Validating the Quality of a Big Data Java Corpus
Abstract

Recent research within the field of Software Engineering have used GitHub, the largest hub for open source projects with almost 20 million users and 57 million repositories, to mine large amounts of source code to get more trustworthy results when developing machine and deep learning models. Mining GitHub comes with many challenges since the dataset is large and the data does not only contain quality software projects. In this project, we try to mine projects from GitHub based on earlier research by others and try to validate the quality by comparing the projects with a small subset of quality projects with the help of software complexity metrics.

**Keywords:** mining software repositories, GitHub, GHTorrent, Chidamber & Kemerer metrics, software complexity
Contents

1 Introduction 4
   1.1 Background 4
   1.2 Related work 4
   1.3 Problem formulation 6
   1.4 Motivation 6
   1.5 Objectives 7
   1.6 Scope/Limitation 7
   1.7 Target group 7
   1.8 Outline 8

2 Method 9
   2.1 Mining Software Repositories 9
   2.2 Validate the Quality 9
   2.3 Reliability and Validity 10
   2.4 Ethical Considerations 11

3 Implementation 11
   3.1 Mining Software Repositories 13
   3.2 Software Metrics 15

4 Results 17
   4.1 Literature Review 18
   4.2 The Java Corpus 20

5 Analysis 26
   5.1 Mining Requirements 26
   5.2 The Java Corpus 27

6 Discussion 28

7 Conclusion 29
   7.1 Future work 30

References 31

A Appendix - Logs From Clustering 34

B Appendix - Random Project Sample 38
1 Introduction

In this project, we create a big data java corpus of real-world open source software projects and validate its quality by calculating complexity metrics and compare them with metrics for a smaller set of high-quality projects. To collect a significant amount of projects GitHub, a web-hosted service that lets people host open-source software projects, will be used for mining projects to the corpus.

1.1 Background

Recent research within the field of Software Engineering has applied machine learning and other statistical code metrics on source code from real-world software projects. A lot of these metrics and models require large quantities of source code to give good and trustworthy results [1]. One way to access large amounts of data about software projects is mining it from services hosting open-source projects like GitHub.

GitHub is today the largest hub for open source projects with almost 20 million users and 57 million repositories. Factors like that users can host projects for free, and its social and collaboration features are a part of GitHub’s success [2]. These reasons make it a good source for mining data and source code to be used for research.

Due to GitHub’s rate limit for their API. New solutions for mining data have been created such as GHTorrent, a queryable mirror of data that can be accessed by the GitHub Rest API [3], and GitHub Archive, a record of events from GitHub’s public timeline [1], to better be able to mine large quantities of data.

Software metrics are widely used within Computer Science to see whether a system or parts of a system possesses specific properties to determine the quality, and one of those areas are software complexity [4]. Software complexity can be measured in different ways, and one suite of metrics that are widely used is the Chidamber and Kemerer metrics [5] that look at several measurements and properties of a class in object-oriented languages. The metrics that will be used in this project are explained in detail in chapter 3.2.

1.2 Related work

Most of the related work focuses on the collection process when mining
GitHub, use deep learning to find patterns in source code or investigating the usage of libraries and frameworks. There is no article except Munaiahl et al. [7] that actually verifies the quality of the collected source code from GitHub even though some articles raises issues and solution on how to mine data from GitHub with the repositories metadata.

There's been some different approaches and techniques when collecting data from GitHub. One example is Allamanis and Sutton [1] who used the GitHub Archive, a record of the GitHub timeline, to create a corpus of software projects. Also, Gousious and Spinellis [3] have done work creating GHTorrent an offline mirror based on data of the GitHub REST API for making it easier for other researchers to use GitHub data for research.

Work has also been done in identifying how to distinguish real projects from storage, student assignments, and noise when mining GitHub like Kalliamvakou et al. [2] identifies critical perils when mining data and how to avoid them. Cosentino et al. [6] studied 93 research papers that mined software projects and evaluated their methods, datasets, and limitations found. Munaiahl et al. [7] implemented a program called Reaper to find quality software projects from repositories mined from GitHub. The authors implemented several properties to validate quality regarding the presence of unit tests, continuous integration and other metadata from the projects but still had trouble removing noise in their collected corpus. The difference between their approach and the one discussed in this paper is that this approach tries to measure quality on the actual source code.

A lot of studies that have leveraged a big data corpus of source code have applied deep learning methods for their research. Allamanis and Sutton [1,8] and White et al. [9] have used corpora of Java projects to create language models of source code to find patterns in all the projects. Similar research has been done by Gabel and Su [10] and Nguyen and Rajan [11] who have used large corpora of source code to investigate uniqueness and repetitiveness in projects. Barone and Sennrich [12] have tried to use data gathered in a corpus of Java projects for automated code generation and documentation and Keivanloo et al. [13] to try to detect working code examples.

Other studies have used the source code together with the commit history to determine the usage of frameworks and libraries throughout the lifecycle of a project to gain better knowledge which libraries that work well together and which libraries developers tend to migrate between. For example
Zerouali and Mens [14] that investigated the use of testing libraries in Java over several projects lifetime and also by Goeminne and Mens [15] that did a similar investigation but for database frameworks.

1.3 Problem formulation
The goal of this project is to create a big data corpus that contains Java software projects, validate its quality by measuring software complexity for a set of quality projects and classify how good quality the corpus has by comparing the quality projects with projects in the collected corpus.

