MASTER IN COMPUTER SCIENCE

Knowledge acquisition for machine learning and analysis of software quality in GitHub

Fernández-Grande, David

Award date:
2018

Awarding institution:
University of Namur

Université de Namur
Faculty of Computer Science
Academic Year 2017–2018

Knowledge Acquisition for Machine Learning and Analysis of Software Quality in GitHub

David Fernández-Grande

Supervisor: _____ (Signed for Release Approval - Study Rules art. 40)
Benoît Frénay

Co-supervisor: Benoît Vanderose

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur

Knowledge Acquisition for Machine Learning and Analysis of Software Quality in GitHub

Acknowledgements

I am grateful to my supervisor, Benoît Frénay, and co-supervisor, Benoît Vanderose. I appreciate greatly their support, kindness and knowledge, which have been remarkably helpful and motivating. It has been a real pleasure working with you.

Thanks to Efrén and Mimi for proofreading the manuscript.

Gracias familia por todo vuestro apoyo y amor. Efrenzote, *viejunos*, *pitufo: sin vosotros, habría sido imposible*.

Abstract

Abstract. Software measurement aims at providing a reliable and repeatable method to assess software quality. However, quantifying precisely (i.e. defining meaningful thresholds) the connection between software metrics and higher level quality attributes has been a continuing challenge. Machine learning can be exploited to improve our understanding of software metrics and the relationship with software quality. This requires the analysis of large amounts of data. Fortunately, online social coding platforms such as GitHub make large amounts of software-related data (both source code and metadata) publicly available. This study uses machine learning on GitHub repositories to assess their quality. Specifically, this work (i) makes publicly available a dataset with the metadata of 71,942 GitHub repositories, then uses it to (ii) describe the characteristics of the use of GitHub and (iii) define criteria to select projects relevant to software quality analysis. Additionally, the study (iv) builds an extended version of the dataset with software metrics of 3,074 GitHub repositories exploitable by standard machine learning techniques, (v) examines the style of code in this platform and (vi) creates a machine learning model to analyse the quality of these repositories.

Keywords. Machine learning, Software Measurement, Software Metrics, Software Quality, GitHub, Software Repositories Mining.

Résumé. La mesure logicielle cherche à fournir une méthode fiable et répétable d'évaluation de la qualité logicielle. Néanmoins, la quantification précise (incluant une définition significative des seuils) de la relation entre les métriques logicielles et les attributs de qualité de plus haut niveau reste un défi. L'apprentissage automatique peut être utilisé pour renforcer notre compréhension des métriques logicielles et leur lien avec la qualité logicielle, ce qui requiert l'analyse de grandes quantités de données. Toutefois, certaines plateformes de programmation collective en ligne, telles que GitHub, ont l'avantage de rendre ces données logicielles disponibles au public (aussi bien le code source que les métadonnées). Cette étude utilise l'apprentissage automatique sur des dépôts (*repositories*) de GitHub afin d'évaluer leur qualité. En particulier, ce travail (i) rend public les métadonnées de 71942 dépôts de GitHub, utilisées ultérieurement pour (ii) décrire les caractéristiques de l'utilisation de GitHub et (iii) définir les critères de sélection des projets pertinents à l'analyse de la qualité logicielle. En outre, l'étude (iv) rassemble une version étendue de l'ensemble de données contenant les métriques logicielles de 3074 projets de GitHub exploitables par les techniques standard d'apprentissage automatique, (v) décrit le style de programmation dans cette plateforme et (vi) crée un modèle d'apprentissage automatique pour analyser la qualité de ces projets.

Mots-clés. Apprentissage automatique (*machine learning*), Mesure Logicielle, Métriques Logicielles, Qualité Logicielle, GitHub, Exploration (*mining*) de données de projets logiciels.

Table of contents

I. Introduction....6
II. State-of-the-Art: Using GitHub in Software Quality Research....8
A. GitHub as a Data Source for Software Research....9
B. Replicability in GitHub....11
B1. Limitations of Existing Datasets....13
C. Pitfalls in GitHub for research....13
D. Quality Indicators and Software Metrics in GitHub Research....15
D1. Quality indicators in GitHub research....15
D2. Software metrics in GitHub research....16
E. Massive GitHub Data Processing: Machine Learning....17
F. Summary....18
III. Contributions: Gathering Data and Analysing the Use of GitHub....20
A. Metadata Extraction Tools and Process....21
A1. Dataset Availability....22
B. Characteristics of the Use of GitHub....22
C. Selection of Repositories Criteria....29
D. Conclusion....29
IV. State-of-the-Art: Software Quality and Metrics....31
A. Software Quality....31
B. Software Measurement....33
B1. Software Product Metrics....35
B2. Review of Main Software Metrics....36
B3. Metrics and Fault-Proneness....37
C. Change Metrics....38
D. Software Metrics and Machine Learning....39
E. Summary....40
V. Contributions: Predicting Quality with Machine Learning....41
A. Machine Learning Features and Targets....42
A1. Software Quality Targets....42
A2. Software Metric Features....43
B. Data Extraction....45
B1. Extraction Tools....46
B2. Extraction Process....46
B3. Software Metric Features Extracted....48
B4. Dataset Availability....50
C. Characterisation of Code in GitHub....51
D. Predicting Quality with Software Metrics using Machine Learning....58
D1. Presentation of Results....58
D2. Analysis of Results....61
E. Conclusions....63
VI. Conclusions and Future Work....65

A. Conclusions....65
B. Future Work....66
C. Research Contributions....67
VII. References....68
VIII. Appendix....71
A. Paper submitted to MaLTesQue 2018 72

I. Introduction

Software quality assessment is crucial in software development, which in turn is essential to companies, clients, developers and other entities working with software. Software measurement, together with software inspection, is one of the main methods used to establish meaningful relationships between measurable properties of software artefacts (e.g., lines of code, depth of inheritance tree, coupling between objects, etc.) and high-level software quality characteristics (e.g., reliability, maintainability, efficiency, etc.) [1]–[4].

The academic analysis of software metrics has significantly increased in the last years, but software industry has remained reluctant to incorporate these metrics. The main reason is that the concrete relationships between software metrics and quality have not been yet accurately characterised (i.e. with suitable thresholds to interpret quality based on measurement values), while the limitations of metrics are still under discussion [5]–[7]. The goal of this study is to contribute to improving the current understanding of the connections between software metrics and quality.

The use of numerous and diverse source codes is undoubtedly key to providing significant results in this research. With the increasing popularity of social coding platforms, an enormous amount of data is publicly available which can be used to this purpose. In recent years, GitHub has become the most widespread collective code hosting platform. Therefore, it offers a very rich source of data GitHub the study of software quality and metrics. It provides not only the source code of projects, but also interesting additional metadata such as reported issues, liveliness of the project, etc. Therefore, GitHub can be used both to extract the software metrics from code and to obtain indicators of software quality related to this metadata.

Machine learning provides a great opportunity to process this enormous amount of data and better analyse and understand them. Specifically, machine learning can help in the actual quantification of the relationship between software metrics and high-level quality characteristics. In a classical empirical setting, it would require intractable efforts to do this manually and it would be very difficult to analyse how different metrics are related with each other and with quality. Applying machine learning techniques to software measurements extracted from GitHub has great potential to yield meaningful results to improve the understanding of quality and metrics.

Many standard machine learning algorithms require a set of features and targets. It is therefore crucial to identify those software metrics which can be used as features and those properties of GitHub projects which denote quality and can be employed as targets. This is a challenge in itself: taking a GitHub software repository and identifying relevant properties on which algorithms can be applied in order to detect relevant patterns that characterise it.

To summarise, this study aims at analysing the connections between software quality and software metrics using machine learning with data retrieved from GitHub. To achieve this, a dataset with metadata about the use of GitHub is gathered and made publicly available. This also enables to analyse the properties of the use of this platform. An extended version of the dataset is built, containing relevant software metrics and quality data retrieved from GitHub. This dataset is fed to a machine learning algorithm which contributes to understand the correlation between software metrics and quality and its limitations. The dataset is also used to characterise the properties of source code hosted in GitHub.

The remaining content of this document is as follows: chapter II analyses the state-of-the-art in the use of GitHub for research in software quality and metrics. In chapter III, metadata from GitHub repositories are extracted in order to build a dataset, which are employed to characterise the use of this platform and define the criteria of the set of repositories used to analyse quality and metrics. In chapter IV, the state of the art on software quality and metrics is analysed, specially fault proneness and static software product metrics. Chapter V uses machine learning on an extended dataset, containing software metric features and quality targets retrieved from GitHub. It also explains the process of feature and target extraction and uses the retrieved metrics to characterise the source code in GitHub. In Chapter VI, improvements and future work to deepen the current understanding of software quality are proposed.

The appendix includes a paper that was submitted to the Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTesQuE 2018) in March 2018 in Campobasso, Italy.

II. State-of-the-Art: Using GitHub in Software Quality Research

For the software engineering researcher, social coding platforms and software forges are a very promising subject of study and a precious source of available data. In our study, aimed at improving the understanding of software quality and software metrics, they appear to be a very valuable tool. Among these collective coding platforms, the most widespread is GitHub.

This chapter examines whether GitHub is an appropriate tool for the research on software metrics and software quality. To answer this question, one needs to evaluate whether GitHub has already been used as a subject or a tool for research. It is also necessary to evaluate the amount of data available and compare its characteristics with others possible source of data, such as software engineering research databases. Analysing the risks that the use of GitHub entails in scientific research is also important. For example, one of the main challenges of using this coding platform in research is its constant evolution, which makes it difficult to ensure the replicability of studies.

Analysing the scientific works already carried out with GitHub in this field can help assess whether GitHub is valid tool to analyse software quality and software metrics. Specifically, it is essential to investigate how previous works have measured the software quality of projects in this platform. It is also important to consider which type of software metrics can be extracted and which ones have been so far used in previous studies.

Collecting raw data on software metrics and quality is not enough, as these informations need to be processed and analysed. Identifying and understanding the methods that can be used to handle these data is necessary. Analysing which methods have already been used in preceding works can help in this objective.

In summary, this section endeavours to present the state-of-the-art in research around GitHub, with special interest in software quality and metrics. Specifically, section II.A analyses the potential of GitHub as a source of data in software research, compared to other sources of data, such as those used before the emergence of GitHub. Section II.B analyses how replicability can be ensured for a platform in constant evolution like GitHub. Section II.C examines the pitfalls that have been identified in using GitHub as a subject of research and proposes some measures to avoid these risks. II.D concentrates on studying the software metrics and quality indicators available in GitHub and how they have been exploited in previous works. Finally, section II.E addresses which methods are available and have been used in order to process software quality and metrics extracted from GitHub.

A. GitHub as a Data Source for Software Research

In the Open Source Software (OSS) community, software forges are web-based collaborative platforms, used mostly for version control, code sharing among developers, bug tracking, source code management, documentation management, etc. Various commercial and open source coding platforms exist, such as SourceForge, Subversion, GitHub, GitLab, Bitbucket, Google Code, etc. The largest among these collaborative code hosting platforms is GitHub, started in February 2008. It is based on Git, a distributed version control system which adds social interaction features. Its popularity has greatly increased and, as of July 2018, it has more than 30 million users and around 28 millions public repositories. At the end of 2017 it accounted for a total of 67 millions repositories [8], including private repositories.

The aforementioned web-based software forges have been used for research in software engineering. Other sources of data have been employed, such as public datasets with data about various software projects. Among these, one of the most relevant ones is the PROMISE repository [9]. In 2002 the NASA established the Metrics Data Program (MDP), which gathered static code metrics from their projects. Some of these data were used to create the Promise repository in 2004. Ever since, the number of datasets have increased and covers a wide range of software engineering research datasets.

One of the main problems of software engineering research is that many studies used their own *ad-hoc* datasets, many of which are private. Catal et al. show in Fig. 1 that, before the creation of the PROMISE repository, only 31% of the datasets used in software fault prediction studies were public. From 2005, the number of public datasets in research increased to 52% (Fig. 2) [10]. Notwithstanding this increase, it would be desirable for replicability that public databases were used more often in scientific studies.

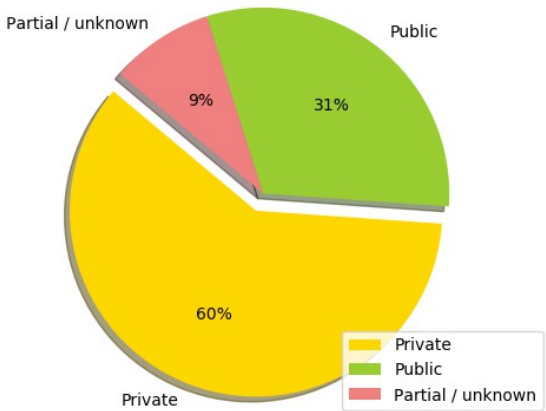

Fig. 1. Distribution of public and private datasets (data from [10]).

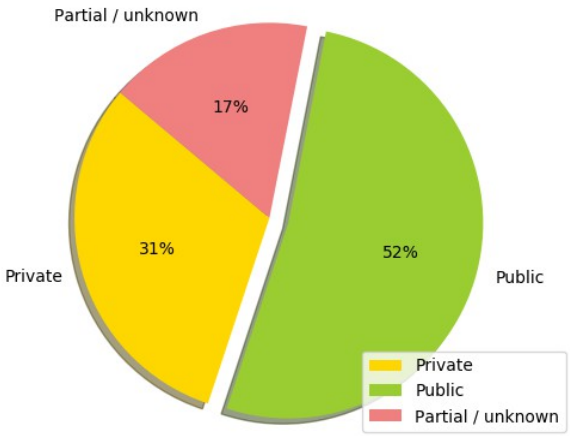

Fig. 2. Distribution of public and private datasets since 2005 (data from [10]).

Although the platform has less than 10 years of existence, its popularity has made GitHub a subject of research in a wide range of topics. Cosentino et al. [11] analyse 315 works involving GitHub. They classify them in three groups, depending on the role of this repository hosting service:

- 167 “source papers”, that use GitHub as a “source of repositories”, but did not study any specific feature of this tool.
- 157 “target papers”, that analysed specific features of this platform and they way they were used.
- 18 “description papers”, that studied GitHub as part of studies that compared open-source platforms.

Among those classified as “target papers”, table I lists the subjects that they covered. Among other topics, it shows a significant interest in the social dimension of GitHub (e.g. analysis of distributed coding tools such as issues and forks, interaction between its users, examining their teams and discussions, etc.).

TABLE I. Areas of "target papers" research.

Areas	Topics	In topic	In area
Software development	Code contributions	21	34
	Issues	6	
	Forking	12	
Projects	Characterization	19	46
	Popularity	14	
	Communities & teams	18	
	Global discussions	12	
Users	Characterization	21	34
	Rockstars	10	
	Issue reporters and assignees	2	
	Followers	10	
	Watchers	6	
Ecosystem	Characterization	8	23
	Transparency	9	
	Relationship with ther platforms	7	

Table included in [11]

B. Replicability in GitHub

Using GitHub as a scientific research tool raises the problem of replicability. Due to its popularity, it is continuously evolving: new repositories and users are added, projects change through time, artifacts are removed from the platform, etc. Furthermore, GitHub's interface, mechanisms and features have changed over time and some functionalities have evolved: these changes hinder replicability.

Given the importance of replicability and the enormous amount of data stored in GitHub, various projects have created datasets of the information contained in this platform at specific moments, mostly used for big data analysis and machine learning. One of the most used is GHTorrent [12], a queriable mirror database hosting a significant part of GitHub metadata, stored both in a SQL database and a set of MongoDB files. The dumps with the data can be downloaded and queried offline, or directly queried online. It is a useful snapshot on which to conduct research in a controlled and replicable manner. GHTorrent holds metadata information about GitHub users, repositories, commits, etc. However, it does not include code or repositories as such.

In a similar manner, the GitHub archive [13] provides a dataset of public events in the GitHub event stream as from 2011. Specifically, it proposes three datasets, depending on the time range they cover: the year dataset that contains activities for each year from 2011, the month dataset and the day dataset.

Another available dataset is the GitHub Java Corpus [14], gathered for the work conducted in [15]. It is a snapshot of all the open-source Java projects with at least one fork that were

publicly available in October 2012 (a total of 14785 projects). This source of data is not a database strictly speaking, but a compilation of the files with extension ".java" used in their research.

Cosentino et al. [11] show in Fig. 3 that GHTorrent is the most frequently used source of data for GitHub research (41.25%). The second most frequent way to retrieve data is by directly querying the GitHub API (31.25%), instead of using previously existing datasets.

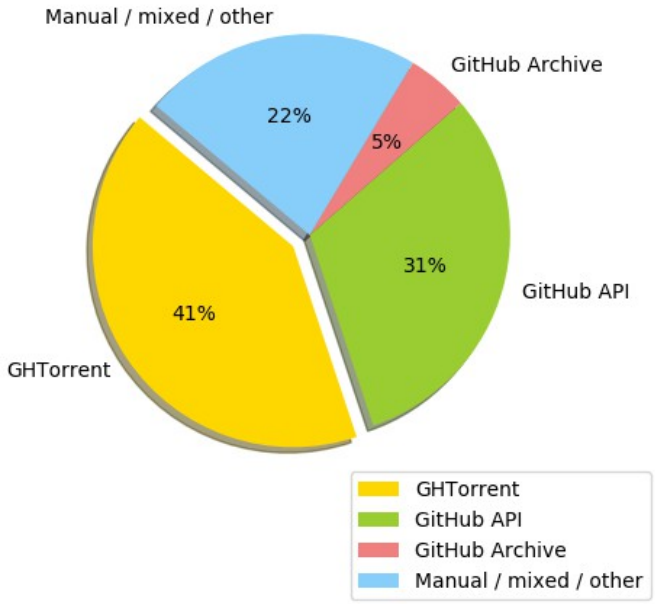

Fig. 3. Sources of data for GitHub research (data from [8]).

A more recent database is the Public Git Archive [16], a dataset consisting of 182,014 repositories (based on the GHTorrent MySQL dump from January 2018) having at least 50 stargazers. Stargazers of a repository are those GitHub users that have bookmarked it, in order to have easier access to it, indicating interest in this project.

The Public Git Archive contains 248,043 files for a total of 3TB of data, along with a CSV file containing metadata about the repositories, which can be queried or downloaded. This facilitates the replicability and retrieval of repositories' source code, metadata and development evolution.

Another approach to build a dataset about GitHub is the one used by Gyimesi et al. [17] and Tóth et al [18], who focus on bug detection: they analysed 13 and 15 GitHub projects respectively, searching commits referencing a bug (as identified by the SZZ algorithm [19]–[21]). This information is gathered throughout different releases, in intervals of six months, along with software product and process metrics.

B1. Limitations of Existing Datasets

A positive and promising trend can be observed in the increase of available GitHub datasets. However, there are limitations to the use of these datasets in the context of software quality and metrics:

- GHTorrent and the GitHub Archive do not contain any information on software metrics. They collect metadata about the use of GitHub, but do not analyse their source code.
- The GitHub Java Corpus is not a database than can be directly exploited for software metrics or quality. It contains a snapshot of Java files from 14,785 projects, but holds no relevant information to evaluate the software quality (such as GitHub metadata). Furthermore, the information contained is not particularly recent (2012) and uses only projects having been forked (which can bias their quality).
- The datasets used in [17] [18] gather detailed software metrics information, but only for a limited number of GitHub projects (13 and 15 respectively), which seem not enough projects to represent a variety of styles and patterns in coding.
- The Public Git Archive is a very valuable and up to date source of data, with information about 182,014 repositories. It contains metadata on the repositories and the files of these repositories. However, like the GitHub Java Corpus, it contains the source files and not their software metrics. When a researcher wants to use these data, he would need to download all the repositories and extract their metrics with an appropriate software metric tool. Furthermore, it contains data only for repositories with a certain popularity (more than 50 stargazers), which can bias the quality of projects.

In spite of the increase in the number of publicly available datasets, it would be desirable to create a dataset of diverse and heterogeneous GitHub projects gathering both software metrics of repositories and their metadata (which can be used to assess its quality).

C. Pitfalls in GitHub for research

In order to use GitHub's repositories for research, it is useful to analyse their characteristics, since they can have an impact on the results of the studies. It is particularly important to be aware of the possible pitfalls and risks for its use in software research.

Firstly, the repositories accessible for research are public repositories. As mentioned before, the 28 millions public repositories accounted for less than half of the total repositories in GitHub (around 67 millions by the end of 2017). This implies that any research carried out on GitHub data is necessarily a partial view of the whole platform.

Another crucial observation reported by Kalliamvakou et al [22] is the fact that, albeit its

original purpose, more than one third of GitHub projects are not software development, as shown in table II. To prevent incorporating these *noisy* projects to research, they propose to consider the types of files in the repositories, as well as the review descriptions and README files.

TABLE 2. Number of repositories per type of use.

Category of use	Number of repositories
Software development	275 (63.4 %)
Experimental	53 (12.2 %)
Storage	36 (8.3 %)
Academic	31 (7.1 %)
Web	25 (5.8 %)
No longer accessible	11 (2.5 %)
Empty	3 (0.7 %)

Table included in [22]

In [22], it is also underlined that most projects have low activity (the median number of commits for a project is only 6 commits) and almost a half of them were inactive (only 54% of them had had commits in the last 6 months).

A relevant aspect of GitHub resides in the use of its social coding features (contributions to source code by different developers). A small subset of projects account for most source interaction in the platform. Specifically, it has been proved that the distribution of code contributions is highly skewed (power-law distribution) towards a small number of projects with numerous contributors [11]. Most projects are developed by small teams.

Furthermore, [22] shows that most projects do not use code contributions at all: 67% of GitHub projects are personal and only have one committer. 87% repositories have two or less committers, 93% have three or less. Therefore, when a researcher intends to avoid personal projects, it is recommended to take into consideration the number of committers in the repository.

Issue tracking is one of the main tools provided by GitHub to allow its community to interact with regards to code. This tool enables users to point out and fix bugs, propose new features, etc. Bissyande et al. [23] analysed in 2013 a hundred thousand GitHub projects to characterise the use of issue trackers and came to the conclusion that the use of this feature was very rare: about 30% of the projects used issues, whereas 3% disabled issue tracking and, remarkably, 66% did not disable them but still did not use them. The use of tags to categorise issues is scarcely used: less than 30% of issues are labelled.

GitHub allows to create repository copies (called *forks*), to which modifications (such as bug fixes) can be made and later proposed to the owner of the project (pull request). Again, the

distribution of forks is very skewed and unevenly distributed among projects: most of them are never forked, but few projects account for a great number of forks [11] [24] [25].

Another precaution to observe when using GitHub as a source of research data is that many projects do not use GitHub exclusively: they also recur to external forms of collaboration and many are mirrors of popular projects [22]. Using external tools is one of the reasons that explains why most projects do not use a considerable number of GitHub features, such as pull requests, issues, etc [11].

D. Quality Indicators and Software Metrics in GitHub Research

As it will be addressed in chapter IV, one of the main challenges in the field of software quality is reinforcing our understanding of the connections between external attributes, perceived by the user of the software, and internal attributes, which describe the characteristics of the software product and its code source. The exact quantification of the relationship between metrics and software quality is still disputed.

