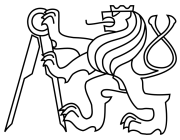
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Analysing JavaScript and NPM at scale
Student:	Bc. Jakub Žitný
Supervisor:	Jan Vitek, M.Sc.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2017/18

Instructions

JavaScript (JS) has evolved significantly over the course of past few years. Besides browsers it is now used on the server-side and in native apps. A lot of useful parts were (and will be) added to the specification. And the NPM package manager has grown to be the largest repository of modules for a programming language. Furthemore, most of the projects available at NPM are open-source with history accessible via GitHub API. This environment of JS modules is an interesting area for a data analysis on how JS is used "in the wild" and provide useful information for optimization of JS runtimes and other areas.

- Create a large dataset of JS codebases.
- Create a program/platform for generating structured JS code datasets.
- Perform data analysis on static code information.
- Perform similarity metrics across different repositories.
- Perform similarity metrics with snippets of code from StackOverflow.

References

Will be provided by the supervisor.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague April 5, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Analysing JavaScript and NPM at scale

Bc. Jakub Žitný

Supervisor: Jan Vitek, M.Sc.

10th January 2017

Acknowledgements

I am extremely grateful for all guidance, support (material and spiritual) and patience I received from my supervisor Jan Vitek. I thank to Petr Maj for pushing the whole project forward when I was busy, lazy or chaotic and for taking the lead. I would also like to thank Petr Brzek for understanding my lack of focus and absence at Avocode. My deepest thanks also go to Eva, João, Honza, Ondrej, Michaela and all my friends that supported me during my studies. I would not finish this without the calming folk tunes of Marhule band either.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 10th January 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Žitný, Jakub. *Analysing JavaScript and NPM at scale.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Táto práca skúma JavaScriptové repozitáre a vývojárske metadáta z viacerých zdrojov. Zozbierali sme niekoľko datasetov a analyzovali ich základné vlastnosti. Vytvorili sme process na detekciu duplikátov miliónov JavaScriptových súborov a zistili sme, že na GitHube je viac ako 90% súborov zduplikovaných z iných projektov. Poskytujeme niekoľko vysvetlení pre veľké množžstvo duplikátov a navrhujeme niekoľko ďalších projektov, ktoré by sa zo zozbieranými dátami mohli robiť.

Klíčová slova big code, github, mining software repositories, ecmascript, javascript, npm, analyza, sourcererCC

Abstract

In this work we looked at different sources of JavaScript code and metadata, collected rich datasets and performed a data analysis on top of them. We have created a pipeline for detecting duplicates within millions of JavaScript files and found that there is more than 90% of JavaScript files in Github projects that are cloned from somewhere else. We discuss the reasons for the high percentage of clones and present few ideas on what analyses we can to in the future with collected data.

Keywords big code, github, mining software repositories, ecmascript, javascript, npm, analysis, sourcerercc

Contents

Introduction **1**
Structure of the following chapters 2

1 Related work **3**
1.1 StackExchange Data Dump 4
1.2 GitHub Archive 5
1.3 GHTorrent 5
1.4 Boa language and infrastructure 7
1.5 npms.io 8

2 Analysis of the problem **9**
2.1 Asserting the quality of dependencies 9
2.2 Searching for clones 11
2.3 Searching for occurrences of StackOverflow snippets 12
2.4 Enhancing the infrastructure 12
2.5 Proposed data analyses 13

3 The data **15**
3.1 GitHub API and its quotas 17
3.2 Git repository contents 19
3.3 NPM API 19
3.4 Code 21

4 Basic data analysis **23**
4.1 JavaScript on GitHub 23
4.2 Popularity on GitHub 31
4.3 NPM 34

5 Clone detection **41**
5.1 SourcererCC 41

5.2 Clone detection pipeline 43
5.3 Results 43

Conclusion **47**

Bibliography **49**

A Acronyms **55**

B Contents of enclosed CD **57**

List of Figures

1.1 JavaScript, CoffeeScript and TypeScript project counts in Boa's 2015 September/GitHub dataset 8

2.1 The NPMS architecture 10
2.2 Proposal for NPMS enhancement 13

3.1 The growth of NPM modules 19

4.1 Distribution of file sizes in the $70k$ dataset 24
4.2 Distribution of LOC per file among all files in the $70k$ dataset . . . 24
4.3 Distribution of token counts in a file in the $70k$ dataset 25
4.4 Distribution of unique token counts in a file in the $70k$ dataset . . 25
4.5 Distribution of whitespace bytes in a file in the $70k$ dataset 26
4.6 Distribution of file counts per project 27
4.7 Distribution of jQuery patch versions in $.min.js$ files in $1M$ dataset 30
4.8 Distribution of the number of stars in projects 31
4.9 Distribution of the number of subscribers in projects 32
4.10 Distribution of the number of forks in projects 33
4.11 Distribution of the number of commits in projects 33
4.12 The share of Git hosting services in NPM modules 35
4.13 The share of open-source licenses in NPM modules 36
4.14 The share of GPL licenses among all GNU GPL derivatives 37
4.15 The share of formats of Readme files in NPM modules 38

List of Tables

3.1 Overview of datasets 16

4.1 Most common module names contained in *node_modules* 28
4.2 Most common *.min.js* files 28
4.3 Summary of stats for star, watch and fork counts 32

5.1 Final clone-detection results for *top*1000, 70*k* and 1*M* datasets . . 45

Introduction

Analysing large codebases such as GitHub can provide interesting insights on how programmers use or misuse programming languages, what kind of programming patterns occur in different development ecosystems, and possibly how authors can further improve their languages or runtimes. Looking at software repositories has been in the eyes and minds of researchers at least since mid 2000s [1]. Every year, the analysis of repositories is becoming more complex and today it requires plenty of human and computational resources. Nowadays, we do not only look at the code, its history in version control systems [2] and build artefacts, but we also take into account the libraries and frameworks that code depends on, the activity of developers, and various popularity indicators of files and projects.

When analysing such large repositories of code we can use an umbrella term "Big Code" [3, 4], similarly as "Big Data" has been used for a general data analyses exceeding certain size or complexity [5].

JavaScript has been rising to the top of programming language popularity indices, such as TIOBE [6], PYPL [7], or IEEE Spectrum's ranking. It has gained the current momentum only recently, hence the usage of modern tools like GitHub or StackOverflow is higher, compared to C, Java, or Python programmers. It is therefore not a coincidence that JavaScript is the most popular language on both platforms, according to GitHut statistics [8] and recent StackOverflow's Developer Survey [9]. Together with the language, the whole ecosystem has been on the rise. JavaScript is now used to write server-side apps thanks to Node.js, native desktop apps thanks to GitHub's Electron, and also mobile apps thanks to Facebook's React Native or Apache Cordova. Most of JavaScript programmers now use NPM to include the libraries and frameworks into their project, in fact, NPM is the largest of all module repositories out there. According to Modulecounts [10], NPM is first with more than twice the modules Maven Central has for Java, Rubygems for Ruby or

Packagist has for PHP.

Analysing JavaScript source code repositories can have not only interesting results but can also lead to creating useful developer tools [4, 3]. This work makes first steps in this direction, several datasets with JavaScript code have been created, analysed and the search for clones within them have been performed and discussed.

Structure of the following chapters

This chapter describes the fundamental motives regarding the purpose and outcome of this work.

The next two chapters review academic work and projects that have been dealing with similar problems, as well as examine the required steps that need to be done before we can look at first interesting problems. Chapter 3 covers the structure of datasets for mentioned purposes and the process of data collection. Chapter 4 provides the reader with basic insights gained from simple dataset analyses. Chapter 5 explores the code duplication that is contained within various repositories in the datasets.

Appendices include further methods and results in order to provide better overview of this topic.

Chapter 1

Related work

There has been work somewhat related to the pursuit of better development tools derived from existing code repositories and the Big Code challenge. The Mining Software Repositories conference has been around since 2004 [1] and the first year had already touched this topic [2]. More relevant work has popped up in past few years, especially when GitHub and StackOverflow polished their APIs and started releasing dump archives dedicated to data analysis.

Wittern E. et al. look at basic statistics of modules hosted in NPM [11]. Authors inspect package descriptions, the dependencies among them, and download metrics, and since NPM packages are usually linked to GitHub source code, they were able to take a look at them as well. In both perspectives, they considered historical data and generated several charts that tell us more about what NPM really stands for. Some of their results are mentioned and built upon in Chapter 4.

A subset of data from GitHub and StackOverflow APIs are also part of publicly available datasets in Google BigQuery. The GitHub part contains repository metadata such as programming languages or licenses, but also the repository contents. BigQuery datasets are publicly available to any logged in user with a Google account. Users can run SQL queries and arbitrary JavaScript code on top of the data. An example of data analysis on top of these datasets is presented in [12]. Authors parsed a billion files from 400000 repositories in 14 programming languages and compared the usage of tabs and spaces for indentation. In results for JavaScript there are 18% of files that use tabs and the rest is spaces. There are similar analyses discussed in [13], [14], [15] and [16].

An example of a tool derived from existing code is JSNice. It is a scalable prediction engine for solving two kinds of problems in the context of JavaS-

cript: predicting (syntactic) names of identifiers and predicting (semantic) type annotations of variables. Experimentally, JSNice predicts correct names for 63% of name identifiers and its type annotation predictions are correct in 81% of the cases. In the first week since its release, JSNice was used by more than 30000 developers and in only few months has become a popular tool in the JavaScript developer community [4].

In 2014, the U.S. DARPA agency initiated a program called Mining and Understanding Software Enclaves (MUSE) [17, 3] which seeks to make significant advances in the way software is built, debugged, verified, maintained and understood. A number of projects have presented their progress or have already been published [18, 19, 20]. *CodePhage* is a system developed for automatic transferring of correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. Experimental result highlight the ability of CP to transfer code across applications to eliminate out of bounds access, integer overflow, and divide by zero errors [18]. *Prophet* is a patch generation system that works with a set of successful human patches obtained from open-source software repositories to learn a probabilistic, application-independent model of correct code. It generates candidate patches, uses the model to rank the them in order of likely correctness, and validates the ranked patches against a suite of test cases to find correct patches [19].

The following sections describe existing datasets and services that can fall into the "Big Code" category. Some of them became the sources of datasets described in the next chapter.

1.1 StackExchange Data Dump

StackExchange Data Dump is an anonymised dataset of all user-contributed content on the Stack Exchange network. Each site is formatted as a separate archive consisting of compressed XML files. Each site archive includes Posts, Users, Votes, Comments, PostHistory and PostLinks [21].

We are interested in the $Posts.xml$ file from $StackOverflow$ part. The compressed size of the single file is 8,3 GB and it includes over a million JavaScript-related posts.

The StackExchange Data Dump is also available at Google's BigQuery.

1.2 GitHub Archive

GitHub Archive is a service recording the public GitHub timeline, archiving it, and making it easily accessible for further data analysis [22]. GitHub Archive records all events from the GitHub API [23], these range from new commits and fork events, to opening new issues, adding new comments and adding new members to a project. All of the events are stored in three datasets by year, month and a day. GitHub Archive datasets are a part of the public datasets on Google's BigQuery.

An insight to the data can be obtained by a simple query. The following query counts the number of forked repositories from last year, for example. The result is 9538608.

