UDC 004.62/.63

V. Yakubiv [1]
D. Terletskyi [2]

MINING SOURCE CODE FROM GITHUB REPOSITORIES

[1] Taras Shevchenko National University of Kyiv
[2] V. M. Glushkov Institute of Cybernetics of the NAS of Ukraine

Анотація

GitHub став однією з найпоширеніших платформ для розміщення коду та інтелектуального аналізу даних з репозиторіїв. Однак видобуток вихідного коду з GitHub'а створює труднощі, зокрема строге обмеження в 30 запитів за хвилину на API пошуку. Ми пропонуємо методологію для видобутку даних з репозиторіїв програмного забезпечення, засновану на філософії UNIX та концепції інструменту збірки Make, ставлячи в пріоритет модульний дизайн, композицію та вивід у текстові файли. Запропонований підхід застосовує прогресивне уточнення та фільтрацію і підходить для збору вихідного програмного коду, коли релевантні репозиторії Git не відомі апріорі.

Ключові слова: аналіз програмного коду, збір репозиторіїв програмного коду, формування проєктних вибірок

Abstract

GitHub has become one of the most widely used platforms for code hosting and repository mining. However, mining source code from GitHub presents challenges, most notably, a strict rate limit of 30 requests per minute imposed on the search API. We propose a methodology for mining software repositories based on the UNIX philosophy and the concept of the Make build tool, prioritising modular design, composition, and text-based output to files. The proposed approach applies progressive refinement and filtering, and is suitable for collecting datasets of any kind of source code where relevant Git repositories are not known a priori.

Keywords: source code mining, mining software repositories, projects sampling

Introduction

Over the past two decades, GitHub has become one of the most popular platforms for code hosting and collaborating on open-source software. Consequently, most researchers use it as a platform for mining software repositories (MSR). However, mining data from GitHub imposes challenges.

Notably, GitHub imposes rate limits on its API, restricting anonymous users to 1 000 requests per hour and authenticated users to 5 000 requests per hour. The search API is subject to a stricter limit of 30 requests per minute. Multiple efforts have been made in the MSR area to overcome these challenges. In the following sections, we overview these and propose our approach for mining source code from GitHub.

Related research

Mining of GitHub repositories can be organised into two broad categories: mining repository contents (i. e., Git data), and mining platform events (i. e., GitHub-specific data), or a combination of both. Approaches in the first category focus on data extractable from Git-repositories themselves, including file contents and commit metadata. The second category of approaches encompasses a broader range of metadata produced throughout the software development process, including issues, pull requests, comments, and events arising from other platform activities; this data is inherently GitHub-specific.

There is a variety of MSR tools, which implement approaches from the mentioned categories. GHTorrent [1–3] is the most prominent and widely adopted tool [4], which falls strictly into the second one. The service

collects GitHub events including information about users, commits, issues, pull requests, etc., and provides a downloadable dataset. It effectively helps MSR researchers overcome the GitHub API limit of 5 000 requests per hour [1].

GitHub Search [5] and SEART Data Hub [6] are efforts of the Software Analytics Research Team (SEART) that together employ a hybrid approach — collecting Git repositories and augmenting it with GitHub-specific data. GitHub Search provides an alternative way to search GitHub repositories, addressing the 30 requests-per-minute rate limit that is widely reported in the MSR research [4]. GitHub Search mirrors GitHub whilst restricting its index to repositories with at least 10 stars. SEART Data Hub is a repository crawler and query engine enabling users via a web interface to request a source code dataset built on demand. Built on top of GitHub Search, SEART Data Hub constrain its index only to repositories containing at least one Java or Python file.

These tools represent complex infrastructural projects comprising back-end servers and databases. Databases in particular require specialised software and potentially local infrastructure to support human validation and review. Furthermore, whilst these projects are well-suited to major use cases, their predefined filters make them unsuitable for certain edge cases. For instance, the SEART Data Hub excludes approximately 95% of Java repositories indexed on GitHub [5]. When outlying projects have to be collected, or when source code in another programming language is required — such as Rust, C++, or Bash — an alternative data collection tool is necessary.

The present work falls within the first category: mining Git-specific data whilst using GitHub solely as a source. In contrast to other efforts, our approach is built around UNIX philosophy, chaining a number of specialised programmes such as Git, Make, and GNU core utilities, with each step producing a text-based file artefact for both the next step and the human validation. We discuss this in the next section.

