DéjàVu: A Map of Code Duplicates on GitHub

CRISTINA V. LOPES, University of California, Irvine
PETR MAJ, Czech Technical University
PEDRO MARTINS, University of California, Irvine
VAIBHAV SAINI, University of California, Irvine
DI YANG, University of California, Irvine
JAKUB ZITNY, Czech Technical University
HITESH SAJNANI, Microsoft Research
JAN VITEK, Northeastern University

Previous studies have shown that there is a non-trivial amount of duplication in source code. This paper analyzes a corpus of 4.5 million non-fork projects hosted on GitHub representing over 428 million files written in Java, C++, Python, and JavaScript. We found that this corpus has a mere 85 million unique files. In other words, 70% of the code on GitHub consists of clones of previously created files. There is considerable variation between language ecosystems. JavaScript has the highest rate of file duplication, only 6% of the files are distinct. Java, on the other hand, has the least duplication, 60% of files are distinct. Lastly, a project-level analysis shows that between 9% and 31% of the projects contain at least 80% of files that can be found elsewhere. These rates of duplication have implications for systems built on open source software as well as for researchers interested in analyzing large code bases. As a concrete artifact of this study, we have created DéjàVu, a publicly available map of code duplicates in GitHub repositories.

CCS Concepts: • **Information systems → Near-duplicate and plagiarism detection**; • **Software and its engineering → Ultra-large-scale systems**;

ACM Reference format:
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 (October 2017), 27 pages.
https://doi.org/10.1145/3133908

1 INTRODUCTION

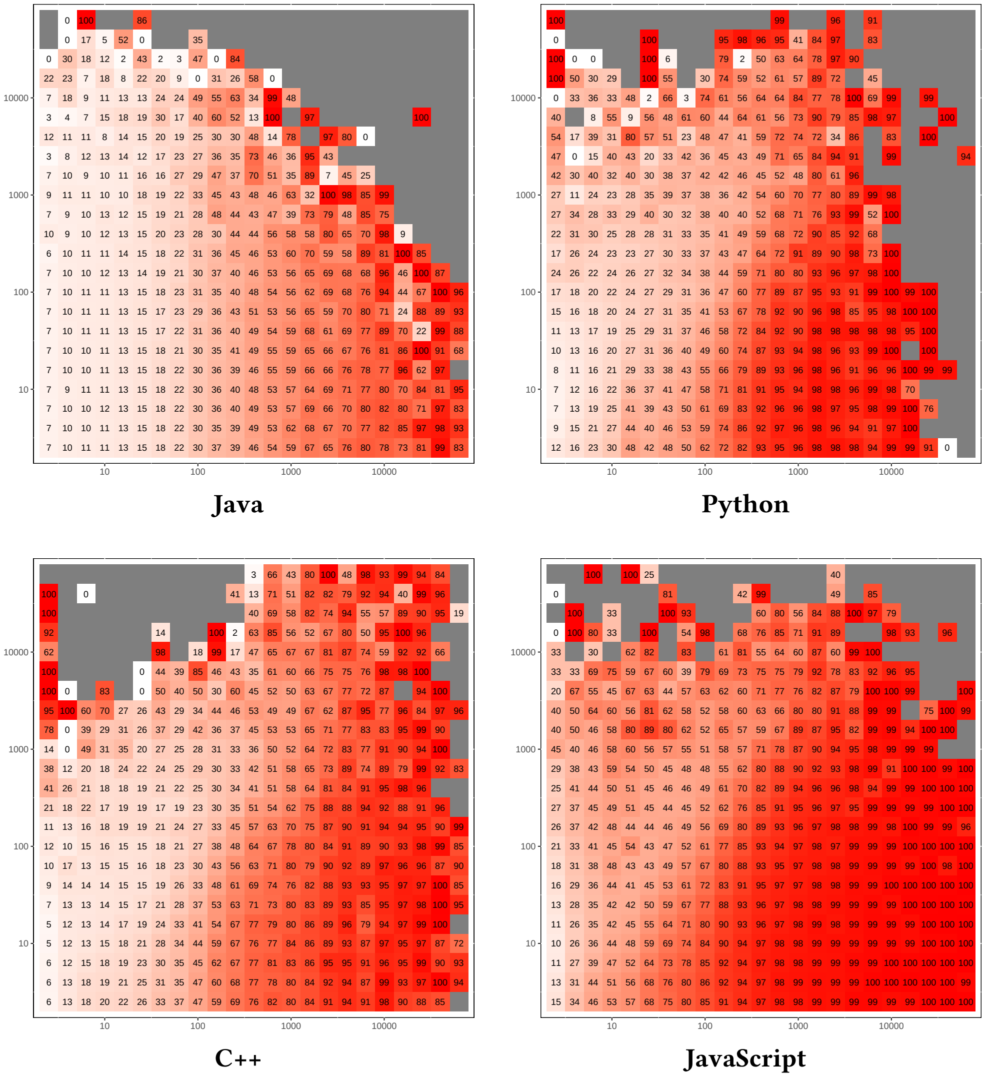
The advent of web-hosted open source repository services such as GitHub, BitBucket and SourceForge have transformed how source code is shared. Creating a project takes almost no effort and is free of cost for small teams working in the open. Over the last two decades, millions of projects have been shared, building up a massive trove of free software. A number of these projects have been widely adopted and are part of our daily software infrastructure. More recently there have been attempts to treat the open source ecosystem as a massive dataset and to mine it in the hopes of finding patterns of interest.

When working with software, one may want to make statements about applicability of, say, a compiler optimization or a static bug finding technique. Intuitively, one would expect that a conclusion based on a software

2475-1421/2017/10-ART84
https://doi.org/10.1145/3133908

corpus made up of thousands of programs randomly extracted from an Internet archive is more likely to hold than one based on a handful of hand-picked benchmarks such as [Blackburn et al. 2006] or [SPEC 1998]. For an example, consider [Richards et al. 2011] which demonstrated that the design of the Mozilla optimizing compiler was skewed by the lack of representative benchmarks. Looking at small workloads gave a very different picture from what could be gleaned by downloading thousands of websites.

Scaling to large datasets has its challenges. Whereas small datasets can be curated with care, larger code bases are often obtained by random selection. If GitHub has over 4.5 million projects, how does one pick a thousand projects? If statistical reasoning is to be applied, the projects must be independent. Independence of observations



Java

Python

C++

JavaScript

Fig. 1. Map of code duplication. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for all projects in the tile. Darker means more clones.

is taken for granted in many settings, but with software there are many ways one project can influence another. Influences can originate from the developers on the team, for instance the same people will tend to write similar code. Even more common are the various means of software reuse. Projects can include other projects. Apache Commons is used in thousands of projects, Oracle's SDK is universally used by any Java project, JQuery by most websites. StackOverflow and other discussion forums encourage the sharing of code snippets. Cut and paste programming where code is lifted from one project and dropped into another is another way to inject dependencies. Lastly, entire files can be copied from one project to the next. Any of these actions, at scale, may bias results of research.

Several published studies either neglected to account for duplicates, or addressed them before analysis. [Casalnuovo et al. 2015] studied the use of assertions in the top 100 most popular C and C++ projects in GitHub. [Ray et al. 2014] studied software quality using the top 50 most popular projects in 17 languages. Neither addressed file duplication. Conversely, [Hoffa 2016] studied the old "tabs v. spaces" issue in 400K GitHub projects; file duplication was identified as an issue and eliminated before analysis. [Cosentino et al. 2016] present a meta-analysis of studies on GitHub projects where trends and problems related to dataset selection are identified.

This paper provides a tool to assist selecting projects from GitHub. DéjàVu is a publicly available index of file-level code duplication. The novelty of our work lies partly in its scale; it is an index of duplication for the entire GitHub repository for four popular languages, Java, C++, Python and JavaScript. Figure 1 illustrates the proportion of duplicated files for different project sizes and numbers of commits (section 5 explains how these heatmaps were generated). The heatmaps show that as project size increases the proportion of duplicated files also increases. Projects with more commits tend to have fewer project-level clones. Finally JavaScript projects have the most project-level clones, while Java projects have the fewest.

The clone map from which the heatmaps were produced is our main contribution. It can be used to understand the similarity relations in samples of projects or to curate samples to reduce duplicates. Consider for instance a subset that focuses on the most active projects, as done in [Borges et al. 2016], by filtering on the number of stars or commits a project has. For example, the clones for the 10K most popular projects are summarized in Figure 2. In Java, this filter is reasonably efficient at reducing the number of clones. In other languages clones remain prevalent. DéjàVu can be used to curate datasets, i.e. remove projects with too many clones. Besides applicability to research, our results can be used by anyone who needs to host large amounts of source code to avoid storing duplicate files. Our clone map can also be used to improve tooling, e.g. being queried when new files are added to projects to filter duplicates.

Fig. 2. File-hash duplication in subsets.

	10K Stars	10K Commits
Java	9%	6%
C/C++	41%	51%
Python	28%	44%
JavaScript	44%	66%

At the outset of this work, we were planning to study different granularities of duplication. As the results came in, the staggering rate of file-level duplication drove us to select three simple levels of similarity. A *file hash* gives a measure of file that are copied across projects without changes. A *token hash* captures minor changes in spaces, comments and ordering. Lastly, SourcererCC captures files with 80% token-similarity. This gives an idea of how many files have been edited after cloning. Our choice of languages was driven by the popularity of these languages, and by the fact that two are statically typed and two have no type annotations. This can conceivably lead to differences in the way code is reused. We expected to answer the following questions: How much code cloning is there, how does cloning affect datasets of software written in different languages, and through which processes does duplication come about? This paper describes our methodology, details the corpus that we have selected and gives our answers to these questions. Along with the quantitative analysis, we provide a qualitative analysis of duplicates on a small number of examples.

Artifacts. The lists of clones, code for gathering data, computing clones, data analysis and visualization are at: http://mondego.ics.uci.edu/projects/dejavu. Processing was done on a Dell PowerEdge R830 with 56 cores (112 threads) and 256G of RAM. The data took 2 months to download and 6 weeks to process.

2 RELATED WORK

Code clone detection techniques have been documented in the literature since the early 90s. Readers interested in a survey of the early work are referred to [Koschke 2007; Roy and Cordy 2007]. There are also benchmarks for assessing the performance of tools [Roy and Cordy 2009; Svajlenko and Roy 2015]. The pipeline we used includes SourcererCC, a token-based code clone detection tool that is freely available and has been compared to other similar tools using those benchmarks [Sajnani 2016; Sajnani et al. 2016].[1] SourcererCC is the most scalable tool so far for detecting Type 3 clones. Type 3 clones are syntactically similar code fragments that differ at the statement level. The fragments have statements added/modified/removed with respect to each other.

One of the earliest studies of inter-project cloning, [Kamiya et al. 2002] analyzed clones across three different operating systems. They found evidence of about 20% cloning between FreeBSD and NetBSD and less than 1% between Linux and FreeBSD or NetBSD. This is explained by the fact that Linux originated and grew independently. [Mockus 2007] performed an analysis of popular open source projects, including several versions of Unix and several popular packages; 38K projects and 5M files. The concept of duplication there was simply based on file names. Approximately half of the file names were used in more than one project. Furthermore, the study also tried to identify components that were duplicated among projects by detecting directories that share a large fraction of their files. Both [Mockus 2007] and [Mockus 2009] use only a fraction of our dataset and a single similarity metric, as opposed to the 3 metrics we provide.