Mining data from GitHub is not a trivial task since a suitable algorithm is needed to distinguish real software projects from all the noise like repositories on GitHub that might be used for school assignments or code examples [2]. There is also need for an automated way to validate the quality for a collected corpus since the size makes it not feasible to manually validate the projects.

Another challenge is to calculate software metrics for projects on such a large scale since most tools have to compile the source code before they can calculate the metrics which is not feasible on this scale. A tool is needed that give good results on a static analysis of the source code without the need to compile it.

1.4 Motivation
Being able to use a large amount of source code opens up exciting possibilities to find patterns and trends in software development. Results gathered can be used for creating good software development practices [1].

There are also more and more researchers that turn to GitHub for doing research [6], and therefore tools that can mine quality software projects with as little noise as possible enable more researchers to get valid datasets. Most of the tools today focuses on the GitHub dataset and don’t have an easily accessible interface for getting the source code from a large set of projects.

The corpora that are available today have not proven quality like the Allamanis and Sutton corpus [2] (14,000 projects) or have a verified quality but a too small dataset like the Qualitas Corpus [16] (108 projects) to be used for machine learning and natural language processing.
1.5 Objectives
These are the main objectives of the project.

| O1 | Implement a tool for mining and storing data from GitHub |
| O2 | Mine the source code |
| O3 | Implement library/tool for measuring software metrics |
| O4 | Compute software metrics for all projects in corpus |
| O5 | Compute software metrics for Qualitas Corpus |
| O6 | Measure quality by comparing corpus with Qualitas Corpus |
| O7 | Analyze the results |

After the project is completed, we expect to have a corpus with approximately 10,000-20,000 Java projects. We also expect to have a measurement of the quality of the corpus.

1.6 Scope/Limitation
Source code will be collected from GitHub only and not several code hosting services due to the time limitation of this project. The reason for choosing GitHub is that it has the most extensive set of software projects.

A limitation for the data collection is that we only can mine open source projects that are publicly available on GitHub, this might affect the type of projects in the corpus because it might contain more tools and frameworks rather than software applications. GitHub is also used for other things such as storage of files and student tasks and to avoid them in the corpus its needed to include only popular projects that have been developed for a certain amount of time. For this corpus projects that use the Java programming language is only of interest.

Though there are other ways of measure quality on software projects, this research will focus only on the complexity metrics by Chidamber & Kemerer since they are widely used in software development and there are several implementations that can be used.

1.7 Target group
The main target group for this project is researchers that need a big data corpus for researching the field of machine learning and natural language
processing on source code.

Other people that are performing research related to the process of mining software repositories might also be interested in the tool or process of collecting repositories for further improving it.

1.8 Outline
In the next chapter, method, there is an explanation of how to find requirements to verify that we create an application that can mine quality Java projects as well as how to validate the quality of the collected corpus and concerns regarding reliability and validity.

The following chapter, implementation, go more into detail on how the application is built, problems that occurred when mining source code, how the different metrics are computed, which tools that are used for computing metrics and how the clustering was performed.

The implementation chapter is then followed by the result, analysis, discussion, and conclusion chapters where the results will be presented and discussed as well as what can be done in the future with the knowledge gained from this project.
2 Method

To collect a set of quality software projects from GitHub it is required that we know a good but efficient way of categorizing projects so noise can be filtered away.

The complexity metrics are not enough on their own to determine if a project has good quality, so they need to be compared with the metrics from quality projects to be able to classify the projects as quality projects or not.

2.1 Mining Software Repositories

To understand the best practices when it comes to mine source code from GitHub it is essential to understand how other researchers have approached this task. It is important to understand the advantages and disadvantages of tools used for mining GitHub. It can be a limit to the number of requests to an API, available metadata of the projects and completeness.

The other part to get a better understanding of is how other researchers have tried to filter out the noise or non-quality projects and if other considerations need to be taken into account.

To be able to answer all these questions, a literature review was performed. In order to find relevant research articles, the review was based on two articles ‘Mining Software Engineering Data from GitHub’ [3] and ‘Mining Source Code Repositories at Massive Scale using Language Modeling’ [1]. With these two articles, I made a transitive search for related or articles that quoted the two articles on Google Scholar¹. To filter away non-relevant articles only articles that used a big data corpus for research or researched the process of mining software repositories was included in the review.

Based on the results of the review a set of requirements was collected that the application that collects the corpus needs to fulfill only to get quality software projects. These requirements are presented in chapter 3.2 and further explained in chapter 5.2.

2.2 Validate the Quality

To validate that the corpus that was collected consists of quality projects an algorithm for determining if a project is of high quality or not. The Chidamber and Kemerer metrics on their own do not provide guidance if the

¹ https://scholar.google.com
value is of high quality or low quality. A study by Laing and Coleman [17] manually validates projects qualities and compare the quality with the average complexity metrics calculated for each class. Their findings suggest that a high-quality project has a lower WMC, CBO, and RFC compared to a low-quality project. Software projects can, however, look different concerning complexity depending on the problem they try to solve. For example, a framework or a tool might be more complex than an application because it needs to solve more problems to fit a more extensive usage by other applications but might still be of good quality.

With this in mind, the computed metrics by Laing and Coleman was not used directly. Instead, the same metrics were calculated for the Qualitas Corpus and our Corpus. With the projects for the Qualitas Corpus, different clusters were created based on their average complexity metrics. Clustering is a way to group entities that share similar attributes together by measure how close they are to the centroid in an x-dimensional space which is good when the complexity profiles can be a lot different. The quality of our Corpus was then measured by taking each project and see if it was close to any of the clusters that were created. This provides us an indication of the quality of the corpus.