The mining approach

Our methodology is rooted in the UNIX philosophy, prioritising modular design and the composition of minimalist, single-purpose tools. In particular, we build the process around Make[1], and adhere to its core concept that each step must produce a file as output. Consistent with the UNIX philosophy, the methodology leverages text-based formats such as JSON, YAML, and plain text. Such files are intelligible without specialised software; most outputs can be inspected using `cat` or `less`, or edited in a text editor such as Vim. The overall process can be seen as a chain of Make recipes encompassing small programmes or shell scripts that take a file input and transform it into another file output.

The overall repository mining methodkis illustrated by the Figure 1 as a data flow diagram. Nodes with rounded corners represent functions — denoting commands or scripts — and carry identifiers prefixed with *F*. Rectangular nodes with sharp corners represent input data (identifiers prefixed with *I*) and output data (identifiers prefixed with *O*), each materialised as a file. A stacked rectangle indicates that the corresponding function produces multiple outputs, or is invoked once per input file of the given type.

The mining process takes two user-supplied inputs: a list of repository scopes (*I1*) from which mining begins, and a list of repositories to be excluded (*I3*). In the subsequent step (*F2*), each scope is expanded into a list of repositories (*O2*). These repositories are then filtered (*F3*) and cloned (*F4*), yielding a collection of compressed repository archives (*O4*). File listings are extracted from these archives (*F5*) to produce *O5*, which is analysed (*F6*) and reduced to a list of selected files (*O6*). The files identified in *O6* are then extracted from the repositories (*F7*), and the final output (*O7*) constitutes the target file collection. To explicate the process, we will follow an example of collecting a dataset of Rust files.

[1] *Make* is a command-line tool originally designed by Stuart Feldman in 1976. It has a few implementations, sometimes referred to as BSD Make, GNU Make, etc. Despite the variations between those implementations, we refer to the general tool concept.