A few studies have focused on block-level cloning, i.e. portions of code smaller than entire files. [Roy and Cordy 2010] analyzed clones in twenty open source C, Java and C# systems. They found 15% of the C files, 46% of the Java files, and 29% of C# files are associated with exact block-level clones. Java had a higher percentage of clones because of accessors methods in Swing. [Heinemann et al. 2011] computed block-level clones consisting of at least 15 statements between 22 commonly reused Java frameworks consisting of more than 6 MLOC and 20 open source Java projects. They did not find any clones for 11 projects. For 5 projects, they found cloning to be below 1% and for the remaining 4, they found up to 10% cloning. These two studies give conflicting accounts of block-level code duplication.

Closer to our study, an analysis of file-level code cloning on Java projects is presented by [Ossher et al. 2011]. This work, analyzed 13K Java projects with close to 2M files. The authors created a system that merges various clone detection techniques with various degrees of confidence, starting on the highest: MD5 hashes; name equivalence through Java's full-qualified names. They report 5.2% file-hash duplication, considerably lower than what we found. Our corpus is three orders of magnitude larger than Ossher's. Furthermore, intra-project duplication meant to deal with versioning was excluded. They looked at subversion, which may have different practices than git, especially related to versioning. We speculate that the practice of copying source code files in open source has become more pervasive since that study was made, and that sites like GitHub simplify copying files among projects, but we haven't reanalyzed the dataset as it is not relevant to the DéjàVu map.

Over the past few years, open source repositories have turned out to be useful to validate beliefs about software development and software engineering in general. The richness of the data and the potential insights that it represents has created an entire community of researchers. [Kochhar et al. 2013] used 50K GitHub repositories to investigate the correlation between the presence of test cases and various project development characteristics, including the lines of code and the size of development teams. They removed toy projects and included famous projects such as JQuery and Rails in their dataset. [Vendome et al. 2016] study how licensing usage and adoption changes over a period of time on 51K repositories. They choose repositories that (i) were not forks; and (ii) had at least one star. [Borges et al. 2016] analyze 2.5K repositories to investigate the factors that impact their popularity, including the identification of the major patterns that can be used to describe popularity trends.

[1] http://github.com/Mondego/SourcererCC

The software engineering research community is increasingly examining large number of projects to test hypotheses or derive new knowledge about the software development process. However, as [Nagappan et al. 2013] point out, more is not necessarily better, and selection of projects plays an important role – more so now than ever, since anyone can create a repository for any purpose at no cost. Thus, the quality of data gathered from these software repositories might be questionable. For example, as we also found out, repositories often contain school assignments, copies of other repositories, images and text files without any source code. [Kalliamvakou et al. 2014] manually analyzed a sample of 434 GitHub repositories and found that approximately 37% of them were not used for software development. As a result, researchers have spent significant effort into collecting, curating, and analyzing data from open source projects around the world. Flossmetrics [Gonzalez-Barahona et al. 2010] and Sourcerer [Ossher et al. 2009] collect data and provide statistics. [Dyer et al. 2013] have curated a large number of Java repositories and provide a domain specific language to help researchers mine data about software repositories. Similarly [Bissyande et al. 2013] have created Orion, a prototype for enabling unified search to retrieve projects using complex search queries linking different artifacts of software development, such as source code, version control metadata, bug tracker tickets, developer activities and interactions extracted from hosting platform. Black Duck Open Hub (www.openhub.net) is a public directory of free and open source software, offering analytics and search services for discovering, evaluating, tracking, and comparing open source code and projects. It analyzes both the code's history and ongoing updates to provide reports about the composition and activity of project code bases. These platforms are useful for researchers to filter out repositories that are interesting to study a given phenomenon by providing various filters. While these filters are useful to validate the integrity of the data to some extent, certain subtle factors when unaccounted for can heavily impact the validity of the study. Code duplication is one such factor. For example, if the dataset consists of projects that have hundreds and thousands of duplicate projects that are part of the same dataset, the overall lack of diversity in the dataset might lead to incorrect observations, as pointed out by [Nagappan et al. 2013].

3 ANALYSIS PIPELINE

Our analysis pipeline is outlined in Figure 3. The pipeline starts with local copies of the projects that constitute our corpus. From here, code files are scanned for fact extraction and tokenization. Two of the facts are the hashes of the files and the hashes of the tokens of the files. File hashes identify exact duplicates; token hashes allow catch clones up with minor differences. While permutations of same tokens may have the same hash, they are unlikely. Clones are dominated by exact copies, and we did not observe any such collision in randomly sampled pairs. Files with distinct token hashes are used as input to the near-miss clone detection tool, SourcererCC.

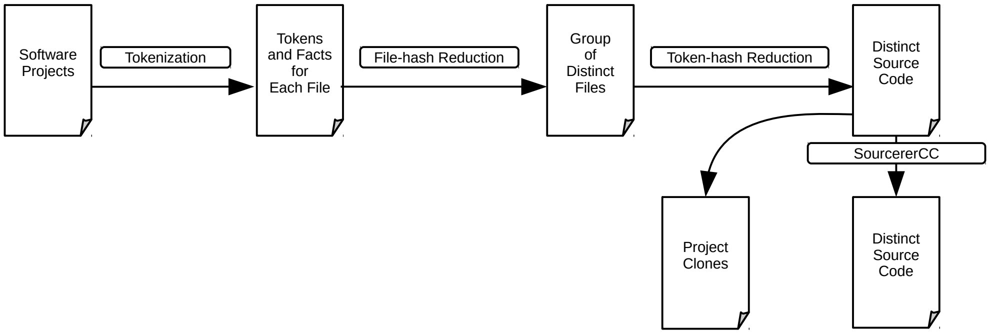


Fig. 3. Analysis pipeline.

While our JavaScript pipeline was developed independently, data formats, database schema and analysis scripts are identical.

3.1 Tokenization

Tokenization transforms a file into a "bag of words," where occurrences of each word are recorded. Consider, for instance, the Java program:

```
package foo;
public class Foo {  // Example Class
  private int x;
  public Foo(int x) { this.x = x; }
  private void print() { System.out.println("Number: " + x)  }
  public static void main() {  new FooNumber(4).print();  } }
```

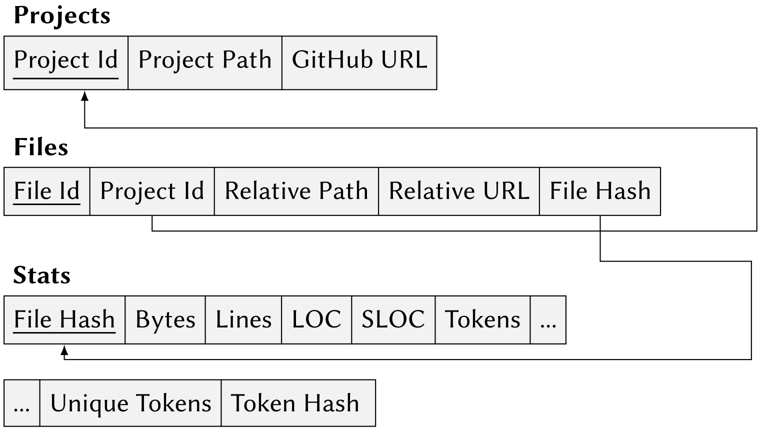
Tokenization removes comments, white space, and terminals. Tokens are grouped by frequency, generating:

```
Java Foo:[(package,1),(foo,1),(public,3),(class,1),(Foo,2),(private,2),(int,2),(x,5),
(this,1),(void,2),(print,2),(System,1),(out,1),(println,1),(Number,1),(static,1),
(main,1),(new,1),(FooNumber,1),(4,1)]
```

The tokens package and `foo` appear once, `public` appears three times, etc. The order is not relevant. During tokenization we also extract additional information: (1) *file hash* – the MD5 hash of the entire string that composes the input file; (2) *token hash* – the MD5 hash of the string that constitutes the tokenized output; (3) *size* in bytes; (4) *number of lines*; (5) *number of lines of code* without blanks; (6) *number of lines of source* without comments; (7) *number of tokens*; and (8) *number of unique tokens.* The tokenized input is used both to build a relational database and as input to SourcererCC. The use of MD5 (or any hashing algorithm) runs the risk of collisions, given the size of our data they are unlikely to skew the results.

3.2 Database

The data extracted by the tokenizer is imported into a MySQL database. The table `Projects` contains a list of projects, with a unique identifier, a path in our local corpus and the project's URL. `Files` contains a unique id for a file, the id of the project the file came from, the relative paths and URLs of the file and the file hash. The statistics for each file are stored in the table `Stats`, which contains the information extracted by the tokenizer. The tokens themselves are not imported. The `Stats` table has the file hash as unique key. With this, we get an immediate reduction from files to hash-distinct files. Two files with distinct file hashes may produce the exact same tokens, and, therefore the same token hash. This could happen when the code of one file is a permutation of another. The converse does not hold: files with distinct token hashes must have come from files with distinct file hashes. For source code analysis, file hashes are not necessarily the best indicators of code duplication; token hashes are more robust to small perturbations. We use primarily token hashes in our analysis.



3.3 Project-level analysis

Besides file-level analysis, we also look for projects with significant overlap with other projects. This is done with a script that queries the database making an intersection of the project files' distinct token hashes. This script produces pairs of projects that have significant overlap in at least one direction. The results are of the form: *A cloned in B at x%, B cloned in A at y%*, where x% of project A's files (in tokenized form) are found also in project B, and y% of project B's files (in tokenized form) are found in project A. Calculating project-level information is done in two steps. First, collect all the files from a project A, say, for example there are 4 files in A: Then find the token-hash duplicates for each of these files in other projects. It might be something like:

```
project A
  File$_1$ - B, B, C
  File$_2$ - B
  File$_3$ -
  File$_4$ - B, D, F
```

There are 3 files from A with duplicates in B, making A a clone of B at 75%. Conversely, there are 4 files in B with duplicates in A; assuming B has a total of 20 files, then B is cloned in A at 20%. A file can be in other project multiple times (e.g. in different directories) as is File 1.

3.4 SourcererCC

The concept of inexact code similarity has been studied in the code cloning literature. Blocks of code that are similar are called near-miss clones, or near-duplication [Cordy et al. 2004]. SourcererCC estimates the amount of near-duplication in GitHub with a "bag of words" model for source code rather than more sophisticated structure-aware clone detection methods. It has been shown to have good precision and recall, comparable to more sophisticated tools [Sajnani 2016]. Its input consists of non-empty files with distinct token hashes. SourcererCC finds clone pairs between these files at a given level of similarity. We have selected 80% similarity as this has given good empirical results. Ideally one could imagine varying the level of similarity and reporting a range of results. But this would be computationally expensive and, given the relatively low numbers of near-miss clones, would not affect our results.

4 CORPUS

The GitHub projects were downloaded using the GHTorrent database and network [Gousios 2013] which contains meta-data such as number of stars, commits, committers, whether projects are forks, main programming language, date of creation, etc., as well as download links. While convenient, GHTorrent has errors: 1.6% of the projects were replicated entries with the same URL; only the youngest of these was kept for the analysis.