A random sampling was then performed for 100 of the projects in the corpus to validate how well the clusters classified the projects as quality or non-quality projects. This random sample was used to understand if a project was correctly classified as well as understand what type of project it was since example code or student assignments are not real projects.

### 2.3 Reliability and Validity

Getting the same dataset twice will be hard to collect. To increase the reliability, it’s possible to record which database dump that was used from GHTorrent to collect projects metadata. It’s also possible to download the source code for the last commit in that dump, so it's more likely that the source code is the same. Even though these two steps are followed it’s possible that not the same projects are included, this is because projects on GitHub can change visibility from public to private and back or they can be archived or removed.

This is maybe not that likely to happen but to have good reliability it’s a good idea to store the source code and metadata mined so it can be used for further experiments. If the source code is stored, the analysis will always
return the same results. For this reason the source code collected will be stored in a database for 6 months and then be archived.

Mined data from GitHub might risk having some validity issues. It’s important to make sure that there are no duplicate projects in the dataset, this can be done by matching commit ids to figure out if a project has duplicates. Other noise also needs to be filtered such as student assignments and storage of code examples.

Another risk with the dataset mined from GitHub is that it only contains open-source projects so findings on this data might not be able to apply to other domains in the software industry. Open-source projects might also be more related to tools, libraries, and frameworks rather than software applications.

2.4 Ethical Considerations

All data collected is already publicly available on GitHub. To avoid violating any user privacy no user data will be collected. Committers or contributors will be referred to as an incremental number, that can not be traced to the user, to identify number of collaborators.

Since GHTorrent is used for gathering the data this project is also affected by their privacy handling. As of Mars 2016 GHTorrent decided not to distribute any personal data by default like names and e-mail addresses [18]. There might be personal information in downloaded source code but since the source code only is used for computing metrics it will not be presented in the result or analysis.
3 Implementation

In this chapter we outline the process of mining GitHub and computing the complexity metrics. Actual results such as the number of projects downloaded, the time it took to download them, and results related to complexity metrics computations, will be presented in the next chapter.

The application developed is called Investigitor and executes a set of tasks for collecting, storing and filtering data from GHTorrent and GitHub. It also uses the tool CK [19] for calculating complexity metrics for each Java file in each project. The reason for picking this tool is that it is the only tool we found that calculates the metrics from static analysis which is crucial since compiling each project will take too much time. In the following chapters each of these tasks will be explained in more detail.

The application is written in Java 1.8 and uses a PostgreSQL 9.4 database for persisting metadata about the different projects and the metrics collected from each java file. Investigitor will not have a graphical user interface or a rest API and all the data collected will instead be queried directly from the PostgreSQL database.

When the application is started it executes a TaskRunner object that contains a number of tasks that inherit the abstract Task class. The TaskRunners job is to keep track and persist status of the different tasks and make sure they are executed in the right order even if the system goes down and, if that happens, can continue from the point where it left of.

![Figure 3.1: Class diagram over the task execution](image-url)
Figure 3.1 shows a class diagram over how the TaskRunner can have one or more Tasks and all the different task classes that inherit Task.

3.1 Mining Software Repositories

Table 3.1 shows a breakdown of each step needed to mine Java projects from GitHub followed by a more in-depth explanation of each step. The motivation of using GHTorrent to get information about the projects as well as the filtering criteria is further explained in Chapter 5.2.

<table>
<thead>
<tr>
<th>Step</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Download archived dump of database from GHTorrent.</td>
</tr>
<tr>
<td>2</td>
<td>Unarchive the database dump.</td>
</tr>
<tr>
<td>3</td>
<td>Delete the archive.</td>
</tr>
<tr>
<td>4</td>
<td>Store project information (project.csv) from database dump that has language Java in our database.</td>
</tr>
<tr>
<td>5</td>
<td>Store project to commit mapping (project_commits.csv) from database dump for each project that already is written to the database.</td>
</tr>
<tr>
<td>6</td>
<td>Store commit history from database dump for each commit that already is written to the database.</td>
</tr>
<tr>
<td>7</td>
<td>Store watchers/star information from database dump for each project that already is written to the database.</td>
</tr>
<tr>
<td>8</td>
<td>Delete database dump to clear up disk space on the server.</td>
</tr>
<tr>
<td>9</td>
<td>Filter projects that have more than one collaborator, more than one watcher/star and has been active for more than 100 days.</td>
</tr>
<tr>
<td>10</td>
<td>Remove projects that share the same commit history to avoid duplicates, keep the most popular project.</td>
</tr>
<tr>
<td>11</td>
<td>Download source code from GitHub with git clone command.</td>
</tr>
<tr>
<td>12</td>
<td>Calculate software metrics for each project.</td>
</tr>
</tbody>
</table>

*Table 3.1: Algorithm for mining GitHub.*

The first task in the application was to download a specified database dump from GHTorrent. This project used the dump from 2018-03-01. The database dump consists of several CSV files containing the data and is downloaded as an archived file.

Next step in the chain of tasks is to unarchive the contents of the archived database dump and store it in a local folder, and when completed the
archive file will be deleted.

Once that is done, the application takes the CSV files or tables that are needed to find the Java projects that we are interested in and inserts them into the PostgreSQL database. That is projects.csv that contains information such as the project name, description, programming language, create date, if the project still exists and an URL to GitHub API for that specific project; commits.csv that contains author identifiers to map out who made the commit and when the commit was done; project_commits.csv that map projects to commits since one commit can belong to several projects; watchers.csv that contain which users that starred or liked the project on GitHub.

