50K-C: A dataset of compilable, and compiled, Java projects

Pedro Martins
University of California, Irvine
pribeiro@uci.edu

Rohan Achar*
University of California, Irvine
rachar@uci.edu

Cristina V. Lopes
University of California, Irvine
lopes@uci.edu

ABSTRACT

We provide a repository of 50,000 compilable Java projects. Each project in this dataset comes with references to all the dependencies required to compile it, the resulting bytecode, and the scripts with which the projects were built.

The dependencies and the build scripts provide a mechanism to re-create compilation of the projects, if needed (to instruct source code for bytecode analysis, for example). The bytecode is ready for testing, execution, and dynamic analysis tools.

CCS CONCEPTS

• **Information systems** → **Information extraction**; • **Software and its engineering** → **Software libraries and repositories**;

KEYWORDS

Large Scale Compilation, Runnable Software Repositories, Software Mining

ACM Reference Format:
Pedro Martins, Rohan Achar, and Cristina V. Lopes. 2018. 50K-C: A dataset of compilable, and compiled, Java projects. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3196398.3196450

1 INTRODUCTION

Large repositories of source code tend to limit their utility to static analysis of the code, as they give no guarantees on whether the projects are compilable, much less runnable in any way. To the best of our knowledge, no public large-scale repository provides guarantees that the projects compile, or run, something very relevant to a wide spectrum of analysis techniques. Those kinds of guarantees can be made on small, manually curated datasets, such as those of the DaCapo [1] and SPEC [10] benchmarks.

The immediate consequence of the lack of large compilable and runnable datasets is that conclusions from research work that require such properties (e.g. [3, 4, 6]) may not generalize beyond the small benchmarks with which they have been evaluated. The scope and impact of these studies would be much greater if they could leverage the vast amounts of open source Java programs available, similarly to what has already been achieved for studies that require vocabulary, software metrics, and static analysis.

We provide a repository of 50,000 compilable (and compiled) `Java` projects taken from GitHub, called 50K-C(ompilable). Each project in this dataset comes with references to all the dependencies required to compile it, the resulting bytecode, and the scripts with which the projects were built. The dependencies and the build scripts provide a mechanism to re-create compilation of the projects, if needed (to instruct source code for bytecode analysis, for example). The bytecode is ready for testing, execution, and dynamic analysis tools.

`Java` is one of the most popular programming languages, and the third most-used programming language in GitHub. [1] The maturity and wide-spread acceptance that `Java` achieved justifies its popularity among the research community, with considerable research focusing on this language. It is therefore a prime candidate for software mining.

All the 50,000 `Java` projects provided come with GitHub origin information, which means that analyses on them can be cross-referenced with other tools operating on GitHub, such as Boa [2] or the map of GitHub clones DéjàVu[7].

On the 50K-C dataset all project dependencies have been resolved, being internal or external to the project. This, in itself, is valuable for many research studies, independent on whether execution is a goal or not. Another advantage is that, even for static analysis, compilation gives solid assurances that the source code is complete and self-contained: having been parsed successfully means that that source code is valid Java code; having been type checked successfully means that all required types, external and internal, are present. For assurances of physical integrity, compilation is the ultimate test.

This paper is organized as follows: immediately following we describe access to information and tools. In Section 2 we describe the methodology used to create the dataset, and in Section 3 the contents of the dataset. In Sections 5 and 4 we describe the static and the dynamic contents of the dataset, respectively. In Section 6 we conclude.

Repositories and Tools Access. The repository and the tools described in this dataset can be found in http://mondego.ics.uci.edu/projects/jbf. In detail the reader will find:

- A repository of 50K `Java` projects as source code.

*The secretary disavows any knowledge of this author's actions.

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196450

[1] https://octoverse.github.com, Dec. 2017

- A repository of bytecode generated from building the 50K projects.
- A repository of `JAR` files required to build the 50K projects.
- A mapping between each 50K project and the GitHub URL it was obtained from.
- The tools used to create the dataset (gather required dependencies and build the projects).
- A Virtual Machine with a tutorial showing the used task of building the 50K projects.