Table 1 gives the size of the different language corpora. We skipped forked projects as forks contain a large amount of code from the original projects, retaining those would skew our findings. Downloading the projects was the most time-consuming step. The order of downloads followed the GHTorrent projects table, which seems to be roughly chronological. Some of the URLs failed to produce valid content. This happened in two cases: when the projects had been deleted, or marked private, and when development for the project happens in branches other than master. Thus, the number of downloaded projects was smaller than the number of URLs in GHTorrent. For each language, the files analyzed were files whose extensions represent source code in the target languages. For Java: `.java`; for Python: `.py`; for JavaScript: `.js`, for C/C++: `.cpp .hpp .HPP .c .h .C .cc .CPP .c++` and `.cp`. Some projects did not have any source code with the expected extension, they were excluded.

The medians in Table 1 give additional properties of the corpus, namely the number of files per (non-empty) project, the number of Source Lines of Code (SLOC) per file, the number of stars and the number of commits of the projects. In terms of files per project, Python and JavaScript projects tend to be smaller than Java and

Table 1. GitHub Corpus.

		Java	C++	Python	JavaScript
Counts	# projects (total)	3,506,219	1,130,879	2,340,845	4,479,173
	# projects (non-fork)	1,859,001	554,008	1,096,246	2,011,875
	# projects (downloaded)	1,481,468	369,440	909,290	1,778,679
	# projects (analyzed)	1,481,468	364,155	893,197	1,755,618
	# files (analyzed)	72,880,615	61,647,575	31,602,780	261,676,091
Medians	Files/project	9 (σ = 600)	11 (σ = 1304)	4 (σ = 501)	6 (σ = 1335)
	SLOC/file	41 (σ = 552)	55 (σ = 2019)	46 (σ = 2196)	28 (σ = 2736)
	Stars/project	0 (σ = 71)	0 (σ = 119)	0 (σ = 99)	0 (σ = 324)
	Commits/project	4 (σ = 336)	6 (σ = 1493)	6 (σ = 542)	6 (σ = 275)

C++ projects. C++ files are considerably larger than any others, and JavaScript files are considerably smaller. None of these numbers is surprising. They all confirm the general impression that a large number of projects hosted in GitHub are small, not very active, and not very popular. Figures 4 and 5 illustrate the basic size-related properties of the projects we analyzed, namely the distribution of files per project and the distribution of Source Lines of Code (SLOC) per file. For JavaScript we give data with and without NPM (it is a cause of a large number of clones). Without NPM means that we ignored files downloaded by the Node Package Manager.

5 QUANTITATIVE ANALYSIS

We present analyses of the data at two levels of detail: file and project level. This section focuses exclusively on quantitative analysis; the next section delves deeper into qualitative observations.

5.1 File-Level Analysis

Table 2 shows a summary of the findings for files. "SCC dup files" is the number of files, out of the distinct token-hash files, that SourcererCC has identified as clones; similarly, "SCC unique files" is the number of files for which no clones were detected. Figure 6 (top row) charts the numbers in Table 2. The duplicated files (dark grey) are the files that are duplicate of at least one of the distinct token-hash files (light grey); further, the distinct token-hash files are split between those for which SourcererCC found at least one similar file (cloned files, grey) and those for which SourcererCC did not find any similar file (unique files, in white).

These numbers show a considerable amount of code duplication, both exact copies of the files (file hashes), exact copies of the files' tokens (token hashes), and near-duplicates of files (SourcererCC). The amount of duplication varies with the language: the JavaScript ecosystem contains the largest amount of duplication, with 94% of files being file-hash clones of the other 6%; the Java ecosystem contains the smallest amount, but even for Java, 40% of the files are duplicates; the C++ and Python ecosystems have 73% and 71% copies, respectively. As for near-duplicates, Java contains the largest percentage: 46% of the files are near-duplicate clones. The ratio of near-miss clones is 43% for Java, 39% for JavaScript, and 32% for Python.

The heatmaps (Figure 1) shown in the beginning of the paper were produced using the number of commits shown in Table 1, the number of files in each project, and the file hashes. The heat intensity corresponds to the ratio of file hashes clones over total files for each cell.

Duplication can come in many flavors. Specifically, it could be evenly or unevenly distributed among all token hashes. We found these distributions to be highly skewed towards small groups of files. In Java 1.5M groups of files with the same token-hash have either 2 or 3 files in them; the number of token hash-equal groups with more than 100 files is minuscule. The same observation holds for the other languages. Another interesting piece

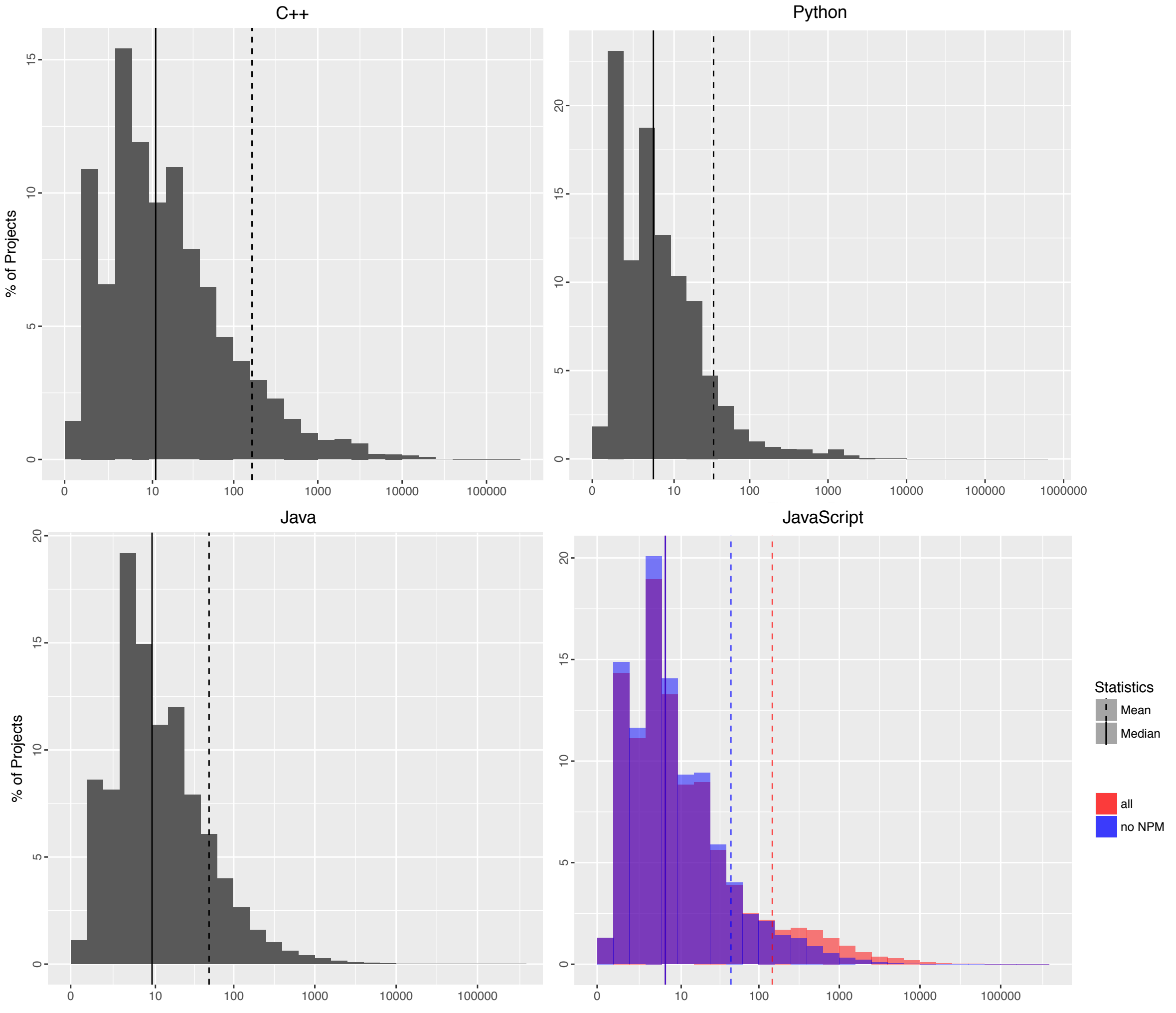


Fig. 4. Files per project.

Table 2. File-Level Duplication.

	Java	C++	Python	JavaScript
Total files	72,880,615	61,647,575	31,602,780	261,676,091
File hashes	43,713,084 (60%)	16,384,801 (27%)	9,157,622 (29%)	15,611,029 (6%)
Token hashes	40,786,858 (56%)	14,425,319 (23%)	8,620,326 (27%)	13,587,850 (5%)
SCC dup files	18,701,593 (26%)	6,200,301 (10%)	2,732,747 (9%)	5,245,470 (2%)
SCC unique files	22,085,265 (30%)	8,225,018 (13%)	5,887,579 (19%)	8,342,380 (3%)

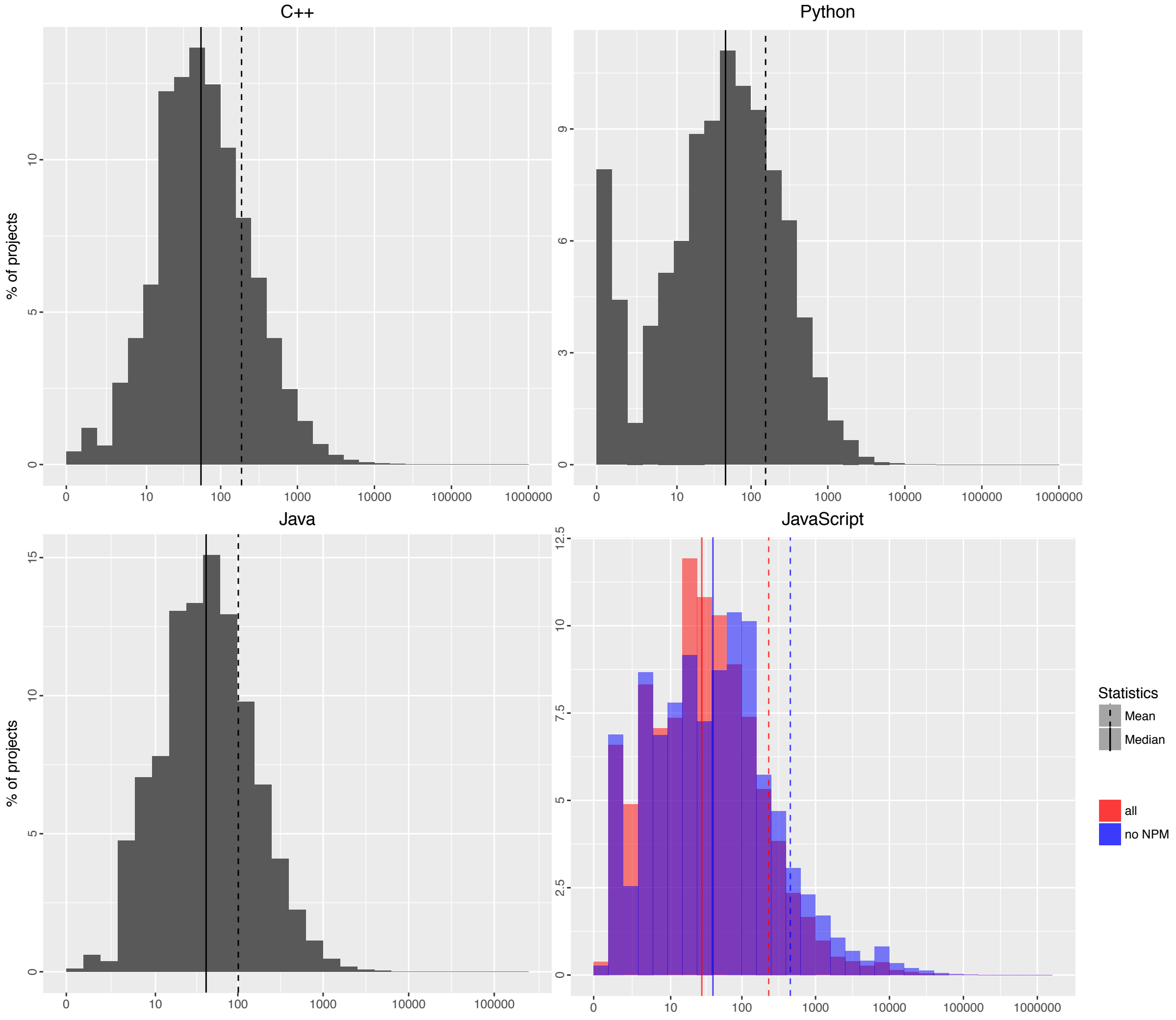


Fig. 5. SLOC per file.

of information about clone groups is given by the largest extreme. In Python, the largest group of file-hash clones has over 2.5M files. In Java, the largest group of SourcererCC clones has over 65K files. In the next section we show which files these are.