Count the number of forks from last year

```
SELECT COUNT(*), repo
FROM [githubarchive:year.2016]
WHERE type = 'ForkEvent'
```

1.3 GHTorrent

One of the sources for our datasets has been GHTorrent. GHTorrent is an independent copy of the data from GitHub API. It monitors the GitHub public event timeline and for each event it retrieves the content and the dependencies, exhaustively. It then stores the raw JSON responses to a MongoDB database, while also extracting the structure in a MySQL database [24].

To access GHTorrent's databases one needs to request access to a SSH tunnel that allows a connection to hosted MongoDB and MySQL instance with the whole dataset. The dataset is also available in Google's BigQuery.

The contents of the dataset inside MySQL tables is following. There are 21 tables:

- commit_comments
- commit_parents
- commits
- followers
- issue_comments
- issue_events
- issue_labels
- organization_members
- project_commits
- project_languages

- project_members
- projects
- pull_request_comments
- pull_request_commits
- pull_request_history
- pull_requests
- repo_labels
- users
- watchers

The *projects* table will be one of the sources for our datasets. The last update to the table was performed in September 2016, it contains 33 million row which takes over 5 GB of disk space. There are 10 columns, the most important ones are *id*, *url* with the link to GitHub API endpoint of given project, *language* with the main programming language of a repository, and the $forked\_from$ and *deleted* flags. The former represents the fact whether project had been created as a fork of another existing one or not. The latter is *true* if the project has been removed from GitHub or its scope has changed from public to private.

When connected to GHTorrent's SSH tunnel we can directly query the MySQL database from a client. An insight to the JavaScript-related data can be given by following queries:

Querying GHTorrent for the counts of JavaScript repositories

```
SELECT COUNT(id)
FROM projects
WHERE language = 'JavaScript';

SELECT COUNT(id)
FROM projects
WHERE language = 'JavaScript'
AND forked_from IS NULL;

SELECT COUNT(id)
FROM projects
WHERE language = 'JavaScript'
AND forked_from IS NULL
AND deleted = 0;
```

The results for the queries are in the order as they were executed 5322424, 2314083 and 2011875. The time GHTorrent database needed to return the first result was 127 seconds, the next two queries have used the cache and ended in milliseconds.

1.4 Boa language and infrastructure

Boa is a domain-specific language and infrastructure that eases mining software repositories. Boa's infrastructure leverages distributed computing techniques to execute queries against hundreds of thousands of software projects very efficiently [25]. The goal of Boa is to ease testing MSR-related hypotheses. The authors came up with it when trying to generally solve a problem with them at a very large scale, at a fine-grained level of detail, and with full history information. [26]

Counting projects in a specific language

```
p: Project = input;
js_count: output sum of int;

foreach (i: int; match(`^javascript$`, lowercase(p.programming_languages[i])))
    jsc_count << 1;
```

The output of Boa job 50839

```
cofeescrpit_count[] = 203531
js_count[] = 1473096
typescript_count[] = 8105
```

The job is publicly available at `https://goo.gl/yatpvZ`.

Boa offers huge comfort for Big Code analysers and already contains both metadata and actual source code in their datasets. Currently they only provide outdated datasets of limited size. The size and richness of Boa datasets is similar to the public ones on Google BigQuery, however, users can not upload their own. The latest Boa dataset from GitHub is from September 2015. Boa is also not nearly as fast as BigQuery.

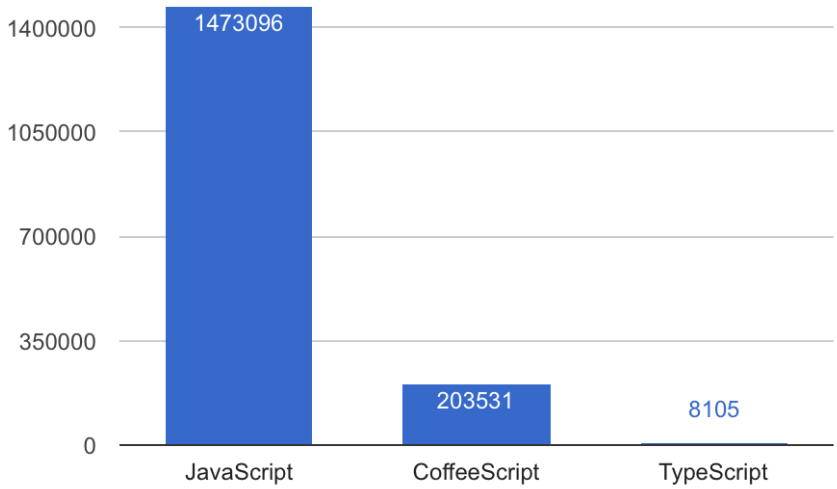


Figure 1.1: JavaScript, CoffeeScript and TypeScript project counts in Boa's 2015 September/GitHub dataset

1.5 npms.io

The npms continuously analyzes the NPM ecosystem, gathering as much information as possible from a variety of sources, including GitHub, David and Node Security Platform (NSP). Using the collected information, a final score for each package is calculated based on four different aspects: quality, maintenance, popularity and personalities. [27]. Users can use NPMS instead of official `https://npmjs.com` to search for NPM modules and learn more about the estimated quality of each one of them.

Chapter 2

Analysis of the problem

In order to follow our goals we need to create a dataset that better suits our needs. It would be even better if we could create a data infrastructure that is extendable and as up-to-date as possible.

The previous chapter has mentioned the GitHub API and projects built on top of it. These are a great start. However, if we want to utilize the full potential of GitHub we have to connect the metadata from the API to the actual source code content of Git repositories that are hosted on GitHub. Furthermore, we should also add metadata from NPM API to the dataset as it contains different popularity indicators than GitHub. These are not always correlated. GitHub API provides developer activity events and number of stars, forks and contributors for each project. NPM API has numbers of downloads for each module. All interesting parts of both APIs will be mentioned in the next chapter.

Besides enriching the actual code with different kinds of available metadata, we would like to be able to regenerate our dataset as often as possible. Since GitHub API is limited we cannot call it too often [28]. This is where the GHTorrent and GitHub Archive are useful. They can give us necessary dumps for every event and we can recreate the picture of data at GitHub without really using our limits for API calls.

In the following sections we are presenting a few possible directions one can take when the foundations of such infrastructure are laid.

2.1 Asserting the quality of dependencies

Can we determine whether JavaScript library or framework is safe to use? How active is the library we include in our project and is it likely to die in

following months or years? What are the quality metrics determining the robustness of a library or module? These are the questions JavaScript developers ask when looking for a module dependency that will be included in their project. There is no JavaScript standard library and even for the smallest things JavaScript programmers tend to depend on an open-source project from NPM or GitHub. We can look at the dataset and expect some of the features to be strong quality indicators. Among them we see the number of stars, forks, subscribers or number of downloads per year, month, and per week. Besides these, we can look at the size of projects, their dependencies, number of maintainers and contributors, issues and whether there are tests or readme files available.

NPMS is one attempt for such assertion. Their formula considers more than 25 quality indicators and calculates a quality score for each NPM module. The quality indicators have been manually chosen to reflect the best practices of JavaScript developers [29]. Figure 2.1 shows the whole pipeline of how NPMS observes the changes on NPM, collects the data and evaluates the score.

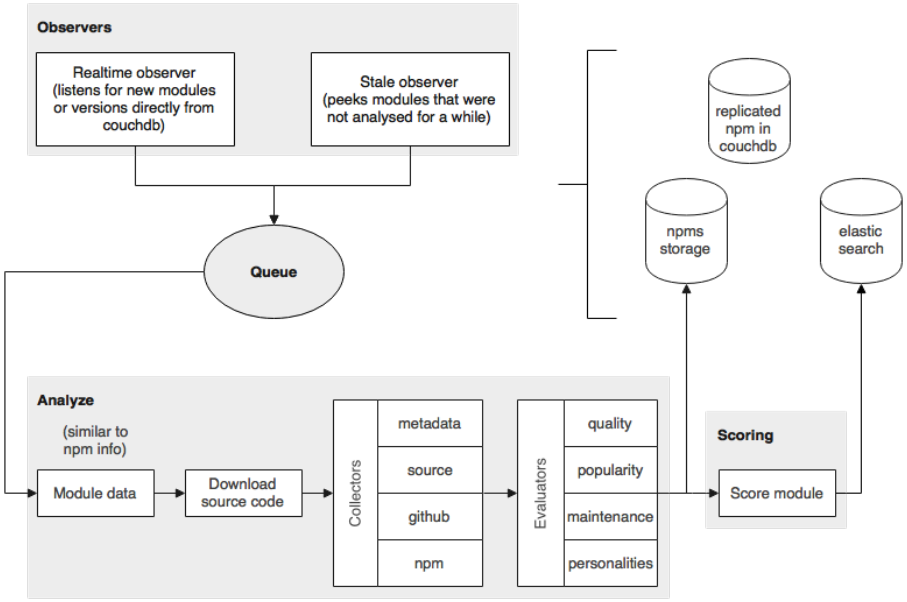


Figure 2.1: The NPMS architecture

The NPMS approach is very popular and more and more developers use it to search for the right dependencies. However, we don't see it scalable and

robust enough. What if the quality indicators change over time? How do we properly balance their value? Could we try training a regression model to calculate a quality score? Would it be possible to get better results than NPMS? Will the size of our dataset be enough for such a complex model? It is definitely worth a try, but creating the right training dataset will be the main challenge. One scenario would be choosing a single ultimate quality indicator, such as GitHub stars or NPM download counts, and trying to learn this from the data as a quality score. Another possibility could be reinforcement learning where the reward would be calculated from the quality score.

The challenge of learning the quality of NPM modules or GitHub projects in general can be reduced to subproblems. We can try to create a classifier for determining the quality of Readme files on GitHub. Readme file content and the list of files in root directory are shown on the default page for each project on GitHub. Readme files contain several quality indicators that NPMS uses in their calculations. Classifying these could be a good subproblem to solve.

2.2 Searching for clones

Another take on the dataset we will have is clone analysis. Once we have such a large and rich dataset of repositories it is interesting to know how can we reduce its size. How many repositories are exact clones of one another? How many repositories are forks of one another? And how many files are copied within and across the repositories? This will help us understand not only the redundancy in the dataset but also the behaviour of the developers and how they are used to the fact that JavaScript now has a module manager [30].

There is a difference between repositories that are forks and repositories that are exact clones of one another. A fork is a copy of a repository that is stored on GitHub (or other Git hosting service) under different namespace [31]. We can determine whether a repository is a fork using GitHub API or we can employ our own algorithm for this. Forked repositories have common history up to certain point but their latest form can be completely diverged and there might be very few common files. An example of such case is when Google forked WebKit's WebCore component and separated the development of new features that were customised for Chrome [32]. Another example is *io.js*, the Node.js fork that had planned to develop separate project under an "open governance model", but let itself back into the Node.js upstream. [33, 34, 35]. Exact clones in our case are repositories that have all files in common. This might be the result of one being a fork of another without any further work being done, it might also be that a programmer manually copied all files to a new project and committed them separately.

We expect cloned files to have different causes. That is why we will do a qualitative analysis of a random sample to see what these might be.

2.3 Searching for occurrences of StackOverflow snippets

If we manage to set up the whole pipeline for detecting clones, we could use it to search for occurrences of StackOverflow snippets. We have already mentioned the StackExchange Data Dump in Chapter 1. We can use the contents of *Posts* and *Answers* from the dataset to create a dataset of JavaScript snippets and search for the clones of these on GitHub.