One of the main difficulties of using GitHub for software quality research is finding indicators of software quality within the data available in the platform. The extraction and analysis of the massive amount of source code in GitHub can be challenging. However, it can be automated and, therefore, practicable. On the contrary, retrieving indicators of external quality attributes (e.g. manual inspection of the quality characteristics of running GitHub software) is not feasible for such an enormous quantity of projects.

D1. Quality indicators in GitHub research

Various works have dealt differently with the automatic extraction of quality indicators from GitHub repositories. They can be summarised in two main approaches: those resorting to indicators of the number of bugs and the default proneness of projects, and those relying on popularity indicators.

Some researchers have opted for using fault-proneness indicators in GitHub. For instance, [17] and [18] use the issue tracking GitHub system, filtering the commits related to bugs and fixes labels, and link them using the SZZ algorithm [19]–[21], intended to determine which changes in code are responsible for a specific bug. Muthukumaran et al. [26] also mine data from GitHub based on bug-related commits (those with a commit message containing the words “bug”, “fix” or “fixed”) and link them to software change metrics.

In general, using issues and commits to quantify the number of bugs is one of the main

external quality indicators in GitHub. Indeed, although labels are not systematically used in GitHub repositories, among those using them the most common tags are bug-related (i.e. containing words such as “bug”, “defect”, “type:bug”, “Browser Bug”, “bugfix", etc.) [23].

The other common approach to GitHub quality measurement is using indicators of the popularity of the project (number of forks, watchers, etc.). For instance, Allamanis and Sutton [15] postulate in their work that “low quality projects are more rarely forked”. Markovtsev et al. [16] built up a GitHub-based dataset for repositories having at least 50 stargazers, as a “proxy on the degree of public awareness and project quality within the community”.

Using the popularity of a project within GitHub as an indicator of its quality seems a more high-risk approach than the one recurring to default-proneness indicators. It does not appear to be an unequivocal and certain correlation between the most popular projects and those having the highest software quality.

The use of these quality indicators, both those axed around default-proneness and around popularity, could be of use for researching software quality in GitHub. However, in the aforementioned studies they have been used as hypothetical indicators of quality, but to our knowledge there are no papers which confirm that they actually reflect the software quality. These different criteria have not been either compared among them to assess which are better predictors of GitHub projects quality. Carrying out these type of studies would help improve our understanding of software quality in GitHub.

D2. Software metrics in GitHub research

Having at their disposal such an enormous quantity of available data for software engineering investigation, various authors have used GitHub to extract software metrics in order to conduct research. In this section, some previous studies which make use of software metrics on data gathered from this social coding platform are analysed.

For instance, Gyimesi et al. [17] and Toth et al. [18] retrieved repositories from GitHub and extracted class-level and file-level static software metrics to create a public bug database for research in the field of fault-prediction. In [18], these data were used for automatic recognition of source code defects with machine learning.

Dwivedi et al. [27] extract 67 different static software object-oriented metrics from source code in GitHub with JBuilder [28]. They used these measurements to automatically predict software design patterns (i.e. Abstract Factory and Adapter) using machine learning algorithms (Layer Recurrent Neural Network and Decision Tree). For their part, [26] resorts to the aforementioned change metrics, easily retrievable from the history of GitHub projects, and feed them into machine learning algorithms to predict buggy files.

Software metrics are also used works using GitHub as a source of data to analyse the specific

characteristics of various families of programming languages. In [29], software metrics like Cyclomatic Complexity are extracted from Scala projects in GitHub in order to illustrate that functional programming repositories have methods with a very low complexity.

To summarise, GitHub projects have already been used to study software metrics. However, previous work has not yielded conclusive results on the quantification of its connection with software quality (e.g. establishing irrefutable thresholds for software metrics).

E. Massive GitHub Data Processing: Machine Learning

It is difficult to analyse individually the vast amount of repositories available in GitHub. Therefore, studies using this platform usually process these repositories automatically, in bulk, and the data obtained are later interpreted. For instance, [11] shows that 71.5% of GitHub studies rely exclusively in the mining and observation of GitHub metadata, while only 12.5% use interviews and surveys and 16.25% mix both approaches.

Among those studies that use GitHub as a data source, it is also very common to resort to automated analysis by machine learning techniques. The enormous amount of data in GitHub can be ideally processed by machine learning tools and algorithms. It is the case of [15], which uses machine learning on GitHub repositories to build a probabilistic language model of source-code, employed to propose and validate new complexity metrics and measure the centrality of a module within the project. In [30], it is shown that using deep learning algorithms on GitHub repositories outperforms other machine learning algorithms in fields such as source code suggestion.

Other works, like [18], use machine learning techniques to predict the fault proneness of 15 GitHub projects. The scope of this work is very similar to our current study. However, in [18], the authors train and test their machine learning models with the data of the projects separately. Therefore, the results of that work cannot be generalised to a wider scope of GitHub projects. Two models trained with the data of two different projects could both yield positive results, but perform poorly predicting the results of the other project. Also [26] applies machine learning algorithms on GitHub repositories to predict faults in source code files according to their change metrics.

In other words, the use of machine learning techniques in the study of data retrieved from GitHub is increasingly popular. This trend in the use of machine learning is not exclusive to GitHub studies. Catal et al. [10] show that machine learning is by far the most frequent method of data analysis in the field of fault prediction (59% of all works, Fig. 4), and this has increased from 2005 up to a 66% of all research (Fig. 5).

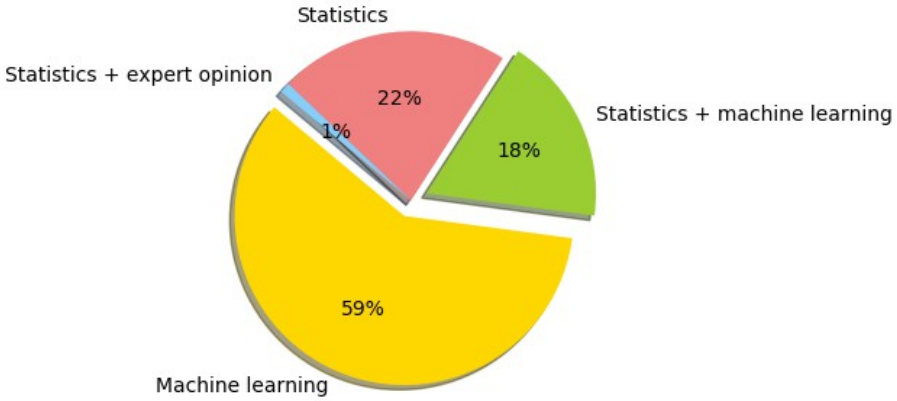

Fig. 4. Methods of data analysis in fault proneness studies [10].

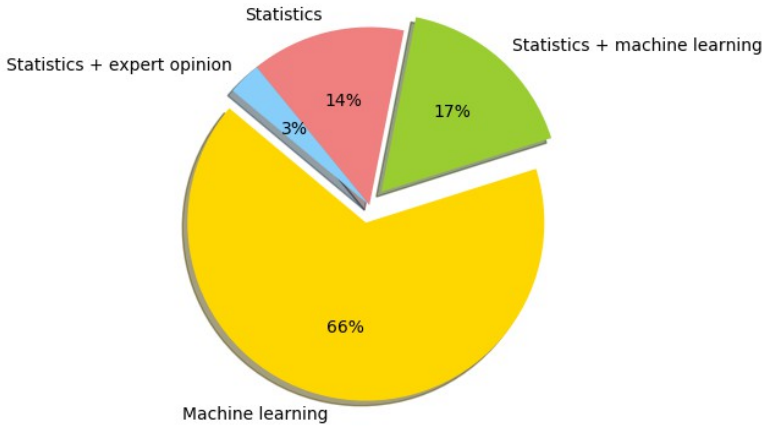

Fig. 5. Methods of data analysis in fault proneness studies since 2005 [10].

F. Summary

In this chapter, it has been shown that GitHub is a valid source of data for studying software metrics and software quality. It offers around 28 million public repositories which can be used for research. This is an unparalleled quantity of data, larger than any of the other sources of information used in software engineering research (such as the datasets gathered in the PROMISE repository [9]). A significant proportion of the datasets used in software research are private, while public GitHub repositories are available to all researchers.

However, the unequalled potential of GitHub as a research tool also pose potential risks that need to be addressed. One of them is the difficulty to ensure replicability of the scientific experiences making use of this platform. To overcome this limitation, in recent years there

has been an increase in the number of available datasets containing GitHub data, offering snapshots of the platform at specific moments.

The increase in the public GitHub datasets is a very promising trend. However, our analysis shows that these datasets have limitations when used in the context of software quality and metrics research. Mainly, they do not include both software metrics and software quality indicators (such as GitHub metadata), or the project selection biases the quality of projects, or they do not hold information about a sufficient number of different projects. Therefore, it is desirable to collect a dataset that overcomes these limitations and which can be exploited in software metrics and quality studies.

Other shortcomings of using GitHub stem from the very uneven and skewed characteristics of its use: most projects are small, personal, not interactive and rarely use many of the GitHub features. However, a small proportion of the projects are very popular, have a substantial size, use the social coding features offered by this platform, etc.

For the use of GitHub in software quality, previous studies use diverse quality indicators, mostly concentrating around two pillars: the fault proneness of the project and its popularity. These indicators, which can be automatically extracted at massive scale in GitHub, seem promising. However, literature has not sufficiently studied whether they reflect the actual software quality of the project, nor has compared them to evaluate which are better predictors.

It has also been shown in this chapter that, in recent years, the use of machine learning to analyse software quality (and specifically fault prediction) has increased, clearly surpassing the use of more classical approaches such as statistic analysis. This seems particularly appropriate in the case of using GitHub, since it provides vast amounts of data which appear to be particularly fit for use in machine learning.

III. Contributions: Gathering Data and Analysing the Use of GitHub

In chapter II, it was shown that GitHub is a valid and useful source of data for our research in software quality and software metrics. However, it is not feasible to manually process all of the great number of its public repositories. Furthermore, not all projects seem equally valuable for our study on software quality and metrics (for example, many repositories are not even software repositories). As a consequence, it is necessary to define a set of criteria to select the subset of repositories which will be mined.

In this chapter we aim at defining the characteristics of the GitHub projects which can be relevant for this research on software metrics and quality. To achieve this, it is necessary to gain a global understanding of GitHub characteristics and users. The literature analysed in chapter II, such as [11], [22] or [23], demonstrate some very interesting characteristics of this platform. Some of the data used in these studies were extracted in the year 2013 and before. Therefore, it is relevant to extract current data from GitHub to analyse other characteristics and to corroborate whether there have been significant changes in GitHub with regards to growth and popularisation in recent years.

It is also relevant to describe the process and tools used to extract and analyse GitHub metadata. This enables replicability for other researchers and allows them, when they need to mine GitHub for other types of metadata, to apply a similar process suited to their specific needs.

It is also desirable to make publicly available the results of this GitHub metadata extraction. In the url https://github.com/david-fdez/metadata_dataset, a set of JSON files with the relevant metadata of the 71.942 repositories mined in this chapter can be found and exploited for research purposes.

In the remaining of this chapter, section III.A explains the process, tools and choices made for the extraction of the GitHub metadata. In III.B, the most relevant results of this process are shown and discussed: the characteristics of the use of GitHub are presented and compared to the results of the previous works. Finally, in III.C these results are used to select a series of criteria to define the subset of repositories that will be used in the study of software quality and metrics in chapter V.

A. Metadata Extraction Tools and Process

In order to define the criteria for the data targets in our research, the characteristics of GitHub need to be analysed. In this section, the choices made to perform this analysis are presented. There are various tools available for gathering large datasets from GitHub. The first and most obvious tool is the GitHub REST API. This API sets limitations to the number of requests that a single user can make: an authenticated user can make up to 5000 requests per hour, whereas a non authenticated one has a limit of 60 requests per hour. Ensuring replicability in GitHub is challenging because the GitHub's API evolves frequently, which can lead to different results for the same calls to the API over time (cfr. section II.B.1). To prevent this, it is advisable to gather datasets and make them accessible for research, as proposed in [10].

Libraries in different languages exist, providing a layer of abstraction to the GitHub REST API, facilitating the interaction with the platform. This study has used the Python library PyGithub[1]. Other libraries in different languages exist to interact with GitHub, such as github3[2] in Python, the GitHub Java API[3] or the GitHub API for Java[4]. These and other similar third-party libraries to interact with the GitHub REST API can be found in [31].

Different attempts have been made to make publicly available snapshots of GitHub, which allow to retrieve the state of the platform at different moments. When a researcher wishes to emulate the results of a previous study, the same data can be accessed. One of the most widespread, as mentioned in section II.C.1, is GHTorrent [12], a set of periodical datasets containing a considerable part of GitHub metadata.

The GHTorrent database could have been used to retrieve the metadata and present the characteristics of GitHub use. However, we preferred to query directly the GitHub API, mainly because for this section it is critical that data are not biased by the date of creation of the repositories. Unfortunately, this bias is present in the GHTorrent data: its data mining process makes that it holds a larger proportion of data from repositories created after 2011, and that it does not contain projects which have not been updated since that date.

Other available GitHub datasets which enable the use of GitHub metadata, already mentioned in the section II.B, are the GitHub archive [13], the GitHub Java Corpus [14] or the recent Public Git Archive [16]. Other datasets, such as the bug dataset of Tóth et al. [18], do not contain directly the GitHub metadata, but already processed software metrics and number of bugs. Section II.B.1 presented the shortcomings of these databases, which make it preferable to query the GitHub REST API for the characterisation of GitHub.

Once chosen to use the GitHub REST API, we select random repositories in order to gain an overview of the characteristics of the GitHub repositories. For that, one can request the

1http://pygithub.readthedocs.io
2https://github.com/sigmavirus24/github3.py
3https://github.com/eclipse/egit-github/tree/master/org.eclipse.egit.github.core
4http://github-api.kohsuke.org/

resource “/repositories” with its pagination parameter “since”, which returns the 100 repositories with an id above the selected one. In our case, we have mined public projects with an id lower than 102775889, ranging from the start of GitHub until 07/09/2017. The current research has fetched a total of 71,942 repositories.

A1. Dataset Availability

The dataset with the metadata of these 71,942 repositories can be accessed at the url https://github.com/david-fdez/metadata_dataset. The data are stored in the form of a zip file, containing a number of JSON files. The name of each of these files is the day and out at which its metadata were extracted (e.g. the file “2017_12_20_07_18_57.json” was extracted the 20th December 2017 at 07:18:57). Each of these files consists of a JSON object, having the id of the repository as the key, and an object with its relevant metadata as its value.

B. Characteristics of the Use of GitHub

In this section, the characteristics of the 71,942 GitHub repositories are presented. One of the surprising results is, as shown in Fig. 6, that 23.3% of the repositories are not software development, i.e. they do not have a main language. Among them, there are empty repositories, as well as non-software files (repositories making use of GitHub as a file storage service). This is a slight increase compared to the state of GitHub before 2014, when Kalliamvakou et al. [22] claimed that 63.4% of repositories manually inspected where software repositories, 12.2% were experimental software code (examples, demos, etc.) and another 5.8% were categorised as “Web” (websites, blogs). A 16% of the repositories they analysed correspond roughly to the “no language” category (“storage”, 8.3%; “academic”, 7.1%; “empty”, 0.7%).

This increment in the number of non-software repositories can be partially explained by the increase of GitHub popularity: being a free hosting platform, one can assume that both developers and non-developers have used it for other purposes than merely collaborative software development. The rough correspondence between the analysis carried out in [22] and ours can also partially justify this increase: Kalliamvakou carries out a manual examination of the GitHub projects, one by one, to split them into seven categories; in our study, the main language of the repository is automatically extracted.

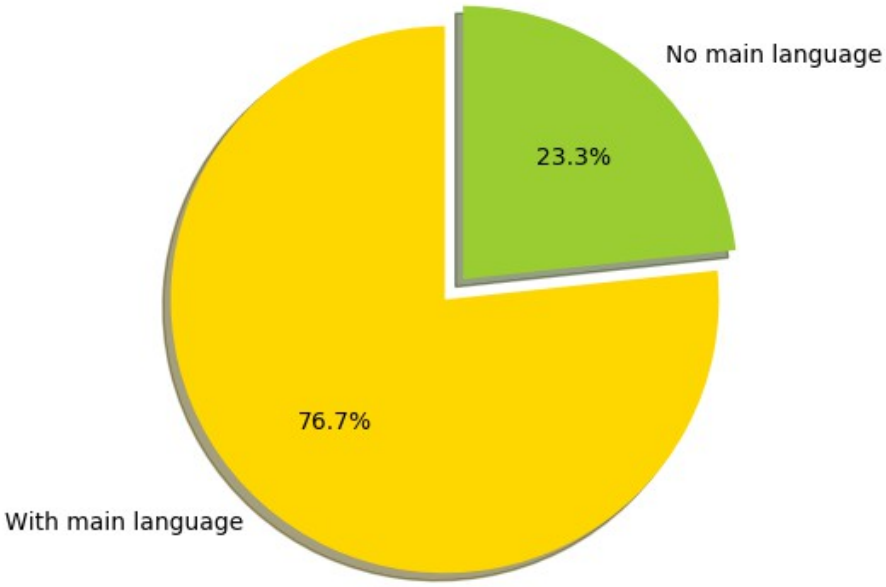

Fig. 6. Repositories with a main programming language.

Fig. 7 shows the distribution of repositories as a function of their main language. The results indicate that, among the main programming languages used in GitHub repositories, the most widespread is Javascript, followed by Java, Python, Ruby and HTML. Among the 71,942 repositories, 16,727 do not have any programming language, 11,076 are mainly written in JavaScript, 8,190 in Java, 5,415 in Python, 3,848 in Ruby and 3,806 in HTML.

As for object-oriented languages, Java is by far the most widespread: it is the main language of 11.38% of all GitHub repositories (Fig. 7). It is interesting to note that, until 2012, Java has increased its presence in GitHub. Before that year, the GitHub ecosystem was dominated by Javascript, Ruby and Python. Since 2013, Java has taken the second place of the most widespread languages [11]. The data extracted in this study confirm this position in 2017.

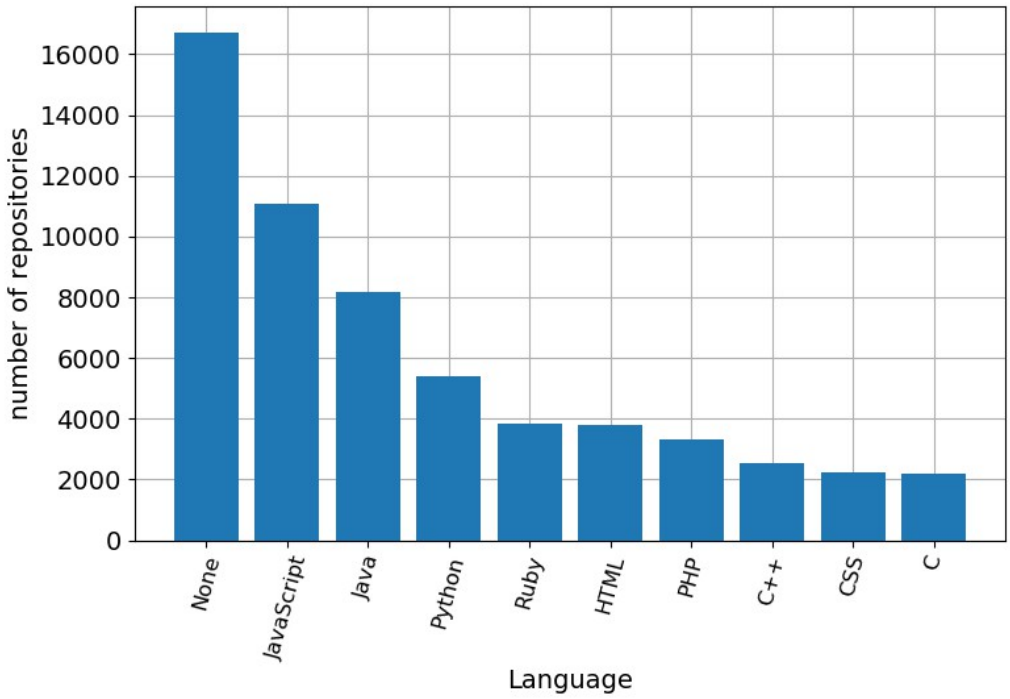

Fig. 7. Distribution of repositories according to their main language.

GitHub stores in its metadata information about the programming languages of each software repository, such as its principal language or the size of code in each used language for a repository. This programming language information is extracted by GitHub using the open source Linguist library [32].

Fig. 8 shows the total size of code written in the 15 most used programming languages in GitHub. It is interesting to note the difference between the main language of repositories and the size of code for each language. While Javascript, Java and Python are most frequently the principal language of software projects, it is surprising that there is more code written in C and C++ languages (Fig. 8). Among the analysed repositories, there are 44,368 TB of code written in C, 13,665 TB in C++, 13,268 TB in Javascript and 9,001 TB in Java.

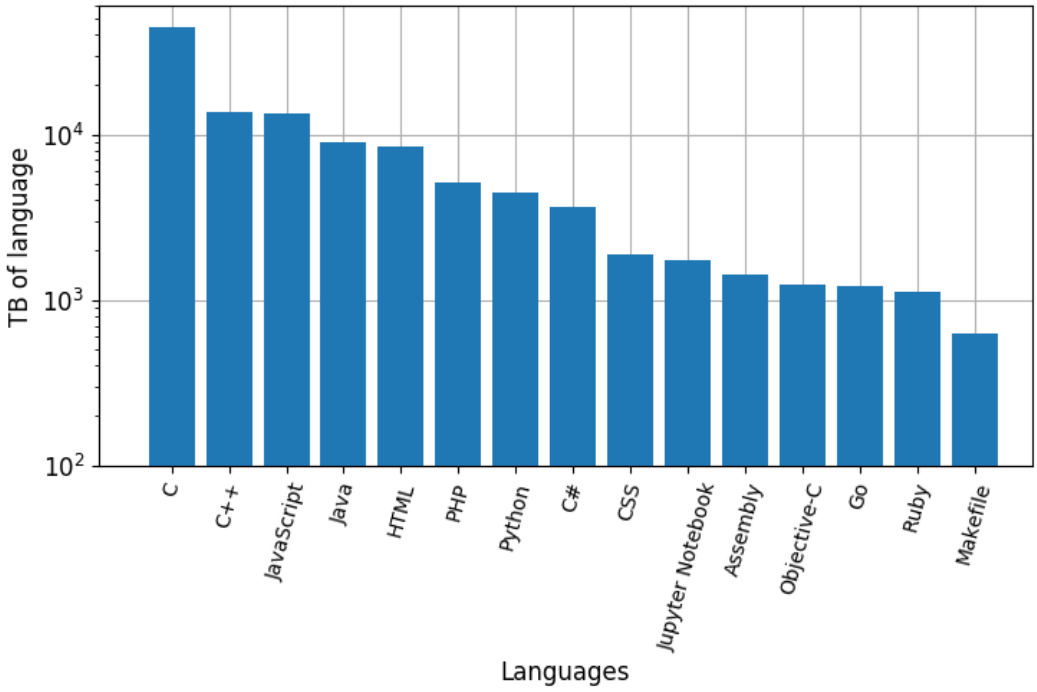

Fig. 8. Programming languages according to their total size in source code.

