Measuring the Reusability of Software Components using Static Analysis Metrics and Reuse Rate Information

Michail D. Papamichail, Themistoklis Diamantopoulos, Andreas L. Symeonidis

Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
Thessaloniki, Greece
mpapamic@issel.ee.auth.gr, thdiaman@issel.ee.auth.gr, asymeon@eng.auth.gr

Abstract

Nowadays, the continuously evolving open-source community and the increasing demands of end users are forming a new software development paradigm; developers rely more on reusing components from online sources to minimize the time and cost of software development. An important challenge in this context is to evaluate the degree to which a software component is suitable for reuse, i.e. its reusability. Contemporary approaches assess reusability using static analysis metrics by relying on the help of experts, who usually set metric thresholds or provide ground truth values so that estimation models are built. However, even when expert help is available, it may still be subjective or case-specific. In this work, we refrain from expert-based solutions and employ the actual reuse rate of source code components as ground truth for building a reusability estimation model. We initially build a benchmark dataset, harnessing the power of online repositories to determine the number of reuse occurrences for each component in the dataset. Subsequently, we build a model based on static analysis metrics to assess reusability from five different properties: complexity, cohesion, coupling, inheritance, documentation and size. The evaluation of our methodology indicates that our system can effectively assess reusability as perceived by developers.

Keywords: Developer-perceived reusability, code reuse, static analysis metrics, reusability estimation

1. Introduction

The introduction of the open-source software development initiative has changed the way software is developed and distributed. Several practitioners and developers nowadays develop their software projects as open-source and store them online (e.g. on GitHub[1] or Sourceforge[2] among others). This has led to the introduction of an agile, component-based software engineering paradigm, where developers rely more and more on reusing existing components in their source code, either in the form of libraries or simply by copying and integrating useful source code snippets/components in their own source code [1]. Adopting these reuse-oriented practices has important benefits, as the time and effort (and subsequently the cost) of software development are greatly reduced.

This new way of developing software has also led to the introduction of even more code hosting services (as those outlined above), question-answering communities (e.g. Stack Overflow[3]), specialized tools, such as code search engines (e.g. Searchcode[4]) or even source code recommendation systems [2, 3, 4, 5, 6], all tailored to the needs of finding source code that can be effectively reused. As a result, locating and retrieving software components that satisfy certain functional requirements has become relatively easy. The most important reuse challenge, however, is that of determining which of the retrieved components are *suitable for reuse*, or, in other words, determining the *reusability* of software components.

Assessing the reusability of a source code component before integrating it into one's own source code is crucial, since components of poor quality are usually hard to integrate, hard to maintain and in some cases they may even introduce faults. However, reusability is actually a quality concept, thus measuring it can be a challenging task; after all, software quality is highly context-dependent and may mean different things to different people [7]. According to the ISO/IEC 25010:2011 quality standard [8], software reusability, i.e. the extent to which a software component is reusable, is related to maintainability. From a rather more finegrained perspective, a component can be considered highly reusable as long as it is modular, exhibits loose coupling and high cohesion, and provides information hiding and separation of concerns [9].

Current research efforts focus on assessing reusability [10, 11, 12] (or, in a broader sense, maintainability and quality as a whole [13, 14, 15, 16, 17, 18, 19]) by using static analysis metrics, such as the known CK metrics [20]. Although these efforts can be effective for assessing different quality attributes, they do not actually provide an interpretable analysis to the developer, and thus do not inform him/her about the source code properties that need improvement. Furthermore, most approaches are either based on expert-defined metric thresholds [12, 14, 15] or

[1]`https://github.com/`
[2]`https://sourceforge.net/`
[3]`https://stackoverflow.com/`
[4]`https://searchcode.com/`

require some type of ground truth quality score (again defined by an expert) that can be used to train a model to produce adaptable metric thresholds [16, 17, 18, 19]. Note, however, that expert help may be subjective, case-specific or even unavailable [21, 22].

