Tooling for Time- and Space-efficient git Repository Mining

Fabian Heseding
fabian.heseding@student.hpi.uni-
potsdam.de
Hasso Plattner Institute, Digital
Engineering Faculty, University of
Potsdam
Potsdam, Germany

Willy Scheibel
willy.scheibel@hpi.uni-potsdam.de
Hasso Plattner Institute, Digital
Engineering Faculty, University of
Potsdam
Potsdam, Germany

Jürgen Döllner
juergen.doellner@hpi.uni-
potsdam.de
Hasso Plattner Institute, Digital
Engineering Faculty, University of
Potsdam
Potsdam, Germany

ABSTRACT

Software projects under version control grow with each commit, accumulating up to hundreds of thousands of commits per repository. Especially for such large projects, the traversal of a repository and data extraction for static source code analysis poses a trade-off between granularity and speed.

We showcase the command-line tool pyrepositoryminer that combines a set of optimization approaches for efficient traversal and data extraction from git repositories while being adaptable to third-party and custom software metrics and data extractions. The tool is written in Python and combines bare repository access, in-memory storage, parallelization, caching, change-based analysis, and optimized communication between the traversal and custom data extraction components. The tool allows for both metrics written in Python and external programs for data extraction. A single-thread performance evaluation based on a basic mining use case shows a mean speedup of 15.6× to other freely available tools across four mid-sized open source projects. A multi-threaded execution allows for load distribution among cores and, thus, a mean speedup up to 86.9× using 12 threads.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; **Software maintenance tools**; **Software version control**.

KEYWORDS

Mining Software Repositories, Python, Git

1 INTRODUCTION

With software playing a key role in the modern world, gaining insights about software becomes crucial [9]. Such insights entail quality metrics about the software as well as data about the development process. One approach to gaining insights about software, especially its source code, is deriving it from their repositories

that are governed by version control systems (VCS) such as git [2]. Mining Software Repositories (MSR) focuses on extracting and analyzing data available in software repositories to uncover interesting, helpful, and actionable information about the software system [14]. One goal is to quantify and monitor the quality of the developed software employing software metrics [8], another is to explore new ways of making sense of the data. However, traversing and gathering the vast amount of data poses a challenge [4, 9–11].

For an exemplary analysis of a software repository, we assume a process that covers the following parts: (1) Acquiring the software repository (e.g., `git clone`), (2) Provisioning the files to run an analysis on (e.g., `git checkout`), (3) Running an analysis on the files (applying existing tools or custom scripts that derive quality metrics), and (4) Reporting the results (e.g., by creating reports or interactive visualization). During this process, the analysis usually covers parts or the whole repository and, thus, requires some sort of traversal (Figure 1). With a list of revisions to analyze, each revision is checked out to the working directory. Upon each checkout, tools that calculate software metrics are run and the results are logged. After handling all revisions, the results are merged and reported. For example, a linter may run on a working directory and output a quality score of the source code. Another example is to measure the source code using software metrics that report higher-level information. Tracking those scores across versions allows for detecting changes, trends, and insights about a software project's development status and quality.

If an applied software developments process includes regular software analysis, the data mining from the repository gets demanding over time. This demand occurs through (1) ever-growing source code, (2) the number of changes created during the overall development, (3) the number of concurrent development branches, e.g., through multiple developers or multiple maintained versions, (4) the complexity of the analyses performed, and (5) the expectation for timely results for effective use in the development process.

In this paper, we showcase a low-level analysis tool for efficient git repository mining named pyrepositoryminer [5] and discuss architectural and design decisions that improve the execution speed of the tool. It is written in Python, available at PyPi, and serves as a framework that provides repository traversal and allows to use custom metrics provided as scripts and integrate existing metric calculation tools. A first analysis using the mid-sized open-source projects numpy, matplotlib, pandas, and tensorflow shows greatly improved execution times compared to other low-level source code repository MSR tools. Although provisioned as a separate tool focusing on Python and git, the discussed approaches are applicable to other MSR tools and VCS, too.
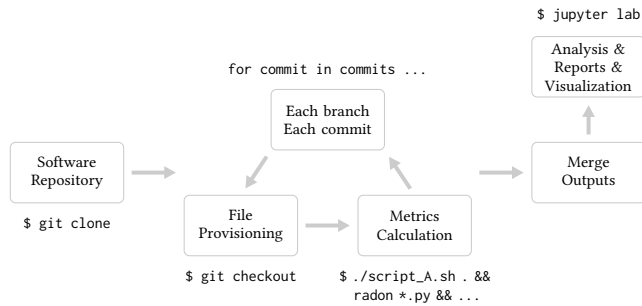
Figure 1: Example process for software analysis that requires repository mining across multiple branches and commits, while computing multiple metrics using third-party and custom tools. The results can than be used for further analysis, static reports, or interactive visualization.