5.2 File-Level Analysis Excluding Small Files

One observation that emerged immediately from all the language ecosystems was that the most duplicated file is the empty file – a file with no content, and size 0. In the Python corpus alone, there are close to 2.2M occurrences of this trivial file, and in the JavaScript corpus there are 986K occurrences of that same file. Another frequently occurring trivial file in all ecosystems is a file with 1 empty line. Indeed, a common pattern that emerged was that the most duplicated files tend to be very small. Once we detected that, we redid the analysis excluding small files. Specifically, we excluded all files with less than 50 tokens.[2] Table 3 and Figure 6 (bottom row) show the results.

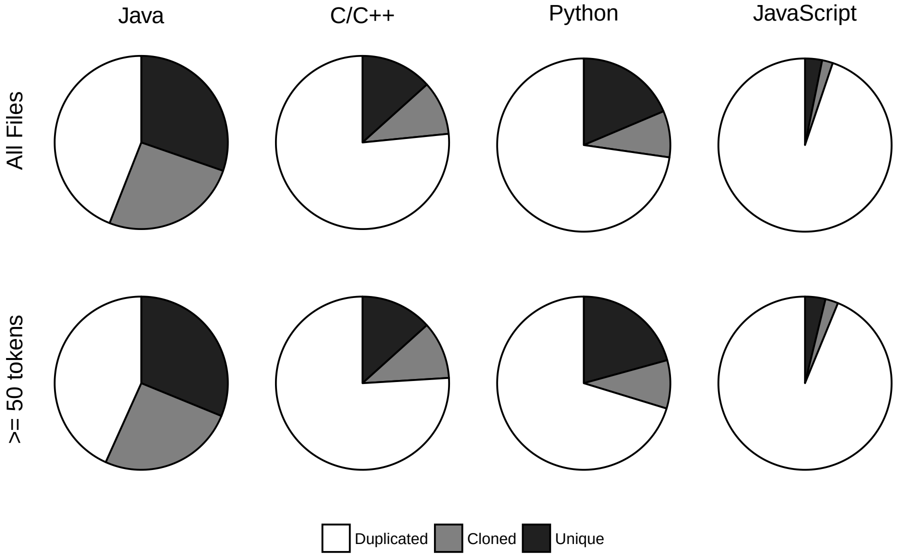


Fig. 6. File-level duplication for entire dataset and excluding small files.

Although the absolute number of files and hashes change significantly, the changes in ratios of the hashes and SCC results are small. When they are noticeable, they show that there is slightly less duplication in this dataset than in the entire dataset. Comparing Table 3 with Table 2 shows that small files account for a slightly higher presence of duplication, but not that much higher than the rest of the corpus.

5.3 Inter-Project Analysis

So far, we investigated how code duplication is rampant at the file level. The next question is how this finding maps into projects: how many projects are exact and near-duplicates of other projects, even though they are not technically forks? This is called **inter**-project cloning. For that, and as explained in Section 3, we computed the

[2] This threshold is arbitrary. It is based on our observations of small files; other values can be used.

Table 3. File-level duplication excluding small files.

	Java	C++	Python	JavaScript
# of files	57,240,552	49,507,006	23,382,050	162,136,892
% of corpus	79%	80%	74%	62%
File hashes	34,617,736 (60%)	13,401,948 (27%)	7,267,097 (31%)	11,444,667 (7%)
Token hashes	32,473,052 (58%)	11,893,435 (24%)	6,949,894 (30%)	10,074,582 (6%)
SCC dup files	14,626,434 (26%)	5,297,028 (10%)	2,105,769 (9%)	3,896,989 (2%)
SCC unique files	17,848,618 (31%)	6,596,407 (13%)	4,844,125 (21%)	6,177,593 (4%)

overlap of files between projects, as given by the files' **token hashes**. We used the entire corpus, including the small files, as these are important for the projects. The results are shown in Table 4 and Figure 7.

Table 4. Inter-project cloning.

	Java	C++	Python	JavaScript
# projects (analyzed)	1,481,468	364,155	893,197	1,755,618
# clones ≥ 50%	205,663 (14%)	94,482 (25%)	159,224 (18%)	854,300 (48%)
# clones ≥ 80%	135,168 (9%)	58,906 (16%)	94,634 (11%)	546,207 (31%)
# clones 100%	87,220 (6%)	24,851 (7%)	51,589 (6%)	273,970 (15%)
# exact dups	73,869 (5%)	19,809 (5%)	43,501 (5%)	198,556 (11%)
# exact dups (≥ 10 files)	37,722 (3%)	10,286 (3%)	7,331 (1%)	78,972 (4%)

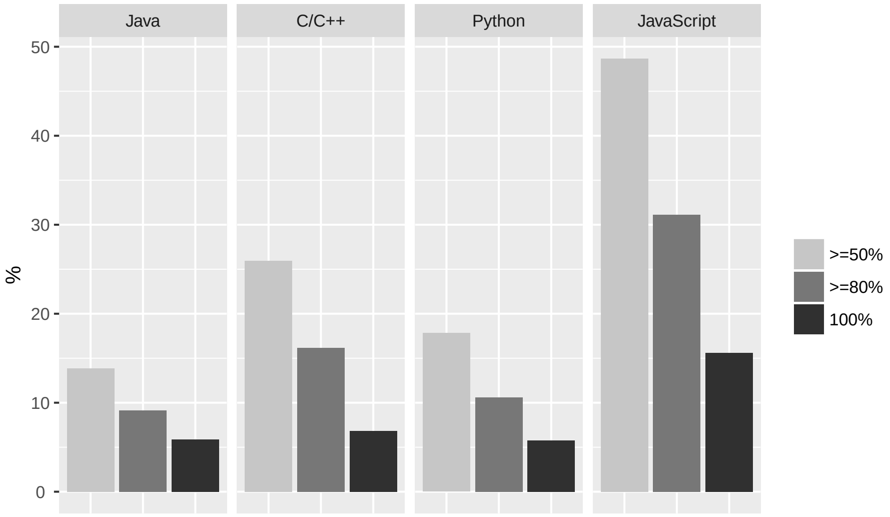


Fig. 7. Percentage of project clones at various levels of overlap.

Table 4 shows the number of projects whose files exist in other projects at some overlap threshold – 50%, 80% and 100%, respectively. A normalization of these numbers over the total number of projects for each language is shown in Figure 7. JavaScript comes on top with respect to the amount of project-level duplication, with 48% of projects having 50% or more files duplicated in some other project, and an equally impressive 15% of projects being 100% duplicated.[3] Not surprisingly, the percentage of project-level duplication tracks the percentage of file-level duplication, as shown in Figure 6. The differences seen between the language ecosystems do not seem to be related to the size of projects: Table 1 in Section 4 shows that the median files per project in JavaScript is slightly higher than in Python (so, JavaScript projects tend to have more files), but the inter-project cloning is much higher for JavaScript than for Python. We will dive more into this in the next section.

The last two rows of Table 4 show the number of projects that are token-hash clones of some other project (apart from differences in white space, comments, and terminal symbols). This is different, and more constrained,

[3] Again, we remind the reader that our dataset does not contain forks.

than being cloned at 100% elsewhere: it requires bidirectionality. With the exception of JavaScript at 11%, the ratios are all 5%, but it is still surprising that there are so many projects that are exact copies of each other. As the last row shows, though, many of those are very small projects, with less than 10 files. The number for projects with at least 10 files that are exact copies of some other project is considerably smaller, but still in the thousands for all languages.

6 MIXED METHOD ANALYSIS

The numbers presented in the previous section portray an image of GitHub not seen before. However, that quantitative analysis opens more questions. What files are being copied around, and why? What explains the differences between the language ecosystems? Why is the JavaScript ecosystem so much off the charts in terms of duplication? In order to answer these kinds of questions, we delve deeper into the data.

With so much data, our first heuristic was size. As seen in the previous section we noticed that the empty file was the most duplicated file in the entire corpus, among all languages. We also noticed that the top duplicated files tended to be very small and relatively generic. Although an intriguing finding, very small, generic files hardly provide any insightful information about the practice of code duplication. What about the non-trivial files that are heavily duplicated? What are they?

This section presents observations emerging from looking at specific files and projects using mixed methods. We divide the section into four parts: (1) an analysis of each language ecosystems looking for the most duplicated files in general; (2) file duplication at different levels (file hashes, token hashes and near duplicates with SourcererCC); (3) the most reappropriated projects in the four ecosystems; and (4) an in-depth analysis of the JavaScript ecosystem.

6.1 Most Duplicated Non-Trivial Files

As stated above, we wanted to find out if the size of the files had an effect on their duplication. For example, are small files copy-pasted from StackOverflow or online tutorials and blogs, and large files from well-known supporting libraries? In order to make sense of so much data, we needed to sample it first, so that interesting hypotheses could emerge, and/or we could find counter-examples that contradicted our initial expectations. This is territory of qualitative and mixed methods [Creswell 2014].

6.1.1 Methodology. We used a mixed method approach consisting of qualitative and quantitative elements. Based on our quantitative analysis, we hypothesized that size of the files, and whether the duplication was exact or token-based, might have an effect on the nature of duplication; for example, the empty file certainly is not being copy-pasted from one project to another, it simply is created in many projects, for a variety of reasons. Maybe we could see patterns emerge for files of different sizes. The following describes our methodology:

- **Quantitative Elements.** We split files according to the percentiles of the number of tokens per file within each language corpus, and create bins representing the ranges 20%-30% (small), 45%-55% (medium), 70%-80% (large), and greater than 90% (very large). So, the 45%-55% bin contains files that are between the 45% percentile and the 55% percentile on the number of tokens per file of a certain language. The number of tokens for the bins can be seen in Table 5. For example in Java, the first bin includes files containing 47 to 72 tokens, and so on. The gaps between these percentiles (for example, no file is observed between the 30% and the 45% percentile) ensure buffer zones that are large enough to isolate the differently-sized files, should differences in their characteristics be observed. For each of these bins, we analyzed the top 20 most cloned files; this grouping was performed twice, using file hashes and token hashes, and this was done for all the languages. In total, for each language, 20(top-20 cloned)x4(4 bins)x2(file and token-hash)=80 files were analyzed, for a total of 320 files.