There are various reasons for the surprising fact that C and C++ are the languages with more code in GitHub, but are only the 9th and 7th in the ranking of main languages in GitHub repositories. These languages are more ubiquitous and can be found across different types of projects, regardless of their principal language. Different types of projects can make use of libraries written in C or C++. For example, it seems more likely that a Java project uses a third party component written in C, than it would be to find, for instance, PHP or Python code.

Furthermore, C and C++ languages are more verbose than most of their counterparts. This contributes to the surprisingly high quantity of code in these two languages present in GitHub. Finally, some specially big projects, such as operating systems that are hosted in GitHub (e.g. the Linux kernel[5]), are fundamentally written in C.

As a social coding platform, the data on interaction between developers in GitHub are a precious source of information for characterising the platform's ecosystem. Furthermore, interaction information has been used as an indicator of the quality of the repository (either as a manner to identify and fix bugs, or as an expression of its popularity).

A limitation of these data derives from the fact that 72% of all repositories are personal (i.e. have only one committer) and/or do not use the mechanisms foreseen for this interaction, such as pull requests and issue tracking [11] [22]. One of the central GitHub features to enable social interaction is called forking. The platform allows to create copies (called forks) of a repository (base repository) owned by another user. On the fork, one can make

5https://github.com/torvalds/linux

modifications, such as bug fixes, that will later be proposed to the owner of the project (pull request). In turn, forks can also have their own forked copies.

Fig. 9 shows the distribution of base and fork repositories: a 43.4% are forks. Base repositories have more forks, issues and pull requests than forks. The reason for this seems to be that the developer community tends to interact more with the main, original repository, than with a fork (many of which are used only to add a specific feature or correct a bug of the base repository).

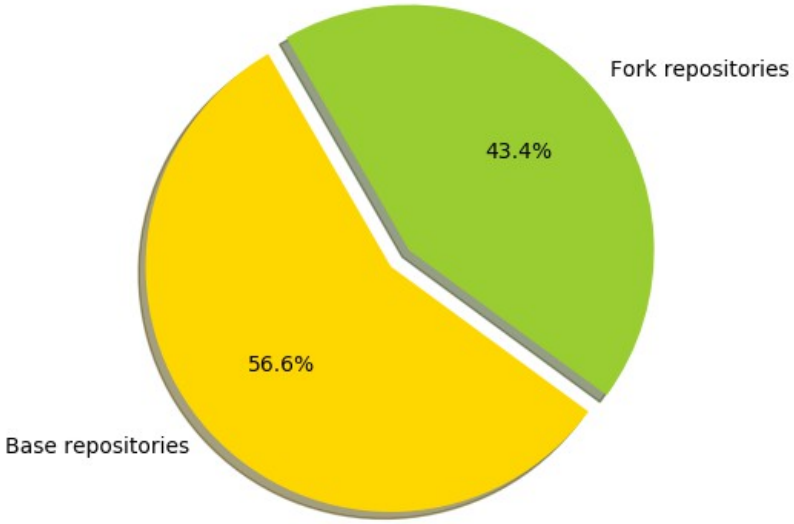

Fig. 9. Distribution of base and fork repositories in GitHub.

Fig. 10 (which, like other figures in this paper, uses logarithmic scale) shows the distribution of GitHub repositories based on their number of pull requests. Even if more than half of the repositories are base repositories, only 5.6% of the repositories analysed in this study have pull requests. One reason to explain this is that pull requests distribution is very skewed: few projects concentrate most pull requests [22]. Furthermore, many forks are used as copies not intended to be pulled to the base repository.

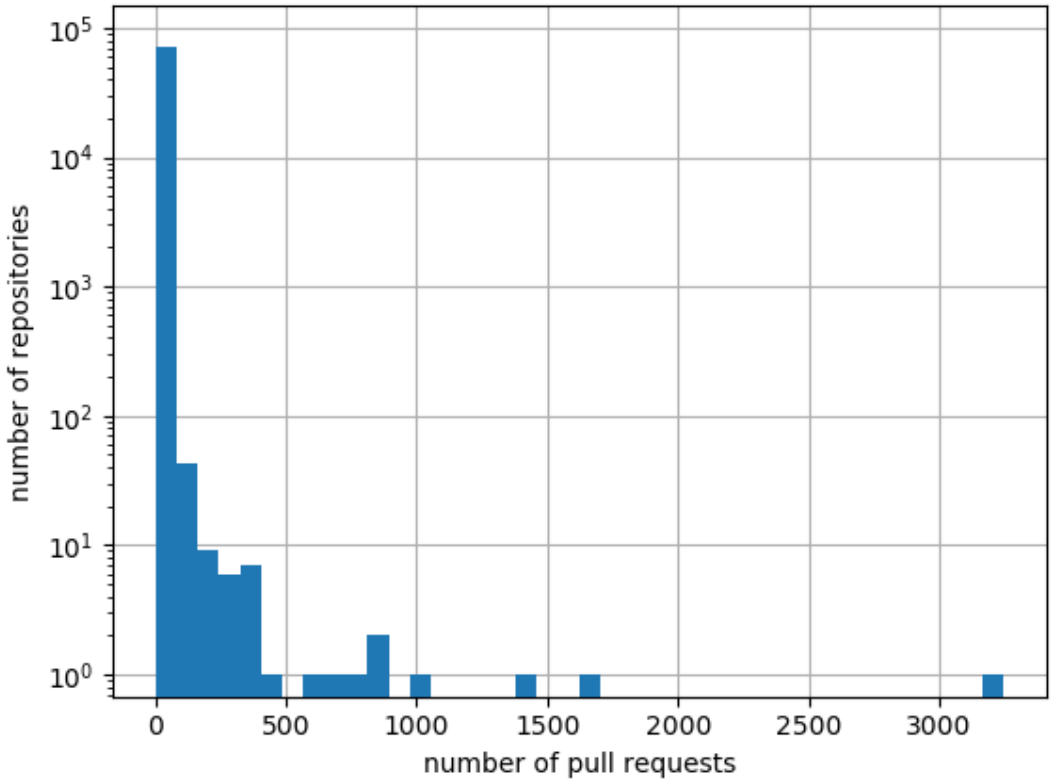

Fig. 10. Distribution of GitHub repositories by their number of pull requests.

In addition, many projects are experimental, empty or not related to software development. Evidence shows that direct code contributions by a user with write access to a repository (pushed commits) are more frequent than indirect contributions by means of pull requests [11] [33]. Finally, our data show that most projects are personal projects, not collective, hence the absence of pull requests from other users.

Another tool for interaction between developers in GitHub are issues. It is a tracking system mainly conceived as central bug tracker, but it can also be used to propose new features, enhancements to the software, etc. Issues are opened when they are created and closed when the work related to the issue is accomplished or the proposal is rejected.

Issue trackers are scarcely used. In 2013, about 30% of the projects had issues, whereas 3% disabled issue tracking and, remarkably, 66% did not disable them but still did not use them [23]. This situation has changed significantly in the data collected in 2017. Fig. 11 shows that 43.6% of repositories disable issue trackers, 48.5% do not disable them but don't use them and only 7.9% actually use issues.

The decline of issue tracking can be explained by the increasing number of personal projects in GitHub due, among other factors, to the popularity of the platform. It can also be partially the result of the fact that [23] excludes, unlike this study, projects with less than 1000 lines of code.

The number of issues per repository confirm that issue-tracking is seldom used in GitHub projects. Among the analysed repositories, the one with the maximum number of issues accounted for a total of 5,928. However, this was clearly an outlier. Fig. 12 (in logarithmic scale) illustrates that repositories rarely account for more than 250 issues in total.

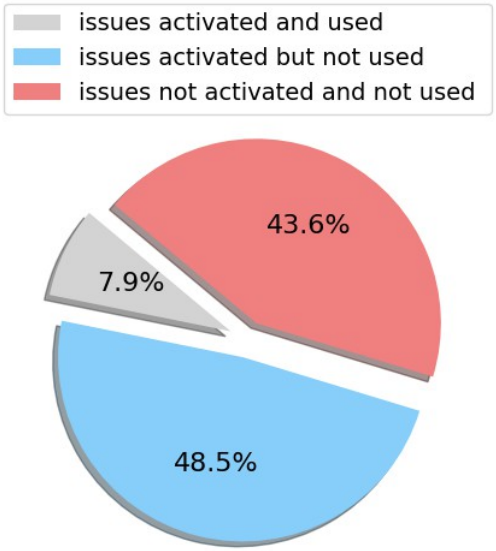

Fig. 11: Use of issue tracking in GitHub repositories.

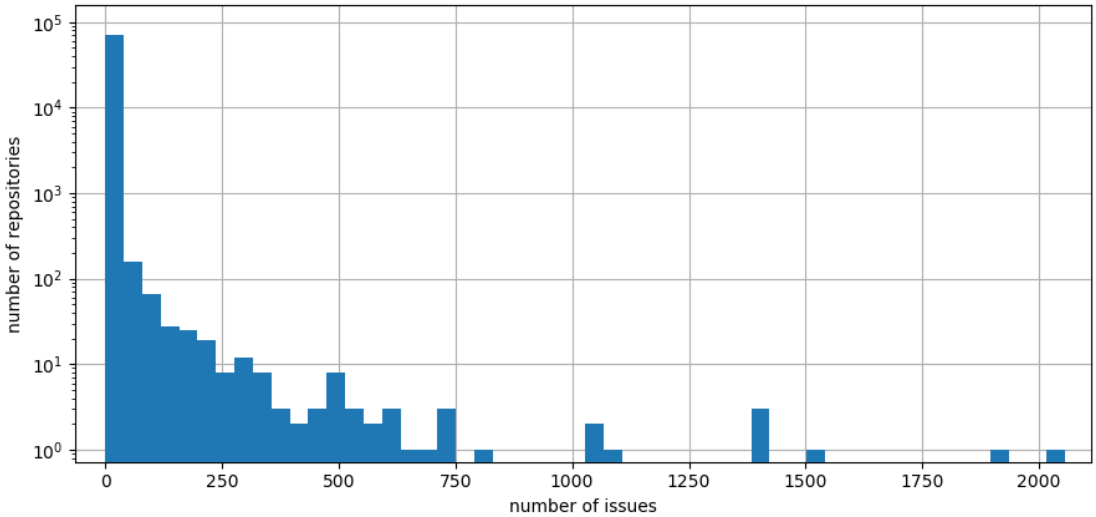

Fig. 12. Distribution of repositories by number of issues.

C. Selection of Repositories Criteria

This section examines the criteria to select the subset of GitHub repositories, which will be later used to analyse software quality and metrics.

Firstly, it seems a coherent choice to study object-oriented languages. They are widely accepted in industry and academia, and many software metrics are adapted to these kind of languages. As illustrated in section III.B, Java is by far the most widespread object-oriented language in GitHub. Its presence has increased significantly since 2012. Furthermore, it is widely used in industry and code metric tools can often analyse it. Therefore, this research will analyse repositories having Java as their main language.

Secondly, only not-fork repositories are selected. Analysing the same code several times (original repositories and forks) can introduce a bias for the most popular projects in GitHub. Also, base repositories are more prone to interaction among different users than forks, in terms of issues, pull requests, comments, etc. This social interaction gives a better insight into the quality of the code, via issues related to bugs, fixes, pull requests, etc.

Finally, only repositories using issues are selected. Analysing issues can be a fruitful manner to measure code quality, as they can be related to fault proneness of the repository. Furthermore, repositories having used the issue tracker system are more likely to be collective, collaborative repositories, instead of personal projects. Collective projects are less interesting in this research, since the lack of interaction between developers makes it difficult to assess the quality of the code using the repository metadata.

To summarise, three selection criteria have been retained:

- The repository has Java as its main language.
- It is a base repository, not a forked one.
- It uses the GitHub issue-tracking system.

Among the 71,942 repositories whose metadata were analysed, only 498 fulfilled the three criteria (0.69%). Extrapolated to the number of public GitHub repositories, approximately 28 millions, the number of repositories fulfilling these criteria would be of around 193,200 repositories.

D. Conclusion

This chapter has explained the process of extracting the GitHub metadata of 71,942 public repositories, in order to characterise the use of this platform. It introduces a number of tools that can be used to interact with the GitHub data. In order to analyse GitHub metadata, it is

preferable to query directly the GitHub REST API (in this case, using the PyGithub library for facility) instead of the existing databases: the later do not contain the relevant GitHub metadata, or do not cover a sufficient number of different projects or projects are unevenly distributed depending on their date of creation (GHTorrent).

Once the data were gathered, they have been made publicly available at the url https://github.com/david-fdez/metadata_dataset.

Analysing these data, it has been shown that the number of non-software repositories has increased since 2014. Among the object-oriented languages, Java is clearly the most widespread in GitHub. This study confirms that Java remains the second most used languages in GitHub, position that it holds since 2013. Remarkably, some languages such as C and C++, although they rarely are the main language of repositories, are very present in the GitHub ecosystem.

This section also highlights some characteristics of GitHub that are particularly relevant for studying software quality and metrics. A great majority of GitHub repositories are personal (interaction data cannot be used to assess its software quality). Almost half of the repositories are *forks* (copies of another repository) and they use less the interaction features of GitHub than the rest of the GitHub projects.

Another remarkable finding is the fact that the use of issue tracking in GitHub has remarkably decreased. In 2013, around 30% of GitHub projects used them, while in 2017 scarcely 8% does.

Finally, based on this characterisation of GitHub, the criteria to create a subset of GitHub repositories on which to analyse software metrics and quality has been defined. The criteria are: repositories must have Java as their main language, must be *base* repositories (not forked) and must actually use the GitHub tracking system. The number of repositories fulfilling these three criteria amounts to a 0.69% of GitHub repositories (193,200 repositories approximately for the total of public GitHub repositories).

IV. State-of-the-Art: Software Quality and Metrics

The analysis and improvement of software quality is crucial for all the stakeholders in software development. The main goal of this part of the study is to contribute to increase our understanding of software quality, improve its assessment and delve into its connection with software metrics. The use of software measurement and metrics are the main methods, together with software inspection, to evaluate the quality of software.

This chapter seeks to analyse the state of the art in software quality and metrics, in order to better understand how can quality be assessed. One of the main ways to assess quality is static software product measurement. Specifically, it is important to understand the role of static software product metrics, which summarise characteristics of source code.

Among the software qualities that denote software quality, one of the most widespread is fault proneness. This section also presents the current situation of fault proneness research and analyses how it is related to software quality and metrics.

In the remaining of this chapter, IV.A presents various models of software quality. In section IV.B different ways in which software quality can be assessed recurring to software measurement are discussed. IV.B.1 and IV.B.2 describe the main type of metrics used in software measurement: software product metrics. Section IV.B.3 analyses fault proneness, one of the main quality attributes in software, and how it can be measured. Section IV.C discusses another possible type of software metric, named change metrics, which analyses the modifications made to files in order to predict its future faults. Finally, IV.D deals with the use of machine learning by previous works on software metrics and quality.

A. Software Quality

Software quality is of paramount concern for developers, companies, users, etc. There are multiple aspects to software quality. Does the software meets its requirements? Which aspects should be taken into account for quality assessment? The user satisfaction? The efficiency of the developer? Is it mainly concerned with the functional requirements and features of the product? Is it instead connected with its non-functional dimension and/or the subjective impression on the user?

Software quality is a polysemic concept with unclear boundaries and different meanings

depending on the viewpoint of the person who coins it. For the International Organization for Standardization (ISO), software quality is defined by six characteristics: functionality, reliability, usability, efficiency, maintainability and portability [34]. Authors like Juran [35] underline two aspects of software quality, customer satisfaction (meeting the customers needs) and absence of errors and deficiencies, to summarise software quality as "fitness for purpose".

Numerous quality models have been proposed, each of which sets out a number of quality attributes. These quality attributes are measurable characteristics of software, either gathered by the subjective experience of the user or by using objective measurements.

These quality attributes can be gathered in two main groups: external and internal attributes [36]. External attributes are those which can be noticed while software is being executed. Most of this type of attributes can be observed directly by the user of the software product, not necessarily by an expert. For instance, a quality attribute such as "efficiency" can be assessed *externally* by measuring the time it performs a specific functionality.

Internal attributes, on the other hand, are those that can be measured statically from the software product itself (mostly by analysing the source code). They express the inner characteristics of software, such as its structural properties, mostly due to its architectural and programming choices.

A crucial assumption in software quality is that both internal and external attributes are related and, more specifically, that external attributes are (at least partially) the result of internal attributes. That is to say: the inner characteristics of the code or architecture of software have an effect on its actual execution behaviour.

According to [34], internal attributes influence external attributes, "so that there is both an external aspect and an internal aspect to most characteristics". As an example, it mentions reliability, which can be externally quantified by the number of failures during a period of execution time. It can also be internally measured by inspecting the specifications and source code in order to assess the fault tolerance of the software product. "The internal attributes are said to be indicators of the external attributes", it concludes, as represented in Fig. 13.

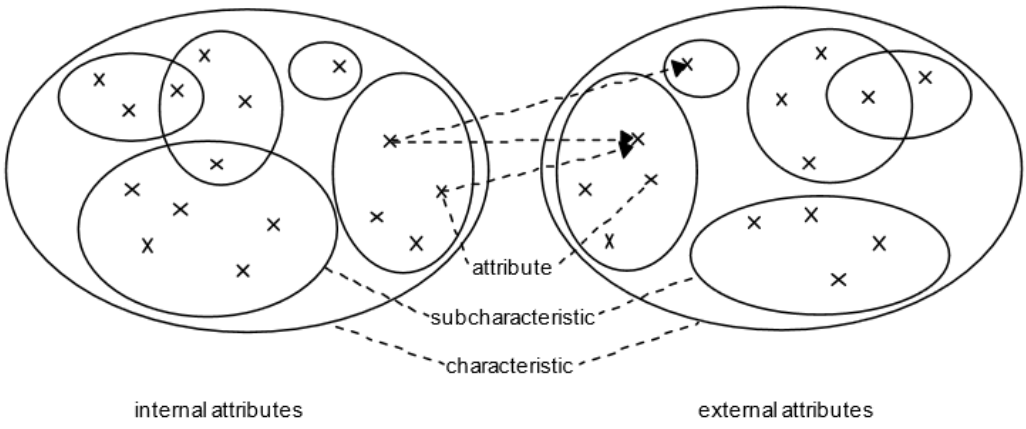

Fig. 13. Relationship between internal and external attributes (included in [34]).

As mentioned above, numerous quality models (and their corresponding set of quality attributes) have been proposed. The six main attributes of software quality proposed in [34] have already been mentioned. Boehm et al. [37] proposed their well-known software quality characteristics tree, shown in Fig. 14, in which its most specific characteristics (the leaves of the tree) are a necessary condition of some of the more general characteristics (closer to the root of the tree).

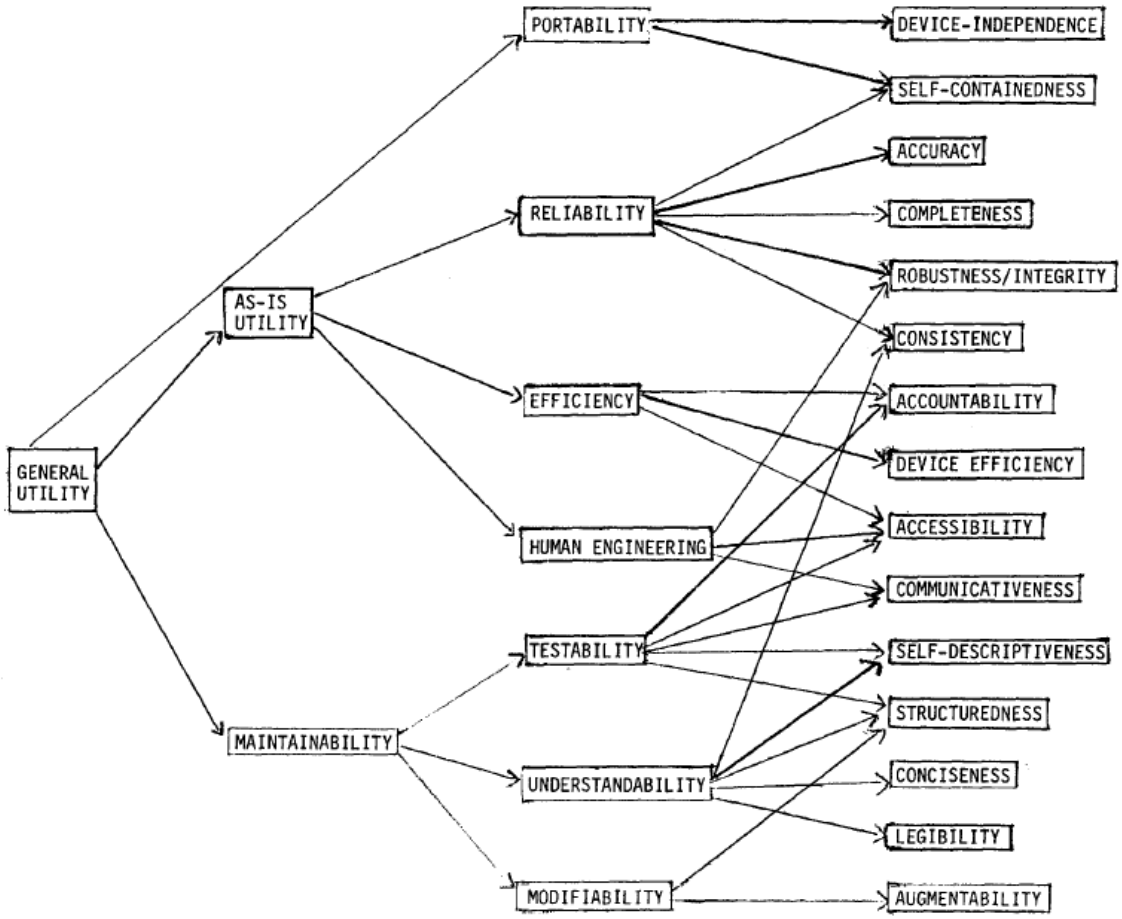

Fig. 14: The software quality characteristics tree (included in [37]).

B. Software Measurement

After having defined software quality, it is necessary to analyse how it can be assessed. In the software development process, quality is evaluated during the verification and validation process. It is generally considered that verification consists of checking that the implemented software corresponds to the requirement specifications, while validation means verifying that the software complies with the actual users needs. In the celebrated formulation of Boehm [38], verification answers the question “Are we building the product right?”, while validation replies to “Are we building the right product?”.

In industry, two activities stand out to evaluate the quality of software: tests and inspections. Testing consists of observing the dynamic behaviour of software during execution, usually with test data. Software inspections consist of the static analysis of software by an expert or an automated tool. It can concern the source code of the software product or other representations (models and abstract representations of the system).

To transcend a purely subjective approach and enable us to compare software products, it is necessary to resort to the field of software measurement: the extraction of numerical values from software, which can be used to summarise its characteristics and its quality. In other words, these measurements allow us to quantify the aforementioned quality attributes.