2.4 Enhancing the infrastructure

As we mentioned earlier, we intend to put forward the whole infrastructure for analysing JavaScript repositories. The infrastructure should be scalable and extendable. Newly calculated data should be reusable for further analyses and new data from the APIs should be available as soon as possible. This will make it even more robust than NPMS.

In addition to the observers, collectors, evaluators and the databases that NPMS has, we propose adding an additional layer of API that will enable users to upload new or derived data. This could work in a similar way that BigQuery offers.

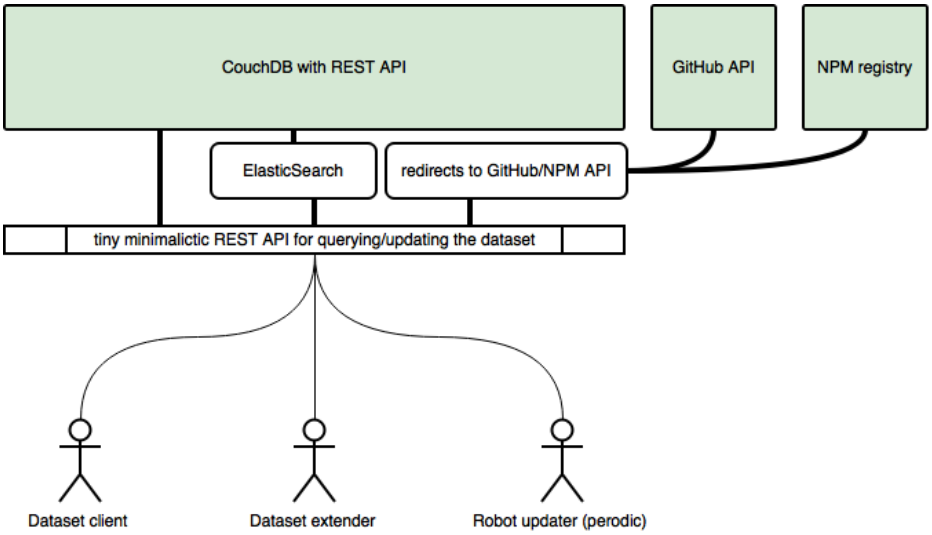


Figure 2.2: Proposal for NPMS enhancement

2.5 Proposed data analyses

We have mentioned four possible directions we can take to follow our goal. Due to time constrains, it is not possible to implement all of them within the scope of this work. We will start with creating the datasets, we will implement an analysis to get to know the basic properties of the data and we will create the pipeline for clone detection within the data.

Chapter 3

The data

The data we intend to use come from several sources. The two main ones are

1. JavaScript Git repositories from GitHub with whole Git history and GitHub metadata

2. metadata from NPM API [30].

None of these are currently in a format that we consider appropriate for analyses we want, however, we do help our case with the already pre-processed dataset from GHTorrent that we can access without limits [24].

Besides these, we will also be putting together a dataset of StackOverflow JavaScript snippets. This dataset will consist of JavaScript code from StackOverflow questions and answers, enriched with metadata from StackExchange API [36].

As we will be using SourcererCC [37] clone detector for detecting clones, we have connected all the sources into a single dataset, similar to the one that authors used for their cause. In their dataset, there is a Git bare repository for working with code history and Git metadata, latest snapshot of the code from Git, usually master branch, GitHub metadata in a special folder and also NPM metadata in another special folder. This enables us to connect the results from various sources and possibly use the historical data from Git too.

Note that Git metadata are different than GitHub metadata. The former represents the metadata from Git itself, such as commit messages, dates of contributions, names of branches, etc., and the latter consists of the data taken from GitHub API [38] where we can see for example the number of project's issues, pull requests, stars or contributors.

Data	Projects	Files	Collection time
top1000	1000	84433	hours
70k	72325	18940371	days
1M	916082	41652400	weeks

Table 3.1: Overview of datasets

We have created several sub-datasets so we can work faster on smaller scale and so that we see the results on the most popular projects separately without distortion from small or non-maintained projects. There is an extreme amount of JavaScript projects on GitHub. In fact, together with repository forks there are over 5 million of them. At NPM there are over 300000 projects and more than 90% of them are linked to their GitHub code. These numbers will be discussed thoroughly in Chapter 4.

We created five datasets as follows.

1. *top*1000 - consists of the most starred 1000 GitHub projects written in JavaScript,
2. $70k$ - consists of randomly picked 70000 GitHub projects written in JavaScript
3. $1M$ - consists of 1 million randomly picked JavaScript GitHub projects that are not forks
4. the NPM dataset - NPM project metadata from the NPM API
5. the $StackOverflow$ JavaScript code snippets

We have collected the datasets in two batches. In the beginning, our goal was to collect the first two datasets only, the $70k$ being the main one. We did not have a good estimate of the size it will take neither the time we will need.

For generating the *top*1000 dataset we only needed the GitHub API. The */search* endpoint is limited to exactly 1000 results [39]. We can specify a language we want the results to be in and also the sorting criterion. In this case we want top 1000 JavaScript projects sorted by popularity. One of the popularity indicators on GitHub is the number of *stars*. A *curl* request for getting these looks like this:

Curl request for top1000 project from GitHub API

```
curl -ni \
"https://api.github.com/search/repositories?sort=stars&q=language:JavaScript"
```

We present detailed description of the GitHub API endpoints we use in the following section.

3.1 GitHub API and its quotas

GitHub API is REST API that returns JSON payloads. We can query it with *curl* command line application or with any HTTP client from any programming language. The API endpoints that we are interested in are

- **/search**
- **/repos/:owner/:repo** or *repo* for short
- **/repos/:owner/:repo/commits** or *commits* for short

We will be querying the *repo* and *commits* endpoints to get meta information about the repositories. Especially the popularity indicators — stargazer, fork and subscriber counts, but also available dates, and commit counts.

Shortened responses for *repos/jakubzitny/big.js* look as follows. The important parts we will be analysing in this work are emphasized.

GitHub API response from /repos/jakubzitny/big.js endpoint


```
{
  "id": 48099297,
  "name": "big.js",
  "full_name": "jakubzitny/big.js",
  "owner": { ... },
  "private": false,
  "html_url": "https://github.com/jakubzitny/big.js",
  "description": "Big.js",
  "fork": false,
  "url": "https://api.github.com/repos/jakubzitny/big.js",
  "created_at": "2015-12-16T09:06:13Z",
  "updated_at": "2016-01-21T16:46:35Z",
  "pushed_at": "2016-07-31T09:48:38Z",
  "git_url": "git://github.com/jakubzitny/big.js.git",
  "ssh_url": "git@github.com:jakubzitny/big.js.git",
  "clone_url": "https://github.com/jakubzitny/big.js.git",
  "size": 8,
  "stargazers_count": 0,
  "language": "CoffeeScript",
  "forks_count": 0,
  "default_branch": "master",
  "subscribers_count": 0,
  ...
}
```

GitHub API response from /repos/jakubzitny/big.js/comits endpoint

```
[
  {
    "sha": "5c87c7e503eae82fd8636189c28a60eaadb5d452",
    "commit": {
      "author": { ... },
      "committer": { ... },
      "message": "add basic crawling in MUSE format",
      "tree": { ... },
      "comment_count": 0
    },
    "author": { ... },
    "committer": { ... },
    "parents": [ ... ]
  },
  ...
]
```

The limitation when querying the GitHub API is 5000 requests per hour for authenticated user [28]. This scales to 70000 or more repositories with a difficulty. We would need 14 hours to get only the basic metadata from one endpoint for each repository.

We also want to get count of all commits for a repository from GitHub API. To do this, we have to look into the HTTP headers of the response from *commits* endpoint. The *commits* endpoint returns the list of commits from the newest one on pages by 30 commits. The *Link* header contains a link to the *next* and *last* page. Let us define the number of pages as *pages*, and the number of commits on the last page as *lp*.

The total number of commits n equals to

$$n = (pages - 1) * \ 30 + lp$$

For getting this information from the GitHub API we need to issue two separate requests. If we want to save the extra request and we do not need the exact n, we can create an upper bound n' that is equal to

$$n' = pages * 30$$

.

When the whole repository clone is available, one can also use the *git* CLI application to get the exact n.

Git CLI command to count commits in a repository

```
git rev-list --all --count
```

3.2 Git repository contents

Besides the meta information from GitHub API we are also cloning the contents of the default branch in the Git repository. We have the Git URL from the API and for simple analysis we need only the latest version of it. We use the *git* CLI client directly with $--depth 1$ argument. To clone contents of the repository from the examples above we will need the following:

Git CLI command to get latest repository without history

```
git clone --depth 1 https://github.com/jakubzitny/big.js.git
```

3.3 NPM API

Figure 3.1 shows the exponential growth of modules published on NPM. According to [10], NPM grows by almost 340 new packages every day. There has been little work done on top of the NPM API though. It does not have any documentation and the only official client is the *npm* CLI application. In Chapter 1 we have mentioned four projects and one publication [11] that did analysis on top of NPM API. Most of them are focusing on visualisation of the NPM package versions [40], download counts [41, 42, 11] and dependency relationships [11, 43].

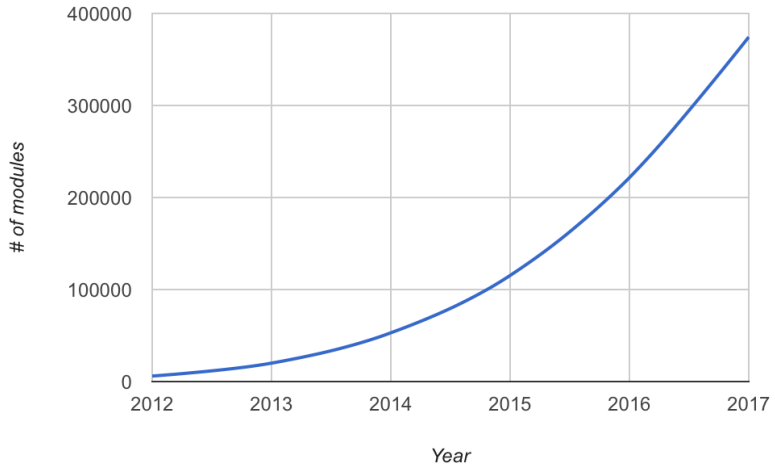


Figure 3.1: The growth of NPM modules

By documenting the API, generating the dataset and examining it we hope to provide better insight into what kind of content NPM really hosts.

We had to look into the source code of the NPM CLI application to see the API endpoints it requests [44]. There are two APIs available at `https://registry.npmjs.org` and at `https://skimdb.npmjs.com`. The former includes all data that NPM CLI app uses, the latter is a CouchDB with module metadata [45].

The endpoint for a single module metadata is **/:name/:version/**. Shortened response to *GET* request to `https://registry.npmjs.org/react/latest` returns following.

NPM API response with module metadata


```
{
  "_id": "react@15.4.2",
  "_nodeVersion": "6.3.1",
  "_shasum": "41f7991b26185392ba9bae96c8889e7e018397ef",
  "name": "react",
  "description": "React is a JavaScript library for building user interfaces.",
  "version": "15.4.2",
  "main": "react.js",
  "dependencies": { ... },
  "bugs": { ... },
  "directories": { ... },
  "dist": { ... },
  "engines": { ... },
  "files": [ ... ],
  "homepage": "https://facebook.github.io/react/",
  "keywords": [ "react" ],
  "license": "BSD-3-Clause",
  "maintainers": [ ... ],
  "repository": {
      "type": "git",
      "url": "git+https://github.com/facebook/react.git"
  },
  "scripts": { ... },
  ...
}
```


There is a lot of interesting information from the *package.json* file of the module [46] and additional internal NPM keys with _ prefix such as *_id*, *_nodeVersion*, *_resolved*, *_shasum*, *_npmOperationalInternal*, or *_npmUser*.