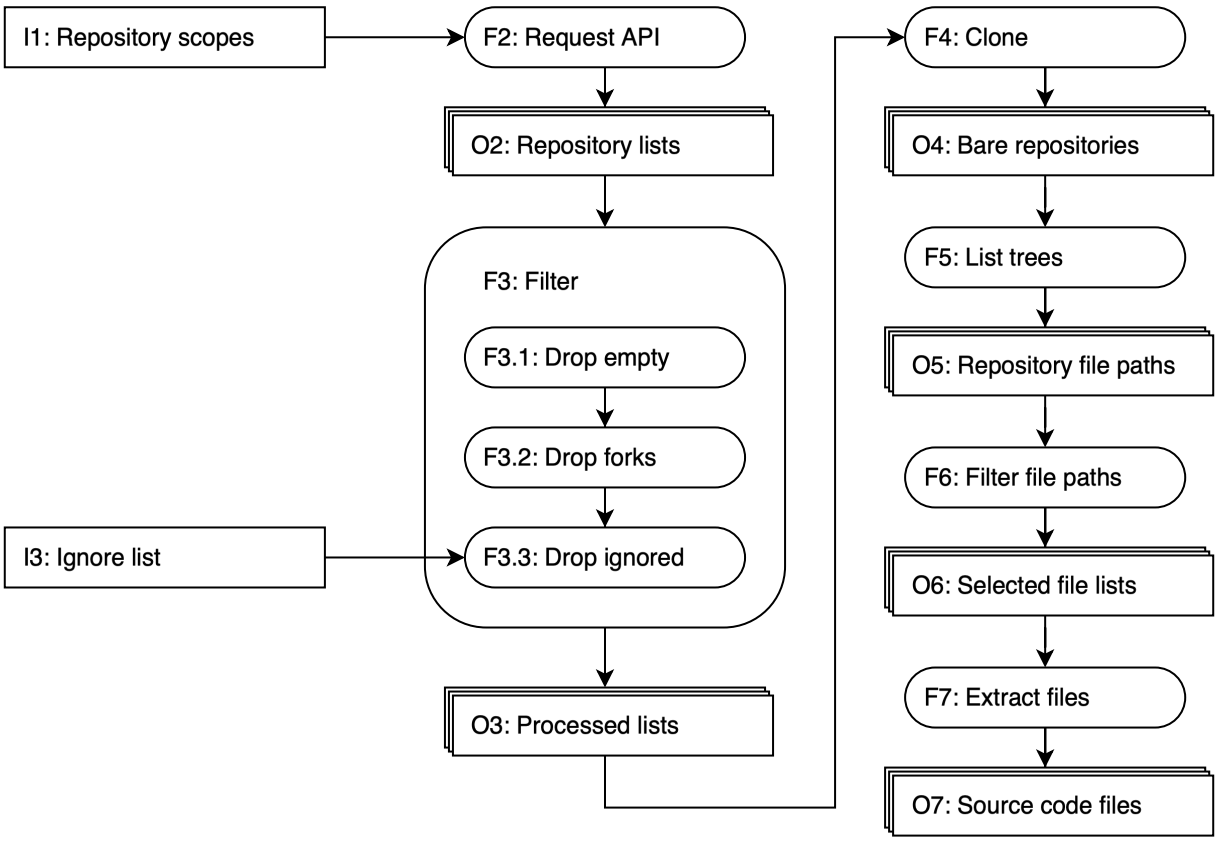

Figure 1: The repository mining approach

The initial user input (*I1*) is a list of repository scopes. The *scope* can be seen as a directory with repositories. On GitHub, public repositories are scoped by their owner — either a user or an organisation. Scopes are fewer in number than repositories, often by one or two levels of magnitude. Providing scopes as input is therefore preferable when the repositories containing relevant files are not known in advance. The process prioritises cloning all repositories within the scope, followed by relevance analysis and cleanup. Being familiar with the Rust programming language, we may know that Mozilla's repositories would be a good place to search for such code.

The first function of the process (*F2*) expands each repository scope into a list of repositories that can be further cloned. Following the analogy, it can be seen as a call of the `ls` command. In practice, this is a request to the repository listing endpoint of the GitHub REST API. As the result, a JSON file with the raw API response is stored (*O2*). In our example, if in the input list *I1* contains `mozilla` GitHub organisation, the API request on *F2* would produce a file `mozilla.json` as the output *O2*.

Repository list filtering (F3) constitutes the central stage of the mining process. At this step, the list of repositories is narrowed down by heuristically excluding irrelevant entries. On the Figure 1, this function is illustrated as a multi-step action producing a single output. In fact, the filter comprises a list of predicates against which each repository is evaluated: if any one holds, the repository is excluded. In our method, three filtering predicates are proposed: *F3.1* whether repository has a size of 0 bytes (i. e., is empty); *F3.2* whether repository is a fork; and *F3.3* whether repository included to the ignore list (*I3*). Zero-sized repositories (*F3.1*) contain no files and are therefore excluded as unworthy of cloning or analysis. Forks (*F3.2*) are likewise omitted. A *fork* is a GitHub mechanism for managing write-access rights in cross-organisation collaboration; it is effectively a duplicate of an existing repository residing under a different scope. Finally, repositories present in the ignore list (*F3.3*) are excluded. This list comprises full repository names that are considered irrelevant. It is initially empty and may be populated incrementally as cloned repositories are found to lack relevant content, such as `*.rs` source files in our example. Additionally, the ignore list may contain repositories that are considered too large and unsuitable for the target dataset. For instance, `mozilla/REALISE-Performance`

occupies 4 GB of disk space but consists predominantly of CSV data files, so could be ignored from mining by its inclusion to *I3*. In our example, the raw repository list `mozilla.json` (*O2*) contains 2465 items. After step *F3.1*, 47 empty repositories are excluded; step *F3.2* excludes 279 more repositories that are forks, and the step *F3.3* excludes the ignored `mozilla/REALISE-Performance`.

After filtering, a condensed list of repositories (*O3*) is constructed and passed for cloning (*F4*). By default, Git clones a repository by creating a work tree and storing repository metadata within a `.git` subdirectory, referred to as the *bare repository*. A bare repository consists of compressed object archives storing commit metadata and file contents. The decompressed work tree can consume substantial disk space; as not all files may be required for a given task, cloning only the bare repository avoids unnecessary extraction and decompression, reducing disk usage and the processing time. This is particularly storage-efficient when the target artefacts constitute a small fraction of the overall repository — for example, shell scripts are typically sparse in Rust code bases. Disk usage can be reduced further via a shallow clone, which omits repository history. For MSR purposes, however, retaining the full history may be beneficial. By this step, we have 2 139 bare repositories cloned within the Mozilla organisation, e. g., `mozilla/servo.git` or `mozilla/rkv.git`.