Depending on the aspects of software one desires to measure, the type of values used in software measurement range from metrics related to the software process, the programming techniques used, the documentation produced, etc. These measurements can be used mainly for two purposes: to evaluate the quality and characteristics of a software system, or to identify anomalous and sub-optimal components of the product.

Software measurements can mainly evaluate two aspects of software: the attributes of the software product itself or the process of developing this software [36] [39]. Some software quality attributes are difficult to evaluate merely analysing the software product. Since the quality of the software development process has an indirect impact on the quality of the software product, measuring and improving the process quality seems to be positive for the quality of the resulting software. Our study mostly focuses on the software product attributes, since it relates more directly to the quality of software.

In spite of the significant increase in research on software metrics in academia, books or conferences, the software industry at large has been less enthusiastic about adopting these metrics. Fenton [6], for example, observes a certain increase in the use of metrics in professional software development, but he regrets that the “industrial take-up of most academic software metrics work has been woeful” and it has been “based almost entirely on metrics that were around in the early 1970s”. Among other reasons, he underlines that a significant part of the academic work on this field is irrelevant for industry. He also argues that, while using software metrics entails an overhead on software development projects -between 4% and 8%-, the effectiveness of these metrics is still contested within the software industry.

Indeed, some studies have shown that software metrics are related to high-level quality characteristics [5] [40] [41] but the actual quantification of this relationship has never been fully accomplished (e.g. establishing unequivocal thresholds to interpret quality based on measurement values). Furthermore, several limitations to software metrics have been pointed out [6] [5]. Therefore, the concrete connection between metrics and quality remains under discussion both in the software industry and the academy.

Other reasons can be found for the reluctance to use software measurements in industry, such as the difficulty to quantify a return on investment, the lack of standards for software metrics, or the perceived difficulty to include process measurement techniques in the increasingly widespread agile software processes.

B1. Software Product Metrics

This study focuses mainly in one of type of software measurement, software product metrics, since it is more closely related to software quality than other types of measurements (e.g. development process metrics). Software product metrics is a very wide area of research, including those metrics common to all procedural and imperative languages, as well as those specific to object-oriented languages. In recent years, object-oriented metrics have been subject to more attention in research.

Object-oriented metrics can be defined at different levels of *granularity*: class-level, method-level, system-level, etc. For example, Catal et al. analysed in [10] fault proneness literature and came to the conclusion that most works made use of method-level granularity, such as the metrics proposed in the 1970s by Halstead [4] and McCabe [3]. Fig. 15 shows the clear predominance of this fine granularity (60% of the studies), followed by class-level metrics, such as those proposed in the 1990s by authors such as Chidamber and Kemerer [1], or Brito e Abreu and Carapuca [42]. The other levels of granularity are very rarely used.

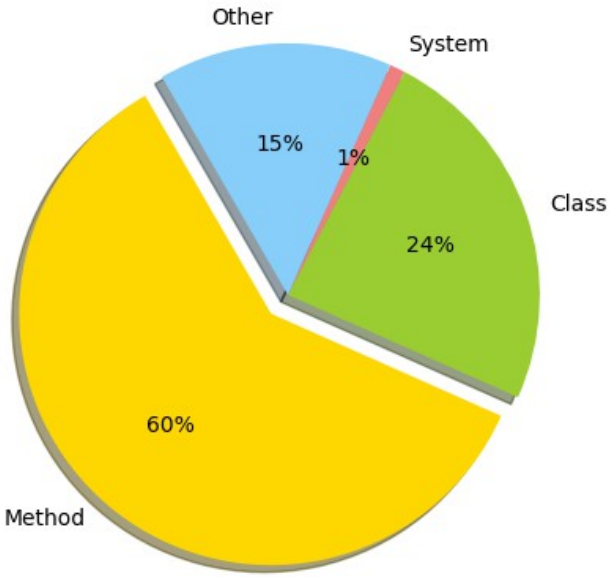

Fig. 15. Distribution of metrics granularity in fault proneness studies (data from [10]).

The less common granularities (i.e. those which are not method-level or class-level) could be further investigated in order to explore their potential in understanding software quality. Mainly, the system-level granularity seems particularly interesting: often the perceived

quality of a software artifact is not related exclusively to one specific method or class, but to the overall quality of its code, style and architecture patterns and the connection between its different components.

The most frequent method for extracting software product metrics is *static analysis*: deriving static metrics from the software artifact. Most static metrics are obtained from the analysis of the software source code, although they can also be elicited from design diagrams or other static representations of the software product design or implementation [43].

It is also possible to use "dynamic metrics", as proposed by [43], extracted from a running program during its execution. They are less frequently used than static metrics, mainly because they can be more difficult to extract. Furthermore, unlike static metrics, they cannot be applied in early stages of the software development process [5] [36].

B2. Review of Main Software Metrics

Since the late 1960's, lines of code (LOC) were used to roughly describe the size and complexity of a software product, together with KLOC (thousands of lines of code) [36] [6]. By the mid-1970s, the limitations of this metric became apparent. Mainly, due to the increasing variety of programming languages with different degrees of abstraction and verbosity, which became very difficult to compare among them.

The mid-1970s saw a great interest in finding new manners to assess the characteristics of software. Among these new metrics, one of the most popular is the complexity metrics proposed by McCabe [3] and specially his cyclomatic complexity V, defined by

$$V = E - N + 2\mathrm{P} \quad ,$$

where E is the number of edges of the control flow graph, N its number of nodes and P its number of connected components.

In 1977, Halstead proposed another relevant suite of metrics, which measured complexity, effort and understandability using four measures: the number of distinct operators, the number of distinct operands, the total number of operators and the total number of operands [4].

With the popularity of object-oriented languages, new class-level metrics were proposed in the 1990s, some of which will be used in chapter V. One of the most relevant are those proposed by Chidamber and Kemerer [1].

Their Lack of Cohesion in Methods (LCOM) measures the cohesiveness of a class: it quantifies the number of shared attributes used by different methods. This metric received critics. As a consequence, some improvements have been proposed, such as LCOM4

suggested by [44], which takes into account not only the shared class-level variables, but also the calls among methods.

High cohesiveness is recommended and the ideal value of LCOM and LCOM4 is 1, meaning a completely cohesive class, whereas other values are discouraged: higher values entail lower cohesiveness and, in LCOM4, a value of zero implies that the class does not contain any method.

Ref. [1] also proposes Coupling Between Objects (CBO) as a measure of the coupling of a class. Low coupling is recommended, since it increases the simplicity, understandability and maintainability of software. The CBO of a class is measured by counting the number of other classes to which it is coupled, i.e. the classes of the methods or variables it uses.

It also introduces the Depth of Inheritance Tree (DIT) metric. The DIT of a class is the length of the longest path from this class to the root of its inheritance tree (whose DIT value is 0). Inheritance is one of the characteristics of object-oriented languages and enhances the reuse of code. However, an excessive use of inheritance can increase the complexity of a software product. Some studies propose a threshold maximum value of 6 for DIT [5].

Other metrics proposed by Chidamber and Kemerer include the Weighted Methods per Class (WMC), the Response for a Class (RFC) or the Number of Children (NOC).

Brito e Abreu et al. [2] [42] also propose a set of metrics which has been widely adopted for software measurement: the Metrics for Object Oriented Design (MOOD) suite. In section V, the total Coupling Factor (CF) of repositories will be discussed. CF measures coupling: a class is considered coupled to another when it includes at least one non-inheritance reference to it. Unlike CBO, which also measures coupling, it does not take into account coupling related to inheritance and bidirectional coupling is counted twice instead of only once.

Other metrics included in the MOOD suite are the Method Hiding Factor (MHF), the Attribute Hiding Factor (AHF), the Metric Inheritance Factor (MHF), the Attribute Inheritance Factor (AIF) and the Polymorphism Factor (PF).

B3. Metrics and Fault-Proneness

Among the external quality attributes discussed in chapter II, one of the most frequently used is fault-proneness and presence of errors, defaults and bugs in software. This is usually considered an indicator of software reliability. The impact of software metrics on fault-proneness is due to the cognitive complexity, which leads to poor understandability and maintainability. Cognitive complexity depends on the coding style, which in turn is reflected in software metrics [43].

Different studies have shown empirical connections between software product metrics and fault-proneness. For instance, El-Emam et al. found a correlation between the probability of a class to have a fault and metrics such as the total number of attributes (NAI), depth of inheritance (DIT) and coupling [43].

However, the specific results in various studies vary depending on the type of metrics and software used. Clear, widely accepted evidence on the specific impact of software has not been attained yet and further study to consolidate conclusions is necessary [43].

At the same time, interest in fault prediction has increased significantly. Catal et al. [10] underline that most papers (86%) dealing with fault prediction in software have been published after the year 2000. As shown in Fig. 16, only 14% of the papers they analysed are previous to that year.

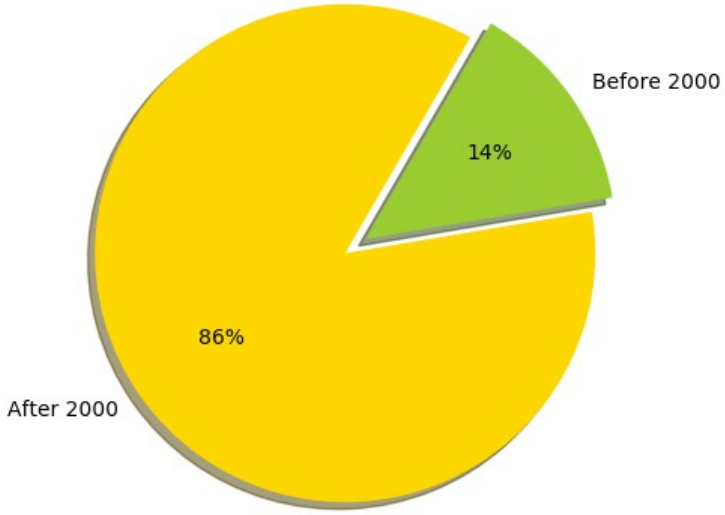

Fig. 16. Fault prediction papers published before and after the year 2000 (data from [10]).

C. Change Metrics

This study focuses mainly in software product metrics. Other recent studies focus on a different approach to measure software: *change metrics*. Unlike static software product metrics, change metrics do not analyse the content of the source code. Instead, they quantify the changes made to the code files over time. Usually, these data are retrieved from analysing the software version control system, such as CVS, Subversion or Git. For instance, Muthukumaran et al. [26] use code change metrics to analyse different versions of the Eclipse JDT project hosted in GitHub, in order to predict the number of bugs.

The advantages of using code change metrics can be their accuracy. According to Moser et

al., “overall change data are effectively better indicators of the presence or absence of software defects than static code attributes” [45]. Other researches come to a more nuanced conclusion. For example, D’Ambros et al. [46] carry out a comparison among the main bug prediction approaches, where static object-oriented attributes (such as Chidamber and Kemerer class metrics) predictive power performs equally or slightly better than the change metrics proposed by Moser et al., while their explanatory power is lower.

In our research we have preferred to use static software product metrics, mainly because, even though change metrics can help predict faults, they do not give any insight on the content of source code. They can help predict that a specific file is more likely to include bugs, but it does not provide any information about how should the developer code to avoid these bugs. Therefore, they do not contribute to a better understanding of software quality and its connections to software metrics and good coding practices. This will be addressed in detail in section V.A.2.

D. Software Metrics and Machine Learning

Machine learning has already been consistently used in research around software metrics and quality. Allamanis and Sutton use machine learning in [15] to elaborate new complexity metrics, that they call *data-driven complexity metrics*. They also predict the centrality of a module within a repository, in order to identify classes more likely to include utility reusable code.

In [18], data on software metrics and bugs are gathered from Java projects in GitHub, using the SZZ algorithm. They use this information to predict faulty components using 13 machine learning algorithms. This is a very interesting attempt to use machine learning to improve our understanding of software metrics and fault proneness. However, the limited number of projects analysed (15 projects) and the fact that the models are trained and tested on each project separately threaten the validity of the results for a wider scope of GitHub projects.

Muthukumaran et al. [26] extract a series of change software metrics from 5 different versions of the Eclipse JDT project in order to predict the number of bugs in the project, using five different machine learning algorithms (Gaussian Naïve Bayes, Decision Tree, Logistic Regression, Naïve Bayes Tree).

In [27], Dwivedi et al. extract 67 different object oriented metrics from source code from GitHub. They are fed as feature vectors to machine learning algorithms (Layer Recurrent Neural Network and Decision Tree) in order to identify software design patterns.

Malhotra et al. [47] use the Xerces dataset, included in the PROMISE repository [9], to compare the performance of 17 different machine learning algorithms in predicting defect prone classes based on the object-oriented metrics of these classes.

To summarise, software metrics have already been used in previous research to predict software quality. However, none of the works carried out used a sufficient number of different GitHub repositories to predict software quality using static software product metrics.

E. Summary

This chapter presents an analysis of the state of the art in software quality and metrics. It defines quality and discusses some of its main definitions in previous work. It discusses the different ways in which quality can be measured, mainly by quantifying either the product artifact or its development process.

There has been a significant increase in the use of software metrics in academia. However, this academic surge has not been followed in the software industry, which is more reluctant to the use of software measurement and, when it is not, it tends to use old metrics. The reasons for this reluctance stems, among other reasons, from the absence of clear quantifications of the relation between software metrics and quality.

Studies on software quality and fault proneness use almost only method-level and class-level metrics. Other granularities, mainly at system-level, seem thus to have a great potential for being further developed and explored in order to understand their connection to software quality. Some of the main software product metrics, which will be later used in chapter V, were introduced.

In this chapter, change metrics have also been presented. Although according to some of the existing literature, they are better indicators of fault proneness than software product metrics, and this could present some advantages (e.g. for application in industry), their characteristics are not appropriate for improving our understanding of software code quality. Fault proneness is one of the main software quality attributes used in previous works and recent years have seen an increasing interesting in it.

V. Contributions: Predicting Quality with Machine Learning

In section III.C, three criteria were chosen in order to select a subset of repositories for further investigating software quality and metrics. After analysing state-of-the-art in the field of software quality and metrics in IV, this chapter analyses the the software metrics and quality indicators of the subset of repositories from GitHub that fulfil the criteria from III.C.

Machine learning techniques seem specially fit to deal with the great amount of data that can be extracted for GitHub. Many standard machine learning techniques require a feature extraction phase prior to their application. In the case of this study, the features are software metrics, whose connection with software quality wishes to be better understood.

The feature extraction process is challenging: it requires taking a complex entity and identifying the relevant properties on which the machine learning algorithms can be applied. Therefore, it is important to select the appropriate features and the quality targets that will be used for machine learning. Once this has been defined, the appropriate tools must be used to extract these data from GitHub repositories.

While the GitHub metadata analysed in chapter III gave an overview of the use of GitHub (interactions between users, use of GitHub features, etc.), the software metrics extracted in this section from 3,074 GitHub repositories give a clear view on the source code hosted in this platform. This is valuable information to understand the coding practices and styles of GitHub users, as well as to better understand their quality.

The application of machine learning algorithms to the features and targets extracted can give an insight on whether the quality targets chosen are valid to reflect the actual quality of repositories, and whether these are related to their software metrics.

Specifically, V.A analyses the choice of the software quality targets and the metrics features that are to be extracted from GitHub repositories. Section V.B presents and discusses the tools (V.B.1) and process (V.B.2) established to extract these features (V.B.3) and targets (V.B.4). Section V.C analyses the software metrics extracted, which help give an overview of the properties of source code in GitHub. Finally, V.D examines the results of applying a linear machine learning algorithm on these data and proposes improvements for improving these results in future work.

A. Machine Learning Features and Targets

This section analyses which are the targets and features extracted from GitHub to build a dataset exploitable by standard machine learning algorithms.

A1. Software Quality Targets

Using GitHub for improving the understanding of the connection between software quality and software metrics requires establishing indicators of software quality. Given the massive amount of data provided by GitHub and exploitable by machine learning, it is important that these quality indicators can be automatically extracted instead of on an individual or manual basis.

Two main groups of indicators can be retrieved from GitHub data, those based on fault proneness and those connected to the popularity of the project (cfr. section II.D.1). This research explores both types of data.

Among the fault proneness indicators, we resort to the issue-tracking GitHub system. The total number of issues can be interpreted as an indicator of fault proneness and *bugginess* of the repository. However, some issues can not be related to bugs, but to additional features or improvements. A notion of "*buggy* issues" seems appropriate to elude this limitation. We define *buggy* issues as those having a message which contains any of the following words: "bug","fix","error","exception","problem","fail".

Other quality indicators are related to the popularity of the project: number of watchers, stargazers and forks.

To summarise, the following quality targets are retained:

- total number of issues
- total number of open issues
- total number of closed issues
- total number of *buggy* issues
- percentage of buggy issues over all issues
- total number of open *buggy* issues
- total number of closed *buggy* issues
- total number of watchers
- total number of stargazers
- total number of forked repositories

A2. Software Metric Features

There are different possibilities as to how and which software metrics to extract from GitHub repositories. These choices refer to whether the software metrics are static or dynamic, their granularity and their approach to the evolution of the repository.

Software metrics can be the result either of static analysis, which measures the static software artifact (mainly its source code or diagrams synthesising its implementation characteristics), or dynamic analysis, which extracts results from the execution of the software product (cfr. section IV.B).

This study will focus on static metrics. It would be impractical to use dynamic metrics, which required the compilation and execution of each repository in order to measure it. Furthermore, unlike dynamic metrics, static metrics can be gathered from early stages of the development lifecycle. When extracting massive data from GitHub repositories, it is not possible to know in advance with certainty the stage of the development of each repository. Static metrics therefore seem a better choice for this study than dynamic metrics, which require an extraction process very difficult to automate.

It is also necessary to decide whether to use software product metrics or change metrics. In this study, static software product metrics have been preferred to change metrics. While the latter can be accurate at predicting faulty components in software (classes, files, etc.), they provide no information about which characteristics of code make them more prone to faults. The use of change metrics can be valuable for industrial use, helping developers know which components should be receive special attention as being more likely to contain bugs. However, change metrics are less useful to find connections between software coding patterns and quality. Furthermore, change metrics are only applicable at late stages of development, once a significant number of modifications have been made to source code, enabling to gather the change metrics of the different software components.

Another choice for the software metrics concerns their granularity. In section IV.B.1, it was mentioned that object-oriented metrics can be measured at different levels of abstraction: system-level, class-level, method-level, etc. [43]. Most research papers (60%) on fault-prediction make use of the finest grained approach [10]: method-level metrics, such as those proposed by Halstead [4] and McCabe [3].

In the case of our study, system-level metrics have been chosen. Specifically, software product metrics are extracted at GitHub repository-level. This choice benefits from a number of advantages compared to other metrics, but also entails some risks.

Firstly, finer granularity metrics are very difficult to connect with automatically extracted software quality targets from GitHub repositories. The most widespread quality indicators of quality in GitHub refer to the project as a whole and not to a specific method or class. For instance, when one uses popularity indicators as quality targets (e.g. the number of GitHub

watchers, forks or stars), they are not applicable to specific classes or components of the repository, but to the totality of the project.

Fault indicators of quality (such as number of bug-fixing issues or commits) are also difficult to relate unequivocally to one specific method or class. For instance, the SZZ algorithm [19]–[21] retrieves the specific lines of code modified by a bug-fixing commit and use this information to track the commit which initially inserted this bug.

The SZZ algorithm is a very promising and smart approach, but it makes a strong assumption on the locality of bugs. In our view, often bugs are not so specifically and clearly located: a bug is the result of the difficulty to understand and apprehend the complexity of a program's source code, and the resolution of a specific bug can be made by modifying very different parts of the source code. The fault proneness of these specific parts of the code is not necessarily related to its concrete method or class metrics.

Our hypothesis is that of a more *holistic* approach to fault proneness: the *bugginess* of a repository is linked to its general coding style, patterns and understandability, rather than to finer-grained metrics.

System-level metrics have rarely been used fault prediction studies (1% according to Catal [10]) and in software quality research in general. Consequently, it seems an approach that could benefit from further exploration.

Using system-level metrics also entails risks. The main risk of using this coarser grained approach is that quantification can be less precise. For example, in a large repository with many classes, the impact of specific outliers (i.e. classes with abnormal class-level metrics) can be blurred by the other classes characterised by *average* metrics.

Specific system-level metrics have been proposed in the last years, since the years 2000s, and they are still subject to discussion among the community and researchers are still working on them [10]. Since they are still recent and subject to discussion, few software metric tools include these new specific system-level metrics.

On the contrary, some metric tools such as Analizo [48] take class-level metrics (such as those proposed by Chidamber and Kemerer [1] or Brito e Abreu and Carapuca [42]) and apply them to system-level. This allows to measure aggregate values of the class metrics for the component. For instance, this enables to quantify the average, median, variance, average or skewness of the classes of a project, among other relevant metrics.

In this research, since specific class-level metrics are still under discussion and are not included in most existing software metric tools, class-level metrics are used with a system-level scope.

Another choice that needs to be made is our approach with regards to the impact of the

repository evolution in the extraction of metrics (features for the machine learning algorithm) and software quality criteria (targets of the machine learning algorithm). GitHub is built upon the version control system Git. As such, repositories hosted in this platform evolve continuously. These modifications of code are grouped in *commits*.

There are mainly two options when handling the extraction of software metrics and the impact of the repository evolution. One, that we call *diachronic* approach, takes different commits as *snapshots* of the state of the repository at different moments in time. In order to analyse the connection between software quality and software metrics, this approach enables to compare the variations, for the same repositories, of metrics and quality throughout time. This *diachronic* approach is used in studies such as [49], [26], [17] or [18].

The other possible option can be labelled *synchronic* approach. It extracts the software metrics and quality criteria once, instead of using different snapshots over time. Under this perspective lies the assumption that the correlation between quality and metrics is not immediate or easily divisible over time: eventual bugs coming from undesirable software metrics can arise non immediately after the code has been modified. In a similar fashion, some quality criteria based on popularity (number of forks, watchers, etc.) seems to match better with this approach: most repositories are increasingly popular over time and changes in software metrics do not seem to have an immediate effect on popularity.

This *synchronic* approach will be more prone to compare the metrics and results of different projects, regardless of its evolution over time, instead of comparing different snapshots of the same repository throughout its lifecycle (as the *diachronic* approach would be inclined to do).

These are the reasons (a more holistic approach to the impact of evolution on repositories and a better connection with some quality criteria available in GitHub) why this research has chosen to use the approach we have called *synchronic*. However, both approaches seem valid. It would be revealing to explore and compare them in future work.

To summarise, this research extracts from GitHub software metrics that fulfil the following criteria:

- Static software product metrics, instead of dynamic or change metrics
- Class-level metrics extended to system-level granularity
- *Synchronic* approach to repository evolution

B. Data Extraction

This section discusses the tools used for retrieving the software product metrics and the extraction process from GitHub. The various software metrics extracted are documented and examined and the resulting dataset is presented.

B1. Extraction Tools

Different tools for metric evaluation exist. For example, Kayarvizhy reviews and compares a series of 10 object oriented metric tools in [50]. Different metric tools, provided with the same source code, can offer different results due to differences in the specificities of their implementation [51] .