The CouchDB metadata uses the default CouchDB REST API for querying [47]. So, one can use the **/_all_docs** endpoint to look for all available packages and the **/registry/:name** to look for all versions of a module. There is also another special API endpoint at `https://registry.npmjs.org` which is used by the NPM CLI app when searching and caching the registry. It is **/-/all** and it returns an array of metadata for all packages at once. We will be using this response as the source of our NPM dataset.

3.4 Code

All of the datasets we mentioned above and used in this thesis are available at `https://github.com/jakubzitny/big.js`.

Chapter 4

Basic data analysis

In this chapter we are focusing on datasets we have created. We are gathering insights and exploring interesting properties of the data. We look at the code statistics in the first part, the GitHub metadata in the second part and the NPM metadata at the end.

4.1 JavaScript on GitHub

We present the properties of the GitHub 70k and 1M datasets in this section. We chose to start working on a smaller scale so that we get a glimpse of the data, fix bugs, and make the platform more stable.

First we look at the general file statistics. What kind of JavaScript files do we have in our dataset? How big the files are? During the tokenization we have been collecting the line counts and the sizes for each file. Figure 4.1 shows the distribution of file sizes in the $70k$ dataset. Median value is $2,4\ kB$ per file and mean is $20,3\ kB$. The largest file in this dataset is over 90 MB large and it is available from [48].

Figure 4.2 shows the distribution of lines of code (LOC) per file among all files in the $70k$ dataset. We have surprisingly large number of one-liners, the reasons will be discussed in following sections. Median value is 41 lines, mean is 422.2 and the longest file in terms of lines of code is almost 2.8 million lines long. The file is available from [49]. The difference between the mean and median shows that there is a lot of extremely big or long files.

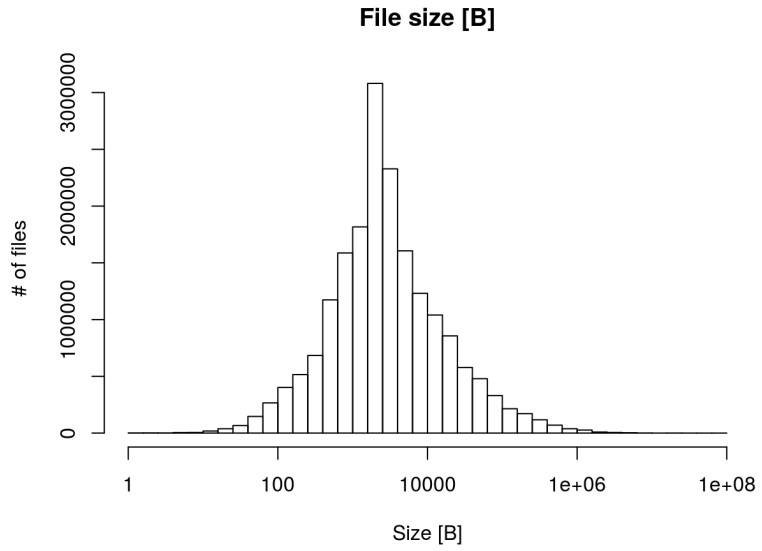


Figure 4.1: Distribution of file sizes in the $70k$ dataset

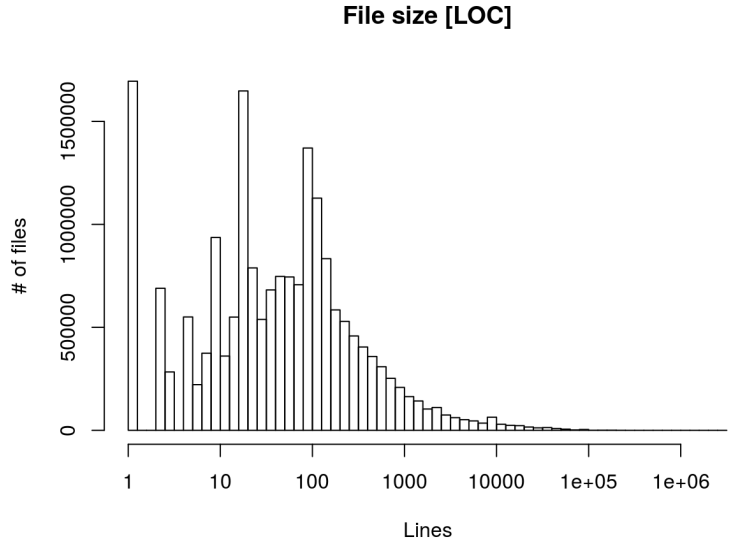


Figure 4.2: Distribution of LOC per file among all files in the $70k$ dataset

Figures 4.3 and 4.4 show the distribution of token counts in a file. The first sums all the tokens together and the second one counts only the unique tokens to capture the diversity of them. Median number of tokens in a file in the $70k$ dataset is 177 and the median of unique tokens is 83.

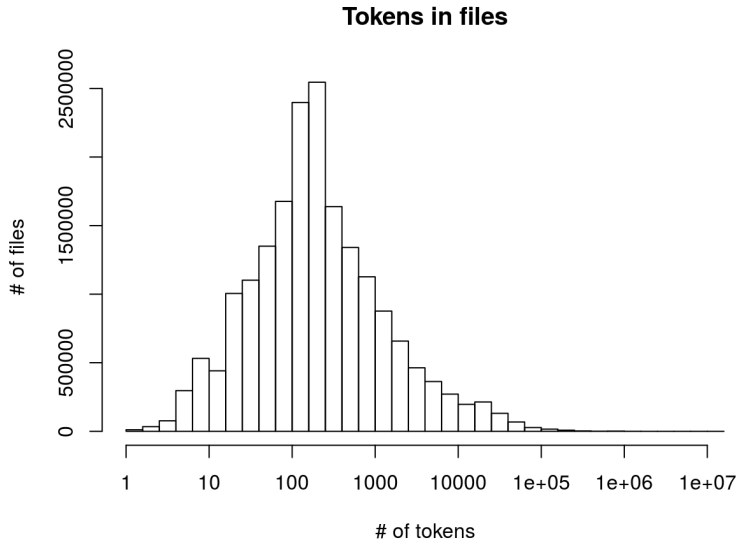


Figure 4.3: Distribution of token counts in a file in the $70k$ dataset

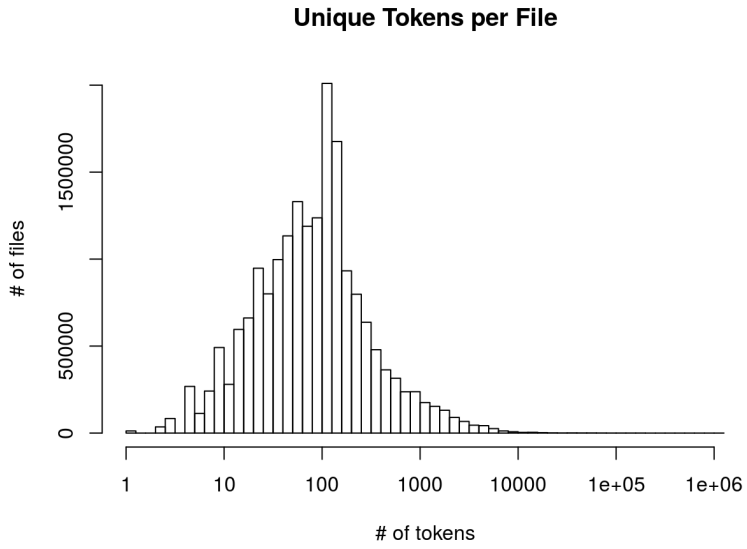


Figure 4.4: Distribution of unique token counts in a file in the $70k$ dataset

Besides the line counts and the sizes for each file, our tokenizer also kept track of the white spaces and comments. We can compare how many percent of the file contents are actually code and how many is only white spaces and comments. On average, there is 6% of the JavaScript file comments and 14% white spaces in the 70k dataset.

On Figure 4.5, we can see that most of the files have less than 10% of the comments per file.

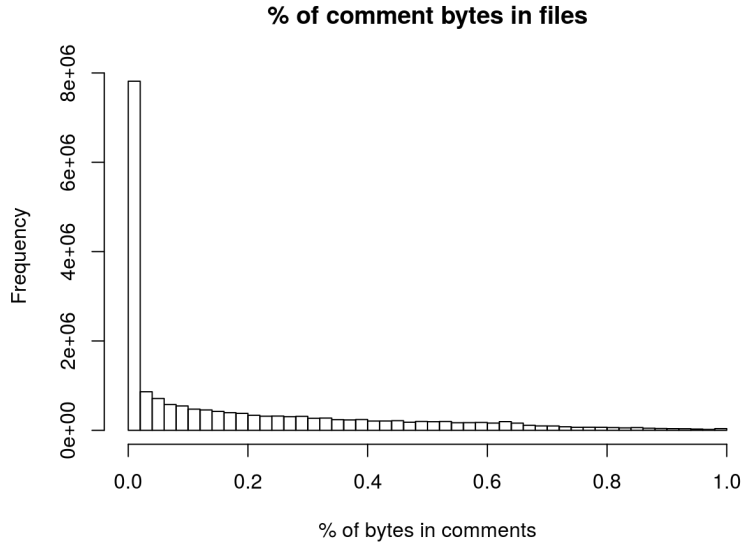


Figure 4.5: Distribution of whitespace bytes in a file in the 70k dataset

To see how projects look compared to each other, we will look at the project-level insights. Figure 4.4 shows the distribution of file counts in our 70k dataset. We can see that most of the projects have 1 to 10 files. The median is 15, the mean is 261.9 and the maximum is 409000 files. Almost half a million files is quite a lot for a single project. We will investigate it further in this chapter.

After looking at the numbers and cherry-picking projects that contain large or a lot of files we have decided not to include forks in the larger scale. This should be the first step in reducing the numbers of duplicated files. We have

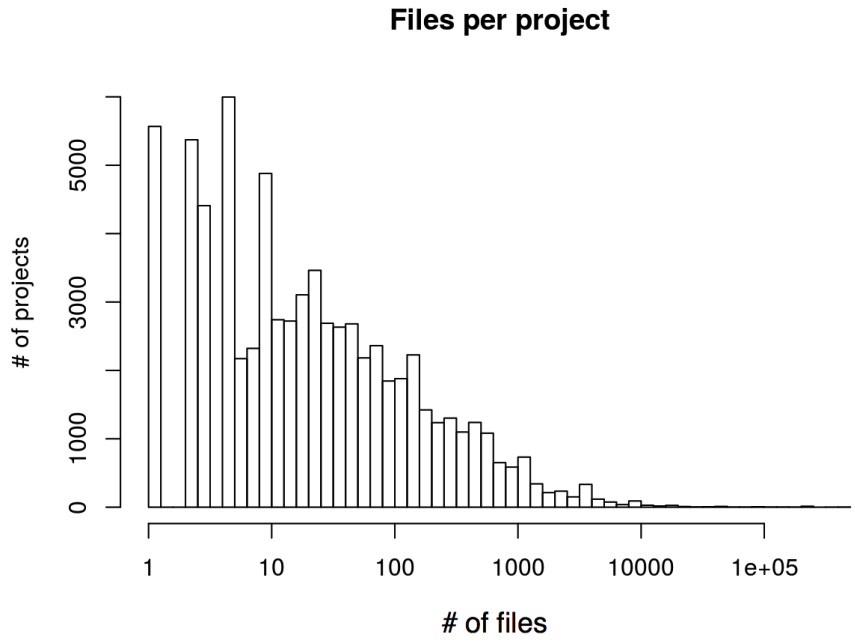


Figure 4.6: Distribution of file counts per project

also identified a few other possible sources of duplication in the datasets. For instance, people often commit minified and obfuscated versions of JavaScript client libraries into their repositories. Even worse case is when commits contain the whole working directory with the complete tree of dependencies from NPM or Bower.