To mitigate the need for expert help, one may resort to *benchmarking*. Benchmarking involves constructing a dataset of software components that are externally (and objectively) evaluated, and using it to model the behavior of the different metrics, and subsequently determine their optimal values within certain thresholds. The effectiveness of benchmarking approaches depends mainly on the statistical potential of the dataset and the objectiveness of the ground truth quality value (if any is provided). Although there are several interesting approaches in this area [22, 23, 24, 25, 26], several of them actually employ again expert-based values, while others focus only on the descriptive power of the benchmark dataset, thus not taking into account any ground truth value. As a result, their estimation may be easily skewed and their outcome may be arbitrary. We argue that using a ground truth value can be quite beneficial for the modeling, as it accounts for the differences between various components and, if chosen correctly and objectively, provides an effective target for training the model.

In previous work, we have attempted to provide such an effective ground truth quality value, by associating the extent to which a software component is adopted/preferred by developers, i.e. its popularity, with the extent to which it is reusable [27]. We initially employed GitHub stars and forks to build a target quality score at repository level, and then proposed different mechanisms, using heuristics [28, 29] and statistical binning/benchmarking [30], in order to build a target reusability score at component level. Based on the evaluation of those works, one may conclude that this target score can be effectively used to train models for estimating the reusability of source code components. The main point of argument in this line of work lies in the way the ground truth value for each component is computed. The number of forks and the number of stargazers indeed provide some informal proof for the reuse rate of repositories, however they still have their limitations. The fact that they refer at repository level dictates that modeling reusability at component (class or package) level requires some type of averaging, which is prone to introducing errors. Furthermore, one may argue that popularity-based metrics may be easily skewed by personal preference or trend (especially the number of stars), this way skewing also the final reusability score.

In this work, we aspire to extend the aforementioned line of research by adopting the actual *reuse rate* of each software component as a measure of its reusability. To do so, we harness the power of open-source code hosting facilities and code search engines to find out the extent to which individual components are actually reused, and, most importantly, to determine what are the quality characteristics that influence the reuse rate of a software component. The ground truth information in this case comprises the reuse rate of each component, or, as we may name it, the reusability as perceived by developers (developer-perceived reusability). Upon formulating our ground truth, we propose a methodology for decomposing the influence of various static analysis metrics on the reusability degree of software components. We compute a large set of metrics both at class and package level and model their behavior to translate their values into a reusability degree. As a result, the outcome of our analysis is not only an aggregated reusability score, but also a comprehensive analysis on metrics belonging to six axes for the source code of each component (class or package), including scores on its complexity, cohesion, coupling, size, degree of inheritance, and quality of documentation.

The rest of this paper is organized as follows. Section 2 reviews current approaches on quality and reusability estimation. Section 3 describes our benchmark dataset that comprises values of static analysis metrics and presents our reusability scoring scheme, which is based on the reuse rate of source code components. The construction of our models for reusability assessment is presented in Section 4, while Section 5 presents the evaluation of our methodology on different projects. Furthermore, Section 6 presents how our reusability evaluation methodology can be exploited by the community, while Section 7 discusses potential threats to validity. Finally, Section 8 concludes this paper and provides useful insight for further research.

2. Related Work

The widely recognized potential of software reuse as a way to reduce development cost and effort [10, 31] has drawn the attention of researchers for years. The concept of reuse dates back to 1968, when Douglas McIlroy proposed the massive production of software using reusable components [32]. Today, the rise of the open-source development initiative and the introduction of online source code repositories have provided new opportunities for reuse; hence, the challenge now lies not only in finding functionally adequate components, but also in ensuring that these components are suitable for reuse. According to the ISO/IEC 25010:2011 quality standard [8], software reusability is a quality characteristic related to maintainability and refers to the degree to which a software artefact can be used in more than one system, or for building other assets. As a result, several methodologies have been proposed to assess the reusability of source code components using static analysis metrics [10, 11, 33], and practically define *reusability metrics* using known quality attributes [34, 35].

Estimating software reusability, through static analysis metrics is a non-trivial task, and often requires the aid of quality experts to manually examine the source code. Obviously, the manual examination of source code can be very tedious, or even impossible for large and complex software projects and/or projects that change on a regular basis. This is the case for the vast majority of today's projects, given the constantly increasing demands both in terms of functional and/or non-functional requirements.