When all the data that are of interest is stored in the database, the first filtering process starts. Here the application removes all projects that are not labeled to use Java as the programming language and all projects that are deleted. After each filtering task, all the other tables that contain a reference to a project that has been removed are removed as well.

The next step in the filtering process is to try to filter out projects that are not quality software projects with the requirements based on the results from the literature review that can be found in chapter 5.1.

Once the application has a set of Java projects that we are interested in it will start downloading, or in git terms clone, them synchronously. This is done by running the git clone command with the URL to the repository. If a repository cannot be downloaded, it will be removed from the dataset and logged so the removed projects can be tracked. For each downloaded project, we also filter out all files that do not have anything to do with the java code such as images, HTML and XML files. The reason for this is not to exceed limitations on storage in the environment.

Finally when all the source code is collected the application will start its final task which is computing the complexity metrics for each file in each project, and once that is done it does the same procedure for the Qualitas Corpus.

In the first attempts to run the application, the import tasks took too long time to complete where the import of the project commit mapping was running for more than two weeks. It was caused by three issues that both took some time to discover.

The first issue was that the PostgreSQL database was not optimal configured and did more IO tasks than required and also that the application ran on a virtual machine which had limited number of IO operations per
second. The second issue was that reading each row and bulk inserting with the INSERT command was not efficient enough; this was solved by using the COPY command which is more effective when it comes to importing data in PostgreSQL [20]. The final issue was that indexes were defined on the tables which slows down the importing, this was solved by removing indexes from the start and adding them when they were needed.

To further speed up the process of importing data were to initially filter the CSV files so they only contained projects where language was Java before they were added to the database which improved the filtering queries a lot.

3.2 Software Metrics

The first metric of the four metrics that are part of the Chidamber and Kemerer suite that was computed is Weighted Methods per Class (WMC) that is a sum of McCabe's cyclomatic complexity metric for methods and constructors of a class [21]. A higher score points to that a class is complex and might be hard to maintain. Cyclomatic complexity is a metric that was developed by Thomas J. McCabe in 1976, and it measures code complexity by looking at a program's control flow as a graph where some nodes and edges affect the complexity [22]. The complexity can be calculated as displayed in Equation 1 below.

\[ V(G) = E - N + 2 \]  

(1)

To give a better understanding in how it works a normal sequence has 2 nodes and 1 edge which results in a cyclomatic complexity of 1 and a while loop/sequence contains 3 nodes and 3 edges which gives a complexity of 2 as shown in Figure 3.2.

![Normal Sequence and While Loop](image)
One reason that might lead to a low complexity score is the use of inheritance. Therefore it's also interesting to compute Depth in Tree (DIT). A low score shows on poor reusability while a high value might affect the possibility to understand classes behavior which then increases complexity [23]. A class that inherits from Object or an interface will get a score of 1 while a class that inherits from another class will get a score of 2 and if that class also inherits from another class 3 and so on.

Coupling between object classes (CBO) is also an important metric when it comes to software complexity. It is a measure of how coupled a class is. A class is coupled if it uses or holds a reference to another class if there are too many couplings between classes it will increase the complexity because a change in one class will certainly affect several classes.

The last metric that will be computed is response for a class (RFC) which shows how many method invocations a class has. As for CBO a higher number is not wanted since it will increase the complexity.

For the collection of the metrics, the application uses the library CK [19] for parsing the Java files and calculate the metrics. Since this library only does a static analysis without compiling or downloading external resources, it gives some limitations when computing metrics for classes that use external resources (e.g. jar files). Having a library that compiles and downloads external resources would not be feasible for the time frame of this project. The limitations affect the DIT metric where inheriting a class outside of the project will result in a depth of 2 since it can look further up in the inheritance tree, but also the RFC metric will have issues with overloaded methods with the same number of arguments but different types [19]. This might be an issue if the computed metrics are compared with metrics that another tool has computed but in this case, it is compared with another corpus with the same tool.

Since complexity can look different between projects, it is not enough to compare the projects based on the average metrics for the Qualitas Corpus. Instead, clustering was applied to find groups in the data that share similar complexity profile. To create clusters from the Qualitas Corpus based on the computed metrics the application Weka² with version 3.8.2 is used. Weka can apply different algorithms when creating clusters and the one that was used was Expectation–Maximization (EM) Clustering. This algorithm has certain

² https://www.cs.waikato.ac.nz/ml/weka/
advantages over the more widely used K-means clustering algorithm. K-means need you to specify the number of clusters beforehand which requires several runs and a manual validation to find the best match while EM does not need this. In this case, this is good because it can be hard to detect the correct number of clusters in a 5-dimension space. Another benefit with EM over K-means is that it is more flexible when it comes to cluster covariance because of the extra standard deviation parameter [24].
4 Results

In this chapter, the results of the literature review are presented which is the foundation for the decision on how to filter projects which are further described in Chapter 5.1. After the literature review, the results from the mining and the attempt to validate the corpus quality is presented as well as an attempt to further classify the non-quality projects found in the corpus.

4.1 Literature Review

The literature review was conducted on the 28th of December 2017 for the article ‘Mining Software Engineering Data from GitHub’ [3] and on 4th of January 2018 for the article ‘Mining Source Code Repositories at Massive Scale using Language Modeling’ [1].

In total 14 research articles were found and studied. A summary of each article can be found in Chapter 1.2. Of these 14 articles, 5 was focused on the processing of mining data from GitHub, 2 mined GitHub and created a corpus, and the rest used an existing corpus like the corpus create by Allamanis and Sutton [1].