Table 5. Number of tokens per file within certain percentiles of the distribution of file size.

		20%-30%	45%-55%	70%-80%	90%+
Tokens	Java	46-71	120-167	279-419	751+
	C/C++	50-77	138-199	372-623	1284+
	Python	29-65	149-236	477-795	1596+
	JavaScript	19-32	68-114	238-431	1127+
Files	Java	7,670,926 (11%)	7,523,679 (10%)	7,335,067 (10%)	7,298,767 (10%)
	C/C++	6,381,850 (10%)	6,228,550 (10%)	6,204,943 (10%)	6,167,647 (10%)
	Python	3,282,957 (10%)	3,205,337 (10%)	3,169,316 (10%)	3,161,325 (10%)
	JavaSript	28,257,319 (11%)	27,306,195 (10%)	26,326,975 (10%)	26,134,513 (10%)

- **Qualitative Elements.** Looking at names of most popular files, a first observation was that many of these files came from popular libraries and frameworks, like Apache Cordova. This hinted at the possibility that the origin of file duplication was in well-known, popular libraries copied in many projects; a qualitative analysis of file duplication was better understood from this perspective. Therefore, each file was observed from the perspective of the path relative to the project where it resides, and was then *hand coded* for its origin.[4] For example, `project_name/src/external/com/http-lib/src/file.java` was considered to be part of the external library `http-lib`. Each folder assumed to represent an external library was matched with an existing homepage for the library, if we could find it using Google. Continuing the running example, `http-lib` was only flagged as an external dependency if there was a clear pointer online for a Java library with that name. In some cases, the path name was harder to interpret, for example: project_name/external/include/internal/ftobjs.h. In those cases, we searched Google for the last part of the path in order to find the origin (in this particular case, we searched i nclude/internal/ftobjs.h). For JavaScript the situation was often simpler: many of the files came from NPM modules, in which case the module name was obvious from the file's location. Some of the files were also minified versions of libraries, in which case the name of the file gave the library name, often with its version (e.g. `jquery-3.2.1.min.js`). Using these methods, we were able to trace the origins of all the 320 files.

6.1.2 Observations. Contrary to our original expectation, we did not find any differences in the nature of file duplication related to either size of the files, similarity metric, or language in the 320 samples we inspected. We also didn't find any StackOverflow or tutorial files in these samples. Moreover, the results for these files show a pattern that crosses all of those dimensions: the most duplicated files in all ecosystems come from a few well-known libraries and frameworks. The Java files were dominated by the ActionBarSherlock and Cordova. C/C++ was dominated by boost and freetype, and JavaScript was dominated by files from various NPM packages, only 2 cases were from jQuery library. For Python, the origins of file cloning for the 80 files sampled were more diverse, along 6 or 7 common frameworks.[5]

Because the JavaScript sample was so heavily (78 out of 80) dominated by Node.js packages, we have performed the same analysis again, this time excluding the Node.js files. This uncovered jQuery in its various versions and parts accounting for more than half of the sample (43), followed from a distance by other popular frameworks such as Twitter Bootstrap (12), Angular.js (7), reveal.js (4). Language tools such as modernizr, prettify, HTML5Shiv and others were present. We attribute this greater diversity to the fact that to keep connections small, many

[4] For a good tutorial on coding, see [Saldaña 2009]

[5] The very small number of libraries and frameworks found in these samples is a consequence of having sampled only 80 files per language, and the most duplicated ones. Many of the files had the same origin, because those original libraries consist of several files.

libraries are distributed as a single file. It is also a testament to the popularity of jQuery which still managed to occupy half of the list.

The presence of external libraries within the projects' source code shows a form of dependency management that occurs across languages, namely, some dependencies are source-copied to the projects and committed to the projects' repositories, independent of being installed through a package manager or not. Whether this is due to personal preference, operational necessity, or simple practicality cannot be inferred from our data.

Another interesting observation was the proliferation of libraries for being themselves source-included in other widely-duplicated libraries. Take Cordova, a common duplicated presence within the Java ecosystem. Cordova includes the source of okhttp, another common origin of duplication. Similarly, within C/C++, freetype2 was disseminated in great part with the help of another highly dispersed supporting framework, cocos2d. This not only exacerbates the problem, but provides a clear picture of the tangled hierarchical reliance that exists in modern software, and that sometimes is source-included rather than being installed via a package manager.

6.2 File Duplication at Different Levels

In this section, we look in greater detail at the duplication in the three levels reported: file hashes, token hashes and SCC clones:

6.2.1 File Hashes. Top cloned files of various sizes were already analyzed in 6.1. To complement, we have also investigated mostly cloned non-trivial files across all sizes to make sure no interesting files slipped between the bins, but we did not find any new information. Instead we tried to give more precise answer to question which files get cloned most often. Our assumption was that the smaller the file, the more likely it is to be copied. Figure 8 shows our findings as a heatmap with number of clones on horizontal axis and size of the file in bytes on the vertical axis. The linear artefacts towards the left egde of the graphs are due to the logarithmic scale of both axes. The heatmaps show that while it is indeed smaller files that get copied most often, with the exception of extremely small outliers (trivial files, such as the empty file), the largest duplication groups can be found for files with sizes in thousands of bytes, with maximum sizes of the clone groups gradually lowering for either larger, or smaller files.

6.2.2 Token Hashes. For a glimpse of the distribution of token hashes, we have investigated the relations between number of files within a token hash group and number of file hashes (i.e. different files). These findings are summarized in Figure 9. The outlier in the top-right corner of each graph is the empty file. The number of different empty files is explained by the fact that when using token hash, any file that does not have any language tokens in it is considered empty. Given the multitude of sizes observed within token hash groups, the next step was to analyze the actual difference in sizes within the groups. The results shown in Figure 10 summarize our findings. As expected, for all four languages the empty file again showed very close to the top. For Java, the biggest empty file was 24.3MB and contains a huge number of comments as a compiler test. For C/C++ the empty files has the second largest difference and consists of a comment with ASCII art. Python's empty file was a JSON dump on a single line, which was commented, and finally for JavaScript the largest empty file consisted of thousands of repetitions of an identical comment line, totaling 36MB.

More interesting than largest empty files is the answer to the question: What other, non-trivial files display the greatest difference between sizes in the same group. Interestingly, the answer is slightly different for each language: for Java, the greatest size differences exist for binary files disguised as java files. In these files, very few tokens were identified by the tokenizer and therefore two unrelated binary files were grouped into a single token group with a small number of very different files. For C/C++ often, we have found source codes with and without hundreds of KB of comments as members of the same groups. An outlier was a file with excessive white-spaces at each line (2.42MB difference). In Python, formatting was most often the cause: a single file multiplied its size 10 times by switching from tabs to 8 spaces. For JavaScript, we observed minified and non-minified versions.

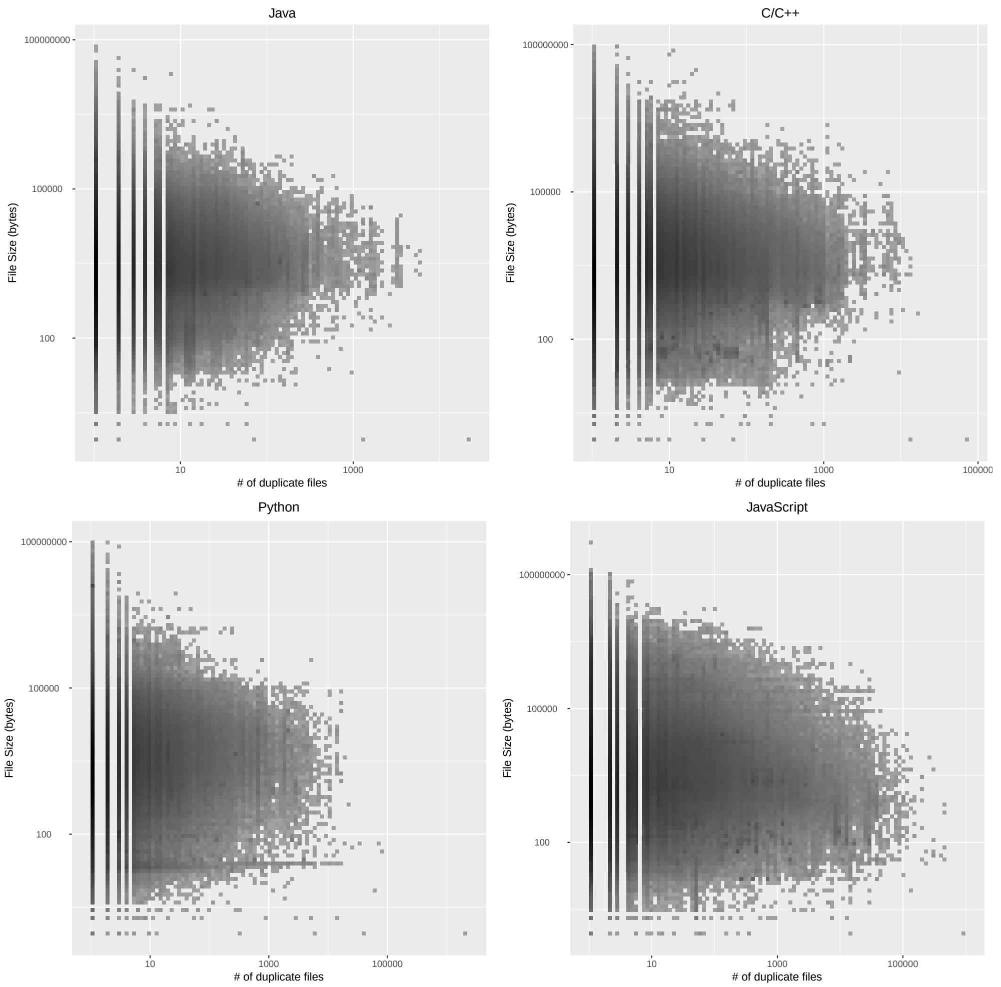


Fig. 8. Distribution of file-hash clones.

Sometimes the files were false positives because complex Javascript regular expressions were treated as comments by the simple cross-language parser.

6.2.3 SourcererCC Duplicates. For SourcererCC, we randomly selected 20 clone pairs and we categorized them into three categories: i) *intentional copy-paste clones*; ii) *unintentional accidental clones*; and iii) *auto-generated clones*. It is interesting to note that the clones in categories ii) and iii) are both unavoidable and are created because of the use of the popular frameworks.