2 METHODOLOGY

The projects were downloaded using the GHTorrent database and network[2] [5] which contains meta-data from GitHub (number of stars, commits, commiters, whether the projects are forks, main programming language, date of creation, etc.) as well as the download links.[3]

We filtered the projects that represent software for the Android ecosystem, since these have special requirements (we would need, for example, to have different versions of the Android JDK.). We did so by searching inside the projects for a `AndroidManifest.xml` file, since according the respective documentation, these files are mandatory for every Android application.[4]

A total of 479,113 `Java` projects were downloaded, an attempt was made to build all of them, and from the successes 50,000 projects were randomly chosen, representing 50K-C. Care was taken to make sure no project clones exist in 50K-C, using the mapping of source code clones DéJàVu [7] and the clone detection tool SourcererCC [9][5].

The build task is split into two main tasks, namely management of dependencies and project compilation, and each of these is further split into sub-tasks. Starting with dependency management, the first task on the pipeline consists of collecting JAR files from the projects. After that, we created an inverted index of Fully Qualified Names (FQN) of Java classes and interfaces available in a JAR. This is done before any compilation takes place, and is used to solve dependencies.

The compilation process is done in up to two rounds for each project. In a first round, an attempt is made to compile it without specifying any external dependencies, just relying on the Java standard libraries. The ones that succeed are marked as so; the ones that fail go through a second round of processing. Compilation errors are detected, and split into the ones related to encoding problems and to missing external dependencies. Both are solved through automatically inserting specific directives in our compilation scripts. For resolving external dependencies, we use the index created in the initial phase. A second round of compilation starts whose results are final: the projects that compile are marked as successes, the failures as failures, and the results of this round together with the results of the first round determine the overall effectiveness of the strategy.

3 DESCRIPTION

The dataset is composed by 3 main smaller sets that require each other for structural integrity.

The first dataset contains all the `JAR` files that were required by the projects as external dependencies for successful builds. The relative `JAR` paths from this dataset (they are organized in subfolders) are used as the index of each respective `JAR` file.

The second dataset contains all the 50,000 projects, in the form of zip files, as if they were extracted directly from GitHub. This is, by size, the largest of the three subsets. Similarly to the `JAR` files, the relative project paths from this dataset (they also are organized in subfolders) are used as the index of each respective project.

The main dataset contains the resulting project classes and builds meta-data, respectively, under the same relative path. For example, the result of building `projects/2/mvmn-Thue-in-java.zip` is in `builds/2/mvmn-Thue-in-java`.

Per project, the following information is provided:

- A folder `build/` with the produced class files,
- The command used to build the project, under `build.command`,
- The exact scripts with which the project was built, on a folder `custom_build_scripts/`,
- And meta information about the building process, saved on the `json` file `build-result.json`. This file is not explained thoroughly[6], but of relevance are the fields:

```
"depends": [
 [
      null,
      null,
      null,
      false,
      "project21/n4upgrade/n4upgrade_0.jar",
      true
 ]
],
 "file": "100/emagnus-ulurulib",
```

The field `depends` contains a list of dependencies per project. Some of these dependencies have as last argument `true`, which means the jar file (`n4upgrade_0.jar` in this case) was obtained inside the projects. A value of `false` would

[2] http://ghtorrent.org

[3] While GHTorrent enabled us to crawl large amounts of repositories, we have found its database to have some errors. Specifically, a residual number of duplicated entries (i.e. projects had the same URL); only the youngest of these was kept for the analysis. We filed an issue report at (URL omitted for anonymity).

[4] https://developer.android.com/guide/topics/manifest/manifest-intro.html, Jan 2018.

[5] More information on http://mondego.ics.uci.edu/projects/dejavu.

[6] For space reasons it is not possible to detail all the fields of this file, but they are detailed in the project website and in the dataset documentation.

Table 1: Project distribution of mean measurements on the respective files.

	Mean	Std.	25%	50%	75%	Max
Mean File Bytes	3,139	4,445	1,327	2,199	3,719	401,232
Mean File Lines	103	117	51	79	124	10,162
Mean File LOC	87	105	41	65	105	9,729
Mean File SLOC	70	83	35	53	84	6,962
Total Files	24	63	7	12	23	5,549

mean otherwise, and all these external dependencies are provided with the dataset.[7]

The field `file` provides the relative path, on the dataset, for the project.

Despite providing an individual `json` file, like the one seen above, for each project, we also provide a large `json` file containing all the meta data for 50K-C. Finally, a mapping between each project and its GitHub URL is also provided.

4 STATIC SOURCE CODE ANALYZES

In this section we provide statistical analyses of static characteristics of the `Java` source code. These include relevant characteristics that help understand the dataset.