2 RELATED WORK

In the last years, many tools for mining git and related systems were proposed and researched. Although a number VCS are suitable for mining of software data, the specific ecosystem around git seems to be a focus. As such, git, a decentralized version control system, and Github, the software project management platform built around git repositories, are driving best practices in software development, software analyses, and repository mining [2, 6]. When faced with an MSR task, researchers and practitioners have the choice to use one of the few existing tools and frameworks that focus on mining, use a higher-level system or infrastructure where they integrate their analysis in, or develop their own tool tailored to their specific needs. Existing tools and infrastructure, however, often require learning a domain-specific language to formulate the analysis in or constrain the analysis to a predefined use case.

MSR Tools. A tool similar in usage to our implementation is PyDriller[1] [14], an open-source Python framework for Mining Software Repositories. PyDriller is a high-level interface around the comparatively low-level GitPython[2] library that offers ease of use for everyday MSR tasks with minimal to no decrease in performance. While PyDriller offers an easy to learn interface for Python developers, its performance is heavily dependent on the implementation of the concrete task at hand. For example, when iterating over all of the blobs of all commits and calculating metrics, unchanged files are not skipped even though the metrics are already calculated. Additionally, this example would run on one core if the researcher does not implement any parallelization.

Another tool that aims to simplify mining tasks is RepoFS[3] [13], a virtual filesystem mapping to a Git repository. RepoFS offers directories for each revision within the filesystem and maps access to a bare repository. Thus, it enables to run filesystem-based mining tools across revisions without incurring disk IO from repeated checkouts. However, it leaves parallelization and efficient calculation logic to the actual mining tool or task implementation.

With regards to mining tools dealing primarily with GitHub, we found tools for efficiently mining artifacts from GitHub, such as ModelMine [12], as well as for efficient classification of commits for downstream analysis, such as GitCProc [3], using primarily regular expressions. While these tools excel in their target use cases, they are not suited for general purpose analysis of a local repository.

MSR Infrastructure. A prominent tool in the MSR research field tool is Boa [4], a domain-specific language and infrastructure for MSR. Boa enables researchers to reproduce data extraction and analysis by specifying mining tasks in the Boa language and running them on the Boa infrastructure. The Boa infrastructure also offers scalability for expensive tasks. Boa, however, does not focus on efficiency in calculations. Additionally, researchers need to learn a new domain-specific language compared to a few shell commands. Finally, flexibility in custom metric logic is limited by what the language allows.

Other tools offering an infrastructure framework include Crossflow[4] [7], SmartSHARK[5] [15], and World of Code [9, 10]. CrossFlow is a domain specific language and framework that offers scalability for MSR tasks; SmartSHARK aggregates data from different sources in a harmonized schema. The World of Code is an infrastructure that enables research on free and open-source software repositories. While these tools offer scalability, they require researchers to learn a new framework and run their analyses on their infrastructure. This requirement restricts the flexibility of these tools for general-purpose analysis. Finally, projects such as the Three Trillion Lines [11] describe the required infrastructure setup to manage large-scale software analysis. While fault tolerance and domain-specific data structures are important to a large-scale MSR endeavor, we describe an approach to make the actual mining of repositories more efficient.

3 APPROACHES

Efficiency in repository mining can be described by executing only required operations, omitting as much as possible intermediate operations that do not directly contribute to the results, as well as leveraging multi-threading capabilities of current hardware.

RAM Disk. As such, physical representation of files on a disk is an intermediate artifact that can be replaced with a RAM Disk that would improve read and write access during traversal of a repository. Eventual writes during mining would also pose no additional strain on the lifetime of HDDs and SSDs. As creation, management, and usage of a RAM Disk is usually done using the operating system and not on a tool-level, this approach is feasible for all MSR tools.