Allamanis and Sutton used the GitHub Archive approach to find metadata about the projects and filtered away all projects that had not been forked before and manually removed duplicate projects that shared the same commit messages not to get duplicated source code.

Munaiah et al. [7] developed a more extensive tool for mining data from Github using GHTorrent. They looked at several properties when training and testing their classification. The properties were number of collaborators, the presence of a continuous integration tool, documentation, commit history, number of issues, licensing and presence of unit tests. They, however, conclude that their application still contained noise and that a more accurate way was to look at the number of watchers or stargazers which is an indication of how popular a project is.

All articles that mined source code from GitHub used the git client and the clone command to download the repositories.

An article written by Cosentino et al. [6] took the approach of studying others research when mining software repositories and 93 papers were included in the research. Below in figure 4.1 is an overview of the most used tools. They also reported that half of the researchers that have done data collection experienced issues with GitHub API due to the quoted limit rate
and also noted that the collection methods that relied on GHTorrent, BOA and GitHub Archive is not up to date due to the fact that they are a mirror of GitHub. GitHub Search API was only used for research on users and social aspects of GitHub.

![Figure 4.1: Percentage of used tools when mining GitHub in 93 research articles.](image)

Kalliamvakou et al. [2] studied the promises and perils of mining GitHub by using the GHTorrent tool and as a result created 10 perils to have in mind when mining GitHub. The perils related to mining software projects and not GitHub's social aspects was a repository is not necessarily a project, most projects have very few commits, a large portion of repositories are not for software development, and two-thirds of projects (71.6% of repositories) are personal. They also provide some strategies to handle these perils. It is good to check collaborators to avoid personal projects, a project can exist in several repositories so it is crucial to keep track of the base and the forked repositories and keep track of the commit history for inactive and projects with few commits.

The approach for collecting the Java corpus was mainly based on the findings by Kalliamvakou et al. but also take the same assumption as Allamanis and Sutton that high-quality projects should be more popular than non-quality projects, the exact criteria for the filtering process is described in
Chapter 5.1.

4.2 The Java Corpus

In total 21,951 projects was collected which contained 3,959,727 files and 555,294,293 lines of code. Of the 3,959,727 files, there was 3,482,067 classes, 413,286 interfaces, and 64,374 enumerations. The corpus is bigger than Allamanis and Sutton's corpus that contained 14,807 projects which contained 2,130,264 files and 352,312,696 lines of code. Below in table 4.1 is a summary of how long time each step in the process took was the most time-consuming tasks were to calculate metrics and download all the source code from GitHub.

<table>
<thead>
<tr>
<th>Step</th>
<th>Task</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DownloadDatabaseDumpTask</td>
<td>1 h 2 min</td>
</tr>
<tr>
<td>2</td>
<td>ExtractDatabaseDumpTask</td>
<td>2 h 3 min</td>
</tr>
<tr>
<td>3</td>
<td>DeleteDatabaseDumpArchiveTask</td>
<td>&gt; 1 min</td>
</tr>
<tr>
<td>4</td>
<td>ImportProjectsTask</td>
<td>7 min</td>
</tr>
<tr>
<td>5</td>
<td>ImportProjectCommitsTask</td>
<td>55 min</td>
</tr>
<tr>
<td>6</td>
<td>ImportCommitsTask</td>
<td>5 min</td>
</tr>
<tr>
<td>7</td>
<td>ImportWatchersTask</td>
<td>2 min</td>
</tr>
<tr>
<td>8</td>
<td>FilterQualityProjectsTask</td>
<td>6 min</td>
</tr>
<tr>
<td>9</td>
<td>FilterDuplicateProjectsTask</td>
<td>16 min</td>
</tr>
<tr>
<td>10</td>
<td>DeleteDatabaseDumpTask</td>
<td>&gt; 1 min</td>
</tr>
<tr>
<td>11</td>
<td>DownloadSourceCodeTask</td>
<td>40 h 36 min</td>
</tr>
<tr>
<td>12</td>
<td>CollectSoftwareMetricsTask</td>
<td>105 h 4 min</td>
</tr>
</tbody>
</table>

*Table 4.1: Time each step took in the mining process.*

Table 4.2 is a summary of the average metrics for all projects both for the collected corpus and for the Qualitas Corpus that was introduced in Chapter 1.4 with its 105 high-quality projects.
<table>
<thead>
<tr>
<th>Type</th>
<th>Qualitas Corpus</th>
<th>Our Corpus</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>32.66 (20.29)</td>
<td>23.7 (40.17)</td>
</tr>
<tr>
<td>DIT</td>
<td>2.34 (0.65)</td>
<td>1.73 (0.71)</td>
</tr>
<tr>
<td>CBO</td>
<td>5.92 (1.91)</td>
<td>6.80 (4.08)</td>
</tr>
<tr>
<td>RFC</td>
<td>18.02 (6.92)</td>
<td>16.46 (12.33)</td>
</tr>
<tr>
<td>LOC</td>
<td>186.99 (75.10)</td>
<td>132.75 (220.76)</td>
</tr>
</tbody>
</table>

*Table 4.2: Average metrics and standard deviation in parenthesis for all projects in each corpus.*

The EM clustering algorithm found 4 clusters based on the projects average complexity metrics that was included in the Qualitas Corpus. Below in Table 4.3, there is an overview of the share of projects that were included in each corpus together with the centroids center points for each metric as well as the standard deviation for each metric. The WMC, CBO and RFC metric seems to increase with the average number of LOC.