When we scaled up from the $70k$ to $1M$ dataset we found similar statistics for file sizes and counts in projects. There were no forked repositories and we still found large number of duplicates and extremely large number of files in some projects. The results of duplicate detection are detailed in Chapter 5. We will discuss the large projects here.

When we inspected the largest projects we found out that a lot of them contain *node_modules* directory. This is a place where *npm* stores the dependencies, however, this should not be committed into Git repository. It takes too much space and it is not a source-code that programmers want to change or track [50]. If programmers need to lock a specific version of dependencies into the code, there are special mechanisms for that, there is no need to commit libraries or frameworks into the version control [51, 52].

It turns out that there is 70% of files in our dataset that are contained within *node_modules* directory. These 70% of the files are located only in 5% of the projects we have in the $1M$ dataset. There is also 2% of files that are inside *bower_components* directory which is managed by alternative Bower module manager for JavaScript. And there is over 10% of minified files inside the whole dataset. Minified JavaScript files are concatenated and sometimes also obfuscated libraries or arbitrary code that is served to client browsers as one big file so it does not have to do a lot of separate requests for assets to a server. Minified files usually have *.min.js* extension.

We have looked at the most popular Node modules and minified files among these. Table 4.1 shows the list of top 5 Node module names. The most common module in this particular subset of NPM programs is *lodash*. Incidentally, it is also one of the most popular NPM modules all together [30].

name	count	percentage of files
lodash	14668258	35%
core-js	10856121	26%
es5-ext	3211456	7,7%
mout	2074284	5%
babel-runtime	1920365	4,6%

Table 4.1: Most common module names contained in *node_modules*

Table 4.2 shows the list of top 10 files with *.min.js* extensions. Programmers used to commit *.min.js* libraries into repository when building simple client applications in the past. The most popular minified libraries here are jQuery and Bootstrap which are the prototype client-side libraries that many developers use and love.

name	count	percentage of min.js files	percentage of files overall
jquery	1226486	28%	3%
bootstrap	312156	7%	1%
angular	290421	7%	1%
kendo	225967	5%	1-%
lodash	142141	3%	1-%
underscore	109292	3%	1-%
rx	54717	1%	1-%
ionic	50838	1%	1-%
moment	35639	1%	1-%
respond	31461	1%	1-%

Table 4.2: Most common *.min.js* files

We looked at the versions of jQuery programmers are using. In our case the committed *.min.js* files are either named *jquery.min.js* or there is a version included in the filename. *jquery* − 1.9.1.*min.js* for example. jQuery follows the Semantic Versioning of packages. This means every new release of a library has three numbers, major, minor, and patch. 1.9.1 for example. Approximately 50% of jQuery occurrences have versions included.

Figure 4.6 shows the distribution of the patch versions in our $1M$ dataset. The most used *patch* version here is 1.9.1, followed by 1.7.1 and 1.8.2. If we merge the patch version counts, 1.8 and 2.1 will be the most popular. Regarding major versions the first one takes 71,8%.

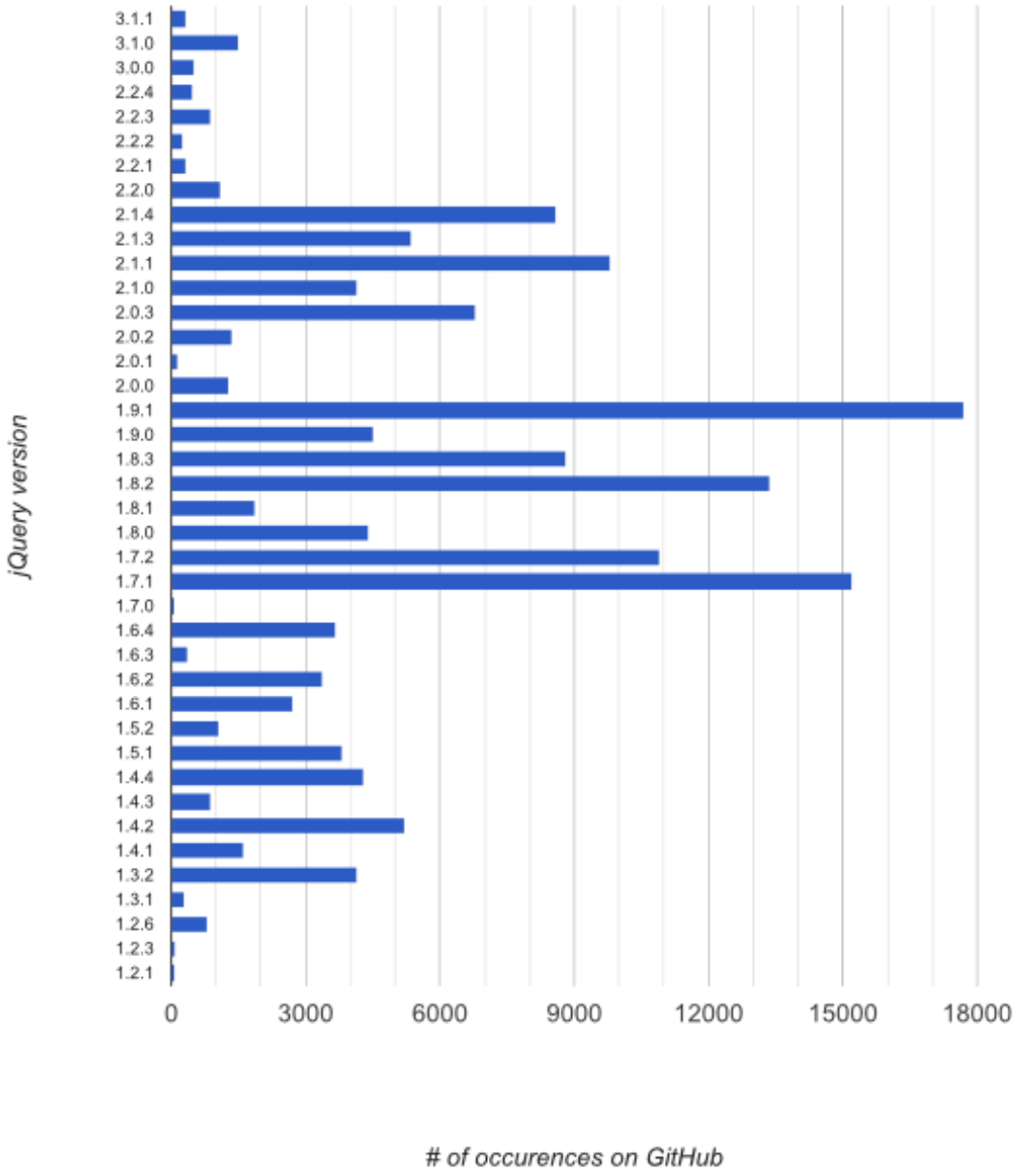


Figure 4.7: Distribution of jQuery patch versions in $.min.js$ files in $1M$ dataset

4.2 Popularity on GitHub

The popularity on GitHub has three main indicators:

1. stars
2. watchers (or subscribers)
3. forks

Starring is a feature that lets users bookmark repositories. *Stars* are shown next to repositories to show an approximate level of interest. Stars have no effect on notifications or the activity feed [**?**]. Users can also *watch* a project. Watching a Repository registers the user to receive notifications on new discussions, as well as events in the user's activity feed [**?**]. Forks are usually connected with contributions to projects. New contributors usually fork main project to their namespace, make local changes and then submit a Pull Request to the upstream.

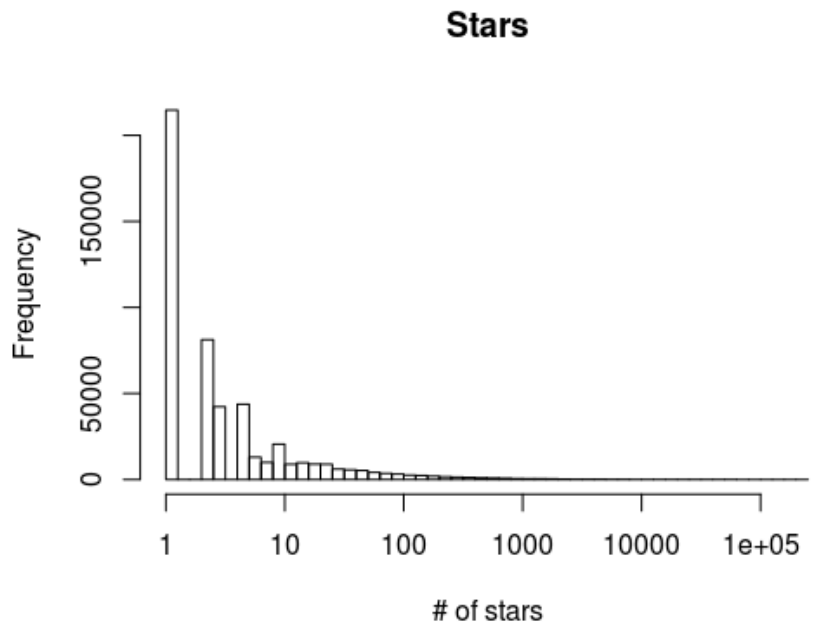


Figure 4.8: Distribution of the number of stars in projects

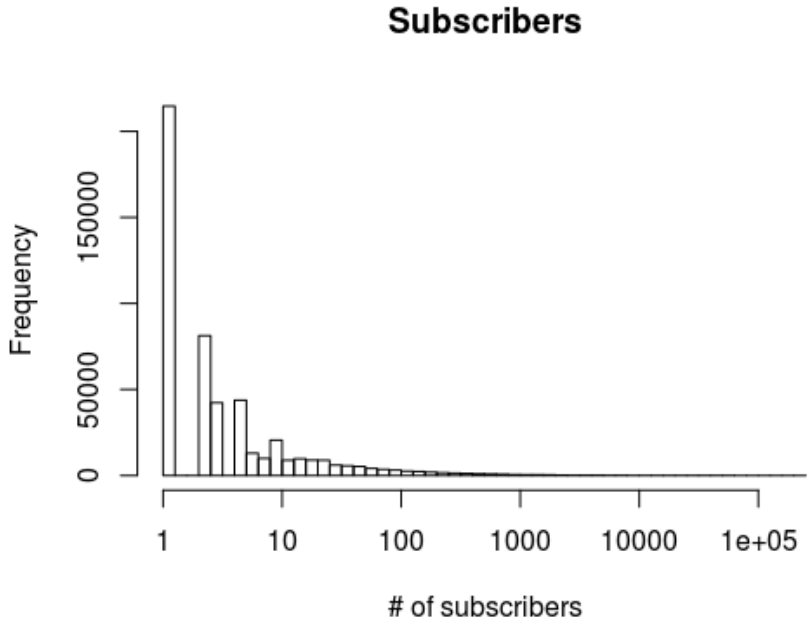


Figure 4.9: Distribution of the number of subscribers in projects

We can see that most of the projects have 0 stars, 0 forks and 0 watchers. This is natural for such a big hosting service as GitHub is. There is many personal projects, that might not be active anymore or are not used by many people, maybe small companies, at schools, etc. There is a fair amount of open-source libraries that are used a lot. The link between the NPM users and GitHub repositories is mentioned in the next section.