In this study, the appropriate software measurement tool is selected based on its characteristics. First of all, among the criteria for selecting repositories, this research effort focused on Java projects. Therefore, some tools which do not support Java analysis are discarded, such as SD Metrics or QMOOD++ [50].

Also, given the number of projects to be analysed, it is not feasible to have to manually compile each repository. Furthermore, many of these projects must be configured before compilation or contain errors which hamper compilation. A tool that can extract data from source non-compiled Java files (".java", not ".class") is needed. This discourages the use of tools such as ckjm, Jdepend [50], Chidamber and Kemerer Java Metrics or Dependency Finder [51]

It is also impracticable to extract the metrics of each repository, one at a time. Instead, given the number of projects, we expect the tool to be able to process repositories in batch. Tools such as the Eclipse IDE plugins (Eclipse Metrics Plugin) are therefore not the best option, since they require importing each project individually into the specific IDE.

An open source solution is preferred to a commercial one. It allows to clearly understand how metrics are measured and facilitates that other researchers share the same tool. Replicability is also better guaranteed when the researcher knows what happens under the hood. This criteria suggests to push aside tools such as RSM, Jhawk, JMT [50] or Understand for Java. Some solutions used in the above mentioned papers seem not to be active any longer, such as Analyst4j or JMT.

Analizo is a tool that fits the conditions of our study. It is an open-source, non commercial tool, created within the scientific community [48]. It extracts a wide range of object oriented (OO) metrics for Java projects and files, which do not need to be compiled or even compilable. Furthermore, it allows to process repositories in batch, and not one by one.

B2. Extraction Process

In chapter III, metadata from GitHub were extracted to characterise the projects hosted on this platform and define a set of criteria to select the relevant projects for this research. Once

this criteria has been settled, the repositories are to be retrieved, i.e. download their actual code. From this code, the software product metrics and relevant metadata will be extracted, in order to define whether the projects fulfils the criteria and to use these metadata as an indicator of the repository criteria.

Retrieving the code of a repository and its relevant metadata requires around 10 or 20 calls to the GitHub API (depending on factors such as the number of pages of issues). Since the hourly limitation is of 5000 calls, approximately 300 repositories per hour can be retrieved. Given that only a 0.69% of the repositories fulfil the three criteria described earlier, only the data for two repositories can be expected per hour. Therefore, this approach, adequate to gather metadata for chapter III, is less adapted for finding repositories fulfilling the criteria established. A more efficient approach is to preselect in GitHub those repositories that fulfil the selection criteria. It is then possible to retrieve the current code and metadata of these repositories.

For this pre-selection, this study has used the GHTorrent database. Instead of querying directly the GitHub API, impractical for the reasons mentioned above, GHTorrent allows to easily query the database in order to retrieve the url of the repositories fulfilling the criteria set in section III.C.

GHTorrent database is a very big database. Already in January 2015, the MongoDB version stored 4TB of JSON data, while the SQL version of the data had more than 1.5 billion rows. The latest SQL versions, once decompressed, take around 270 GB in CSV files. We preferred to use a slightly older version of the database, from January 2016, which took around half of the size, making it easier to handle.

It is not necessary to load and process all the tables in the GHTorrent database. For the scope of this work, only a few tables (projects, issues, issue_labels, project_languages) were used, since they contain the information required to select repositories. The mirror offered a total of 25,364,494 repositories. Among them, 2,243,734 repositories had Java as their main language. Around half of them were not forked (1.109.893). In total, 122,074 projects fitted the three conditions in section III.C.

Once the repositories fulfilling the criteria were pre-selectionned, the process conducted in this study to download the repositories, extract the software product metrics and repositories metadata was the following.

After having identified in GHTorrent the set of repositories fulfilling the selection criteria, a batch script would download the sources of the repository, decompress them, and extract the software metrics with Analizo. This same script would retrieve the relevant metadata: date and time of extraction, number of issues, number of issues with labels or comments related to bugs and errors, etc.

As it had been done at the pre-selection stage, the metadata of the repositories could have

also been obtained by querying GHTorrent instead of retrieving them directly from GitHub. However, the metadata would refer to the state of the repository at the moment in which GHTorrent data were extracted, previous to the download and extraction of the repository. Therefore, it was not advisable to query GHTorrent for metadata, but rather to retrieve them at the same time of downloading the repository and extracting its metrics.

B3. Software Metric Features Extracted

After removing features which often yielded non-numerical values, 170 different features were retained from GitHub repositories. Some of them are directly extracted from the repository, while others are obtained from the individual classes and Analizo extract statistical metrics for the project.

Table III shows the repository-level metrics, together with the reference in which they are defined and the name in the dataset collected for this study. Further definitions and explanations on these metrics can also be found in [48] and [52].

TABLE III. Repository-level metrics extracted.

Repository-level metric	**Defined in**	**Column name in dataset**
Total Coupling Factor	[2]	total_cof
Total Number of Classes	-	total_modules
Total Number of Methods	-	total_nom
Total Lines of Code	[53]	total_loc
Total Number of Classes with at Least One Defined Method	-	total_modules_with_defined_methods
Total Number of Classes with at Least One Defined Attribute	-	total_modules_with_defined_attributes
Total Number of Abstract Classes	-	total_abstract_classes
Total Number of Methods per Abstract Class	-	total_methods_per_abstract_class
Total Number of Effective Lines of Code	[54]	total_eloc
Change Cost	[55]	change_cost

Table IV shows the class-level metrics, collected from the GitHub repositories, from which Analizo extracted statistical metrics for the repository. Specifically, for each of these metrics, it measured the following 10 statistical values:

- Mean

- Sum
- Variance
- Median (“quantile_median” in the dataset)
- Standard deviation
- Minimum (“quantile_min” in the dataset)
- Maximum (“quantile_max” in the dataset)
- Lower quartile (“quantile_lower” in the dataset)
- Upper quartile (“quantile_upper” in the dataset)
- 95th percentile ("_quantile_ninety_five" in the dataset)

TABLE IV. Class-level metrics extracted from repositories.

Class-level metric	Defined in	Column name in dataset
Afferent Connections per Class	[56]	acc
Average Cyclomatic Complexity per Method	[3]	accm
Average Method Lines of Code	-	amloc
Average Number of Parameters per Method	[56]	anpm
Coupling Between Objects	[1]	cbo
Depth of Inheritance Tree	[1]	dit
Lack of Cohesion of Methods	[44]	lcom4
Lines of Code	[53]	loc
Maximum Method Lines of Code	-	mmloc
Number of Attributes	[56]	noa
Number of Children	[57]	noc
Number of Methods	[56]	nom
Number of Public Attributes	[56]	npa
Number of Public Methods	[56]	npm
Response For a Class	[1]	rfc
Structural Complexity	[55]	sc

A significant part of these metrics have already been addressed in section IV.B.2. The definition of the other metrics can easily be found in the research papers mentioned in tables III and IV. It would be cumbersome to reproduce here again each one of these definitions.

The features extracted include the main software quality related metric dimensions. They include measurements related to the size of the repository (e.g. Total Number of Classes, Total Lines of Code, Total Number of Effective Lines of Code, etc.). Also metrics significant to the coupling of the classes are included: Total Coupling Factor, Coupling Between Objects, Afferent Connections, Response for a Class, etc.

Furthermore, metrics measuring the complexity of the project are included, e.g. Average Cyclomatic Complexity, Structural Complexity, etc. They also measure other relevant properties of the repository, such as the use of inheritance (with metrics such as the Depth of Inheritance Tree, Number of Children, Total Number of Abstract Classes), cohesion (e.g.

Lack Of Cohesion of Methods) or encapsulation (e.g. Number of Public Attributes or Number of Public Methods).

B4. Dataset Availability

The dataset extracted in this chapter can be found in https://github.com/david-fdez/metrics_quality_dataset. It has been encoded using the Python library NumPy library[6]. Specifically, using its method `savetxt()`. The features have been saved as a two-dimensional matrix array (each line represents one repository, each column one feature) in the file "features.out".

To load with Python the features, the following two lines of code enable to do it:

```
import numpy as np
features = np.loadtxt('features.out', delimiter=',\t\t')
```

The name of the specific feature of each column (i.e. the software metric) is included in the header of the file (the first line of the document).

The quality targets have been saved in different files, each of which contains a one-dimensional array, i.e. the files named:

- target_issues_count.out
- target_open_issues_count.out
- target_closed_issues_count.out
- target_buggy_issues_count.out
- target_open_buggy_issues_count.out
- target_closed_buggy_issues_count.out
- target_forks_count.out
- target_stargazers_count.out
- target_watchers_count.out

They can be loaded in a similar fashion to the features. In order to extract the targets of a specific type (e.g. for the "target_issues_count.out" file), it is just necessary to execute the following instructions with Python:

```
import numpy as np
targets = np.loadtxt('target_issues_count.out', delimiter=',\t\t')
```

6https://docs.scipy.org/doc/numpy/index.html

C. Characterisation of Code in GitHub

For the scope of this paper, a total of 4,449 repositories were processed. Among them, 726 no longer existed. Analizo was unable to extract the metrics of another 644 repositories. 5 repositories were too big (more than 1Gb of source code). For the remaining 3,074 repositories, all the metrics that Analizo is capable of extracting (described in [52]) are included in the dataset. This section focuses on some of these classical software metrics that allow to characterise the dataset.

Analysing the metrics, it clearly shows a tendency of most repositories to be rather small. Fig. 17 presents the distribution of repositories by their total size and fig. 18 by their number of classes. It is indeed very rare to find repositories with a size bigger than 200MB. A significant number of repositories have few classes, very rarely exceeding systems of more than 200 classes (Fig. 18).

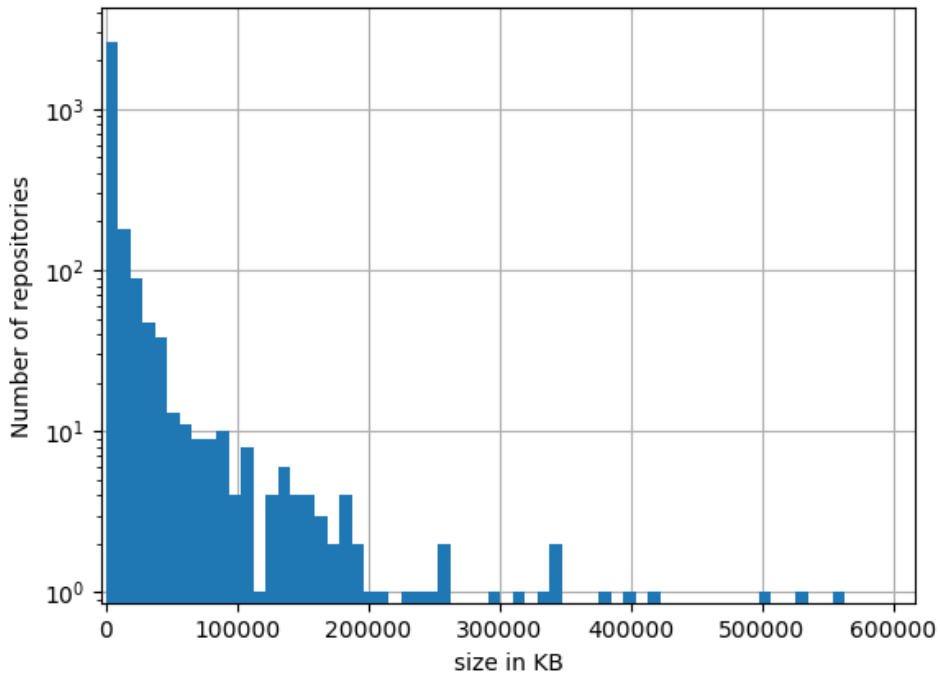

Fig. 17. Distribution of repositories by their total size..

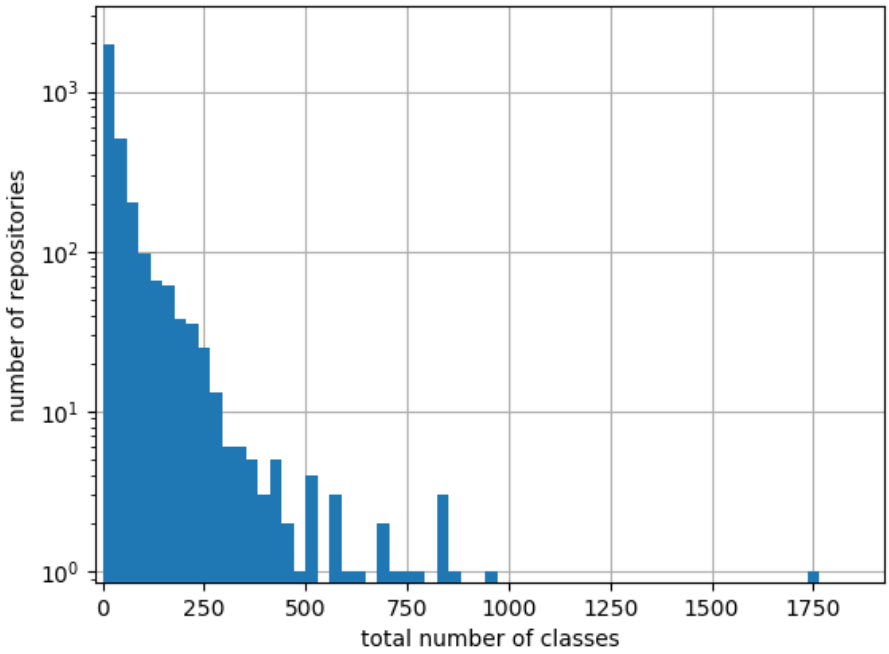

Fig. 18. Distribution of repositories by number of classes.

Fig. 19 shows the distribution of repositories by their total effective lines of code (actual lines of source code, excluding empty lines, comments, etc.). It confirms that it is unusual to find very big projects in GitHub: most of them contain less than 10,000 effective lines of code and it is rare to encounter repositories consisting of more than 40,000.

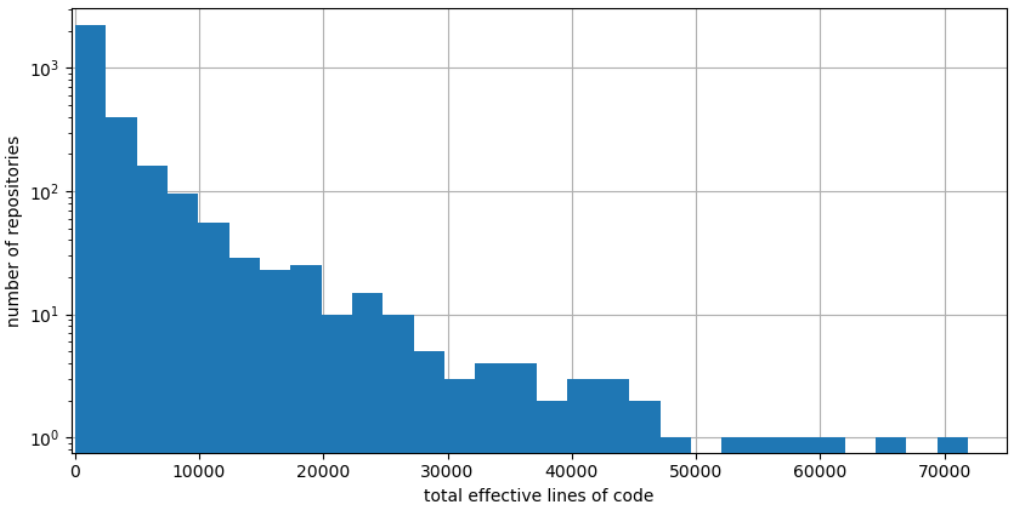

Fig. 19. Distribution of repositories by total effective lines of code.

The complexity of source code has an impact on its quality. High cohesion of classes manifest high quality, by which each class has one, and only one, purpose. One way to measure cohesion is the lack of cohesion in methods (LCOM) metric, proposed by

Chidamber and Kemerer [1] and later improved by Hitz et al. (LCOM4) [44]. It quantifies the number of groups of related methods and fields that exist within a class. It has been described in IV.B.2.

The desirable value for LCOM4 is 1, meaning a highly cohesive class, whereas bigger values reflect classes that should be split. A value of 0 means that there are no methods in the class, which is also not recommended. Fig. 20 shows the number of repositories in GitHub, depending on the mean value of LCOM4 of its classes. Most repositories have non-cohesive classes, having a mean LCOM4 value of around 2.5 (greater than the ideal value: 1.0).

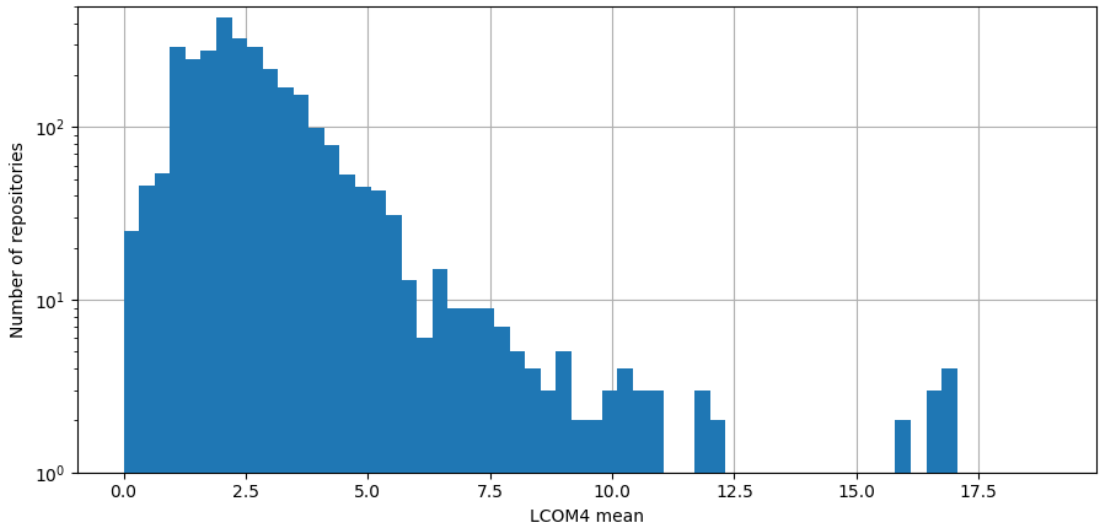

Fig. 20. Distribution of GitHub repositories by their LCOM4 mean.

Analysing the classes with a maximum value of LCOM4 for each project, the non-cohesiveness of GitHub repositories is confirmed. Fig. 21 illustrates that a great majority of GitHub repositories contain classes exceeding a LCOM4 value of 10. In other words, all these projects contain at least one class which should be split into 10 or more smaller and more cohesive classes.

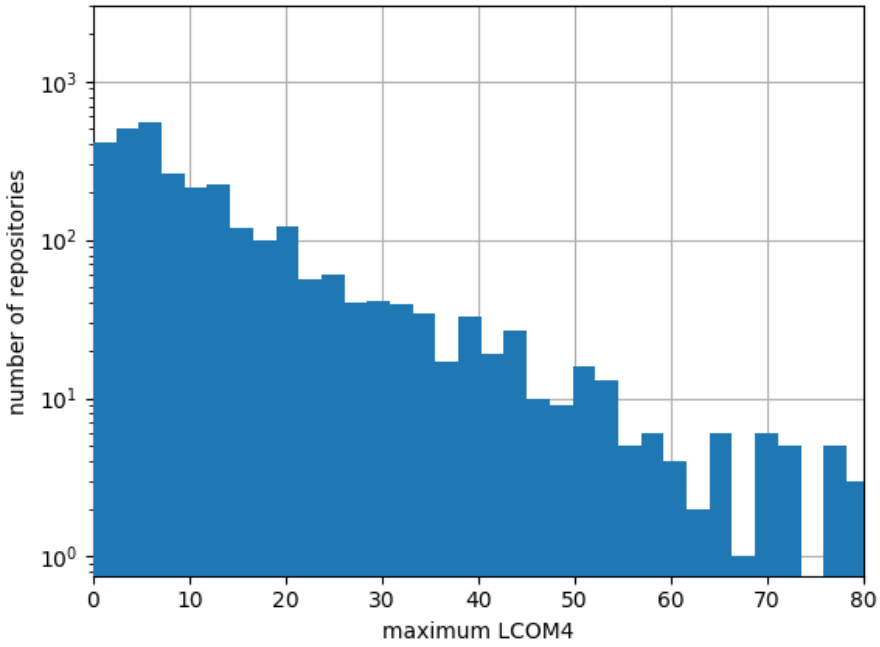

Fig. 21. Distribution of GitHub repositories by their class with a higher LCOM4 value.

Another element to measure dependencies is how coupled classes are to each other. This can be measured using the Coupling Factor (CF), proposed by Brito e Abreu et al. [42], or the Coupling Between Objects (CBO), put forward by Chidamber and Kemerer [1]. Both metrics have been introduced in IV.B.2.

High coupling is undesirable. [58] proposes CF not to exceed 11.2%, while [59] suggests that CBO should not be above 14. It is indeed the case: most of the repositories analysed in GitHub present low coupling.

Fig. 22 shows the CBO mean of the GitHub repositories analysed. It presents two interesting results. Firstly, many repositories have no coupling at all (their CBO mean is close to 0). This is related to Fig. 18 too: when repositories contain few classes, it is more likely that each class depends on few others. Secondly, the mean CBO in repositories is evenly distributed between values 0 and 2. Their classes have a low coupling to each other.

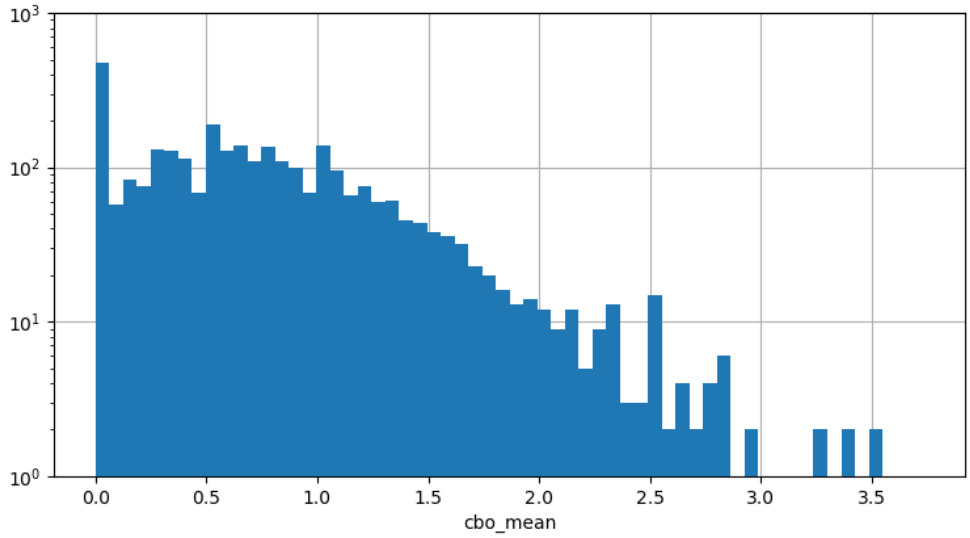

Fig. 22. Distribution of GitHub repositories by their CBO mean.