Consequently, given the amount of effort required for the manual examination of source code, a common practice involves creating a benchmarking code repository using representative software projects [22, 23, 24, 25, 26, 36] and then applying machine learning techniques to derive thresholds and define the acceptable metric intervals [37, 38, 39]. The main weakness of these approaches is that they do not employ some type of ground truth value and using only the descriptive power of the benchmark dataset may easily lead to skewed estimations.

Other research efforts attempt to derive reusability by setting predefined thresholds for quality metrics [12, 14, 15]. For instance, Diamantopoulos et al. [12] proposed a metrics-based reusability scorer that calculates the degree of reusability of software components based on the values of eight static analysis metrics. The assessment depends on whether the values of the metrics for each component exceed certain thresholds, as defined in current literature. Any metric that lies outside the predefined threshold negatively affects the reusability score. In addition, the calculation of the proposed reusability score assumes that each one of the eight static analysis metrics involved in the calculation equally contributes in the reusability degree of software, which is not the case especially for metrics that quantify different source code characteristics.

Since using predefined metric thresholds is also typically limited by expert knowledge and may not be applicable on the specifics of different software projects (e.g. the scope of an application), several approaches have been proposed to overcome the necessity of using predefined thresholds [16, 17, 18, 19, 40]. These approaches involve quantifying reusability through reuse-related information such as reuse frequency [40]. Then, machine learning techniques are employed in order to train reusability estimation models using as input the values of static analysis metrics [16, 17, 18, 19]. An interesting note is that these models also incorporate how the values of static analysis metrics influence reusability.

Although the previous approaches can be effective for certain cases, their applicability in real-world scenarios is limited. At first, using predefined metrics thresholds [12, 14, 15] leads to the creation of models unable to incorporate the various different characteristics of software projects. Automated reusability evaluation systems seem to overcome this issue [17, 18, 19], however they are still confined by the ground truth knowledge of quality experts for evaluating the source code and determining whether the degree of reuse is acceptable.

Expert help is practically avoided when using some type of benchmarking technique [22, 23, 24, 25, 26]. Most approaches in this area are also effective under certain preconditions, however they do not generalize beyond the benchmark dataset itself, as they do not employ some type of ground truth for modeling. We argue that the challenge lies in finding such an effective ground truth quality value and employing it to build robust models that will account for the specifics of different components in an objective expert-free manner. As such, we believe that our previous work lies in a promising direction [27, 28, 29], since it highlights a rather effective analogy between the popularity of source code components and their reusability. As already mentioned, however, this analogy also has its limitations; GitHub stars and forks indeed provide an indication of the reusability for components, however they may be easily skewed by personal preference or trend, while they also cannot offer accurate results in class/package level (given that the stars/forks metrics are determined per repository).

In this work, we design a more thorough methodology, which is based not on arbitrary popularity-based metrics, but rather on the actual reuse rate of source code components. We initially illustrate how one can determine the preference/adoption by developers for source code components at different levels (class and package). By employing this measurement as ground truth, we are able to build models capable of estimating the reusability of a component as perceived by software developers in cases where no further reuse data is available. The main idea of our approach lies in the fact that highly reused components are typically expected to exhibit high reusability degree. We investigate this claim and utilize its potential to construct a system that receives as input the values of static analysis metrics for a software component and estimates its reusability. Apart from a final reusability score, our system further provides an analysis on six source code properties, those of complexity, cohesion, coupling, inheritance, documentation and size.

3. Reusability Modeling

In this section, we design a reusability modeling methodology based on the reuse rate of source code components as well as on static analysis metrics.

3.1. Measuring the reuse rate of source code components

As already mentioned, our methodology is based on employing the reuse rate of software components in order to estimate their reusability as perceived by software developers. We define the reuse rate of a software component as the extent to which the component is preferred/selected for reuse by developers[5]; consequently, a straightforward idea for measuring the reuse rate of a component is to find out how many other components issue calls to its API.