<table>
<thead>
<tr>
<th>Type</th>
<th>Cluster 0</th>
<th>Cluster 1</th>
<th>Cluster 2</th>
<th>Cluster 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share</td>
<td>40 % (40.0)</td>
<td>12 % (69.0)</td>
<td>18 % (33.9)</td>
<td>3 % (2.5)</td>
</tr>
<tr>
<td>WMC</td>
<td>29.00 (4.94)</td>
<td>69.02 (33.94)</td>
<td>42.65 (8.51)</td>
<td>17.34 (4.40)</td>
</tr>
<tr>
<td>DIT</td>
<td>2.53 (0.74)</td>
<td>2.79 (0.73)</td>
<td>2.09 (0.36)</td>
<td>2.05 (0.37)</td>
</tr>
<tr>
<td>CBO</td>
<td>5.60 (1.69)</td>
<td>7.57 (2.53)</td>
<td>6.54 (2.12)</td>
<td>5.32 (1.13)</td>
</tr>
<tr>
<td>RFC</td>
<td>17.64 (4.26)</td>
<td>30.51 (5.62)</td>
<td>20.69 (4.83)</td>
<td>12.04 (2.72)</td>
</tr>
<tr>
<td>LOC</td>
<td>177.13 (27.10)</td>
<td>304.25 (77.46)</td>
<td>250.55 (47.42)</td>
<td>115.67 (32.40)</td>
</tr>
</tbody>
</table>

*Table 4.3: Table over the final cluster centroids for the 4 created clusters with the standard deviation in the parenthesis.*
Figure 4.2: Percentage of how the projects, categorized as domains, got distributed over the different clusters.

Figure 4.2 shows the distribution of projects by their domain to verify if the assumption that complexity can be similar for different types of domains, however, the results show that they are spread over the clusters which make the assumption not valid.
Figure 4.3: Percentage of how the projects, categorized by their Java version, got distributed over the different clusters.

Figure 4.3, similar to Figure 4.2, tries to see the correlation between the clusters and the projects Java versions, there seems to be no correlation between them whatsoever.

To compare each project in the corpus with the cluster a radius around a cluster centroid needed to be set. Using 2*std as cluster radius makes sense if you assume that metrics are normally distributed around their average value. In that case, 2*std will cover 95.4% of all cases [25]. Hence, approximating the cluster as ellipsoids with radius 2*std are likely to capture a majority of the cases. This hypothesis was verified by assigning each project in the Quality Corpus a cluster using this approach and found that 91% of projects ended up in their original cluster which is good enough.

After each project in the corpus was compared with each cluster a total of 11,296 of 21,951 projects matched at least one cluster which gives a quality rate of 51.5% and 10,655 projects that were considered not similar enough to the projects in the Qualitas Corpus.

After that, the random sampling was performed for 100 projects by selecting a random row in the project table and evaluating the information about the project as well as the properties collected. This was done to validate how well a corpus quality can be described by comparing the projects with high-quality projects. Each project also got categorized into what type of project it is. The complete list of the random sampling can be found in Appendix B. The margin of error for 100 projects with a confidence level of 95% is approximately 10%. Table 4.4 below gives an overview of how well projects were classified as quality or non-quality projects. In total it classified 79-99% correct and it seems to be more likely that a quality project will get falsely classified as a non-quality project than a non-quality project get falsely classified as a quality project which is good since we want to reduce the noise.

<table>
<thead>
<tr>
<th>Total</th>
<th>Non-Quality</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>89.0%</td>
<td>80.4%</td>
<td>96.3%</td>
</tr>
</tbody>
</table>

Table 4.4: Percentage of correctly classified projects in total, non-quality and quality (margin of error 10%).
Figure 4.4: Percentage of correctly classified projects in total, non-quality and quality for each category.

In figure 4.4 there is a breakdown of the random sampling classification divided in the type of project. Here we see that applications, databases, testing related tools, code examples and other tools/libraries were easy to classify correctly based on the complexity metrics. Plugins were not because the could have lower complexity than the high-quality projects which mainly is because plugins are usually small projects that solve one thing.

Of the correctly classified non-quality projects three patterns were found to why they were non-quality projects. Some projects had overall high WMC, CBO and RFC which indicated a very high complexity, other projects had just a high CBO which indicates that the classes in the project are very coupled which is not good and finally code examples, that should not be considered as real projects by their simplified way of describing how to use tools and libraries, had a very low complexity together with a low LOC.

To verify these patterns all the correctly classified non-quality projects was used to create new clusters with the same algorithm as for the clustering of the Qualitas Corpus. Three new clusters were then created which verified the patterns. The characteristics for cluster 4 is that it has high complexity overall, cluster 5 has low complexity with the majority of projects as code examples, and cluster 6 has an overall high coupling between objects.
(CBO). The clusters properties can be seen below in Table 4.5.