Most repositories have a low mean CBO, but this does not imply that, within these repositories, there are no classes with a too high coupling. Fig. 23 shows the distribution of repositories by the maximum CBO of their classes. It is remarkable that, unlike the mean CBO which is very low, many repositories have classes with a very high coupling (i.e. above the value of 14 that [56] recommends).

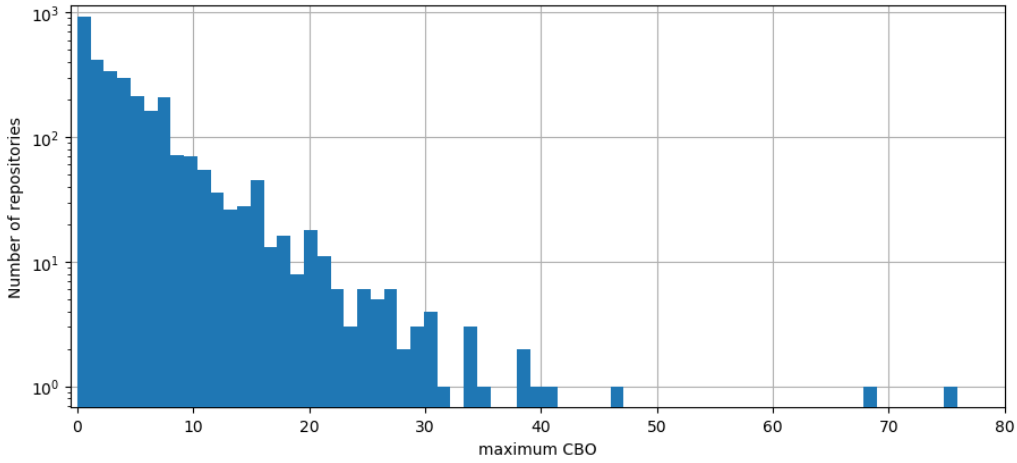

Fig. 23. Distribution of repositories by the maximum CBO of their classes.

Also the analysis of the CF mean illustrates also a rather low coupling among the repositories analysed. The distribution of repositories by their total CF is illustrated in Fig. 24. It indicates that a significant part of the projects had CF values below 0.2. A peak can be observed at value 1 (100% coupling), which is explained by the fact that this value is given to all of the projects which consist of a single class.

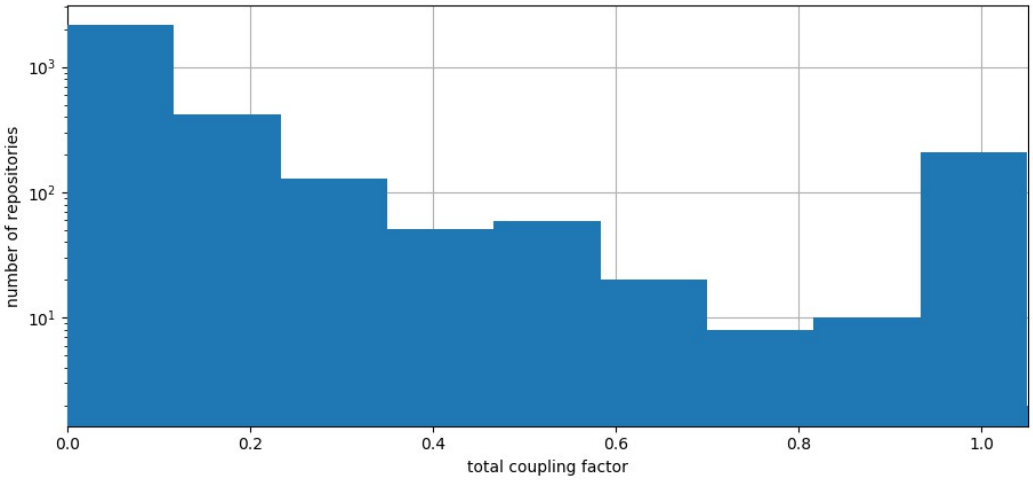

Fig. 24. Distribution of repositories by their total CF.

Terceiro et al. propose in [60] a measure for structural complexity, measured as the product of LCOM4 and CBO. Fig. 25 shows the distribution of GitHub repositories by their structural complexity mean. It confirms that most of the GitHub repositories analysed have low complexity.

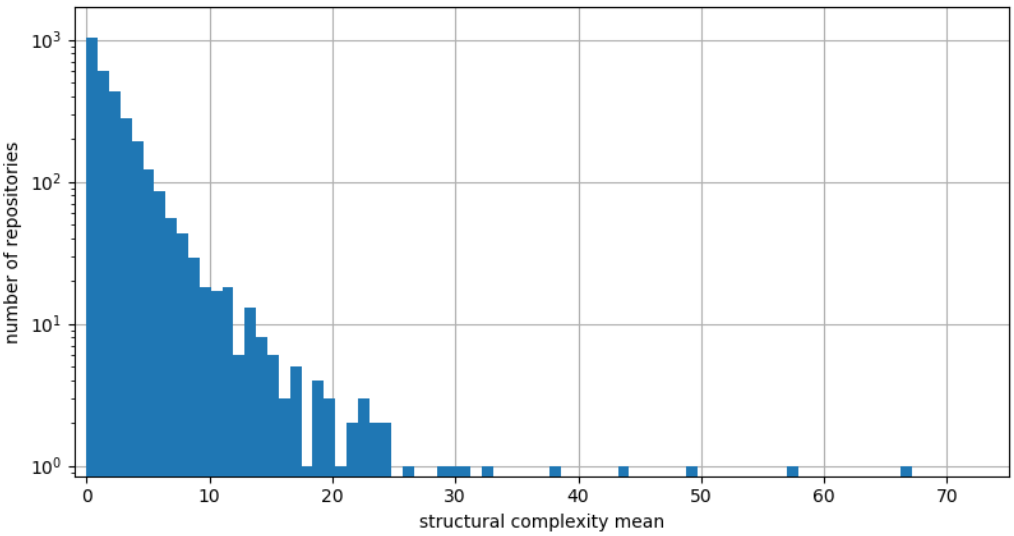

Fig. 25. Distribution of GitHub repositories by their structural complexity mean.

It is also interesting to compare the mean number of methods per class among the different repositories in GitHub, as is shown in Fig. 26. Most projects have less than 10 methods per class, reaching its peak around five methods per class.

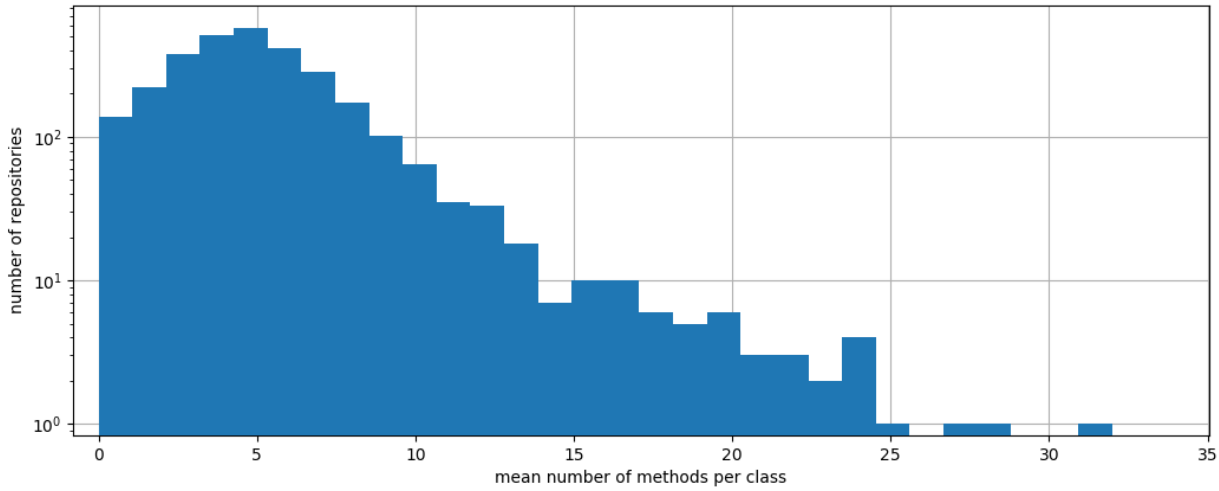

Fig. 26. Distribution of repositories by their mean number of methods per class.

One of the particularities of object oriented languages is the possibility to inherit methods and/or variables from parent classes. While method inheritance promotes reuse of code, it also increases complexity and can make it more difficult to understand the behaviour of a specific class. It is therefore recommended not to have a too big depth of inheritance tree (DIT) [1] metric value. Some studies propose not to exceed a DIT value of 6 [5]. Fig. 27 shows the DIT mean values of GitHub repositories, indicating that most repositories do not use inheritance excessively: their mean DIT values mostly range from 0 to 2.

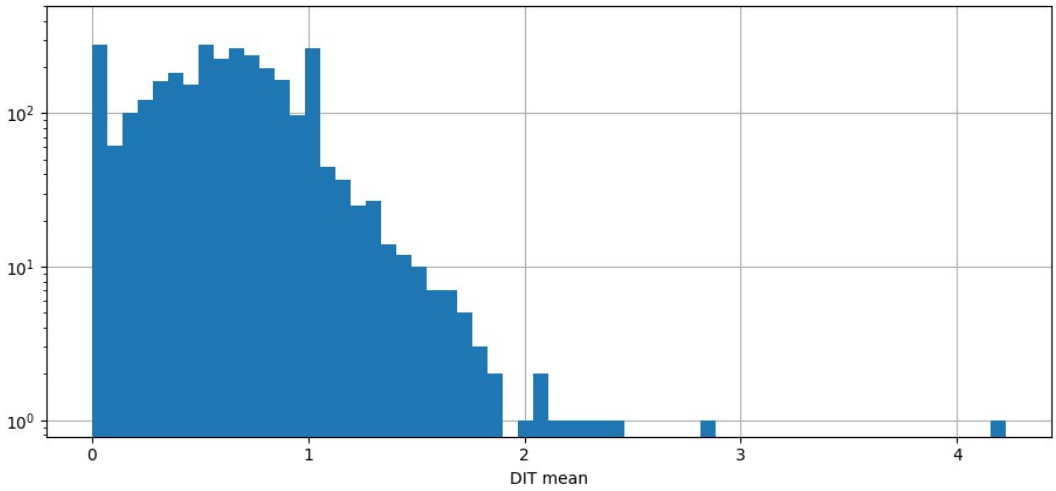

Fig. 27. Distribution of GitHub repositories by their DIT mean.

The fact that inheritance is rarely abused in GitHub is confirmed by examining also Fig.28, which shows the classes with maximum depth of inheritance within each project. It confirms

that the deepest class of most GitHub projects has two or less ancestor classes. It is very rare when the DIT of a repository exceeds a value of 6.

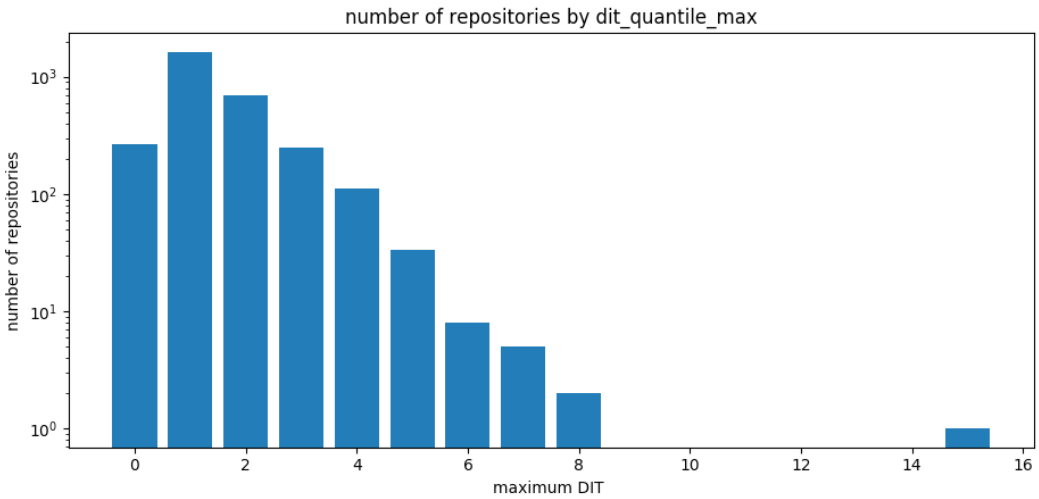

Fig. 28. Distribution of repositories by the maximum DIT among its classes.

In summary, the selected GitHub repositories have a tendency to be small, lack cohesion, not be very coupled, not use inheritance excessively and, in general, not be too complex. However, the dataset contains rich and varied values, making it suitable for analysing its impact on quality assessment indicators (such as the number of bugs, issues or other metadata extracted from GitHub).

D. Predicting Quality with Software Metrics using Machine Learning

This section explains the machine learning algorithm used to predict the quality of the GitHub repositories and analyse the results obtained.

D1. Presentation of Results

After having presented the software metrics characteristics of these 3074 repositories, these data are fed into machine learning algorithms to evaluate its predictive and explanatory power. Different targets and subsets of the features mentioned in section V.C were used.

To assess the efficiency of this score, we chose to use the Lasso algorithm, as applied by the Python machine learning library scikit learn. It is a linear model which tries to minimise the function

$$\min_{\omega} \frac{1}{2\mathrm{m}} \| X\,\omega - Y \|_2^2 + \alpha \| \omega \|_1 \quad ,$$

where m is the number of samples, ω is a vector of the coefficients of the model, X is the matrix of features, Y is a vector of the targets and α is a constant value, which is used as penalty term to limit the number of non-zero coefficients in ω [61].

This penalisation of non-zero coefficients, make this algorithm specially fit for cases, such as this one, in which the number of features is high and it is desired to reduce the number of features taken into account by the model. Furthermore, it can be efficient even when it is not fed with an enormous number of samples.

This Lasso algorithm was applied with 3-fold cross-validation (CV). This means that the training data is split into three groups, which are used to find the optimal value of the (alpha) parameter. Each one of the three subsets is used to test the effectiveness of a model trained with the other two subsets. The average of the three partitions is used to compare the possible values of the α constant.

As for the training and test subsets, a 75% of the dataset samples are used to train the Lasso model, while the other 25% are used to calculate the score of the model. This split of the dataset into two subsets, one for training the model and another one for testing it, is common practice in machine learning. If the tests and the training of a model were to use the same data, they would easily lead to *overfitting*: the model learns too precisely the specificities of the dataset but fails to predict new samples.

The score was calculated using the coefficient of determination R^2 of the predicted output, as calculated by scikit-learn, which yields a result from 0 to 1 (where 1 is a perfect score and 0 means that there is no linear relationship between the predicted outputs and the actual targets). This is given by

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{m-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{m-1} (y_i - \bar{y})^2} \quad ,$$

where m is the number of samples in the test, $\hat{y}_i$ is the value predicted by the model for the i^{th} sample, y_i is the actual target value of that i^{th} sample in the test subset and $\bar{y}$ is the mean of the actual target values in the test subset [62].

Various subsets of features and quality targets were measured, as explained below. For each of these combinations, the results were repeated 20 times, since the coefficients learnt can fluctuate from one time to the other, depending on the split of the training and test subsets. In the tables below, the best and worst score is included. In certain cases, the R^2 score of the

predicted model can be negative (when its predictions are arbitrarily negatively correlated with the actual targets). For those negative cases, the score was calculated as having a value of 0.

Table V shows the minimum, maximum and median R^2 score using each of the 10 possible targets, having trained the Lasso algorithm with the data of the 170 features.

TABLE V. Lasso results using all features.

Target	**Minimum R^2 score**	**Median R^2 score**	**Maximum R^2 score**
total number of issues	0.0	0.027	0.187
total number of open issues	0.0	0.016	0.098
total number of closed issues	0.0	0.015	0.206
total number of *buggy* issues	0.0	0.0	0.200
percentage of buggy issues over all issues	0.007	0.019	0.029
total number of open *buggy* issues	0.0	0.018	0.078
total number of closed *buggy* issues	0.0	0.030	0.293
total number of watchers	0.0	0.0	0.003
total number of stargazers	0.0	0.0	0.0
total number of forked repositories	0.0	0.0	0.008

In general, the results do not show a strong correlation between the software metrics features and the quality targets. The best R^2 scores are yielded when using the total number of closed *buggy* issues (0.293), the total number of closed issues (0.206) and the total number of *buggy* issues (0.2). The weights for the 170 features seem slightly unstable in certain cases, ranging from low scores to higher scores depending on which samples are used in the test and training subsets (e.g., for the target "total number of closed *buggy* issues", R^2 yields scores ranging from no correlation to a 0.293 score). One reason for this instability can be the high number of features used in the training algorithm for a dataset of 3,074 samples.

Using fewer features confirms their impact on the volatility of the trained models. With less features, the R^2 scores tend to become more stable, but also become closer to zero, confirming that there is a weak correlation between the predicted results and the testing set. Specifically, table VI shows the R^2 score using 16 features, which represent the averages of the different software metrics enumerated in V.B.2. It also includes the results using 5 features related to cohesion and complexity which are particularly meaningful in fault proneness studies (average lines of code, average depth of inheritance tree, average coupling between objects, average lack of cohesion of methods and average cyclomatic complexity).

TABLE VI. Lasso results using 16 features.

	R^2 score using 16 features			R^2 score using 5 features	
Target	**Min.**	**Median**	**Max.**	**Min.**	**Median**
total number of issues	0.0	0.015	0.041	0.0	0.022
total number of open issues	0.0	0.016	0.043	0.0	0.025
total number of closed issues	0.0	0.016	0.058	0.0	0.021
total number of *buggy* issues	0.0	0.007	0.057	0.0	0.022
percentage of buggy issues over all issues	0.0	0.011	0.025	0.0	0.014
total number of open *buggy* issues	0.0	0.006	0.022	0.0	0.012
total number of closed *buggy* issues	0.0	0.019	0.050	0.0	0.012
total number of watchers	0.0	0.0	0.003	0.0	0.0
total number of stargazers	0.0	0.0	0.0	0.0	0.0
total number of forked repositories	0.0	0.0	0.0	0.0	0.0

D2. Analysis of Results

The results show that the models trained with these data offer a low predictive capacity: they do not show a strong relation between the software metrics and the quality targets extracted from GitHub. This is an important conclusion since it provides some insight on the future work necessary to fully exploit GitHub as a source of data for the study of software quality and metrics.

Fault-proneness related quality targets clearly outperform quality criteria related to popularity, which offer very poor results. Targets such as “total number of closed *buggy* issues” and “total number of closed issues” have R^2 scores of 0.293 and 0.206 respectively, while the best popularity-related target result only reaches a value of 0.008 (total number of forked repositories).

Using the 170 metrics features yields better results but it also makes them more dependent on the splitting of the training and test subsets, making them more unstable. With less features, the scores were more stable but also were closer to zero (less correlation for the predicted values).

A plausible reason for the low correlation scores is that the criteria targets are not precise enough to reflect the actual quality of each repository. As mentioned in chapters II and III, the popularity, size and activity of GitHub repositories are very unequal and have very skewed distributions. Most projects are personal or have a small community of developers, are small, rarely use interaction features such as issue tracking and have few activity. However, a few projects are very popular or collective, or are very big, or accumulate most of the activity and interactions of GitHub.

This uneven distribution (of the popularity, size and use of GitHub interactive features) has an impact on the quality targets. Issues and *buggy* issues could be a good indicator of default proneness. However, the GitHub projects that are very popular will accumulate a greater numbers of issues than those less popular, regardless of the actual quality of their code, since less popular repositories barely use issue tracking. The target criteria related to popularity, such as the number of forks, stargazers or watchers, are also insufficient. The most popular projects will have more watchers, forks and stargazers than the rest, but that does not necessarily imply that their software quality is greater. In a similar fashion, differences of size or activity between repositories have an impact on these quality targets and bias their connection with software metrics.

These results show the impact of bug reporting in GitHub being done on a voluntary principle. This allows the developers working on a repository to use bug-tracking systems external to the issue-tracking GitHub system, or to not use any bug-tracking system at all. In both cases, using the number of issues of a repository to indicate its fault proneness is misleading. There is not a unique reason that explains why some repositories use issues and others do not. It is safe to assume that the most popular repositories will use more frequently the issue-tracking system, but other factors surely play also a role (e.g. the type of project, the number of developers, their personality and conscientiousness, etc.).

In order to exploit the potential of GitHub, it is crucial to discover better quality indicators that can be automatically extracted at massive scale. To achieve this, one could analyse in detail a subset of repositories for which quality measurements outside GitHub can be found (for instance, expert inspections of these repositories or repositories systematically using bug tracking systems outside GitHub). Other quality indicators could be analysed and evaluate their connection with the actual quality of the software. For example, one could envisage using technical debt (which can be calculated by coding tools such as SonarQube [63]) or resorting to the time which repositories take to fix buggy issues (although this risks being also biased by the popularity of the project).

Another way worth exploring is to group the different repositories in clusters (according to their popularity, size, activity, etc.) and use the machine learning algorithms within them. Besides, in addition to the selection criteria set in section III.C, thresholds could be used to exclude projects with abnormal characteristics with regards to size, popularity, activity, etc.

In addition to the inadequacy of the quality criteria used, another possible reason for the low scores of the machine learning algorithm is the use of class-level metrics at system-level. Applying class metrics to repositories has some perils. For example, non relevant files (such as configuration files, non object-oriented files, etc.) included in the repository can distort the metrics of the repository as a whole. Furthermore, the existence of several classes can make that different classes with values outside the norm are not really taken into account and are finally evened out around the average value.

The absence, to our knowledge, of system-level metrics hinders the exploitation of these type of metrics for GitHub repositories. The use of this granularity is promising but rather recent. In order to fully use the GitHub potential to study software quality and metrics, it is desirable to develop software metric tools including these type of granularity.

Another interesting research path would be to try a combination of class-level and system-level metrics. This could possibly benefit from the advantages of both granularities: the specificity and concretion of class-level metrics, and a more holistic and integrated approach provided by the system-level metrics.

It would also be revealing to compare the efficiency of the *synchronic* and *diachronic* approaches to repositories evolution in GitHub quality studies. We called *diachronic* to the approach that analysed the evolution in a project throughout time (mainly using different commits of different moments of a repository lifecycle). The *synchronic* approach, on the contrary, measures once the aggregated state of the repository and compares it to the aggregated state of other repositories. While other works uses a *diachronic* approach, this one suggests using a *synchronic* approach. This choice was made based on our understanding of bug introduction. Future work could empirically confirm or reject this choice.

E. Conclusions

This chapter examines the relation between software metrics and quality in 3,074 GitHub repositories using machine learning. Specifically, the features and targets to be fed to the regression algorithm were defined: features would be the software metrics and the targets would be the software quality indicators. Software metrics were extracted following some choices which were discussed: the metrics used were static software product metrics, with class-level metrics gathered at system-level component and a *synchronic* approach to the repository evolution (this approach was defined in contrast with a *diachronic* approach).