In order to conduct such a measurement, we decided to use the capabilities of our own code search engine, named AGORA[6]. AGORA is a powerful source code index, which embeds syntax-aware capabilities and has an open API that allows performing queries and easily integrating it with other tools. The index of AGORA is populated with approximately 3000 of the most popular Java repositories of Github, where popularity is determined by the number of stargazers. Our choice of these repositories is supported by the fact that popular projects typically incorporate examples of properly written source code. Indeed, research has shown that highly rated projects (i.e. with large number of stars/forks) exhibit also high quality [27, 28, 29, 30], have sufficient documentation/readme files [45, 46], while they also involve frequent refactoring cycles and maintenance releases [47]. Based on the above, we argue that the projects of AGORA can be used as a proper pool of projects to define our benchmark dataset.

[5]There can also be other interpretations of the metric, such as techniques for extracting reuse-related information of software components [41, 42, 43, 44]. These techniques, however, do not focus on the challenge of reusability assessment from a metrics-based perspective; thus, they lie outside the scope of this work,

[6]`http://agora.ee.auth.gr/`

The next step involves designing a scoring mechanism that would provide a value for the reuse rate of individual components. Note that our mechanism is mostly language-agnostic (as is our methodology as a whole), yet in the context of this paper we provide our proof-of-concept for components written in the Java language. As a result, we focus on two levels of components: classes and packages. For each component we compute its reuse rate by first extracting its fully qualified name and then issuing it as a query in AGORA. The index allows easily forming such a query, given that the imports of all AGORA projects are analyzed and stored in a separate field.

An example query for a component is shown in Figure 1. The import field of AGORA is analyzed and split into tokens (“org”, “xproject”, “XObject”)[7]; thus the query is suitable for finding all possible declarations of the sample XObject component (e.g. import declarations with wildcards). Upon processing the results to determine if the import declarations actually correspond to reuses of the component at hand, we maintain two statistics: (a) the number of source code files in which the component is being reused, and (b) the number of projects in which the component is being reused.

```
{
   ”query”: {
      ”match”: {
         ”code.imports”: ”org.xproject.XObject”
      }
   }
}
```

Figure 1: AGORA query on import declarations

Although these statistics may seem simplistic, they are actually quite indicative of the reusability degree of source code components. In practice, if a component is reused in many different projects, this means that several different developers deem it as reusable and therefore choose to use it in their own code. Furthermore, as implied by the file-level statistic, components that are reused multiple times within each project (and/or a lot of times in total) are most probably designed exactly for that type of usage (e.g. they may include well-written API methods). As a result, we employ these metrics as ground truth in our analysis, which is presented in the following subsections.

3.2. Dataset

We created a benchmark dataset that contains the 100 most popular projects included in the maven central repository[8]. Given the popularity of maven, we argue that these projects are extensively reused and thus can provide a proper benchmark dataset for our methodology. In order to analyze the source code of these maven libraries, we manually examined whether they are linked to a GitHub repository or not, and kept only the ones that are. Upon removing any duplicates and any projects that are not linked to a GitHub repository, our final dataset contains 65 projects with more than 3 million lines of code. Certain statistics on the dataset can be found at Table 1.

Table 1: Dataset Statistics

Statistics of the constructed dataset	
Number of projects	65
Total Number of Methods	223,490
Total Number of Classes	24,800
Total Number of Packages	2,199
Total lines of code (LOC)	3,062,741

Given the source code of these projects, we have computed a large set of static analysis metrics that are widely used by several research efforts that aspire to assess reusability [48, 49, 19]. The computed metrics refer to six primary source code properties: cohesion, complexity, coupling, documentation, inheritance, and size. We have computed these metrics using the Sourcemeter[9] static analysis tool; a complete reference of the computed metrics can be found at Table 2. A this point, it is worth noting that although our approach employs these respective metrics, additional metrics can be easily incorporated without any major change in the proposed strategy.

Given the level of computation of each metric, the reusability assessment at class level involves metrics that refer to all properties, while in the case of packages, only metrics that refer to documentation and size are taken into account. While one might assume that the reusability degree for a given package could simply occur as an aggregation of the reusability degree of its classes, from a metrics perspective the extent to which a software component is reusable is actually relevant to the level of its granularity. Thus, one can first use our methodology at package level to get an overview, and subsequently at class level to get a more detailed view of the components. As a result, we design a reusability scoring mechanism for evaluating both classes and packages based on their individual characteristics.