Bare Repository Access. Further, checking out revisions into a working directory is inefficient as it incurs disk IO by reading the files from the repository folder and writing them to the working directory [13]. Similarly, a working directory creates a copy of data already available in a git repository, which can be avoided if analyses and tools are versatile to handle input data using in-memory interfaces. With bare repository access – a standard concept of git – the used memory on disk is strictly bound to the space requirements of the bare repository. Additionally, using a working

---

directory complicates parallelization, i.e., a parallel computation would require one working directory per thread.

Parallelization. We apply concurrent computation of metrics and source code analyses on a per-commit level. Such parallelization is feasible if the results for different commits can be computed independently. As repository mining is a task that is inherently idempotent and should only require read access to a software repository, this holds on a conceptual level. From an implementation perspective, this also holds if working directories can be omitted, or, alternatively, each thread uses its own working directory.

Caching. Regarding metric calculations, the idempotent property of analyses allow for thorough caching of results. This is a large advantage for software repositories, as each commit usually changes only parts of the source code [1]. If, for example, a blob rarely changes across revisions but is analyzed for each revision nonetheless, these calculations are redundant and should be cached instead.

Change-based Analysis. Likewise, some implementations of analyses or metrics operate on a whole working directory instead of only changed files for a commit. However, unchanged files were presumably mined with an earlier commit during the analysis. This optimization can be leveraged using a cache and filtering for files that were not mined in an earlier commit. Even external tools can profit from this filter.

Optimized Communication between Components. Last, we consider an optimized communication through in-memory string representation of files. This optimization aims to reduce file operations and relate to the access of the bare repository and handling of the repository on a RAM disk. The communication between the traversing component and analysis components is ideally done through contents residing in RAM and not on disk.

4 IMPLEMENTATION

We target all the performance optimizations with our tool and further focus on usability, extensibility, and integration. As a result, we propose a tool that applies the optimization approaches internally – by architecture and design – and that allows further extensions with user-specific analyses and metrics (Figure 2). We implemented the tool as an open-source Python project available on PyPi[6] with its source code hosted on GitHub[7], published under the GPL-3.0 License. We leverage pygit2[8], the python bindings for libgit2[9], for direct interaction with Git objects of the bare repository. The command-line interface is built using the Typer[10] framework and comprises four commands: (1) `clone`, (2) `branch`, (3) `commits`, and (4) `analyze`.

Command-line Interface. The `clone` command clones a bare repository from a URL pointing to a git repository into a local path. The `branch` command returns the branches of a repository. Remote and local branches can be included or excluded using the

---

[6]https://pypi.org/project/pyrepositoryminer/
[7]https://github.com/fabianhe/pyrepositoryminer
[8]https://github.com/libgit2/pygit2
[9]https://github.com/libgit2/libgit2
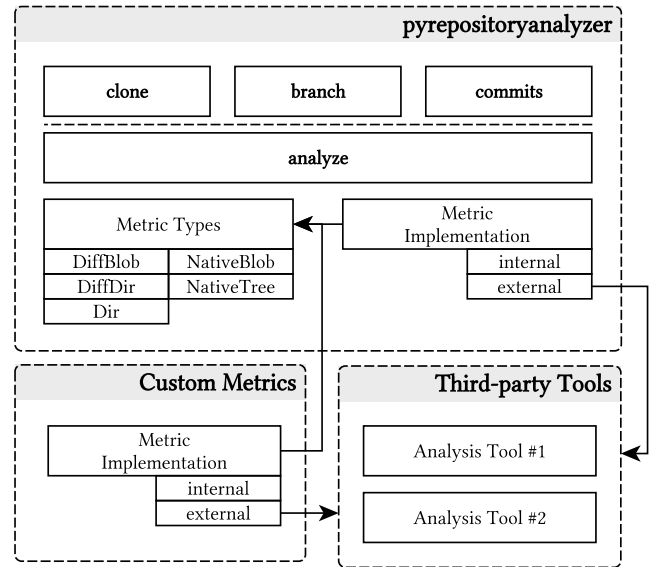[10]https://github.com/tiangolo/typer