We start in Table 1 by showing the distribution of various characteristics of the projects, namely mean values for the size of their files, in bytes, lines, LOC (Lines Of Code) and SLOC (Source Lines of Code, LOC without comments), and the number of files. Note a skewed distribution towards smaller projects, which is unavoidable since it is known to be a characteristic of source code (for `Java` see, for example, the work of Lopes *et al.* [8]). Also note that despite the distribution of project sizes towards small, there are nevertheless cases on both ends of the spectrum: the maximum project size has more than 5,000 files. We hope this variety covers a large size of research domains.

On Figure 1 we can see all the possible distributions of the values described in Table 1. File size indicators are correlated, and note the behavior of `Total Files`, which is never correlated with the other indicators. This means that when subsampling this dataset for larger projects, this will be made in detriment of variability of file size, and otherwise.

5 DYNAMIC CODE ANALYZES

In this section we continue with statistical descriptions of the dataset, but this time on its runnable components.

We start with Table 2, which shows reachable `main` methods and the presence of `junit` on the class files generated for each method. This numbers were obtained by searching for methods names and import names, respectively.

A substantial number of projects has `main` methods, but the presence of `junit` is relatively small. Note however that all the projects have resulting class files, here we are just giving emphasis to the standard entry point in `Java` projects,

[7]The presence of some fields as `null` is justified by their need for tool-specific functions like resuming build processes or managing fault recovery. It is kept here to replicate what the tool generates and avoid confusion on tool usage, despite some loss of clarity.

Table 2: Project distribution of reachable `main` methods and presence of `junit` on the respective files.

	Mean	Std.	25%	50%	75%	Max
Reachable `main`	3	13	0	1	2	770
With `junit`	1	5	0	0	0	374

and to the standard test framework. This suggests that observations and analysis of `junit` might require specially curated datasets, as evidence here suggests that randomly sampled corpus will have a poor presence of unit testing. This is outside the scope of this work.

In Figure 2 we see the correlations between the presence of `main` methods and `junit`, and we had a third factor, the number of class files per project. The results show that none of these variables are correlated, which indicates that their presence might be hard to capture by sub-sampling 50K-C.

6 CONCLUSION

We created 50K-C, a dataset of Java programs together with all their dependencies, with individual building scripts and with the class files generated by them.

We believe this dataset will contribute to software research by its novelty, since as far as the authors awareness is correct there is no comparable corpus, in `Java` or in any other language.

By providing different elements of successful compilations: build instructions, required dependencies, source code, generated class files, mappings to GitHub, we hope to provide to the research community a tool and a corpus of knowledge that will leverage and accelerate research projects.

It is also hoped that this dataset provides a useful tool for research that does not focus exclusively on the facets of building and running source code. By providing source code and bytecode we are providing a mapping between static source code and the runnable components that it generates inviting, for example, techniques of machine learning for bidirectional transformations. This dataset can also be useful for research teams that are interested only on static code but, for which, have strong demands of quality and structural integrity, which we can assure by guaranteeing the source code compiles.

IMPORTANT DISCLOSURE

Please be aware that no special care was taken to analyze the source code towards security threats. This means that