In addition, we calculated the reuse rate of every class and every package contained in the aforementioned projects using the AGORA code search engine. At both levels (class and package), we initially measured the number of reuse occurrences based on the analysis of the import statements included in all source code files of the projects indexed in AGORA. Figure 2 presents an example of the reuse rate calculation. As for the reuse of classes, we account only for classes that are explicitly declared in the import statements. In the case of packages, one may notice that the import declaration of a child package is by definition inherited to the parent package (e.g. a declaration “org.example.mypackage.*” includes not only the package “org.example.mypackage” but also the package “org.example”). Therefore, in order to eliminate the impact of this issue to the purity in our dataset, we performed a post-processing step that involves removing all the inherited declarations from the parent packages.

[7]See the link `https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis.html` for more information about the analysis performed by Elasticsearch.

[8]`https://mvnrepository.com/`

[9]`https://www.sourcemeter.com/`

Table 2: Overview of the computed Static Analysis Metrics

Static Analysis Metrics			Computation Levels	
Property	**Name**	**Description**	**Class**	**Package**
Cohesion	LCOM5	Lack of Cohesion in Methods 5	X	—
Complexity	NL	Nesting Level	X	—
	NLE	Nesting Level Else-If	X	—
	WMC	Weighted Methods per Class	X	—
Coupling	CBO	Coupling Between Object classes	X	—
	CBOI	Coupling Between Object classes Inverse	X	—
	NII	Number of Incoming Invocations	X	—
	NOI	Number of Outgoing Invocations	X	—
	RFC	Response set For Class	X	—
Documentation	AD	API Documentation	X	X
	CD	Comment Density	X	X
	TCD	Total Comment Density	X	X
	CLOC	Comment Lines of Code	X	X
	TCLOC	Total Comment Lines of Code	X	X
	DLOC	Documentation Lines of Code	X	—
	PDA	Total Documentation Lines of Code	X	X
	TPDA	Total Public Documented API	—	X
Inheritance	DIT	Depth of Inheritance Tree	X	—
Size	LOC	Lines of Code	X	X
	LLOC	Logical Lines of Code	X	X
	TLLOC	Total Logical Lines of Code	X	X
	TLOC	Total Lines of Code	X	X
	TNA	Total Number of Attributes	X	X
	NG	Number of Getters	X	X
	TNG	Total Number of Getters	X	X
	TNM	Total Number of Methods	X	X
	TNOS	Total Number of Statements	X	X
	TNPM	Total Number of Public Methods	X	X
	NCL	Number of Classes	—	X

```
// classes (NetInfo, Test) and package android.net are reused
// ReuseRate of NetInfo class += 1
// ReuseRate of NTest class += 1
// ReuseRate of android.net package += 2
import android.net.NetInfo;
import android.net.Test;

// package telephony is reused
// ReuseRate of android.telephony package += 1
import android.telephony;

// package android.content is reused
// ReuseRate of android.content package += 1
import android.content.*;
```

Figure 2: Example of Reuse rate calculation at class and package level

Table 3 provides certain statistics regarding the computed values of the reuse rate for both classes and packages. As shown in the first row, approximately 71% of the classes and 54% of the packages appear to have zero reuse rate. The rest of the table rows refer to the classes and the packages with increasing reuse rate (excluding the ones with zero reuse rate). As expected, the larger the reuse rate, the less the number of classes and packages. This decrease originates from classes in the dataset that may hold private components that are not (to be) reused.

Table 3: Reuse Rate Statistics

Reuse Rate Statistics		
Stats	**Class Level**	**Package Level**
ReuseR. $= 0$	17,426 or 71.74%	1,193 or 54.25%
ReuseR. $\in [1, 2)$	2,191 or 31.92%	126 or 12.52%
ReuseR. $\in [2, 4)$	1,781 or 26.02%	178 or 17.69%
ReuseR. $\in [4, 8)$	1,219 or 17.81%	188 or 18.68%
ReuseR. $\in [8, 16)$	683 or 9.98%	142 or 14.15%
ReuseR. $\in [16, 32)$	395 or 5.77%	139 or 13.81%
ReuseR. $\in [32, 64)$	283 or 4.12%	95 or 9.44%
ReuseR. $\in [64, 128)$	169 or 2.34%	67 or 6.65%
ReuseR. $\in [128, 256)$	80 or 1.16%	36 or 3.59%
ReuseR. $\in [256, 512)$	41 or 0.59%	22 or 2.18%
ReuseR. > 512	21 or 0.21%	13 or 1.29%
Total Components	**24.289**	**2.199**