Ten different quality targets were extracted and discussed, related both to the fault-proneness of the project and to its popularity. The extraction process was reviewed. Various software metric tools were examined and Analyzo was retained.

The data extracted illustrate that most repositories in GitHub have few classes. Their code lacks cohesion, is not very complex and does not overuse inheritance. It also shows that most classes have low coupling. The variety of metric values make the dataset suitable for studying software quality.

The scores of the machine learning model show that the correlation between the predicted values and targets is weak. The main reason is that the quality targets extracted do not seem to reflect precisely the actual quality of the repository, since they are biased by other characteristics of the repository (size, popularity and use of GitHub features). The quality

criteria targets related to fault-proneness perform better than those related to the popularity of the project.

The results shed light into possible improvements of these results. Mainly, they show that it is crucial to find quality target criteria suited for exploring GitHub at massive-scale. Various possibilities are suggested: using clusters to reduce the impact of other properties of repositories, comparing diverse GitHub quality criteria with external indicators of quality (external bug-tracking systems, technical debt as calculated by tools like SonarQube [63], etc.).

The study of GitHub software quality and metrics could also be improved by developing metric tools which extract system-level metrics, in addition to the class-level ones that most of them currently include. Also, further progress could derive from empirically comparing the *diachronic* (the state of a project is matched against its own state at some other time) and *synchronic* (the aggregated state of one project is compared to the aggregated state of another project) approaches to repositories evolution.

VI. Conclusions and Future Work

This section summarises the conclusions of this study, proposes future work derived from its findings and outlines its contributions to academic research.

A. Conclusions

The work reported in this thesis has explored the possibilities of using machine learning on GitHub projects to improve the understanding of software quality and metrics. It shows that GitHub is a valid source of data for this research and that it offers an enormous amount of available data which can be exploited with machine learning. It also highlights some pitfalls related to the use of this platform in scientific research, such as replicability.

The use of public datasets is encouraged and some of the limitations of the current available datasets are analysed. A dataset with the metadata of 71,942 public repositories has been gathered and made publicly available (https://github.com/david-fdez/metadata_dataset). These data allow to draw the current characteristics of GitHub. Non-software and personal projects have increased in the last years, while issue-tracking (the main feature for user interaction and bug tracking in the platform) has remarkably decreased.

In order to create a subset of GitHub repositories relevant for software quality studies, selection criteria were discussed and defined. They must be *non-fork* repositories written mainly in Java that make use of the issue tracking system.

This study also examined the difficulties to incorporate software metrics into the software industry. Mainly, due to the absence of concrete quantification of the relationship between different software metrics and higher level quality properties.

The study analysed different approaches to the extraction of machine learning features and targets from GitHub repositories. The metric features gathered were static software product metrics, with class-level granularity applied at the whole repository and a synchronic approach to the evolution of the code. Ten different quality targets were collected, related either to the fault-proneness of the project or to its popularity.

A dataset containing the metric targets and quality targets of 3,074 has been built and made publicly available (https://github.com/david-fdez/metrics_quality_dataset). This allows to assess the characteristics of code hosted in GitHub. Among other properties, it shows that projects in GitHub are small and have few classes, lack cohesion and have low values of coupling, are not very complex and make a limited use of object-oriented inheritance.

The created dataset dataset was fed into a machine learning algorithm which showed weak correlation between the metric features and the extracted quality targets. This is mainly due to the imprecision of the chosen quality targets, biased by other properties such as the project popularity. It makes clear that, in order to study the quality of GitHub projects, it is crucial to find quality indicators that reflect accurately the actual quality of the project. From this results, other proposals to deepen our understanding of software quality in GitHub were examined.

B. Future Work

The data and analysis conducted in this research contribute to gain insight into the possibilities that GitHub offers to improve our understanding of software quality and metrics using machine learning techniques. They also suggest new paths to continue the investigation in this promising area.

In the selection of features and targets for machine learning, some additional analysis have already been proposed throughout this study:

- Crucially, finding appropriate quality indicators that can be automatically extracted from GitHub projects. It would be interesting to compare the GitHub repositories metadata with external measurements of quality not related to this platform (experts analysis of code, bug tracking systems, technical debt, etc.).
- Using clusters of repositories with similar characteristics (i.e. size, popularity, number of issues, etc.) to prevent them from biasing the data.
- Attempting other approaches to software metrics extraction: comparing synchronic and diachronic approaches, combining class-level and system-level granularities. It would also be useful to develop software metric tools which provide system-level granularity metrics.

Once the selection of features and targets in GitHub is perfected, a larger dataset could be extracted and made publicly available. Various machine learning algorithms should be applied in order to evaluate the performance of existing metrics and to determine which of them are more efficient in predicting software quality. This would also help propose evidence-based optimal values for software metrics, probably the Achilles' heel of these metrics.

A further step could be to use machine learning (multivariate approach) to identify relevant software metrics combinations. By doing so, new software metrics could be created, aimed at yielding better results than classical metrics.

C. Research Contributions

Here below, some of the contributions to academic research are briefly summarised.

In chapter II :

- Compilation and presentation of the current state of research in social coding platforms such as GitHub, with a special focus on the use of machine learning techniques and software metrics.

In chapter III:

- Making publicly available an exploitable database containing metadata of 71,942 GitHub projects. These metadata contain information about the use and characteristics of the GitHub ecosystem.
- Presentation of the current characteristics of GitHub as a social coding platform.
- Selection of criteria to mine GitHub repositories relevant to software quality and metrics.

In chapter IV:

- Compilation, presentation and analysis of the state of the art in software quality and software metrics.

In chapter V:

- Making publicly available an exploitable database containing software metrics and diverse quality indicators of 3,074 public open-source repositories in GitHub.
- Presentation and analysis of the process of extracting massive-scale software metrics features and quality targets from GitHub, exploitable for machine learning tools.
- Analysis of the general patterns of coding in Java projects in GitHub, particularly with regards to their software metrics values. Quality assessment of code in GitHub Java projects.
- Use of machine learning to predict the quality of GitHub repositories based on their software metrics.
- Suggestion of improvements in the selection of machine learning features and quality targets in GitHub.

VII. References

[1] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[2] F. Brito e Abreu and W. Melo, “Evaluating the impact of object-oriented design on software quality,” in *Proceedings of the 3rd International Software Metrics Symposium.* IEEE, 1996. p. 90-99

[3] T. J. McCabe, “A Complexity Measure,” *IEEE Trans. Software Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.

[4] M. H. Halstead, *Elements of software science.* Elsevier Science Inc., New York, 1977.

[5] K. El-Emam, “Object-Oriented Metrics: A Review of Theory and Practice,” in *Advances in Software Engineering*, 2002, pp. 23–50.

[6] N. E. Fenton and M. Neil, “Software metrics: successes, failures and new directions,” *J. Syst. Softw.*, vol. 47, no. 2–3, pp. 149–157, 1999.

[7] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, “Identifying thresholds for object-oriented software metrics,” *J. Syst. Softw.*, vol. 85, no. 2, pp. 244–257, 2012.

[8] *GitHub Octoverse 2017*. [Online]. Available: https://octoverse.github.com/. [Accessed: 27-Jul-2018].

[9] T. Menzies, R. Krishna, and D. Pryor, “The promise repository of empirical software engineering data,” *The Promise Repository of Empirical Software Engineering Data*. North Carolina State University, Department of Computer Science, 2016.

[10] C. Catal and B. Diri, “A systematic review of software fault prediction studies,” *Expert Syst. Appl.*, vol. 36, no. 4, pp. 7346–7354, 2009.

[11] V. Cosentino, J. L. Canovas Izquierdo, and J. Cabot, “A Systematic Mapping Study of Software Development With GitHub,” *IEEE Access*, vol. 5, pp. 7173–7192, 2017.

[12] G. Gousios, “The GHTorent dataset and tool suite,” in *2013 10th Working Conference on Mining Software Repositories (MSR).* IEEE Press, 2013. p. 233-236.

[13] I. Grigorik, “The github archive.” 2012. [Online]. Available: https://githubarchive.org [Accessed: Jul-2018].

[14] M. Allamanis and C. Sutton, “GitHub Java Corpus,” *GitHub Java Corpus*. [Online]. Available: http://groups.inf.ed.ac.uk/cup/javaGithub/. [Accessed: Jul-2018].

[15] M. Allamanis and C. Sutton, “Mining source code repositories at massive scale using language modeling,” in *2013 10th Working Conference on Mining Software Repositories (MSR).* IEEE Press, 2013. pp. 207-216.

[16] V. Markovtsev and W. Long, “Public git archive,” in *Proceedings of the 15th International Conference on Mining Software Repositories - MSR ’18*, 2018.

[17] P. Gyimesi, G. Gyimesi, Z. Tóth, and R. Ferenc, “Characterization of Source Code Defects by Data Mining Conducted on GitHub,” in *Lecture Notes in Computer Science*, 2015, pp. 47–62.

[18] Z. Tóth, P. Gyimesi, and R. Ferenc, “A Public Bug Database of GitHub Projects and Its Application in Bug Prediction,” in *Lecture Notes in Computer Science*, 2016, pp. 625–638.

[19] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, p. 1, 2005.

[20] S. Kim, T. Zimmermann, K. Pan, and E. Jr. Whitehead, “Automatic Identification of Bug-Introducing Changes,” in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06).* IEEE, 2006. p. 81-90.

[21] C. Williams and J. Spacco, “SZZ revisited,” in *Proceedings of the 2008 workshop on Defects in large software systems - DEFECTS ’08,* ACM, 2008. pp. 32-36.

[22] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “The promises and perils of mining GitHub,” in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014,* ACM, 2014. pp. 92-101.

[23] T. F. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. Le Traon, “Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub,” in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 2013. p. 188-197.
[24] A. Lima, L. Rossi, and M. Musolesi, “Coding Together at Scale: GitHub as a Collaborative Social Network,” in *The International AAAI Conference on Weblogs and Social Media (ICWSM)*.
[25] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang, “Why and how developers fork what from whom in GitHub,” *Empirical Software Engineering*, vol. 22, no. 1, pp. 547–578, 2016.
[26] K. Muthukumaran, A. Choudhary, and N. L. Bhanu Murthy, “Mining GitHub for Novel Change Metrics to Predict Buggy Files in Software Systems,” in *2015 International Conference on Computational Intelligence and Networks.* IEEE, 2015. pp. 15-20.
[27] A. K. Dwivedi, A. Tirkey, R. B. Ray, and S. K. Rath, “Software design pattern recognition using machine learning techniques,” in *2016 IEEE Region 10 Conference (TENCON)*. IEEE, 2016. p. 222-227.
[28] D. Neuendorf and R. Wiener, “JBuilder 8,” *The Journal of Object Technology*, vol. 2, no. 2, p. 205, 2003.
[29] R. Coleman and M. A., “A Study of Scala Repositories on Github,” *International Journal of Advanced Computer Science and Applications*, vol. 5, no. 7, 2014.
[30] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, “Toward Deep Learning Software Repositories,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories.* IEEE Press, 2015. p. 334-345
[31] *Libraries. GitHub Developer Guide*. [Online]. Available: https://developer.github.com/v3/libraries/. [Accessed: Jul-2018].
[32] “About repository languages,” *GitHub Help*. [Online]. Available: https://help.github.com/articles/about-repository-languages/. [Accessed: Jul-2018].
[33] G. Gousios, M.-A. Storey, and A. Bacchelli, “Work practices and challenges in pull-based development,” in *Proceedings of the 38th International Conference on Software Engineering - ICSE ’16.* IEEE, 2016. p. 285-296.
[34] Joint Technical Committee ISO/IEC JTC and Information Technology, *ISO/IEC 9126-1:2001, Software engineering: product quality*. 2001.
[35] J. M. Juran and F. M. Gryna, *Juran’s Quality Control Handbook*. McGraw-Hill Companies, 1988.
[36] E. Arisholm, L. C. Briand, and A. Foyen, “Dynamic coupling measurement for object-oriented software,” *IEEE Trans. Software Eng.*, vol. 30, no. 8, pp. 491–506, 2004.
[37] B. W. Boehm, J. R. Brown, and L. Mlity, “Quantitative evaluation of software quality,” in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 592–605.
[38] B. W. Boehm, “Verifying and Validating Software Requirements and Design Specifications,” *IEEE Softw.*, vol. 1, no. 1, pp. 75–88, 1984.
[39] P. Goodman, *Practical Implementation of Software Metrics*. McGraw-Hill Companies, 1993.
[40] R. Subramanyam and M. S. Krishnan, “Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects,” *IEEE Trans. Software Eng.*, vol. 29, no. 4, pp. 297–310, 2003.
[41] K. El Emam, W. Melo, and J. C. Machado, “The prediction of faulty classes using object-oriented design metrics,” *J. Syst. Softw.*, vol. 56, no. 1, pp. 63–75, 2001.
[42] F. B. e Abreu, F. B. e. Abreu, and R. Carapuça, “Candidate metrics for object-oriented software within a taxonomy framework,” *J. Syst. Softw.*, vol. 26, no. 1, pp. 87–96, 1994.
[43] K. El Emam and N. F. Schneidewind, “Methodology for Validating Software Product Metrics,” in *Encyclopedia of Software Engineering*, 2002.
[44] M. Hitz and B. Montazeri, “Measuring product attributes of object-oriented systems,” in *Lecture Notes in Computer Science*, 1995, pp. 124–136.
[45] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Proceedings of the 13th international conference on Software engineering - ICSE ’08.* ACM, 2008. p. 181-190.
[46] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR*

2010). IEEE, 2010. p. 31-41.
[47] R. Malhotra, S. Shukla, and G. Sawhney, "Assessment of defect prediction models using machine learning techniques for object-oriented systems," in *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO).* IEEE, 2016. p. 577-583.
[48] A. Terceiro *et al.*, "Analizo: an Extensible Multi-Language Source Code Analysis and Visualization Toolkit," in *1st Brazilian Conference on Software*.
[49] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 897–910, 2005.
[50] K. N and N. Kayarvizhy, "Systematic Review of Object Oriented Metric Tools," *Int. J. Comput. Appl. Technol.*, vol. 135, no. 2, pp. 8–13, 2016.
[51] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08.* ACM, 2008. p. 131-142.
[52] "Analizo documentation," *Analizo documentation*. [Online]. Available: http://www.analizo.org/documentation.html. [Accessed: Jul-2018].
[53] T. M. Khoshgoftaar and J. C. Munson, "The lines of code metric as a predictor of program faults: a critical analysis," in *Proceedings., Fourteenth Annual International Computer Software and Applications Conference.* IEEE, 1990. p. 408-413
[54] "SLOCCount," *SLOCCount*. [Online]. Available: https://www.dwheeler.com/sloccount/. [Accessed: Jul-2018].
[55] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Manage. Sci.*, vol. 52, no. 7, pp. 1015–1030, 2006.
[56] P. R. M. Meirelles, "Monitoramento de métricas de código-fonte em projetos de software livre," Universidade de São Paulo, 2013.
[57] L. H. Rosenberg and L. E. Hyatt, "Software quality metrics for object-oriented environments," *Crosstalk journal*, vol. 10, no. 4, pp. 1–6, 1997.
[58] J. Al-Ja'afer and K. Sabri, "Metrics for Object Oriented Design (MOOD) to assess Java programs," presented at the International Arab Conference on Information Technology.
[59] Sahraoui, Sahraoui, Godin, and Miceli, "Can metrics help to bridge the gap between the improvement of OO design quality and its automation?," in *Proceedings International Conference on Software Maintenance ICSM-94.* IEEE, 2000. pp. 154.
[60] A. Terceiro, L. R. Rios, and C. Chavez, "An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects," in *2010 Brazilian Symposium on Software Engineering.* IEEE, 2010. p. 21-29.
[61] "Generalized linear models," *Scikit-learn 0.19.2 documentation*. [Online]. Available: http://scikit-learn.org/stable/modules/linear_model.html#lasso. [Accessed: Jul-2018].
[62] "Model Evaluation: quantifying the quality of predictions," *Scikit-learn documentation 0.19.2*. [Online]. Available: http://scikit-learn.org/stable/modules/model_evaluation.html#r2-score-the-coefficient-of-determination. [Accessed: Jul-2018].
[63] "Technical Debt - SonarQube-5.2." [Online]. Available: https://docs.sonarqube.org/display/SONARQUBE52/Technical+Debt. [Accessed: Jul-2018].

VIII. Appendix

A. Paper submitted to MaLTesQue 2018

The following paper was submitted to the Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTesQuE 2018) in March 2018 in Campobasso, Italy.

Knowledge Acquisition from GitHub for Software Quality Assessment

Abstract— **Software measurement intends to provide a reliable and repeatable method to assess software quality. However, defining meaningful thresholds for quality metrics values has been a continuing challenge for researchers. Machine learning can be exploited to improve our understanding of software metrics and their relationship with software quality. This requires the analysis of large amounts of software-related data. Fortunately, online code hosting platforms such as GitHub make such large amounts of raw data (both source code and metadata) publicly available. This paper investigates the process of building a relevant dataset that (i) extracts data from Java software projects hosted on GitHub, (ii) uses software metrics as its feature extraction process and (iii) is exploitable by standard machine learning techniques. Criteria are defined to select relevant repositories from GitHub and challenges related to data acquisition are investigated. This paper exposes findings regarding the characteristics of object-oriented projects hosted on GitHub and provides an overview of the collected software metrics. Most repositories in this platform are small, not very cohesive and have low complexity and coupling. The results show that GitHub is a useful tool to produce a dataset on which to conduct further studies.**

Keywords— **Machine learning, Software Measurement, Software Quality, GitHub, Software Repositories Mining**

I. Introduction

Software quality assessment is a crucial part of software development. Besides software inspection, software measurement is one of the main methods used to assess the quality of software. Multiple software metrics have been designed to establish meaningful relationships between measurable properties of software artefacts (e.g., lines of code, cyclomatic complexity, etc.) and high-level software quality characteristics (e.g., robustness, evolutivity, maintainability, etc.) [1][2][3]. Unfortunately, these relationships have yet to be accurately characterised (i.e. with suitable thresholds to interpret quality based on measurement values) and several limitations to metrics have been pointed out [4][5].

Machine learning provides new opportunities to dive into software artefacts to better analyse and understand them. While machine learning is often used as an improved form of software inspection applied directly to the source code (e.g., for defect prediction), many standard machine learning techniques require a feature extraction phase prior to their application. This feature extraction process is a challenge in itself, taking a complex entity and singling out relevant properties on which algorithms can be applied in order to detect relevant patterns that characterize the entity. In the case of software quality, software measurement provides a promising candidate to conduct a relevant feature extraction. Indeed, software metrics are known to be related to high-level quality characteristics [5] [6][7] while the actual quantification of this relationship has never been fully accomplished. In a classical empirical setting, it would require a huge effort to do it manually and it would be difficult to analyse how different metrics interact with each other. Applying machine learning techniques to extracted software measurement values should therefore yield meaningful results regarding software quality.

Both software measurement and machine learning techniques require a large amount of data in order to extract relevant patterns from the studied domain. In order to study software quality, the availability of numerous source codes is key to provide significant results. In recent years, GitHub has become the most widespread collective code hosting platform and therefore provides a very rich source of data providing not only the source code of a project but also interesting additional data such as reported issues, liveliness of the project, etc.

The goal of our research is therefore to go beyond just using classical metrics, i.e. (i) combine metrics and better understand them and (ii) create new metrics that overtake classical metrics. This paper constitutes the first milestone of this ongoing research. It investigates the process of building a relevant dataset that (i) extracts data from relevant Java software projects hosted on GitHub, (ii) uses software metrics as its feature extraction process and (iii) is exploitable by standard machine learning techniques.

II. Related works

GitHub has been increasingly used in research. Cosentino et al. carry out a thorough and complete compilation of the methods and findings of 80 relevant works around GitHub [8]. Kalliamvakou et al. [9] analyse the characteristics of the use of GitHub and underline some of the perils and challenges that mining this platform can entail for researchers.

Issue tracking is one of the main tools provided by GitHub to allow its community to interact with regards to code. This tool enables users to point out and fix bugs, propose new features, etc. Bissyande et al. [10] analyse a hundred thousand GitHub projects to characterise the use of issue trackers.

Machine learning tools have already been used to treat data and code gathered from GitHub. Muthukumaran et al. [11] extract a series of change metrics from 5 different versions of Eclipse JDT project in order to predict the number of bugs on the project. In [12], Allamanis and Sutton gather 14,807 open source Java projects (the “GitHub Java corpus”), used to train a n-gram language to predict Java tokens and analyse its code. In

a similar fashion, after tokenizing 16,221 GitHub Java projects, White et al. [13] train a deep learning model for uses such as code suggestion.

In [14], Dwivedi et al. extract object oriented metrics as feature vectors fed to machine learning algorithms (layer recurrent neural network and decision tree) in order to identify software design patterns. Malhotra et al. [15] use the Xerces dataset to compare the performance of 17 different machine learning algorithms in predicting defect prone classes based on the object oriented metrics of these classes.

Table 1 shows how machine learning is used in the above works, in particular whether each project uses GitHub as a primary data source, analyses multiple projects and extracts software metrics to feed machine learning algorithms. As we can see, several research efforts exploit GitHub and/or machine learning and/or software metric to better understand software quality. However, none of them use the three in conjunction as this research proposes.

III. Data and code collection in GitHub

In this section, we present the tools and criteria used in this paper to retrieve relevant repositories in GitHub.

A. Tools for Collection in GitHub

The present study retrieves code from GitHub repositories in order to extract the appropriate object oriented metrics, along with metadata that could indicate the quality of the code (e.g. number of issues, errors, pull requests, number of forks, etc.). Before doing so, we analyse the characteristics of repositories hosted in GitHub, in order to define the selection criteria: what are the characteristics of the repositories deemed relevant in this research. Filtering repositories is a necessary step in order to limit the enormous number of repositories in GitHub and select those which are more relevant for this study.

There exist various options to retrieve large datasets from GitHub. The first and most obvious tool is the GitHub REST API. This API sets limitations to the number of requests that a single user can make. An authenticated user can make up to 5000 requests per hour, whereas a non authenticated one has a limit of 60 requests per hour. Libraries in different languages exist, providing a layer of abstraction to the GitHub REST API, facilitating the interaction with the platform. This study has used the Python library PyGithub, because of its simplicity and because Python has a strong machine learning ecosystem.

TABLE I. Related Papers Scope

ref.	uses GitHub	analyses several projects	extracts software metrics
[11]	Yes	No (only Eclipse JDT)	No (extracts change metrics)
[12]	Yes	Yes (14,807 projects)	No (treats code directly)
[13]	Yes	Yes (16,221 projects)	No (treats code directly)
[14]	Unclear	No (only JHotDraw)	Yes
[15]	No (Xerces dataset)	Yes	Yes

GHTorrent is a powerful tool for the GitHub data miner [16]. It is a queriable mirror database of a significant part of GitHub data, stored both in a SQL database and a set of MongoDB files. The dumps with the data can be downloaded and queried offline, or directly queried online. It is a useful snapshot on which to conduct research in a controlled and replicable manner. GHTorrent holds information about GitHub users, repositories, commits, etc., but does not store the files and code of the repositories. It is therefore insufficient in our objective to extract software metrics. In a similar manner, the GitHub archive [17] provides a dataset of events in the GitHub event stream as from 2011 [9].