Figure 2: Architecture of the pyrepositoryminer. Custom metrics can be developed as (1) internal metrics using standard Python libraries and source code, or (2) external metrics by wrapping third-party tools, both implemented by subclassing one of the metric types in Python scripts. The custom metrics can be developed, maintained and stored independently from the pyrepositoryminer.

command options. The `commits` command gets the commit hashes of commits on the input branches. By default, merge commits are simplified to their first parent commit, duplicates are eliminated, and the commits are sorted in topological order. This behavior can be changed using command options. Additionally, the output can be limited to the first *n* commits encountered. The `analyze` command analyzes all input commits. Arguments to the analyze command are names of the predefined metrics to use. This tool further supports command options, e.g., to load implementations of custom metrics by using the `--custom-metrics` option, or to configure the number of worker processes for parallel analysis (`--workers`).

Tool Operation. As an example, we demonstrate how to analyze the numpy repository hosted on GitHub with regards to the Halstead metrics and the total lines of code on all commits of the main branch (see also Listing 1): This example (1) mounts a RAM disk of size 250 MB, (2) clone the numpy repository from GitHub to the newly created RAM disk using the `clone` command, (3) collects local branches, (4) and their reachable commits – topologically sorted –, and (5) perform computation of the Halstead and Lines-of-Code metrics on each commit.. The results are piped to a file for reporting. As cleanup, and the RAM disk is unmounted.

Metric Types. Metrics are implemented as subclasses of their respective metric type. Each superclass corresponds to the input format and intermediate calculations that can be shared across all metrics of that metric type. We provide classes for file blobs (`NativeBlobMetric`), commit trees (`TreeMetric`), and directories (`DirMetric`), each with the option of only using the touched blobs

```
1  > ./mount-ramdisk-macos.sh 512000
2  > pyrepositoryminer clone https://github.com/numpy/numpy \
3      /Volumes/RAMDisk/numpy.git
4  > pyrepositoryminer branch /Volumes/RAMDisk/numpy.git \
5      | pyrepositoryminer commits /Volumes/RAMDisk/numpy.git \
6      | pyrepositoryminer analyze /Volumes/RAMDisk/numpy.git \
7      halstead loc > out.jsonl
8  > ./unmount-ramdisk-macos.sh disk3
```

Listing 1: Running the pyrepositoryminer on macOS. The content of the ramdisk scripts are available in the repository of the tool.

of a commit (`DiffBlobMetric`, `DiffTreeMetric`, `DiffDirMetric`). We encourage to write metrics that use native git objects rather than a working directory for analysis. Metrics that request a checked-out working directory, i.e., subclasses of the `DirMetric` and the `DiffDirMetric`, are handled by providing a working directory nonetheless. For example, the class of a directory of touched blobs - `DiffDirMetric` - finds all touched blobs of a revision and checks out a working directory on which filesystem-based tools can run.

User-defined Metrics. Custom metrics can be easily implemented by using the library interface of the tool. The tool provides five superclasses to subtype from – each corresponding to one of the supported metric types – such that own analyses and metrics can be implemented in scripts and get loaded together with the tool. Existing tools can be integrated by implementing them as a metric. For example, an executable of a static source code analysis tool such as Tokei[11] can get wrapped as a metric by using the `DiffDir` metric and inter-process communication and string conversion.

5 EVALUATION

We evaluate the pyrepositoryminer using two performance measurements: (1) the time required to iterate all commits of a repository and (2) the runtime of a basic mining use case. The measurements are compared to similar implementations using PyDriller. We measured on an Intel Core i9 CPU with 18 cores at 3.00 GHz and 128 GB of main memory on an Ubuntu 20.04. The software repositories used for the measurements are four publicly available mid-sized GitHub repositories (Table 1). Before measuring either tool, we loaded the repositories onto a RAM disk.

Commit Iteration Throughput. In our commit iteration use case, the task is to get and print each revision id on a repository's main branch. We aim to measure the framework overhead cost without a specific software metric. Using the pyrepositoryminer, this is accomplished by running and piping the inputs and outputs of the `branch` and `commits` commands without any further configuration. Using PyDriller, this is accomplished by running the `traverse_commits` method. As a result, the pyrepositoryminer achieves a minimum throughput of 23 229 commits/s and a maximum of 28 899 commits/s, whereas PyDriller achieves a throughput between 4191 commits/s and 6155 commits/s (see Figure 3). The mean speedup achieved by the pyrepositoryminer compared to PyDriller is at 4.8×. When analyzing large software repositories with

---

[11]https://github.com/XAMPPRocky/tokei

Table 1: Open source projects used for evaluation. The reported number of commits include all commits reachable by all remote branches.