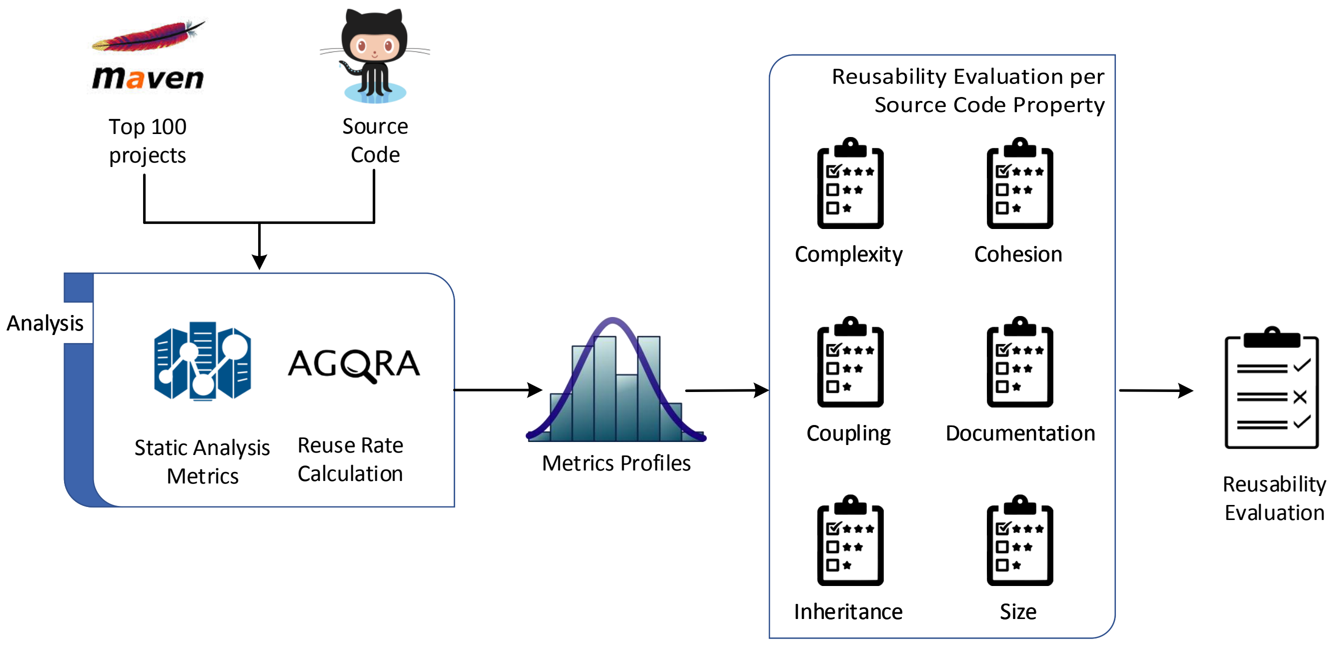
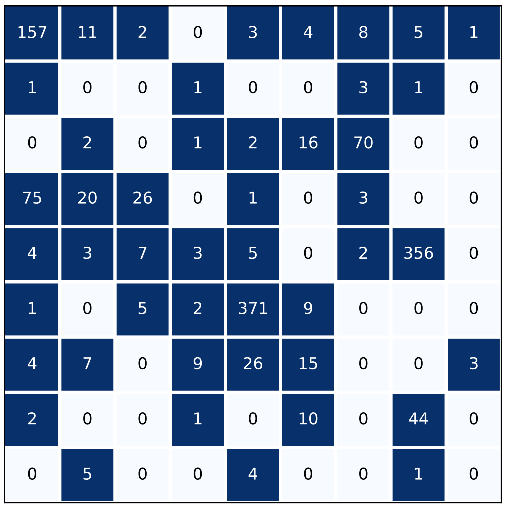


Figure 3: System Overview.