It is also worth mentioning the GitHub Java Corpus [18] gathered for the work conducted in [12]. It is a snapshot of all the open-source Java projects with at least one fork that were publicly available in October 2012. This amounts to a total of 14,765 projects, summing up to approximately 352 million lines of code. Using only repositories having at least one fork aims to filter repositories with higher quality. In this study, we have used GitHub REST API directly and PyGithub, in order to retrieve repositories and their metadata. GHTorrent has been used to preselect repositories relevant for this study.

B. A Characterisation of GitHub Repositories

In order to build the dataset, criteria to decide which repositories are included must be defined. The aim of the present study is to treat a set of object oriented projects, randomly chosen from a subset of projects that are meaningful for extracting information about software quality. For that, characteristics of repositories on GitHub are analysed.

In order to select random repositories, one can request the GitHub REST API resource "`/repositories`" with its parameter "`since`" in order to retrieve random repositories. In our case, we have mined public projects with an id lower than 102775889, ranging from the start of GitHub until 07/09/2017. The current research has fetched a total of 71,942 repositories.

One surprising result is that 23.3% of the repositories are not software development, as shown in Fig. 1. They do not have a main language. Among these cases, we find empty repositories, as well as non-software files (repositories making use of GitHub as a file storage service).

Among the projects with a main programming language, the most widespread is Javascript, followed by Java, Python, HTML and Ruby. Being interested in the analysis of object oriented languages, Java seems the appropriate choice: it is the main language in 11.38% of all GitHub repositories (see Fig. 2

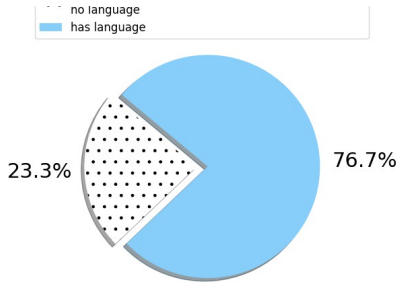

Fig. 1. Repositories with and without a main language.

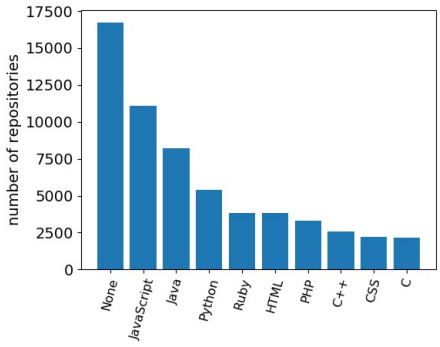

Fig. 2. Number of repositories by main language.

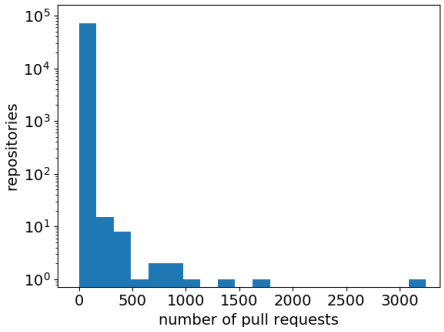

Fig. 3. Number of repositories by number of pull requests.

for all languages) and it is widely used in industry. Being a social coding platform, interaction between developers in GitHub is a valuable source of information to assess the quality of code. However, data reveals that 72% of all repositories are personal [9] (i.e. have only one committer) and/or do not use the mechanisms foreseen for this interaction, such as pull requests and issue tracking. For this paper, personal projects are less interesting, since the lack of interaction between developers makes it difficult to assess the quality of the code.

GitHub allows to create copies (called *forks*) of a repository (*base repository*) to which one can make modifications, such as bug fixes, that will later be proposed to the owner of the project (*pull request*). In turn, forks can also have their own forked copies. Of all repositories analysed, 43.7% are forks. Base repositories have more forks, issues and pull requests than forks. The reason for this seems to be that the developer community tends to interact more with the main, original repository, than with a fork (many of which are used only to add a specific feature or correct a bug of the base repository).

Although more than half of the repositories are base repositories, only 5.6% of them have pull requests, as shown in Fig. 3 (which, like other figures in this paper, uses logarithmic scale). The reasons are that few projects concentrate most pull requests [9], that many forks are used as copies not intended to be pulled to the base repository and that a lot of projects are experimental, empty or not related to software development.

Issue trackers are also scarcely used in GitHub (Fig. 4). In 2013, about 30% of the projects had issues, whereas 3% disabled issue tracking and a whole 66% did not disable them but still did not use them [10]. This situation has changed remarkably in 2017: 43.6% of repositories disable issue

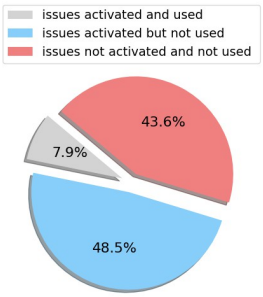

Fig. 4. Use of issue trackers in GitHub.

trackers, 48.5% do not disable them but don't use them and only 7.9% actually use issues. The decline of issue tracking can be explained by the increasing number of personal projects in GitHub related to the popularity of the platform. It can also be partially the result of the fact that [10] excludes, unlike this study, projects with less than 1000 lines of code.

C. *Criteria for Repositories Selection*

Based on section III.B, we have selected three criteria to filter the repositories treated in this paper:

- **The repository has Java as its main language.** The study focuses mainly on object oriented metrics, so choosing an object oriented language is a coherent choice. This criteria excludes all non-software repositories. Furthermore, Java is one of the most widespread languages in GitHub and one that code metric tools can often analyse.
- **Only not-fork repositories are selected**. Analysing the same code several times (original repositories and forks) can introduce a bias for the most popular projects in GitHub. Also, base repositories are more prone to interaction among different users than forks, in terms of issues, pull requests, comments, etc. This social interaction gives a better insight into the quality of the code, via issues related to bugs, fixes, pull requests, etc.
- **Only repositories using issues are selected**. Analysing issues can be a fruitful manner to measure code quality. Furthermore, repositories having used the issue tracker system are more likely to be collective, collaborative repositories, instead of personal projects, making it easier to quantify code quality.

Among the 71,942 repositories whose metadata were analysed, only 498 fulfilled the three criteria (0.69%).

D. *Mining Repositories*

Once settled the criteria to select the relevant projects for this research, the repositories are to be retrieved, i.e. their code and relevant metadata (language, number and type of issues, pull requests, whether it is a forked repository, etc.).

Retrieving the code of a repository and its relevant metadata requires around 10 or 20 calls to the GitHub API (depending on factors such as the number of pages of issues). Since the hourly limitation is of 5000 calls, approximately 300 repositories per hour can be retrieved. Given that only a 0.69%

of the repositories fulfil the three criteria described earlier, only the data for two repositories can be expected per hour. Therefore, this approach, adequate to gather metadata for section III.B, is less adapted for finding repositories fulfilling the criteria set in III.C. A more efficient approach is to *preselect* in GitHub those repositories that fulfil the selection criteria. It is then possible to retrieve the current code and metadata of these repositories.

GHTorrent database is a very big database. Already in January 2015, the MongoDB version stored 4TB of JSON data, while the SQL version of the data had more than 1.5 billion rows. The latest SQL versions, once decompressed, take around 270 GB in CSV files. We preferred to use a slightly older version of the database, from January 2016, which took around half of the size, making it easier to handle.

It is not necessary to load and treat all the tables in the GHTorrent database. For the scope of this work, only a few tables (projects, issues, issue_labels, project_languages) were used, since they contain the information required to select repositories according to section III.B.

The mirror offered a total of 25,364,494 repositories. Among them, 2,243,734 repositories had Java as their main language. Around half of them were not forked (1.109.893). In total, 122,074 projects fitted the three conditions in III.C.

IV. Object oriented metrics evaluation

In this section we discuss the tools and process to extract software metrics from GitHub and present relevant results.

A. Tools for Metrics Evaluation

Different tools for metric evaluation exist. Kayarvizhy reviews and compares a series of 10 object oriented metric tools [19]. Provided with the same code, different metric tools offer different results due to implementation specificities [20]. In this study, we selected the appropriate software measurement tool based on its characteristics.

First of all, this paper chooses to analyse Java code. Therefore, some tools which do not support Java analysis are discarded, such as SD Metrics or QMOOD++ [19].

Also, given the number of projects to be analysed, it is not feasible to have to manually compile each repository. Furthermore, many of these projects must be configured before compilation or contain errors which hamper compilation. A tool that can extract data from source non-compiled Java files (".java", not ".class") is needed. This discourages the use of tools such as ckjm, Jdepend [19], Chidamber and Kemerer Java Metrics or Dependency Finder [20].

It is also impracticable to extract the metrics of each repository, one at a time. Instead, given the number of projects, we expect the tool to be able to process repositories in batch. Tools such as the Eclipse IDE plugins (Eclipse Metrics Plugin) are therefore not the best option, since they require importing each project individually into the specific IDE.

An open source solution is preferred to a commercial one. It allows to clearly understand how metrics are measured and facilitates that other researchers share the same tool. Replicability is also better guaranteed when the researcher knows what happens *under the hood*. This criteria suggests to push aside tools such as RSM, Jhawk, JMT [19] or Understand for Java. Some solutions used in the above mentioned papers seem not to be active any longer, such as Analyst4j or JMT.

We concluded that Analizo is a tool that fits the conditions of our study. It is an open-source, non commercial tool, created within the scientific community [21]. It extracts a wide range of object oriented OO metrics for Java projects and files, which do not need to be compiled or even compilable. Furthermore, it allows to treat repositories in batch, and not one by one.

B. Mining and Metrics Extraction Process

The process conducted in this study to download the repositories, extract the OO metrics and repositories metadata is the following. First, GHTorrent was queried offline to obtain the set of repositories fulfilling the selection criteria mentioned in section III.C. Once these repositories were identified, a batch script would download the sources of the repository, decompress them, and extract the OO metrics with Analizo. This same script would retrieve the relevant metadata: date and time of extraction, number of issues, number of issues with labels or comments related to bugs and errors, etc.

The metadata could be obtained by querying GHTorrent instead of retrieving it directly from GitHub. However, the metadata would refer to the state of the repository at the moment in which GHTorrent data was extracted, previous to the download and extraction of the repository. Therefore, it is not advisable to query GHTorrent for metadata, but rather to retrieve it at the same time of downloading the repository and extracting its metrics.

V. Metrics Results

For the scope of this paper, we treated a total of 4,449 repositories. Among them, 726 no longer existed. Analizo was unable to extract the metrics of another 644 repositories. 5 repositories were too big (more than 1Gb of source code). For the remaining 3,074 repositories, all the metrics that Analizo is capable of extracting (described in [21]) are included in the dataset. In this section we focus on some of these classical software metrics that allow to characterise the dataset.

Analysing the metrics, it is clear that there is a tendency of most repositories to be rather small. It is indeed very rare to find repositories with a size bigger than 200MB, as shown in Fig. 5. A big number of repositories have few classes (Fig. 6).

The complexity and cohesion of the code can be measured in different manners, all of them relevant to measure the quality of the code. High cohesion of classes manifest high quality, by which each class has one, and only one, purpose.

One way to measure cohesion is the lack of cohesion in methods (LCOM) metric, proposed by Chidamber and Kemerer [1] and later improved by Hitz et al. (LCOM4) [22]. It quantifies the number of groups of related methods and fields that exist within a class. The desirable value for LCOM4

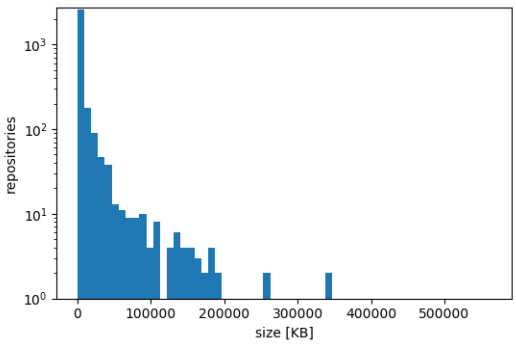

Fig. 5. Number of repositories by size.

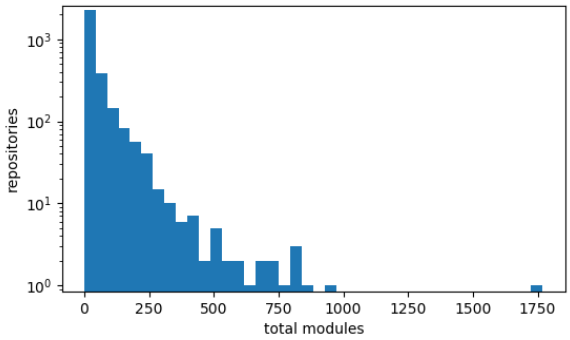

Fig. 6. Number of repositories by number of modules.

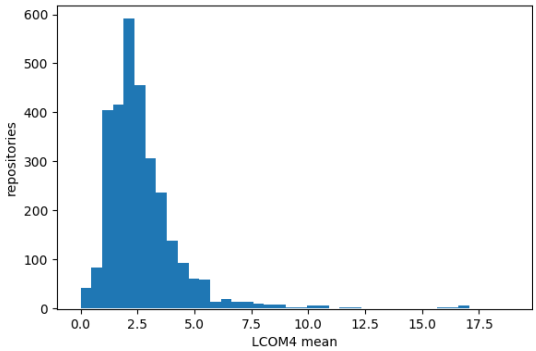

Fig. 7. Number of repositories by LCOM4 mean.

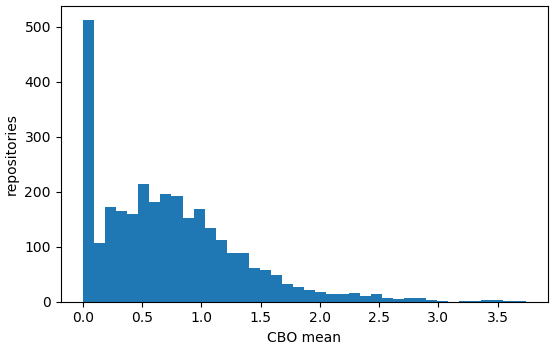

Fig. 8. Number of repositories by CBO mean.

is 1, meaning a highly cohesive class, whereas bigger values reflect classes that should be split.

Fig. 7 shows the number of repositories in GitHub, depending on the mean value of LCOM4 of its classes. It reveals that most repositories have non-cohesive classes, with most repositories having a mean LCOM4 value of around 2.5 (greater than the ideal value: 1.0).

Another element to measure dependencies is how coupled classes are to each other. This can be measured using the Coupling Factor (CF) proposed by Abreu et al. [2] or the Coupling Between Objects (CBO) put forward by Chidamber and Kemerer [1]. Sharma et al. [23] conclude it is preferable to use CBO instead of CF. High CBO is undesirable, in particular when it exceeds a threshold value of 14, according to [24].

The GitHub repositories analysed, as Fig. 8 shows, present two interesting results. Firstly, many repositories have no coupling at all (their CBO is 0). This is related to Fig. 6 too: if repositories contain few classes, it is more likely that each class depends on few others. Secondly, Fig. 8 shows that the mean CBO in repositories is evenly distributed between values 0 and 2. Their classes have a low coupling to each other.

Terceiro et al. propose in [25] a measure for structural complexity, measured as the product of LCOM4 and CBO. Fig. 9 shows that most of the GitHub repositories analysed have low structural complexity.

It is also interesting to compare the mean number of methods per class among the different repositories in GitHub. Most projects have less than 10 methods per class (Fig. 9), reaching its peak in 5 methods per class.

One of the object oriented languages particularities is the possibility to inherit methods and/or variables from parent classes. While method inheritance promotes reuse of code, it also increases complexity and can make it more difficult to understand the behaviour of a specific class. It is therefore recommended not to have a too big depth of inheritance tree (DIT) [1] metric value. Fig. 10 shows that most repositories do not overuse inheritance, with mean DIT values mostly ranging from 0 to 2. This values refer to the mean of the whole project, some classes within them can of course exceed these values.

In summary, the selected GitHub repositories have a tendency to be small, lack cohesion, not be very coupled, not overuse inheritance and, in general, not be too complex. However, the dataset contains rich and varied values (see figures), making it suitable for analysing its impact on quality assessment indicators (such as the number of bugs, issues or other metadata extracted from GitHub).

VI. CONCLUSIONS AND FUTURE WORK

This paper is the first milestone of a wider research aimed at better understanding and improving software quality measurement by using machine learning algorithms on a large amount of data extracted from GitHub. This paper analyses the process and tools used to extract a relevant dataset of software metrics that can be treated by machine learning techniques. It also proposes criteria to select relevant repositories in GitHub in order to analyse software metrics.

We also presented a characterisation of Java repositories on GitHub and showed some of their *classical* software metrics. Results show that most repositories in this platform are small, personal, rarely use issue tracking. The software metrics

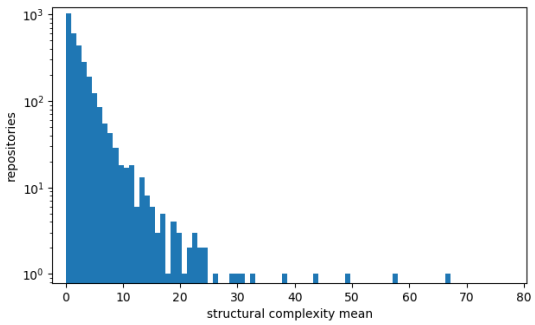

Fig. 9. Number of repositories by structural complexity mean.

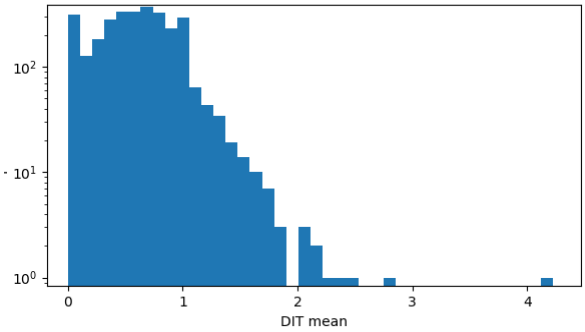

Fig. 10. Number of repositories by Depth of Inheritance Tree.

extracted from the dataset revealed that most code in GitHub lacks cohesion, is not very complex and does not overuse inheritance. It also shows that most classes have low coupling. The variety of values still make the dataset suitable for further analysis of its effects on code quality indicators retrieved from GitHub.

Future work includes analysing correlation between metrics and GitHub metadata, as well as trying to use machine learning tools to identify relevant metrics combinations (multivariate approach). Before that, more specifically, we will try to extract software metrics from more GitHub repositories and understand the reasons why some repositories could not be correctly processed by Analizo in order to finalise (and then make publicly available) a dataset as exhaustive as possible.

REFERENCES

[1] S.R. Chidamber and C.F. Kemerer. "A metrics suite for object oriented design," IEEE Transactions on software engineering 20.6, 1994, pp 476-493.

[2] Abreu,, and Rogério Carapuça. "Object-oriented software engineering: Measuring and controlling the development process," Proceedings of the 4th international conference on software quality, Vol. 186, 1994, pp 1-8.

[3] F.B. Abreu. "Using OCL to formalize object oriented metrics definitions," INESC, Software Engineering Group, Technical Report ES007/2001, 2001.

[4] N.E. Fenton and M. Neil. "Software metrics: successes, failures and new directions," Journal of Systems and Software 47.2, 1999, pp 149-157.

[5] K. El-Emam. "Object-oriented metrics: A review of theory and practice," Advances in software engineering, Springer New York, 2002, pp 23-50.

[6] R. Subramanyam and M.S. Krishnan. "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," IEEE Transactions on software engineering 29.4, 2003, pp 297-310.

[7] K. El Emam, W. Melo and J.C. Machado. "The prediction of faulty classes using object-oriented design metrics," Journal of Systems and Software 56.1, 2001, pp 63-75.

[8] V. Cosentino, J.L. Cánovas Izquierdo and J. Cabot. "A Systematic Mapping Study of Software Development With GitHub," IEEE Access 5, 2017, pp 7173-7192.

[9] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D.M. Germanand D. Damian. "The promises and perils of mining GitHub," Proceedings of the 11th working conference on mining software repositories, ACM, 2014, pp. 92-101.

[10] T.F. Bissyandé, D. Lo, L. Jiang, L. Réveillere, J. Klein and Y. Le Traon. "Got issues? who cares about it? a large scale investigation of issue trackers from GitHub," Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on IEEE, 2013, pp. 188-197.

[11] K. Muthukumaran, A. Choudhary, and N.B. Murthy. "Mining GitHub for novel change metrics to predict buggy files in software systems," Computational Intelligence and Networks (CINE), 2015 International Conference on, IEEE, 2015, pp. 15-20.

[12] M. Allamanis and C. Sutton. "Mining source code repositories at massive scale using language modeling," Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, 2013, pp. 207-216.

[13] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. "Toward deep learning software repositories," Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, 2015, pp. 334-345.

[14] A. Dwivedi, A. Tirkey, R. Ray and S. Rath. "Software design pattern recognition using machine learning techniques," 10.1109/TENCON.2016.7847994, 2016, pp. 222-227.

[15] R. Malhotra, S. Shukla and G. Sawhney. "Assessment of defect prediction models using machine learning techniques for object-oriented systems," Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), 5th International Conference on. IEEE, 2016, pp. 577-583.

[16] G. Gousios. "The GHTorent dataset and tool suite," Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, 2013, pp. 233-236.

[17] I. Grigorik. The GitHub archive [Online, accessed Dec. 2017]. Mar. 2012. Available: https://www.githubarchive.org

[18] M. Allamanis and C. Sutton. GitHub Java Corpus [Online, accessed Dec. 2017], 2012 [dataset]. University of Edinburgh: School of Informatics. Available: http://dx.doi.org/10.7488/ds/1690.

[19] N. Kayarvizhy. "Systematic Review of Object Oriented Metric Tools," Bakar 135.2, 2016, pp 8-13.

[20] R. Lincke, J. Lundberg and W. Löwe. "Comparing software metrics tools," Proceedings of the 2008 international symposium on Software testing and analysis, ACM, 2008, pp. 131-142.

[21] A. Terceiro, J. Costa et al. "Analizo: an extensible multi-language source code analysis and visualization toolkit," Brazilian conference on software: theory and practice (Tools Session), 2010.

[22] M. Hitz and M. Behzad. "Measuring coupling and cohesion in object-oriented systems," 1995, pp 25-27.

[23] A.K. Sharma, A. Kalia and H. Singh. "Metrics identification for measuring object oriented software quality," International journal of soft computing and engineering 2.5, 2012, pp 255-258.

[24] H. Sahraoui, R. Godin and T. Miceli. "Can metrics help bridging the gap between the improvement of OO design quality and its automation," Proceedings of the International Conference on Software Maintenance, ICSM, 2000.

[25] A. Terceiro, L.R. Rios and C. Chavez. "An empirical study on the structural complexity introduced by core and peripheral developers in free software projects," Software Engineering (SBES), 2010 Brazilian Symposium on, IEEE, 2010, pp. 21-29.