	Min	Q1	Med	Mean	Q3	Max
Stars	0.00	0.00	0.00	8.81	1.00	210925.00
Forks	0.00	0.00	0.00	2.06	0.00	26776.00
Watchers	0.00	1.00	1.00	2.42	1.00	5690.00

Table 4.3: Summary of stats for star, watch and fork counts

Results presented in this section are results for the whole set of 2 million non-forked JavaScript GitHub repositories that are listen in GHTorrent's project table. Stars are not present in the GHTorrent dataset so we had to use our scripts from the third chapter to generate these.

The maximum number of both stars and subscribers in a JavaScript project goes to FreeCodeCamp [53], maximum number of forks has Angular.js [54].

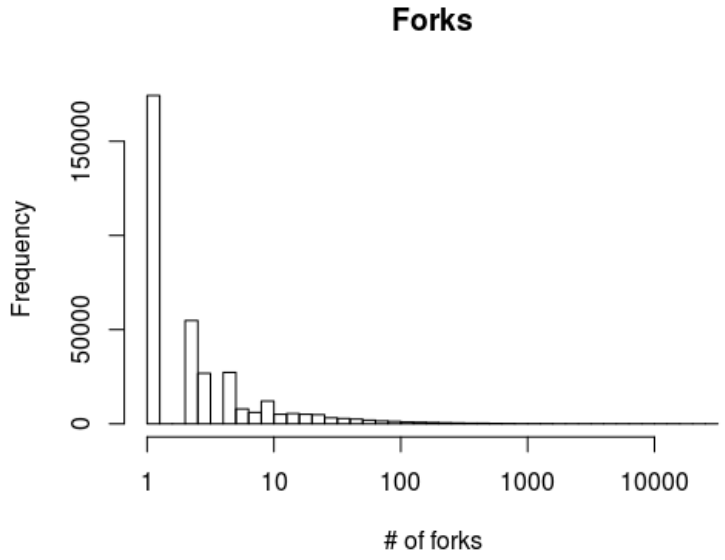


Figure 4.10: Distribution of the number of forks in projects

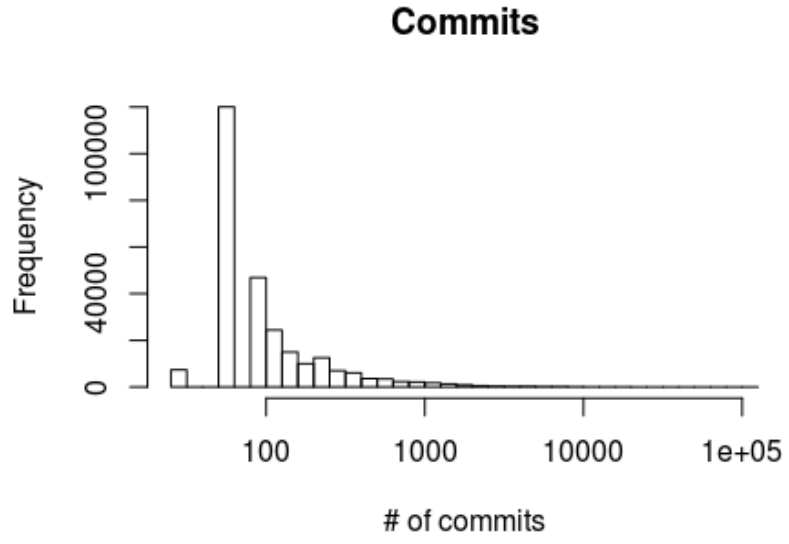


Figure 4.11: Distribution of the number of commits in projects

4.3 NPM

Figure below shows the exponential growth of modules published on NPM. In this section we are analysing the NPM dataset that is described in previous Chapter. NPM grows by almost 340 new packages every day [10].

One of the ways to start a NPM project is to use the *npminit* command. The NPM CLI then asks developer to enter the following information, which is saved in the *package.json* file at the root level of the repository:

- name
- version
- description
- entry point
- test command
- git repository
- keywords
- license

The name, version, description, repository info, keywords and license are also present in our dataset.

4.3.1 Git and version control systems

We have found that 82% of modules have the *repository* field present in their *package.json* files. Out of these, 99% have type Git. It is understandable that this number is so high as NPM command line application specifically asks developers to enter Git repositories. In the missing 1% there are a few SVN, Mercurial and Bazaar repositories. 98% of the Git repositories resolve to endpoints at GitHub, 0.8% to BitBucket and 0.5% to GitLab.

4.3.2 Licenses

79% of the modules have the *License* field filled out. The NPM command line application uses the ISC license automatically, however, the most used license is MIT, followed by ISC, Apache, BSD family, and GPL-like licenses.

97% of Apache licenses are Apache 2.0.

32% of the BSD family occurrences is 3-clause license under names ”revised”, ”new”, or ”modified”, 30% is 2-clause license also known as ”simplified”, or ”FreeBSD License” and 38% of the cases it is not specified whether authors desire BSD-3-Clause or BSD-2-Clause. Only 9 modules explicitly ask for the original 4-clause BSD License.

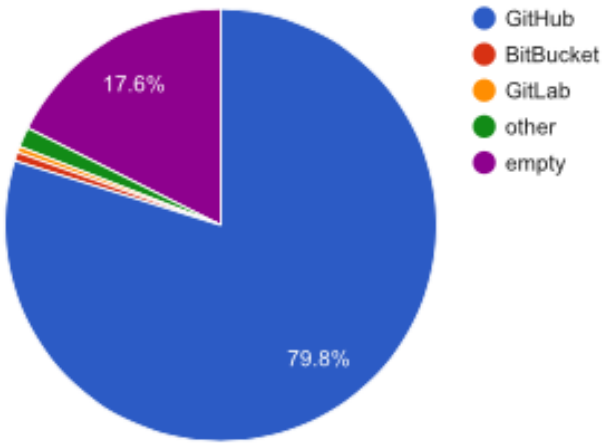


Figure 4.12: The share of Git hosting services in NPM modules

57% of the GPL-like licenses link the latest GPLv3 license [55], 17.5% are GPLv2 [56] and 1% goes to both LGPLv3 and AGPLv3 [57, 58], while old LGPLs and AGPLs are almost not used at all [59, 60, 61, 62].

The rest of the modules include Creative Commons licenses [63], other custom or modified licenses or deliberately ”unlicensed” modules. Worthy of mention is also WTFPL license with a 0.4% share [64].

GitHub released the statistics of overall License usage across all programming languages and revealed that the MIT License had 44,69%, GPL-likes 23%, and BSD family had 6%. The most popular among BSD licenses had been version 2 [65].

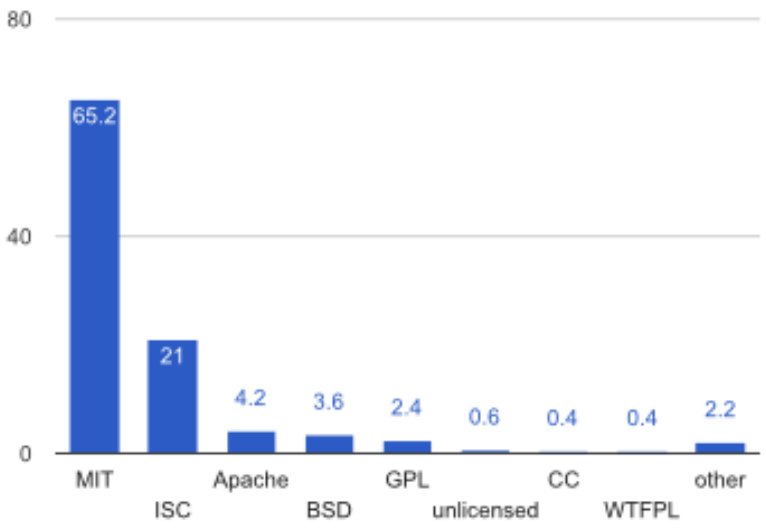


Figure 4.13: The share of open-source licenses in NPM modules

4.3.3 Descriptions

95% of all modules have non-empty description in metadata. The average length of description is 7.7 words per description and the mean is 6. After removing the stop words the 10 most popular words in the corpus of all descriptions are:

1. node
2. simple
3. plugin
4. module
5. library
6. javascript
7. api
8. your
9. react
10. package

We can see that almost all of the words have somewhat general meaning. It is either a synonym to ”module” — ”plugin”, ”library”, or ”package” or it is related to the platform (”node”, ”javascript”). The only instance of a specific library is ”react” [66].

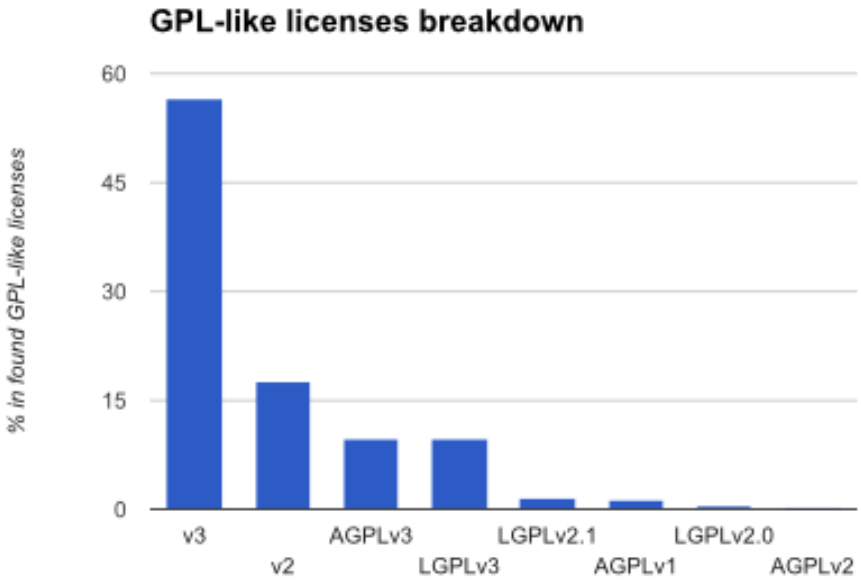


Figure 4.14: The share of GPL licenses among all GNU GPL derivatives

4.3.4 Keywords

We have checked the most used keywords in all modules. 69% of modules have non-empty keyword array in metadata and surprisingly "react" is the most used one.

1. react
2. api
3. javascript
4. css
5. node
6. gruntplugin
7. json
8. yeoman-generator
9. cli
10. plugin
11. express
12. angular
13. react-component
14. test
15. gulp
16. html
17. browser
18. ember-addon
19. framework
20. http

Front-end developers often compare increasingly popular frameworks React, Angular and Ember. In this simple statistic, in over 350000 modules, there is 24952 modules with "react", 9095 with "angular" and 3690 with "ember" keyword occurrences. React already attracted our attention in previous section.

Another interesting comparison might be the number of keywords with *grunt* and *gulp* occurrences. These are runners that a lot of JavaScript project use for building, testing, installing and other configuration tasks. Both of can be used as Make or CMake in the C/C++ world [67]. JavaScript developers often compare these as well [68]. Right now the NPM contains 8619 modules in which keywords match $.*grunt.*$ regular expression. There is 8752 gulp matches.