<table>
<thead>
<tr>
<th>Type</th>
<th>Cluster 4</th>
<th>Cluster 5</th>
<th>Cluster 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share</td>
<td>19 %</td>
<td>56 %</td>
<td>24 %</td>
</tr>
<tr>
<td>WMC</td>
<td>74.39 (36.12)</td>
<td>8.63 (5.68)</td>
<td>25.89 (10.57)</td>
</tr>
<tr>
<td>DIT</td>
<td>2.27 (0.52)</td>
<td>1.31 (0.35)</td>
<td>2.87 (2.12)</td>
</tr>
<tr>
<td>CBO</td>
<td>16.05 (2.20)</td>
<td>4.85 (2.64)</td>
<td>13.42 (2.61)</td>
</tr>
<tr>
<td>RFC</td>
<td>47.45 (8.19)</td>
<td>7.64 (4.40)</td>
<td>25.00 (8.01)</td>
</tr>
<tr>
<td>LOC</td>
<td>333.73 (130.50)</td>
<td>54.06 (25.18)</td>
<td>149.84 (46.09)</td>
</tr>
</tbody>
</table>

*Table 4.5: Table over cluster centroids with standard deviation.*

When classifying the projects in the corpus against the old and the new clusters a label was added to divide the results into groups of what earlier results indicated. Matching against Cluster 0-3 is considered as quality projects, cluster 4 and 6 are considered as non-quality projects, cluster 5 is considered to be low complexity examples and non-matching projects are considered as projects that we can not estimate the quality of. The results can be found below in Table 4.6.

<table>
<thead>
<tr>
<th>Label</th>
<th>Number of Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality projects</td>
<td>11,296 (51.5 %)</td>
</tr>
<tr>
<td>Non-quality projects</td>
<td>2,785 (12.7 %)</td>
</tr>
<tr>
<td>Low complexity (coding examples)</td>
<td>5,253 (23.9 %)</td>
</tr>
<tr>
<td>Other (not able to match cluster)</td>
<td>2,617 (11.9 %)</td>
</tr>
</tbody>
</table>

*Table 4.6: Number of projects for each classification label.*
5 Analysis

In this chapter, the mining requirements and the reason for picking them based on the literature review are presented following an analysis of the created corpus, the quality of the corpus and how well the cluster worked when classifying the corpus.

5.1 Mining Requirements

The four most popular tools for mining data from GitHub are GitHub Archive, GHTorrent, GitHub API and GitHub Search API. Though GitHub API is the most used tool, it is not something that can be used when mining data of this magnitude due to the rate limits. The rate limit is 5,000 requests per hour [26] and in total there was around 5,000,000 active Java projects which leaves us with at least that many requests that would take around 41 days to complete and this is not including downloading of the source code. GitHub Search API is not used for collecting information about repositories and is also disregarded. Both GHTorrent and GitHub archive are mirrors of GitHub and can, therefore, contain slightly older data. This is however not a problem since we want to collect projects that have been active for a longer time. GHTorrent has the advantage that each dataset is separated into separate files this means that the application does not need to process data such as user information that is of no interest for our corpus and is the tool that was used for this project. The way to download the repositories will be to install and use the GIT client with the clone command since this was done by all research papers that mined source code from GitHub.

Based on the findings by the Reaper project and by Kalliamvakou et al. the filters applied to find real-world software projects were focused around the number of collaborators, the popularity of the project and commit history. A project should have more than one collaborator otherwise it is more likely that the project is a student assignment, example or hobby project. It should also have some popularity by being starred by other users, in this case, more than 1 star was included in the corpus. The last and most important filter is the time range between the project creation and the latest commit, a project should have been active more than 100 days to be considered a real project. The assumption is that the corpus might contain some non-quality projects and a larger number of projects which make it easier to validate how the different filters affect the inclusion and exclusion of projects. The filters can
later be tweaked to give a smaller corpus but with higher quality once the effect of each filter is analyzed. The best way would probably be to calculate the metrics for all Java projects on GitHub and then try to find patterns for the projects with good quality, but that is too time-consuming for this work, so some initial filtering needs to be done.

An important aspect raised by Allamanis and Sutton is the importance of avoiding duplicate projects in the corpus since this can skew the results. The way to remove duplicates in an automated way will be to find projects that share some commits and find out which of the projects that are the most popular one and remove the others from the corpus.

5.2 The Java Corpus
From the 60 million repositories on GitHub, around 5 million of them were labeled as Java repositories that were filtered down to 21,951 projects that resulted in 11,296 high-quality projects.

The size of our corpus (21,951 projects) is a bit bigger than the corpus collected by Allamanis and Sutton (14,807 projects) even though the filtering should be more strict, and it can be explained by the fact that their corpus was collected in 2013 and that projects on GitHub have increased a lot since then.

Comparing the average values with the Qualitas Corpus shows that the WMC and RFC are lower for the collected corpus. This is probably due to all the smaller example repositories that passed the filters applied. Interestingly the CBO is higher which means that the collected projects are more coupled.

Figure 4.2 and 4.3 shows that software complexity is something unique for each project and cannot be mapped to a particular domain or Java version and this assumption was maybe a bit naive. The reason for how the cluster looks like seems to be more tied to the average LOC which then seems to increase WMC and RFC.

The 51.5 % quality rate of the corpus was a bit lower than expected even though earlier research points to that it is hard to mine good quality projects from GitHub. The random sampling makes it even more clear that using only metadata for filtering projects is not good enough. Simple examples can have high popularity because they show users how to implement specific tools or solve specific problems, this can also lead to that they are updated over an extended period and also by different people. This is probably the reason that the corpus contained around 23 % of those cases it
also explains why the average WMC and RFC were lower than the Qualitas Corpus.

Also, the high values of mainly CBO but the other metrics as well points to smaller projects that are not well written and potentially contain source code that can skew learning algorithms.

Finally, a concrete output of this thesis project is a database containing the 21,950 projects with information such as description, commit history, popularity and how they were finally classified by comparing them with the created clusters quality/non-quality/low-complexity/other. The database also contains the reference to where the source code can be found on disk for each project and the computed complexity metrics for each Java file in the corpus.
6 Discussion

This paper is yet another proof that mining GitHub for quality projects is hard despite learning from Kalliamvakou et al. [2] recommendations and Munaiah et al. [7] on how to avoid noise on GitHub. Just using metadata about the projects is not good enough which the found code example repositories show as well as the number of projects with very high complexity.