Java. We have categorized 30% (6 out of 20) of the clone pairs into the *intentional copy-paste clones* category. It included instances of both inter-project and intra-project clones. Intra-project clones were created to test/implement functionalities that are similar while keeping them isolated and easy to maintain. Inter-project clones seemed to come from projects that look like class projects for a university course and from situations where one project was almost entirely copy-pasted into the other project. We found 2 instances of *unintentional cloning*,

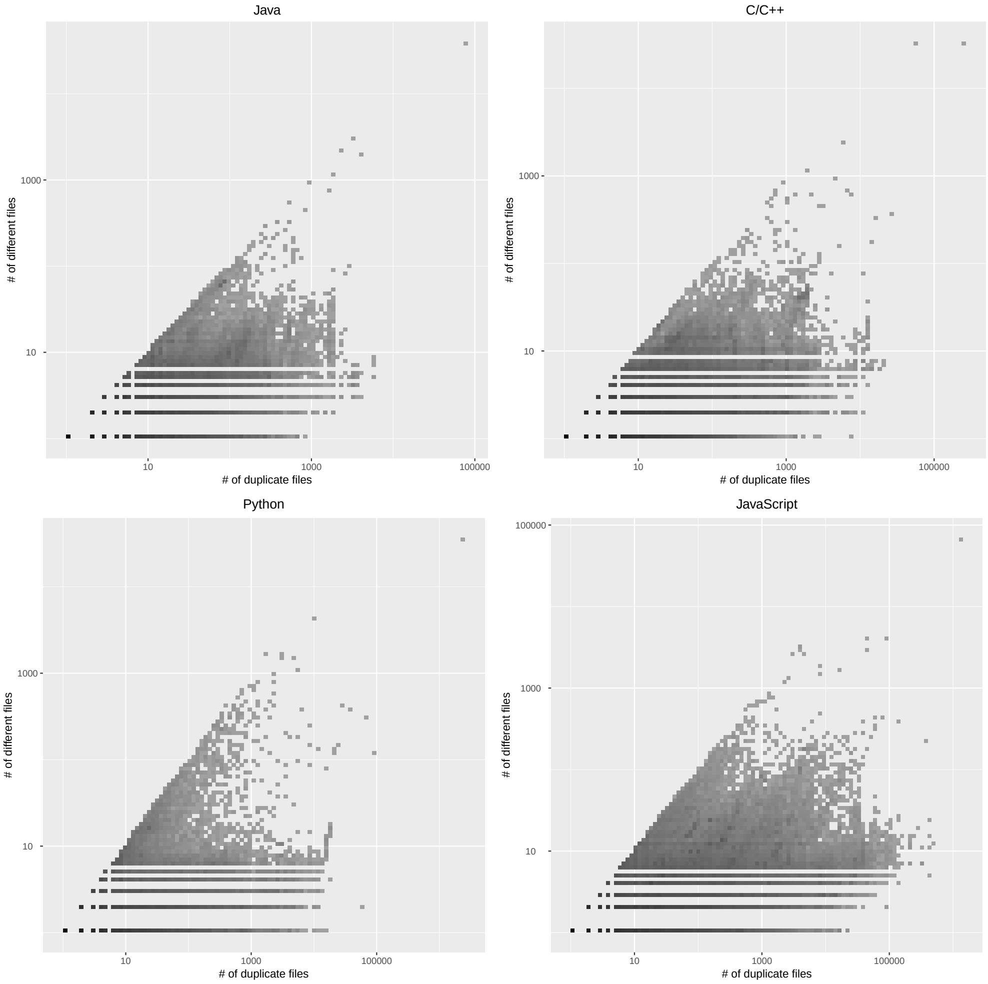


Fig. 9. Distribution of token-hash clones.

both inter-project. The files in such clone pairs implement a lot of similar boilerplate code necessary to create an Android activity class. We categorized the majority (12 out of 20) of the clone pairs into the *auto-generated clones* category. The files in this category are automatically generated from the frameworks like Apache Axis (6 pairs), Android (2 pairs), and Java Architecture for XML Binding (4 pairs). The unintentional and auto-generated clones together constitute 70% of the sample.

C/C++. The sample was dominated by *intentional copy-paste clones* (70%, 12 pairs). The origin for these file clone pairs seems to be the same, independent of these being inter of intra-project clones, and relates to the reuse of certain pieces of source code after which they suffer small modification to cope with different setups or support different frameworks. Five pairs were classified as *unintentional cloning*. They represented educational situations (one file was composed in its large part by the skeleton of a problem, and the difference between the files clones was the small piece of code that implements the solution). Two different versions of the same file were

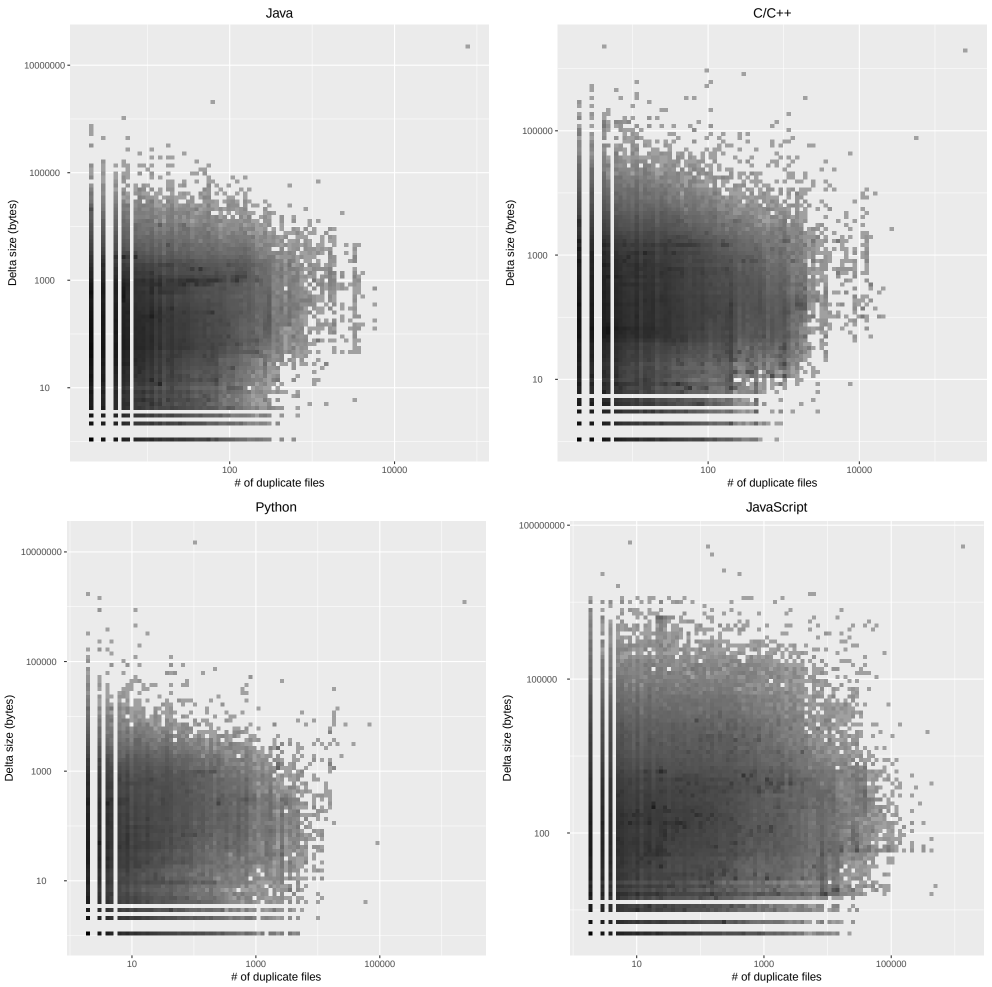


Fig. 10. Δ of file sizes in token hash groups.

also found (libpng 1.0.9 vs. libpng 1.2.30). Files from two projects sharing a common ancestor (bitcoin vs dotcoin) were also observed. The *auto-generated clones* were present in three pairs, 2 of them from the Meta-Object compiler [6]. The unintentional and auto-generated clones accounted for 40% of the sample.

Python. The sample was dominated by uses of the Django framework (17 pairs), all variants of auto generated code to initialize a Djagno application. We classified them as *auto-generated clones*. Two pairs were *intentional copy-paste clones* intra-project copy-paste of unittests. The last pair belonged to the same category was a model schema for a Django database.

JavaScript. Only one *intentional copy-paste clones* example has been found, which consisted of a test template with manually changed name, but nothing else. Five occurrences of *unintentional cloning* comprised of pairs

[6] http://doc.qt.io/qt-4.8/moc.html

of different file versions for jQuery(2), google maps opacity slider, modernizr, and angular socket service. The remaining 14 pairs (70%) have been classified as *auto-generated clones*. Dominated by angular.js project files(7), project files for express.js(3), angular locales and different gruntfiles (builder files for node.js projects) were present. All of the Angular project files are created with Yeoman, a tool for creating application skeletons with boilerplate code used also by the Angular Full Stack Generator. The last pair classified as autogenerated was also the only inter-project clone and consisted of two very similar JSON records in a federal election commission dump stored on Github. In total, 95% of the pairs were unintentional or auto-generated.

6.3 Most Reappropriated Projects

We look for projects duplicated in bulk without any addition or change, i.e. with 100% of their files present in a single host project. This captures the practice of reappropriation. Since versioning systems offer features that should be used instead of reappropriation (such as Git submodules) we were interested in how prevalent and for what purposes reappropriation exists. A simple query into the database gave us some insights. Note our analysis is not exhaustive; projects originating from outside GitHub may not be found unless an abandoned project that just reappropriated them exists. But if the project's exact copy will be missed, the files themselves will be identified as clones between projects using the same library.

For Java, we found that *Minecraft-API* and *PhoneGap* are the two most reappropriated projects. Looking for clues online, we found that the original Minecraft-API project was not hosted in GitHub until 2012, so the copies may have been from developers who used GitHub at the time. Also, on further inspection we found that *PhoneGap* is related to *Apache Cordova*. These frameworks might not have been in GitHub from the beginning.

For C++, GNU ISO C++ Library, homework templates, and Arduino examples have been reappropriated the most. The homework case is interesting: it seems that some instructor created a body of code that was then cloned by several dozen students, instead of being forked in GitHub, as one might expect. All clones were exactly the same, which seems to indicate the students didn't push their changes back. This an unorthodox, and somewhat abusive use of GitHub.

For Python, the top 3 most reappropriated projects are Cactus, which is a static site generator using Django templates, Shadowsocks, a fast tunnel proxy that helps bypass firewalls, and Scons, a software construction tool.

Finally, for JavaScript the most reappropriated project is the Adobe *PhoneGap*'s Hello World Template, [7], which has been found intact in total of 1746 projects. PhoneGap is a framework for building mobile applications using the web stack and it dominates the most frequently cloned projects - the top 15 most cloned projects are all different versions of its template. PhoneGap is followed by the *OctoPress* [8] blogging framework and by a template for the IBM *BlueMix* [9] microservices.

These observations show that project reappropriation exists for a variety of reasons: simple reappropriations that could be addressed by Git submodules (e.g. Minecraft API, Arduino), seemingly abandoned derivative development (Cactus, PhoneGap), true forks with addition of non-source code content (OctoPress) and even unorthodox uses of GitHub (the C++ homework).

6.4 JavaScript

JavaScript has the highest clone ratio of the languages studied. Over 94% of the files are file-hash clones. We wanted to find out what is causing this bloat. After manually inspecting several files, we observed that many projects commit libraries available through NPM as if they are part of the application code.[10] As such, we analyzed the data with respect to the effect of NPM libraries, and concluded that this practice is the single

[7] http://github.com/phonegap/phonegap-template-hello-world

[8] http://octopress.org/

[9] https://www.ibm.com/cloud-computing/bluemix/

[10] `npm` is the package manager used by the very popular Node.js framework.