4.3.5 Readme files

The statistics for Readme file metadata are straightforward. Most of the projects on NPM, similarly as on GitHub are initialized with a Readme.md file. This file is also displayed on module's default page. Markdown with extension ".md" is the preferred markup language. For historical reasons, there are also several text, RST and ADOC Readme files.

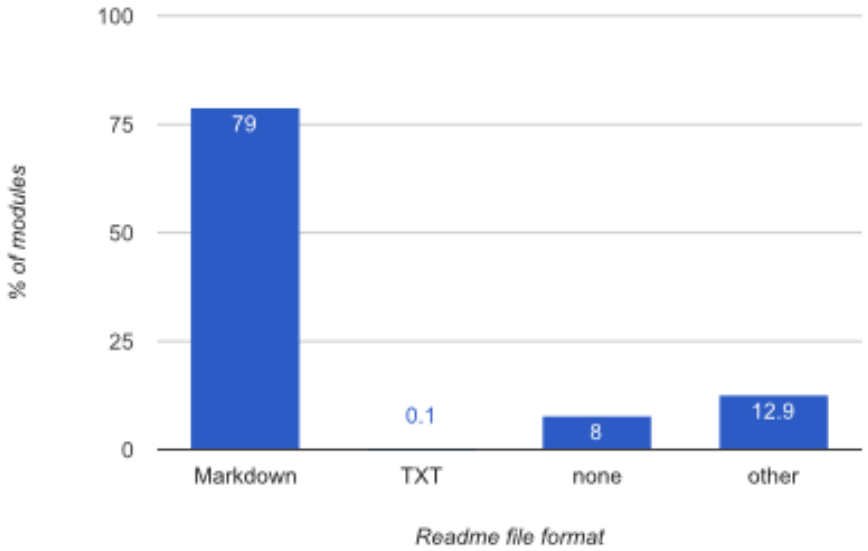


Figure 4.15: The share of formats of Readme files in NPM modules

4.3.6 Analysis code

The code we used for calculating the numbers and generating the charts is available at

- `https://github.com/jakubzitny/big.js`
- `https://github.com/reactorlabs/js-reports`

Chapter 5

Clone detection

There are three different types of clones we want to look for: $file - hash$, $token - hash$ and $sourcerer - cc$ clones. Two files that are $file - hash$ clones have equal MD5 hash. That means that they are exact copies of one another. $Token - hash$ clones have the same MD5 hash of a *token-count* pairs string which captures changes in comments and whitespaces. $Sourcerer - cc$ clones are files reported by SourcererCC clone detector with a 70% threshold. Following sections give details about SourcererCC and the *token-count* pairs.

Incidentally, we can define inverse labels for different types of unique files: $file - hash$ unique file will be a file that is not a $file - hash$ clone. It goes similarly for $token - hash$ and $sourcerer - cc$ uniques.

5.1 SourcererCC

[69] and [37] present efficient clone-detection mechanism and application called SourcererCC. Instead of understanding the whole programming language, SourcererCC calculates the occurrences of tokens in a file, stores them in inverted index and uses it to quickly search for clones. It finds clones where amount of duplicates is above a defined range, 70% for example. This means we can still detect files with whitespaces and comments added, removed, or some variable names and strings changed. And since we are not analysing the syntax or semantics of the source code we can do this very fast and scale the clone detection to Big Code. The clone detection can be performed in different levels of granularity. We can compare files, functions or blocks. For the purpose of this work we will only work with files.

Authors released the source code of SourcererCC at GitHub [70]. The main part — the $clone - detector$ — is written in Java. The input to the $clone - detector$ is preprocessed version of files we want to compare. The pre-

processed version is generated by a tokenizer. A sample tokenizer that works for various programming languages is provided in the source code, it is written in Python. For several reasons we have implemented two different tokenizers for the JavaScript dataset. We will describe them later in this chapter.

Once we have our datasets, the preprocessing consists of splitting each source file by separators to a string of *token-count* pairs. Separators vary from one programming language to another. For JavaScript the separators are operators and whitespaces. The unique id of a file together with *token-count* pairs are written as a one-line string into a file that is fed into $clone-detector$.

Example code in Java

```
/**
 * Execute all nestedTasks.
 */
public void execute() throws BuildException {
  if (fileset == null || fileset.getDir(getProject()) == null) {
    throw new BuildException("Fileset was not configured");
  }
  for (Enumeration e = nestedTasks.elements(); e.hasMoreElements();) {
    Task nestedTask = (Task) e.nextElement();
    nestedTask.perform();
  }
  nestedEcho.reconfigure();
  nestedEcho.perform();
}
```

Example output of a tokenizer for Java code

```
2,1,@#@for@@::@@1,"Fileset@@::@@1,perform@@::@@2,was@@::@@1,configured"@@::
    @@1,throw@@::@@1,if@@::@@1,elements@@::@@1,null@@::@@2,nextElement@@::
    @@1,nestedTask@@::@@2,execute@@::@@1,e@@::@@3,nestedTasks@@::@@1,
    throws@@::@@1,getDir@@::@@1,void@@::@@1,Enumeration@@::@@1,nestedEcho@@
    ::@@2,not@@::@@1,new@@::@@1,getProject@@::@@1,fileset@@::@@2,
    hasMoreElements@@::@@1,Task@@::@@2,public@@::@@1,reconfigure@@::@@1,
    BuildException@@::@@2
```

There are three types of separators here:

1. @#@ - occurs only once, splits the unique fileId and the contents
2. , - splits each *token-count* pair
3. @@ :: @@ - splits tokens from the token count

5.1.1 Custom tokenizers

Since we plan to continue with detection of clones on lower levels of granularity in the future, sooner or later we will need a custom tokenizer that understands JavaScript. The notion of a block is different in JavaScript and Python, for example.
Another reason for implementing our own tokenizer is that our initial testing of the SourcererCC code with Python tokenizer exposed some problems. One

of the main ones was performance of tokenizing minified JavaScript files that are present in our datasets. Exact numbers and details on these were mentioned in previous chapter.

We have developed and tested a JavaScript tokenizer that is built on top of Esprima library [71]. This tokenizer is available on GitHub at [72] and has already been offered to the authors of SourcererCC for adding it to their official repository.

After detecting clones on the small *top*1000 dataset we have found out that scaling it to hundreds of thousands or million projects will not be feasible.

We decided to rewrite the tokenizer one more time in C, collect the data on the fly and detect $file-hash$ and $token-hash$ clones by the tokenizer already. This way the tokenizer will be even faster and large chunk of the dataset will be filtered out after the first preprocessing. We call this the *reactor* tokenizer, its implementation was not part of this work so the detailed description is available in appendix. We have used this tokenizer to run clone detection on top of the two largest datasets (1M and 2M).

5.2 Clone detection pipeline

We have our datasets from Chapter 3 and we do preprocessing with a tokenizer. During this phase, the tokenizer generates various information as side effects of file tokenization. We capture numbers of

- tokens
- unique tokens
- lines and bytes of code
- lines and bytes of comment code
- whitespace lines and bytes

After writing the statistics on the side, the main output of the tokenizer goes to SourcererCC. SourcererCC reports clone groups that it has found. We load the results into the database and analyze.

5.3 Results

The first iteration of our clone detection was run with our custom JS tokenizer on the *top*1000 dataset. The results showed that there over 90% of cloned files

with 70% threshold. The $70k$ dataset had even higher number of clones.

We explain the high number of clones in the $70k$ dataset by having the forked repositories in it. We have not filtered the forks from the GHTorrent source and we chose random 70000 projects from all 5008554 projects there. Since there is 45% of projects that are cloned and additional 11% are moved, deleted or made private since the GHTorrent snapshot was taken, there is over 50% chance of project being a fork there. Chapter two explains this dataset and the numbers more thoroughly. After the $70k$ analysis we have decided to remove forks in $1M$ and $2M$ datasets.

When exploring the largest project clones we found out that many of them were forks of a *cdnjs* repository that stores tens of thousands JavaScript libraries. The original repository is available at `https://github.com/cdnjs/cdnjs` and the project still works as a CDN for front-end development at `https://cdnjs.com`.

After cherry-picking some files from the clone groups in results from the $70k$ dataset we have found that the most cloned file — $is-implemented.js$ — occurs in 11346 files spread across 114 projects. Now, there is already an emerging pattern. The file, and it is not only the most cloned file, but several others as well, occur in the same project multiple times. This should not happen in a project. And if so, it should not happen at this scale — so many times and in so many different projects. After we had gathered results for the 1M dataset we noticed that there is a large amount of library code from NPM contained directly in a lot of repositories. We did not notice this right after analysing the $70k$ results. There is one more indication that this was happening and it is the average and median number of files in a project. On average there was supposed to be $261,9$ files inside a project while the median was only 15.

We have collected the results for the 1M dataset as well. This was done on different machines and we have fully utilised the Reactor tokenizer. In over 41 billion files spread across almost 1 million JavaScript projects we found 6.79% file-hash unique files, 5.86% token-hash unique files and 4.46% files that have less than 70% similarity in tokens with other files.

There are several reasons why the cloning is so high in our datasets. First and foremost, there is large amount of people that commit the library or framework dependencies right into their Git repository. This is discouraged in the JavaScript community and very likely in others as well. This leads to a presence of *node_modules* or *bower_components* directories in the roots of a lot of repositories. To be exact, this amounts to over 70% of the files in our $1M$ dataset. The reasons why this is the cause of such a high number of clones

Data	Projects	Files	File-equals	Hash-equals	SCC- equals
top1000	1000	84433	86%	87.5%	90%
70k	72325	18940371	90,5%	91,9%	95.2%
1M	916082	41652400	93.2	94.1	95.5

Table 5.1: Final clone-detection results for $top1000$, $70k$ and $1M$ datasets

will be explained in following subsection. Another share of the $file - hash$ clones is contained in copied minified library files that are also committed directly into the repositories.

5.3.1 Hardware

The whole $top1000$ analysis took us 36 hours with a tokenizer that was not optimized for heavy loads yet. The analysis was run on a quad-core laptop with 16GB of RAM.

The tokenization of the $70k$ dataset took us one day and the SourcererCC clone detection ran for four days on an Amazon machine with 64GB of RAM and 16 CPU cores.

The tokenization of the 1M dataset ran on a desktop PC with 32 GB of RAM and 3TB of disk. It took us 15 days. The SourcererCC part was run on even larger machine with 112 CPU cores and 252GB of memory and we needed 3 days for this to finish.

Conclusion

We have collected a number of new datasets for analysing the JavaScript and NPM ecosystem — a dataset with top 1000 most starred JavaScript repositories from GitHub together with its metadata, dataset with random 70000 projects from GitHub that is good for testing analyses and getting a glimpse of the data that is present on GitHub, and finally a dataset with 1 million projects with their metadata.

We have calculated surprising code duplication across the whole GitHub. 95% of the JavaScript files on GitHub have a clone somewhere else. The reasons for such a high number of clones are mostly developers' unfamiliarity with Git or the module ecosystem. They tend to copy and commit a lot of libraries and frameworks directly into the source-code version control systems. This is discouraged in most of the module managers for other programming languages and it is discouraged in JavaScript as well [50].