However, even though not perfect, the created clusters for identifying quality and non-quality projects could be applied as a secondary step to reduce noise in large corpora further. Using complexity metrics as a primary filter is not feasible due to the amount of time it takes to calculate them. Using the clusters to classify projects will not remove the noise entirely but seems to improve it a lot.

A useful feature with classifying projects from the clusters is that the researcher can choose if they want to include code examples or not which can be useful to have when creating an extensive database of code examples like Keivanloo et al. [13] but not when trying to understand real software projects.
7 Conclusion

Using the clusters could help in further filtering java corpora. It will not create a perfect corpus or remove all the noise but will reduce the number of example projects and remove projects with high complexity that probably contains lousy code.

This way of applying the cluster classification as a second filter is highly relevant for research where you want to minimize lousy quality code or remove code examples from large java corpora.

The collected corpus, together with the classified projects by the clusters, could also be used by other researchers that do not have the resources or time to mine their own and then they can choose what to include depending on the classification of the projects.

7.1 Future work

Regarding future work, there is a lot that can be done. First thing would be to start classifying other corpora like the Allamanis and Sutton corpus [1] to find out the best way to do a primary filter based on a project's metadata.

Also, manually evaluating projects with the random sampling would help to understand better what the 11.9 % other unclassified projects are. Tweaking the clusters could be a way to improve the classification further. A better way to create the clusters would be to not just include the average metrics for a project but also include the standard deviation for all the classes.

Finally testing earlier research, that has been done on corpora that have been collected without validating the quality, with the corpus we collected and see how the results differ could be of interest.
References


Appendix - Logs From Clustering

Below are the logs from Weka after performing the two clustering algorithms. This information can be useful when trying to reproduce the clusters.

Quality clusters

=== Run information ===

Scheme: weka.clusterers.EM -I 100 -N -1 -X 10 -max -1 -ll-cv 1.0E-6 -ll-iter 1.0E-6 -M 1.0E-6 -K 10 -num-slots 1 -S 100
Relation: quality_corpus_with_metrics-2
Instances: 108
Attributes: 7
  wmc
dit
cbo
rfc
loc
Ignored:
  id
Test mode: evaluate on training data

=== Clustering model (full training set) ===

EM

Number of clusters selected by cross validation: 4
Number of iterations performed: 36

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Attribute</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(0.4)</td>
<td>(0.12)</td>
<td>(0.18)</td>
<td>(0.3)</td>
</tr>
</tbody>
</table>
wmc
mean        29.0031   69.016   42.6458  17.3446
std. dev.    4.9424  33.9403   8.5074  4.4001

dit
mean          2.525   2.7925   2.0868   2.0506
std. dev.    0.7366   0.7258   0.3604   0.3695

cbo
mean         5.6035   7.5731   6.5445    5.318
std. dev.    1.6925   2.5316   2.1162   1.1292

rfc
mean         17.644  30.5071  20.6905  12.0444
std. dev.    4.2628   5.6215   4.8322    2.719

loc
mean       177.1324 304.2483 250.5466 115.6731
std. dev.    27.107  77.4643  47.4217  32.3984

Time taken to build model (full training data) : 0.49 seconds

=== Model and evaluation on training set ====

Clustered Instances

0        44 ( 41%)
1        12 ( 11%)
2        19 ( 18%)
3        33 ( 31%)

Log likelihood: -14.89089
Non-quality clusters

==== Run information ====

Scheme: weka.clusterers.EM -I 100 -N -1 -X 10 -max -1 -ll-cv 1.0E-6 -ll-iter 1.0E-6 -M 1.0E-6 -K 10 -num-slots 1 -S 100
Relation: correctly-classified-non-quality-projects
Instances: 37
Attributes: 7
   wmc
dit
cbo
rfc
loc
Ignored:
id
name
Test mode: evaluate on training data

==== Clustering model (full training set) ====

EM

Number of clusters selected by cross validation: 3
Number of iterations performed: 10

<table>
<thead>
<tr>
<th>Cluster Attribute</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(0.19)</td>
<td>(0.56)</td>
<td>(0.24)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wmc</th>
<th>mean</th>
<th>std. dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>74.3887</td>
<td>36.1174</td>
</tr>
<tr>
<td>dit</td>
<td>mean</td>
<td>std. dev.</td>
</tr>
<tr>
<td></td>
<td>2.2703</td>
<td>36.1174</td>
</tr>
</tbody>
</table>
std. dev.  0.518 0.3458 2.1188

cbo
mean   16.0486 4.8516 13.4156
std. dev.  2.2022 2.6411 2.6102

rfc
mean   47.4461 7.6346 24.9987
std. dev.  8.1925 4.4017 8.0144

loc
mean   333.7265 54.0641 149.8406
std. dev.  130.4973 25.1761 46.0946

Time taken to build model (full training data) : 0.29 seconds

==== Model and evaluation on training set ====

Clustered Instances

0   7 ( 19%)
1  21 ( 57%)
2   9 ( 24%)

Log likelihood: -16.1115
Appendix - Random Project Sample

This is a table of all the random samples that were manually evaluated. The information contains name, stars/watchers that shows popularity, how long the project has been active, the average complexity metrics, if the project was classified low quality (not matching any cluster), if the classification was correct, what type of project we think it is and a comment about the project in relation to the classification as high or low quality.