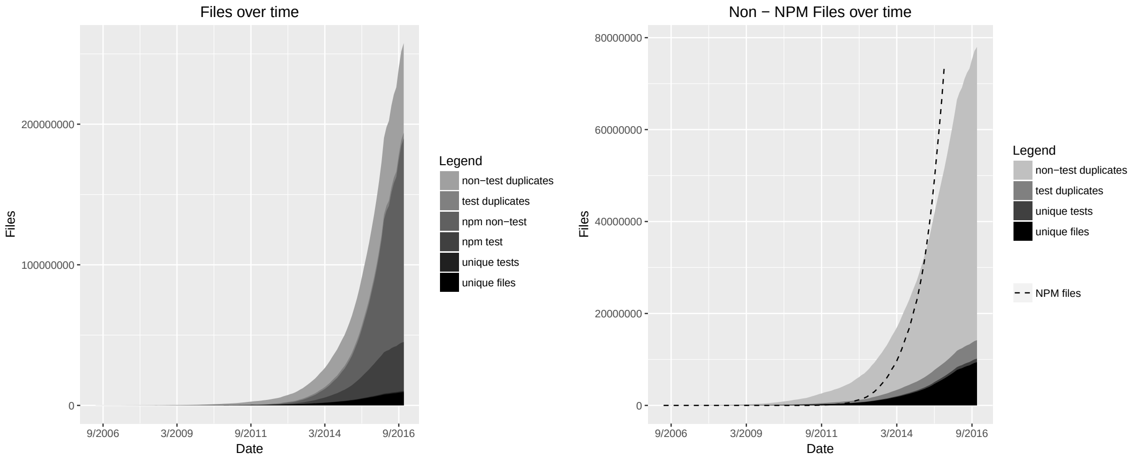


Fig. 11. JavaScript files over time, with and without NPM files.

biggest cause for the large duplication in JavaScript. What follows are some mostly quantitative perspectives on the effect of NPM libraries, along with some qualitative observations pulled from additional sources.

Figure 11 on the left shows the composition of JavaScript repositories over time with respect to unique files and tests and token-hash clones and (we considered any file in `test` folder to be a test) compared with files & tests coming from unorthodox use of NPM. Figure 11 on the right shows the corpus in the same categories, but without the NPM files, whose number is indicated by the dashed line which quickly surpasses all other files in the corpus. The huge impact of NPM files can be seen not only in the sheer number of files, Figure 12 shows the percentage of token-hash clones for different subsets of the files over time. To help assess influence, the background of the graph shows the numbers of total and NPM files at given times.

Few files predate the NPM Manager itself (January 2010). We have found similar outliers in the rest of the files (small amount of them predating not just GitHub and Git, but even JavaScript itself). As soon as NPM files started to appear in the corpus, they took over the global ratio (solid line), while the rest of the files slowly added original content over time. Interesting is the higher originality of tests – when people copy and paste the entire files, they tend to ignore the associated tests.

6.4.1 NPM Files. When npm is used in a project, the `package.json` file contains the description of the project including its required packages. When the project is built, these packages, are loaded and stored in the '`node_modules`' directory. If the packages themselves have dependencies, these are stored under the package name in another nested '`node_modules`' directory. The '`node_modules`' folder will be updated each time the project is built and a new version of some of the packages it transitively requires is available. Therefore it should not be part of the repository itself - a practice GitHub recommends[11]. Since NPM allows dependencies to link to specific versions of the package, there is no need to include the '`node_modules`' directory even if the application requires specific package version.

Even more surprising than the sheer number of NPM files in the corpus is the number of packages responsible for them. 41.21% (732991) projects use NPM package manager, but only 6% (106582) projects include their '`node_modules`' directory. These 6% projects are ultimately responsible for almost 70% of the entire files. It is

[11] https://github.com/github/gitignore/blob/master/Node.gitignore

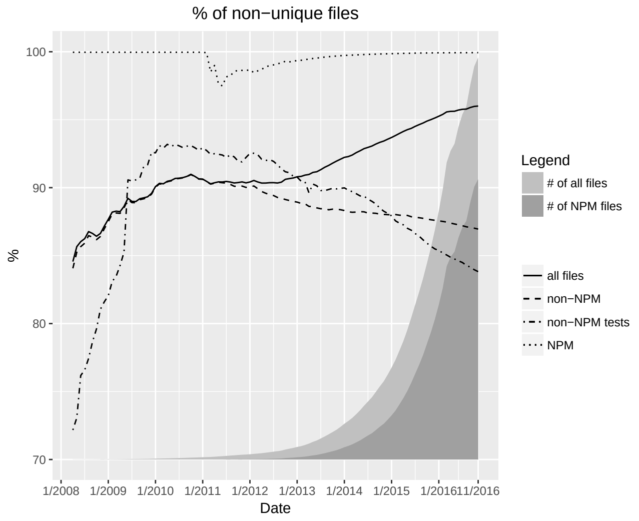


Fig. 12. Percentage of clones over time

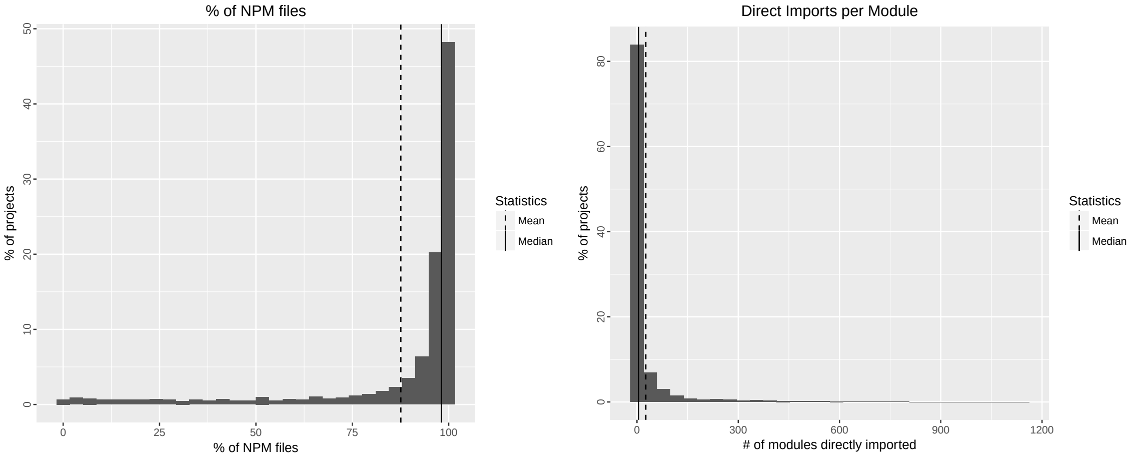


Fig. 13. % of NPM files in projects and directly imported NPM packages

therefore not surprising that once a project includes its NPM dependencies, its file number is overwhelmed by the packagesâĂŹ files as shown in Figure 13 on the left.

There are even projects that seem to contain only NPM files. Often a project is created using an automated generator which installs various dependencies, pushed to Github with the `node_modules` directory and never used again. The largest of such projects[12] contains only NPM modules used in other project of the same author and has 46281 files. If the project is written a dialect of Javascript that does not use the `js` extension (such as

[12] https://github.com/kuangyeheng/workflow-modules

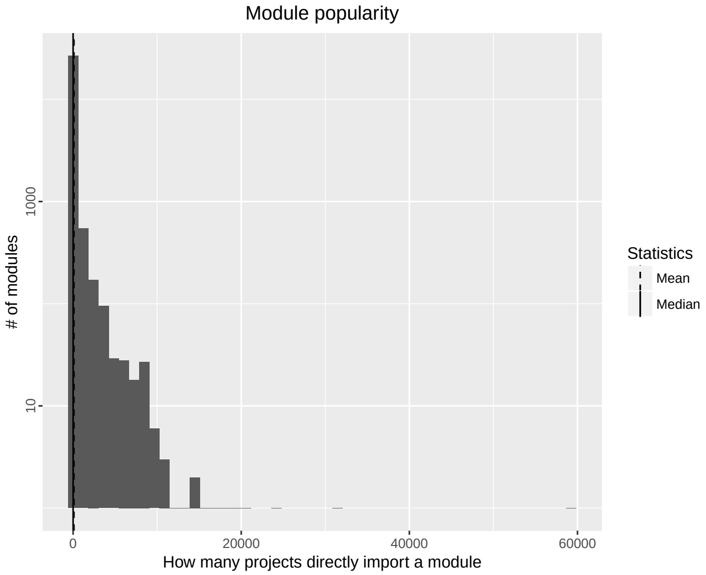


Fig. 14. Popularity of NPM modules.

jsx or TypeScript) it would appear all its files come from NPM. This is the case of the second largest npm-only project[13] consists of 16761 JS files from NPM and a handful of jsx files discovered by manual inspection.

We have also analyzed the depth of nested dependencies in the NPM packages. In the worst case we have observed this nesting to be 47 modules deep with median of 5. The number of unique projects included has median of 63 and maxes out at 1261, but this includes the nested dependencies as well. The direct imports, i.e. modules specified as dependencies in the package.json file is in general much smaller as shown in Figure 13 on the right.

There are however outliers which come close to the max number of unique projects included. The largests of them has been created by the Angular Full Stack Generator[14], an automated service for generating Angular.JS applications[15]. Other projects with extraordinarily large direct dependencies are created using similar automated generators, such as Yeoman.

In terms of module popularity (Figure 14) (note the log scale on y axis) most modules are imported by a small percent of projects, however there are some massively popular ones: Express[16] (59277 projects) is a minimalist web UI framework, body parser[17] (31807 projects) a HTTP response body parser and debug[18] (24413 projects), a debugging utility for node.js applications.

Surprisingly, many of the NPM packages contain a great deal of tests in them, as shown in Figure 11, which seems unnecessary, as these should be release versions of the packages for users, not package for developers.

[13] https://github.com/george-codes/react-skeleton

[14] https://github.com/angular-fullstack/generator-angular-fullstack

[15] It is perhaps ironical that the project itself was according to its authors created to let people âĂIJquickly set up a project following best practicesâĂİ

[16] https://www.npmjs.com/package/express

[17] https://www.npmjs.com/package/body-parser

[18] https://www.npmjs.com/package/debug

7 CONCLUSIONS

The source control system upon which GitHub is built, Git, encourages forking projects and independent development of those forks. GitHub provides an easy interface for forking a project, and then for merging code changes back to the original projects. This is a popular feature: the metadata available from GHTorrent shows an average of 1 fork per project. However, there is a lot more duplication of code that happens in GitHub that does not go through the fork mechanism, and, instead, goes in via copy and paste of files and even entire libraries.

We presented an exhaustive investigation of code cloning in GitHub for four of the most popular object-oriented languages: Java, C++, Python and JavaScript. The amount of file-level duplication is staggering in the four language ecosystems, with the extreme case of JavaScript, where only 6% of the files are original, and the rest are copies of those. The Java ecosystem has the least amount of duplication. These results stand even when ignoring very small files. When delving deeper into the data we observed the presence of files from popular libraries that were copy-included in a large number projects. We also detected cases of reappropriation of entire projects, where developers take over a project without changes. There seemed to be several reasons for this, from abandoned projects , to slightly abusive uses of GitHub in educational contexts. Finally, we studied the JavaScript ecosystem, which turns out to be dominated by Node.js libraries that are committed to the applications' repositories.