This work has been part of a wider research focused on GitHub's code duplicates. The authors of SourcererCC [37] are building on their previous work expanding the clone detection for C/C++, Java, and Python ecosystems. Some of the datasets we have created will be used in Darpa's MUSE program [17]. This work is also a part of JavaScript initiative within Programming Research Laboratory at CTU with long-term goal of improving JavaScript tooling and runtime by understanding JavaScript BigCode.

In the future we will collect the clone detection results for all JavaScript projects on GitHub and compare them with C/C++, Java, and Python results. We also plan to look deeper into history of Git repositories and search for more complex properties of the clones. We are already gathering results for occurrences of snippets from our StackOverflow dataset in JavaScript files from GitHub. We have described the StackOverflow dataset in Chapter 3. Our future work also includes the unfulfilled proposals from Chapter 2: asserting the quality of JavaScript modules and enhancing the whole infrastructure for

"live" analysis. Eventually we will be trying to accept or reject the hypothesis that machine learning might enable new opportunities in the area of program analysis for development, security or runtime performance enhancements in a similar way that [4, 18, 19, 20] do.

Bibliography

[1] Hassan, A. E.; Holt, R. C.; et al. *International Workshop on Mining Software Repositories*. Proceedings of the 26th International Conference on Software Engineering (ICSE'04), 2004.

[2] Rysselberghe, F. V.; Demeyer, S. *Mining Version Control Systems for FACs*. International Workshop on Mining Software Repositories, 2004.

[3] MUSE Envisions Mining "Big Code" to Improve Software Reliability and Construction. `http://www.darpa.mil/news-events/2014-03-06a`, accessed: 2017-02-10.

[4] Raychev, V.; Vechev, M.; et al. *Predicting Program Properties from "Big Code"*. 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2015.

[5] Cavanillas, J. M.; Curry, E.; et al. *New Horizons for a Data-Driven Economy*. SpringerOpen, 2016, ISBN 978-3-319-21569-3.

[6] TIOBE Index for January 2017. `http://www.tiobe.com/tiobe-index/`, accessed: 2017-02-10.

[7] PYPL PopularitY of Programming Language. `http://pypl.github.io/PYPL.html`, accessed: 2017-02-10.

[8] GitHut. `http://githut.info`, accessed: 2017-02-10.

[9] StackOverflow Developer Survey Results 2016. `http://stackoverflow.com/research/developer-survey-2016`, accessed: 2017-02-10.

[10] Modulecounts. `http://www.modulecounts.com`, accessed: 2017-02-10.

[11] Wittern, E.; Suter, P.; et al. *A look at the dynamics of the JavaScript package ecosystem*. Proceedings of the 13th International Conference on Mining Software, 2016.

[12] 400,000 GitHub repositories, 1 billion files, 14 terabytes of code: Spaces or Tabs? https://medium.com/@hoffa/400-000-github-repositories-1-billion-files-14-terabytes-of-code-spaces-or-tabs-7cfe0b5dd7fd#.3apqgmxqg, accessed: 2017-02-10.

[13] Static JavaScript code analysis inside a SQL query: JSHint + GitHub + BigQuery. https://medium.com/google-cloud/static-javascript-code-analysis-within-bigquery-ed0e3011732c#.pb2dhvg9b, accessed: 2017-02-10.

[14] Imports in Java from 2013 to 2016: Winners and losers. https://medium.com/google-cloud/imports-in-java-from-2013-to-2016-winners-and-losers-2a9640056022#.kbq4thdum, accessed: 2017-02-10.

[15] Top angular directives on github, including custom directives. https://kozikow.com/2016/07/01/top-angular-directives-on-github/, accessed: 2017-02-10.

[16] Analyzing Go code with BigQuery. https://medium.com/google-cloud/analyzing-go-code-with-bigquery-485c70c3b451#.3moova43k, accessed: 2017-02-10.

[17] Mining and Understanding Software Enclaves (MUSE). http://www.darpa.mil/program/mining-and-understanding-software-enclaves, accessed: 2017-02-10.

[18] Sidiroglou-Douskos, S.; Lahtinen, E.; et al. *Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications*. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015.

[19] Long, F.; Rinard, M. *Automatic Patch Generation by Learning Correct Code*. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016.

[20] Mining and Understanding Bug Fixes for App-Framework Protocol Defects. http://www.cs.colorado.edu/~bec/talks/fixr-feb2016-techslides.pdf, accessed: 2017-02-10.

[21] Stack Exchange Data Dump. https://archive.org/details/stackexchange, accessed: 2017-02-10.

[22] GitHub Archive. https://www.githubarchive.org, accessed: 2017-02-10.

[23] GitHub V3 API: Event Types and Payloads. https://developer.github.com/v3/activity/events/types/, accessed: 2017-02-10.

[24] Gousios, G. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, Piscataway, NJ, USA: IEEE Press, 2013, ISBN 978-1-4673-2936-1, pp. 233–236. Available from: http://dl.acm.org/citation.cfm?id=2487085.2487132

[25] About the Boa Language and Infrastructure. http://boa.cs.iastate.edu, accessed: 2017-02-10.

[26] Dyer, R.; Nguyen, H. A.; et al. *Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories*. Proceedings of the 2013 International Conference on Software Engineering, 2013.

[27] NPMS About. https://npms.io/about, accessed: 2017-02-10.

[28] GitHub V3 API: Rate Limit. https://developer.github.com/v3/rate_limit/, accessed: 2017-02-10.

[29] NPMS Architecture. https://github.com/npms-io/npms-analyzer/blob/master/docs/architecture.md, accessed: 2017-02-10.

[30] What is npm? https://docs.npmjs.com/getting-started/what-is-npm, accessed: 2017-02-10.

[31] GitHub V3 API: Forks. https://help.github.com/articles/fork-a-repo/, accessed: 2017-02-10.

[32] Blink: A rendering engine for the Chromium project. https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html, accessed: 2017-02-10.

[33] Why io.js decided to fork Node.js. http://www.infoworld.com/article/2855057/application-development/why-iojs-decided-to-fork-nodejs.html, accessed: 2017-02-10.

[34] io.js FAQ. https://iojs.org/en/faq.html, accessed: 2017-02-10.

[35] Node.js Foundation Combines Node.js and io.js Into Single Codebase in New Release. https://nodejs.org/en/blog/announcements/foundation-v4-announce/, accessed: 2017-02-10.

[36] Stack Exchange API v2.2. https://api.stackexchange.com/docs, accessed: 2017-02-10.

[37] Sajnani, H.; Saini, V.; et al. *SourcererCC: Scaling Code Clone Detection to Big Code*. Proceedings of the 38th International Conference on Software Engineering, 2016.

[38] GitHub V3 API. https://developer.github.com/v3, accessed: 2017-02-10.

[39] GitHub V3 API: Search. https://developer.github.com/v3/search/#rate-limit, accessed: 2017-02-10.

[40] NPM by numbers. http://npmbynumbers.bocoup.com, accessed: 2017-02-10.

[41] NPM Stats. http://www.npm-stats.com, accessed: 2017-02-10.

[42] NPM Stat. http://www.npm-stat.com, accessed: 2017-02-10.

[43] npm Discover. http://www.npmdiscover.com, accessed: 2017-02-10.

[44] all-package-metadata.js. https://git.io/vM8x1, accessed: 2017-02-10.

[45] New npm Registry Architecture. http://blog.npmjs.org/post/75707294465/new-npm-registry-architecture, accessed: 2017-02-10.

[46] Specifics of npm's package.json handling. https://docs.npmjs.com/files/package.json, accessed: 2017-02-10.

[47] Apache CouchDB 2.0 Documentation: HTTP API Reference. http://docs.couchdb.org/en/2.0.0/http-api.html, accessed: 2017-02-10.

[48] 1m.js. https://git.io/vM4YO, accessed: 2017-02-10.

[49] 1.full.js. https://git.io/vM4ip, accessed: 2017-02-10.

[50] Things Every Node Developer Should Add to Their .gitignore. https://chaseadams.io/2015/07/my-gitignore-conventions/, accessed: 2017-02-10.

[51] Lock down dependency versions. https://docs.npmjs.com/cli/shrinkwrap, accessed: 2017-02-10.

[52] yarn.lock. https://yarnpkg.com/en/docs/yarn-lock, accessed: 2017-02-10.

[53] FreeCodeCamp/FreeCodeCamp. https://github.com/FreeCodeCamp/FreeCodeCamp, accessed: 2017-02-10.

[54] angular/angular.js. https://github.com/angular/angular.js, accessed: 2017-02-10.

[55] GNU General Public License. https://www.gnu.org/licenses/gpl.html, accessed: 2017-02-10.

[56] GNU General Public License, version 2. https://www.gnu.org/licenses/old-licenses/gpl-2.0.html, accessed: 2017-02-10.

[57] GNU Lesser General Public License. https://www.gnu.org/licenses/lgpl-3.0.en.html, accessed: 2017-02-10.

[58] GNU Affero General Public License. https://www.gnu.org/licenses/agpl-3.0.en.html, accessed: 2017-02-10.

[59] GNU Library General Public License, version 2.0. https://www.gnu.org/licenses/old-licenses/lgpl-2.0.html, accessed: 2017-02-10.

[60] GNU Lesser General Public License, version 2.1. https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html, accessed: 2017-02-10.

[61] AFFERO GENERAL PUBLIC LICENSE. http://www.affero.org/oagpl.html, accessed: 2017-02-10.

[62] AFFERO GENERAL PUBLIC LICENSE, Version 2. http://www.affero.org/agpl2.html, accessed: 2017-02-10.

[63] About The Licenses. https://creativecommons.org/licenses/, accessed: 2017-02-10.

[64] Do What the Fuck You Want to Public License. http://www.wtfpl.net/about/, accessed: 2017-02-10.

[65] Open source license usage on GitHub.com. https://github.com/blog/1964-open-source-license-usage-on-github-com, accessed: 2017-02-10.

[66] GitHub: facebook/react. https://github.com/facebook/react, accessed: 2017-02-10.

[67] The Universal Makefile for JavaScript. http://kvz.io/blog/2016/02/18/a-universal-makefile-for-javascript/, accessed: 2017-02-10.

[68] Gulp vs Grunt. Why one? Why the Other? https://medium.com/@preslavrachev/gulp-vs-grunt-why-one-why-the-other-f5d3b398edc4#.rd8fuu7vr, accessed: 2017-02-10.

[69] Sajnani, H.; Lopes, C. *A Parallel and Efficient Approach to Large Scale Clone Detection*. 7th International Workshop on Software Clones (IWSC), 2013.

[70] GitHub: Mondego/SourcererCC. https://github.com/Mondego/SourcererCC, accessed: 2017-02-10.

[71] Esprima. http://esprima.org/doc/index.html, accessed: 2017-02-10.

[72] GitHub: jakubzitny/SourcererCC. https://git.io/vMWX1, accessed: 2017-02-10.

Appendix A

Acronyms

AGPL Affero General Public License

API Application Programming Interface

CC Creative Commons

CDN Content Delivery Network

CLI Command Line Interface

FAC Frequently Applied Change

GPL General Public License

LGPL Lesser General Public License

LOC Lines of Code

MIT MIT License, originating at the Massachusetts Institute of Technology

MSR International Conference on Mining Software Repositories

MUSE Mining and Understanding Software Enclaves

NPM Node Package Manager

NSP Node Security Platform

VCS Version Control System

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `big.js` ..... implementation scripts
  - `SourcererCC` ..... SourcererCC sources with our tokenizers
  - `js-reports` ..... R scripts for generating summaries and charts
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format