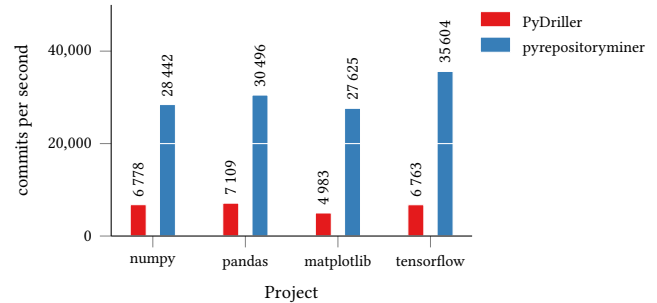| Identifier	Repository	# Commits
numpy	https://github.com/numpy/numpy.git	17 122
matplotlib	https://github.com/matplotlib/matplotlib.git	17 183
pandas	https://github.com/pandas-dev/pandas.git	25 464
tensorflow	https://github.com/tensorflow/tensorflow.git	76 299

Figure 3: Throughput of commit iteration on all reachable commits on all remote branches.
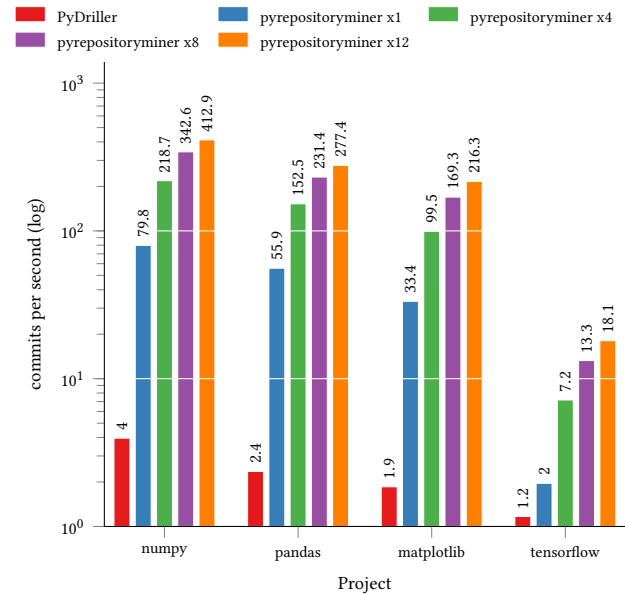

Figure 4: Throughput of calculating all lines of code per reachable commit on all remote branches.

more than 60 000 commits, the total runtime of iterating commits stays within a few seconds.

Basic Mining Use Case Throughput. In our basic mining use case, the task is to calculate the total lines of code for each blob in each commit. In the pyrepositoryminer, this is accomplished by implementing a blob level metric that counts each new line character

in a blob's source code. The pyrepositoryminer ensures that they run in parallel only on modified files. With the PyDriller implementation, each commit is iterated in a loop and analyzed using the built-in `loc` attribute on modified files only. As a result, the pyrepositoryminer achieves a single-threaded minimum throughput of 2 commits/s and a maximum of 79.8 commits/s on the tested repositories, whereas PyDriller achieves a throughput between 1.2 commits/s and 4 commits/s (see Figure 4). The mean single-threaded speedup achieved by the pyrepositoryminer compared to PyDriller is at 15.6×, while it is 68.9× as high using 12 cores.

Discussion. The results from the performance measurements seem promising. However, iterating through a repository and providing the source code files is not the main source of computation time when applying sophisticated analysis. Our tool aims to provide a base for fast runtimes by eliminating omittable operations and allowing for concurrent computations between commits, files, and metrics. Additionally, our tool can be used as a wrapper to improve runtimes on existing tools, e.g., using the `DiffDir` metric, leveraging a subset of the performance optimization approaches. The current limitations include a focus on Python for internal metrics and git as VCS. However, alternatives such as Boa [4] are enticing because they offer the infrastructure to perform research. Additionally, using a custom implementation offers flexibility in research and optimization beyond what any framework approach can offer. On the other hand, our concrete implementation suffers from the drawback that researchers need to learn our framework's interface to use it. Finally, further measurements need to be taken to better understand the achievable speedup compared to current tools.