To mitigate internal threats to validity, we provide our entire dataset and scripts to repeat every step of its creation so that our results can be validated independently. As for external threats, we make no generalization or assumptions regarding the use of GitHub projects for research.

This study has some important consequences. First, it would seem that GitHub, itself, might be able to compress its corpus to a fraction of what it is. Second, more and more research is being done using large collections of open source projects readily available from GitHub. Code duplication can severely skew the conclusions of those studies. The assumption of diversity of projects in those datasets may be compromised. DéjàVu can help researchers and developers navigate through code cloning in GitHub, and avoid it when necessary.

Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement 695412), from the United States Defense Advanced Research Agency under the MUSE program, and was partially support by NSF award 1544542 and ONR award 503353.

REFERENCES

T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere. 2013. Orion: A Software Project Search Engine with Integrated Diverse Software Artifacts. In *International Conference on Engineering of Complex Computer Systems*. https://doi.org/10.1109/ICECCS.2013.42

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/1167473.1167488

Hudson Borges, André C. Hora, and Marco Tulio Valente. 2016. Understanding the Factors that Impact the Popularity of GitHub Repositories. (2016). http://arxiv.org/abs/1606.04984

Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. 2015. Assert Use in GitHub Projects. In *International Conference on Software Engineering (ICSE)*. http://dl.acm.org/citation.cfm?id=2818754.2818846

James R. Cordy, Thomas R. Dean, and Nikita Synytskyy. 2004. Practical Language-independent Detection of Near-miss Clones. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. http://dl.acm.org/citation.cfm?id=1034914.1034915

V. Cosentino, J. L. C. Izquierdo, and J. Cabot. 2016. Findings from GitHub: Methods, Datasets and Limitations. In *Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR.2016.023

John W. Creswell. 2014. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE.

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 *International Conference on Software Engineering (ICSE).* http://dl.acm.org/citation.cfm?id=2486788.2486844

Jesus M. Gonzalez-Barahona, Gregorio Robles, and Santiago Dueñas. 2010. Collecting Data About FLOSS Development: The FLOSSMetrics Experience. In *International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS).* https://doi.org/10.1145/1833272.1833278

Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Working Conference on Mining Software Repositories (MSR).* https://doi.org/10.1109/MSR.2013.6624034

Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. 2011. *On the Extent and Nature of Software Reuse in Open Source Java Projects.* Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-21347-2_16

Felipe Hoffa. 2016. 400,000 GitHub repositories, 1 billion files, 14 terabytes of code: Spaces or Tabs? (2016). https://medium.com/@hoffa/400-000-github-repositories-1-billion-files-14-terabytes-of-code-spaces-or-tabs-7cfe0b5dd7fd

Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Working Conference on Mining Software Repositories (MSR).* https://doi.org/10.1145/2597073.2597074

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.* 28, 7 (2002). https://doi.org/10.1109/TSE.2002.1019480

P. S. Kochhar, T. F. BissyandÃl', D. Lo, and L. Jiang. 2013. Adoption of Software Testing in Open Source Projects–A Preliminary Study on 50,000 Projects. In *European Conference on Software Maintenance and Reengineering.* https://doi.org/10.1109/CSMR.2013.48

R. Koschke. 2007. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software (Dagstuhl Seminar Proceedings 06301).*

A. Mockus. 2007. Large-Scale Code Reuse in Open Source Software. In *First International Workshop on Emerging Trends in FLOSS Research and Development.* https://doi.org/10.1109/FLOSS.2007.10

A. Mockus. 2009. Amassing and Indexing a Large Sample of Version Control Systems: Towards the Census of Public Source Code History. In *Working Conference on Mining Software Repositories (MSR).* https://doi.org/10.1109/MSR.2009.5069476

Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research. In *Foundations of Software Engineering (FSE).* https://doi.org/10.1145/2491411.2491415

J. Ossher, Sushil Bajracharya, E. Linstead, P. Baldi, and Crista Lopes. 2009. SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects. In *Working Conference on Mining Software Repositories (MSR).* https://doi.org/10.1109/MSR.2009.5069501

Joel Ossher, Hitesh Sajnani, and Cristina Lopes. 2011. File Cloning in Open Source Java Projects: The Good, the Bad, and the Ugly. In *International Conference on Software Maintenance (ICSM).* https://doi.org/10.1109/ICSM.2011.6080795

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *International Symposium on Foundations of Software Engineering (FSE).* https://doi.org/10.1145/2635868.2635922

Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA).* https://doi.org/10.1145/2048066.2048119

C. K. Roy and J. R. Cordy. 2007. *A survey on software clone detection research.* Technical Report 541. Queens University.

Chanchal K. Roy and James R. Cordy. 2009. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *International Conference on Software Testing, Verification, and Validation.* https://doi.org/10.1109/ICSTW.2009.18

C. K. Roy and J. R. Cordy. 2010. Near-miss Function Clones in Open Source Software: An Empirical Study. *J. Softw. Maint. Evol.* 22, 3 (2010). https://doi.org/10.1002/smr.v22:3

Hitesh Sajnani. 2016. *Large-Scale Code Clone Detection.* Ph.D. Dissertation. University of California, Irvine.

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *International Conference on Software Engineering (ICSE).* https://doi.org/10.1145/2884781.2884877

Johnny Saldaña. 2009. *The Coding Manual for Qualitative Researchers.* SAGE.

SPEC. 1998. SPECjvm98 benchmarks. (1998).

J. Svajlenko and C. K. Roy. 2015. Evaluating clone detection tools with BigCloneBench. In *International Conference on Software Maintenance and Evolution (ICSME).* https://doi.org/10.1109/ICSM.2015.7332459

Christopher Vendome, Gabriele Bavota, Massimiliano Di Penta, Mario Linares-Vásquez, Daniel German, and Denys Poshyvanyk. 2016. License usage and changes: a large-scale study on GitHub. *Empirical Software Engineering* (2016). https://doi.org/10.1007/s10664-016-9438-4

A APPENDIX: SUMMARY STATISTICS

		Java	C++	Python	JavaScript
Files per project	Min	1	1	1	1
	1st Qu	3	3	2	2
	Median	9	11	5	6
	Mean	49	169	35	147
	3rd Qu	24	40	13	22
	Max	375,859	233,844	410,203	255,902
Bytes per file	Min	0	0	0	0
	1st Qu	1,100	1,399	564	348
	Median	2,334	3,114	2,169	1,123
	Mean	5,714	10,340	8,784	11,326
	3rd Qu	5,166	7,779	6,840	3,758
	Max	80M	100M	105M	576M
Lines per file	Min	1	1	1	1
	1st Qu	38	47	20	10
	Median	75	100	66	33
	Mean	164	279	205	265
	3rd Qu	158	237	195	106
	Max	1,437,949	8,129,599	5,861,049	5,105,047
LOC per file	Min	1	1	1	0
	1st Qu	32	38	16	8
	Median	63	83	54	28
	Mean	142	240	174	232
	3rd Qu	135	200	161	93
	Max	1,436,850	8,129,570	5,860,092	5,105,045
SLOC per file	Min	0	0	0	0
	1st Qu	17	23	12	5
	Median	41	55	46	19
	Mean	101	188	156	173
	3rd Qu	97	147	143	75
	Max	1,436,841	8,129,521	5,859,657	5,105,045
Distinct tokens per file	Min	0	0	0	0
	1st Qu	30	31	27	15
	Median	56	57	77	37
	Mean	86	118	197	128
	3rd Qu	100	111	180	92
	Max	1,320,501	7,338,821	3,525,840	5,945,029

Table 6. Summary statistics for the entire dataset.

		Java	C++	Python	JavaScript
Files per project	Min	1	1	1	1
	1^{st} Qu	3	3	2	1
	Median	7	11	5	3
	Mean	25.12	148.2	29	7
	3^{rd} Qu	19	38	12	7
	Max	66,734	77,730	36,840	53,168
Bytes per file	Min	1	1	0	1
	1^{st} Qu	876	1,149	694	586
	Median	1,984	2,622	1,896	1,605
	Mean	5,189	10,557	10,005	22,711
	3^{rd} Qu	4,513	6,812	5,084	4,683
	Max	80,344,320	100,095,279	104,749,580	576,196,992
Lines per file	Min	1	1	1	1
	1^{st} Qu	33	43	25	20
	Median	67	90	60	50
	Mean	149.3	301	182	441
	3^{rd} Qu	142	221	146	130
	Max	1,437,949	8,129,599	5,861,049	5,105,047
LOC per file	Min	1	1	1	1
	1^{st} Qu	27	35	20	16
	Median	56	74	48	42
	Mean	128.4	260	155	394
	3^{rd} Qu	120	185	119	112
	Max	1,436,850	8,129,570	5,860,092	5,105,045
SLOC per file	Min	0	0	0	0
	1^{st} Qu	18	24	16	13
	Median	41	54	40	35
	Mean	97.4	216	138	320
	3^{rd} Qu	93	145	102	95
	Max	1,436,841	8,129,521	5,859,657	5,105,045
Distinct tokens per file	Min	0	0	0	0
	1^{st} Qu	31	33	34	26
	Median	56	60	70	51
	Mean	84.9	132	144	219
	3^{rd} Qu	99	122	137	105
	Max	1,320,501	7,338,821	3,525,840	5,945,029

Table 7. Summary statistics for the minimum set of files (distinct token hashes).

		Java	C++	Python	JavaScript
Files per project	Min	1	1	1	1
	1^{st} Qu	3	3	2	1
	Median	8	11	5	3
	Mean	28	150	29	8
	3^{rd} Qu	20	38	13	7
	Max	74,144	78,106	37,009	53,168
Bytes per file	Min	0	0	0	0
	1^{st} Qu	899	1,149	662	583
	Median	2,019	2,709	1,838	1,637
	Mean	5,207	10,490	9,792	23,013
	3^{rd} Qu	4,563	6,925	5,008	4,964
	Max	80.3M	100M	105M	576M
Lines per file	Min	1	1	1	1
	1^{st} Qu	34	44	24	18
	Median	68	92	58	48
	Mean	150	299	180	454
	3^{rd} Qu	143	223	143	131
	Max	1,437,949	8,129,599	5,861,049	5,105,047
LOC per file	Min	1	1	1	0
	1^{st} Qu	27	36	19	15
	Median	56	76	47	41
	Mean	129	257	153	405
	3^{rd} Qu	121	187	118	113
	Max	1,436,850	8,129,570	5,860,092	5,105,045
SLOC per file	Min	0	0	0	0
	1^{st} Qu	18	24	15	12
	Median	41	54	39	33
	Mean	97	212	136	324
	3^{rd} Qu	92	145	100	95
	Max	1,436,841	8,129,521	5,859,657	5,105,045
Distinct tokens per file	Min	0	0	0	0
	1^{st} Qu	31	33	32	25
	Median	55	60	68	52
	Mean	84	130	142	218
	3^{rd} Qu	98	121	136	108
	Max	1,320,501	7,338,821	3,525,840	5,945,029

Table 8. Summary statistics for the minimum set of files (distinct file hashes).