Ahmed E. Hassan, and Hajimu Iida. [n. d.]. Review participation in modern code review - An empirical study of the android, Qt, and OpenStack projects. *Empirical Software Engineering (EMSE)* 22, 2 ([n. d.]).
[22] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. 2008. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In *2008 International Conference on Software Testing, Verification, and Validation*, Vol. 3. IEEE, 220–229.