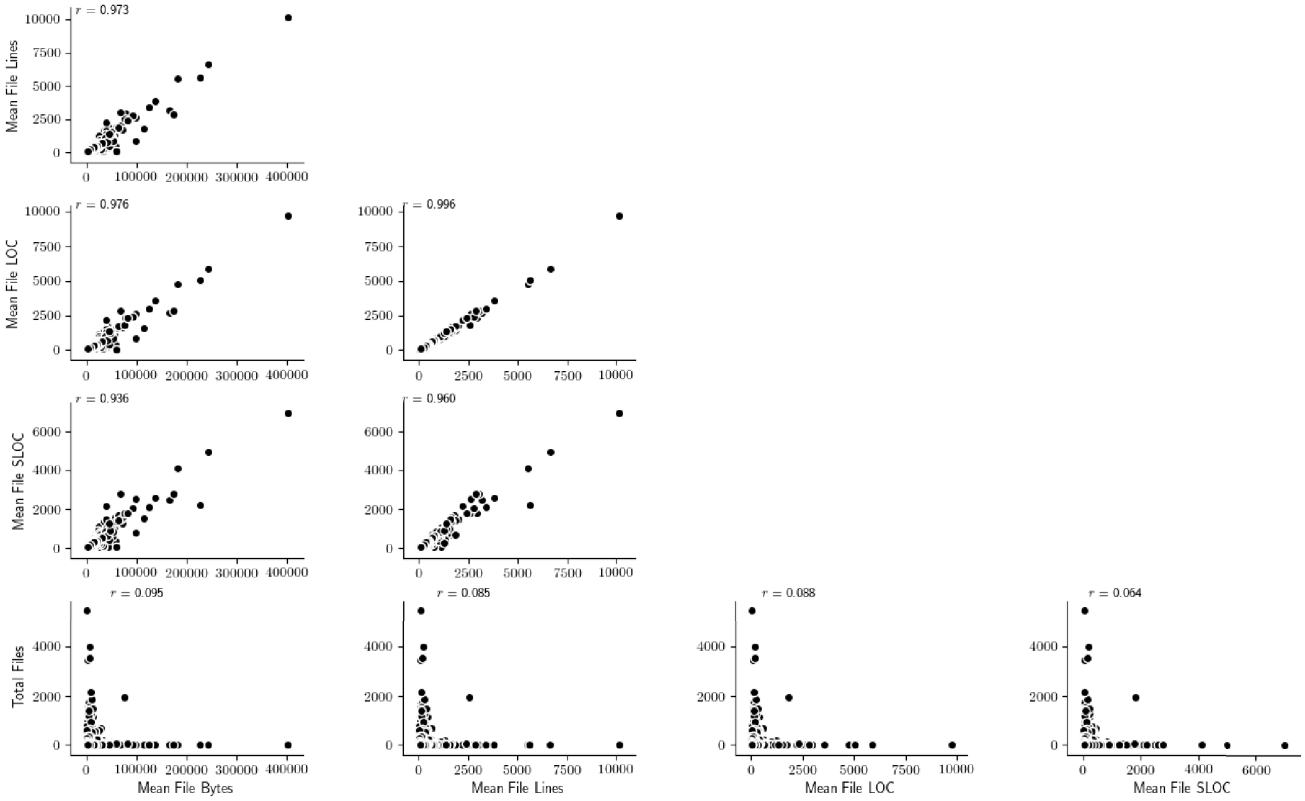

Figure 1: Correlations between all the measurements of Table 1. The value r corresponds to the Spearman's rank correlation coefficient.

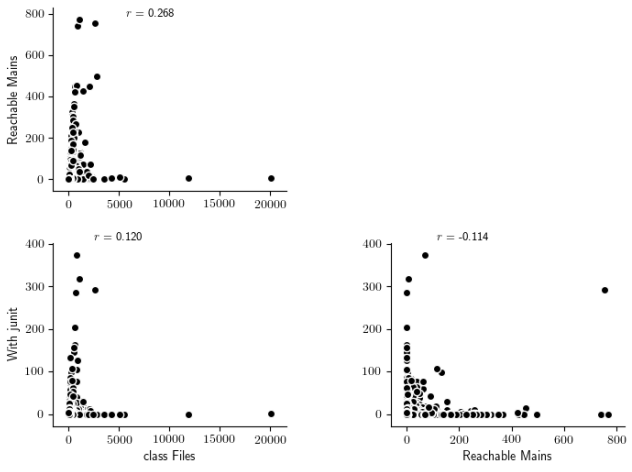

Figure 2: Correlations between all the measurements of Table 2, with the addition of the number of class files per project. The value r corresponds to the Spearman's rank correlation coefficient.

when you run the executable class files or the building scripts that come with the projects you are running unknown code. Act on your own discretion and be careful. At the very least, **sandboxing through system file permissions and limited network access is highly recommended**.

REFERENCES

[1] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[2] 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 *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 422–431. http://dl.acm.org/citation.cfm?id=2486788.2486844

[3] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly Detecting Relevant Program Invariants. In *Proceedings of the 22Nd International Conference on Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 449–458. https://doi.org/10.1145/337180.337240

[4] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[5] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[6] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 110–120. https://doi.org/10.1145/1542476.1542489

[7] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéJàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 84 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133908

[8] Cristina V. Lopes and Joel Ossher. 2015. How Scale Affects Structure in Java Programs. *SIGPLAN Not.* 50, 10 (Oct. 2015), 675–694. https://doi.org/10.1145/2858965.2814300

[9] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[10] SPEC [n. d.]. The SPEC Benchmarks. Standard Performance Evaluation Corporation. ([n. d.]). https://www.spec.org/benchmarks.html