At step *F5*, the file listing is extracted for each repository and stored alongside it (*O5*). For decompressed work trees, the `find` command would achieve this result; however, `git ls-tree` appears to be at least 8 times more efficient in our experiments, as Git maintains a file list index and no file system traversal is required. *F5* produces 2 139 text files, each corresponding to a cloned repository, and together listing 1 811 668 files.

At step *F6*, the file lists are filtered to retain only files of interest — for example, those matching the `*.rs` pattern. Within Mozilla's repositories, this step reduces the *O5* lists to total 26 760 Rust files. The resulting selection *O6* is then processed for content extraction using `git cat-file`, producing 26 760 Rust files for further research. While collecting Rust from Mozilla's repositories served as an example, the proposed method is suitable for mining source code of any kind available within GitHub repositories.

Conclusions

We presented an approach for mining software repositories based on the UNIX philosophy and built around concepts of Make, where each step performs a single, well-defined action and produces a text-based file output. Such outputs require no specialised software or infrastructure to inspect. This design is particularly suited to scenarios in which the relevant repositories are not known a priori, as the pipeline begins from a high-level entry point and applies progressive refinement and filtering.

Future work may include introducing an automated process for exploring and discovering new GitHub scopes to download repositories from. Another improvement to the current process is adding a scheduled job for refreshing the listing of repositories, cloning and analysing freshly found repositories within the selected scopes. The currently proposed process does not account for ongoing changes to collected repositories; this can be improved by introducing another scheduled job that pulls updates for already included repositories and re-runs analysis on the fresher set of files.

Acknowledgements

Viktor Yakubiv thanks Jason Dusek for outlining the key principles of building reliable and auditable data pipelines.

References

1. Gousios G., Spinellis D. GHTorrent: Github's data from a firehose // Proceedings of the 2012 9th IEEE Working Conference on Mining Software Repositories. Zurich: IEEE, 2012. Pp. 12–21.
2. Gousios G. The GHTorent dataset and tool suite // Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR). San Francisco, CA, USA: IEEE, 2013. Pp. 233–236.
3. Gousios G. et al. Lean GHTorrent: GitHub data on demand // Proceedings of the 11th Working Conference on Mining Software Repositories. Hyderabad, India: ACM, 2014. Pp. 384–387.
4. Cosentino V., Luis J., Cabot J. Findings from GitHub: methods, datasets and limitations // MSR '16: Proceedings of the 13th International Conference on Mining Software Repositories. New York, NY, USA: ACM, 2016. Pp. 137–141.
5. Dabić O., Aghajani E., Bavota G. Sampling projects in GitHub for MSR studies // Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, 2021. Pp. 560–564.
6. Dabić O., Tufano R., Bavota G. SEART Data Hub: streamlining large-scale source code mining and pre-processing // Proceedings of the 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME). Flagstaff, AZ, USA: IEEE, 2024. Pp. 888–892.

Якубів Віктор Романович — аспірант другого року навчання факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, ORCID: 0000-0003-3832-9197, email: yakubiv@knu.ua.

Терлецький Дмитро Олександрович — кандидат фізико-математичних наук, старший науковий співробітник відділу методів та технологічних засобів побудови інтелектуальних програмних систем Інституту кібернетики імені В. М. Глушкова НАН України, ORCID: 0000-0002-7393-1426, email: dmytro.terletskyi@nas.gov.ua.

Науковий керівник: ***Провотар Олександр Іванович*** — доктор фізико-математичних наук, професор, завідувач кафедри інтелектуальних програмних систем факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, ORCID: 0000-0002-6556-3264.

Yakubiv Viktor — 2nd year Ph.D. student, Faculty of Computer Science and Cybernetics, Taras Shevchenko National University of Kyiv, ORCID: 0000-0003-3832-9197, email: yakubiv@knu.ua.

Terletskyi Dmytro — Ph.D. in Computer Science, Senior Research Fellow, Department of Methods and Technological Tools for Constructing of Intelligent Software Systems, V. M. Glushkov Institute of Cybernetics of the NAS of Ukraine, ORCID: 0000-0002-7393-1426, email: dmytro.terletskyi@nas.gov.ua.

Supervisor: ***Provotar Oleksandr*** — Doctor of Physical and Mathematical Science, Professor, Head of the Department of Intelligent Software Systems, Faculty of Computer Science and Cybernetics, Taras Shevchenko National University of Kyiv, ORCID: 0000-0002-6556-3264.