6 CONCLUSIONS

While the number of software projects and their sizes increase, and the need for software analysis and timely results remain essential, we depend on highly efficient tooling. We approached efficiency in repository traversal and metric computation by leveraging bare repository access, in-memory storage, parallelization, caching, change-based analysis, and optimized communication between the software components. We showcased these approaches with the tool pyrepositoryminer. This tool is publicly hosted at Github, available at PyPi, allows the use of externally developed scripts and the integration of third-party tools for analysis and software metrics. First performance results are promising and indicate a single-threaded speedup of 15.6× to other freely available tools across four mid-sized open source projects and even a speedup of 86.9× using 12 threads. This allows software repository mining to be applied more frequently, thoroughly, and timely for mid-sized and large projects. For future work, we target a more in-depth analysis of performance impacts on the individual approaches and a broader analysis across open source projects and industry projects. Further, we see the potential for optimization for a change-based analysis on a line level, in contrast to the file level that is currently used. In addition, the tool itself will profit from a broader list of available metrics and supported languages.

ACKNOWLEDGMENTS

REFERENCES

[1] O. Arafat and D. Riehle. 2009. The Commit Size Distribution of Open Source Software. In *Proc. 42nd Hawaii International Conference on System Sciences.* IEEE, 1–8. https://doi.org/10.1109/HICSS.2009.421

[2] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. 2009. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories.* IEEE, 1–10. https://doi.org/10.1109/MSR.2009.5069475

[3] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. 2017. Gitcproc: A tool for processing and classifying github commits. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, 396–399. https://doi.org/10.1145/3092703.3098230

[4] 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 *2013 35th International Conference on Software Engineering (ICSE).* IEEE, 422–431. https://doi.org/10.1109/ICSE.2013.6606588

[5] Fabian Heseding and Willy Scheibel. 2022. pyrepositoryminer. https://doi.org/10.5281/zenodo.5918480 and hosted on github.com/fabianhe/pyrepositoryminer.

[6] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories.* ACM, 92–101. https://doi.org/10.1145/2597073.2597074

[7] Dimitris Kolovos, Patrick Neubauer, Konstantinos Barmpis, Nicholas Matragkas, and Richard Paige. 2019. Crossflow: a framework for distributed mining of software repositories. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* IEEE, 155–159. https://doi.org/10.1109/MSR.2019.00032

[8] Daniel Limberger, Willy Scheibel, Jürgen Döllner, and Matthias Trapp. 2019. Advanced Visual Metaphors and Techniques for Software Maps. In *Proc. 12th International Symposium on Visual Information Communication and Interaction (VINCI '19).* ACM, 11:1–8. https://doi.org/10.1145/3356422.3356444

[9] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of code: an infrastructure for mining the universe of open source VCS data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* IEEE, 143–154. https://doi.org/10.1109/MSR.2019.00031

[10] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2021. World of code: Enabling a research workflow for mining and analyzing the universe of open source vcs data. *Empirical Software Engineering* 26, 2 (2021), 1–42. https://doi.org/10.1007/s10664-020-09905-9

[11] Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2020. Three trillion lines: infrastructure for mining GitHub in the classroom. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming.* ACM, 1–6. https://doi.org/10.1145/3397537.3397551

[12] Sayed Mohsin Reza, Omar Badreddin, and Khandoker Rahad. 2020. Modelmine: a tool to facilitate mining models from open source repositories. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* ACM, 9:1–5. https://doi.org/10.1145/3417990.3422006

[13] Vitalis Salis and Diomidis Spinellis. 2019. RepoFS: File system view of Git repositories. *SoftwareX* 9 (2019), 288–292. https://doi.org/10.1016/j.softx.2019.03.007

[14] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 908–911. https://doi.org/10.1145/3236024.3264598

[15] Alexander Trautsch, Fabian Trautsch, Steffen Herbold, Benjamin Ledel, and Jens Grabowski. 2020. The smartshark ecosystem for software repository mining. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings.* ACM, 25–28. https://doi.org/10.1145/3377812.3382139