Figure 4 illustrates the reuse rate values regarding all classes included in a single project. Each rectangle refers to a certain class, while the label refers to the absolute value of the reuse rate. As one may clearly note, there are several classes that have not been reused (the ones having zero reuse rate), while there are also multiple others that have a rather low reuse rate (e.g. lower than 10 reuse occasions). The former typically refer to private classes, which we took care to remove from our dataset to avoid skewing the results. In contrast, any other components (with either low or high reuse rates) were maintained as they include useful information about the writing style of the library at hand (e.g. high-reuse classes typically are APIs) and their preference by the developers.



157	11	2	0	3	4	8	5	1
1	0	0	1	0	0	3	1	0
0	2	0	1	2	16	70	0	0
75	20	26	0	1	0	3	0	0
4	3	7	3	5	0	2	356	0
1	0	5	2	371	9	0	0	0
4	7	0	9	26	15	0	0	3
2	0	0	1	0	10	0	44	0
0	5	0	0	4	0	0	1	0

Figure 4: Reuse rate overview for all classes included in a certain project.

4. System Design

This section presents our proposed methodology towards evaluating the reusability degree of software components.

4.1. Methodology Overview

A general overview of our system is shown in Figure 3, while Figure 5 illustrates the steps of the proposed methodology for assessing the reusability degree of software components (classes and packages). Our methodology comprises six distinct steps. At first, we analyze the most popular projects included in the maven registry, in order to compute a series of metrics that quantify several aspects of the source code along with their reuse rate (step a). Subsequently, we use the calculated metrics and the reuse rate information to construct the profile of each metric and build a mechanism to translate its values into a reusability score.

Given that our benchmark dataset contains a series of diverse projects in terms of size, complexity, and functionality (also reflected in the values of static analysis metrics), we decided to formulate the ground truth upon which our models will be built on bin level. To that end and in an effort to take advantage of the information included in the benchmark dataset as a whole, our next step (step b) involves the extraction of the general distribution of the values of each metric, which is expressed as series of intervals (bins). Then, we employ the reuse rate information in order to assign a certain reusability score to each bin (step c), this way formulating our ground truth upon which we train our reusability evaluation models using polynomial regression (step d). Each model refers to a certain metric and translates its values into a reusability score. The design choice behind using one model for each metric, instead of one

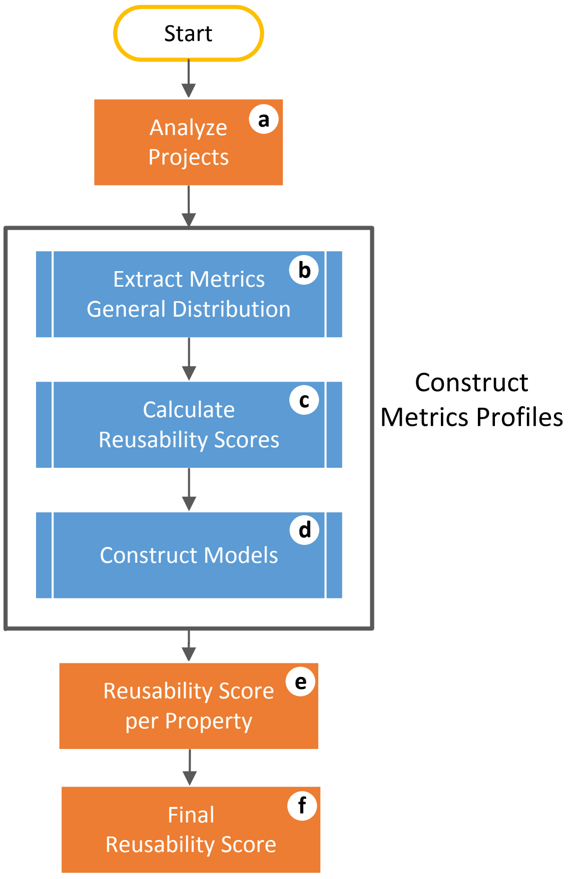

Start
Analyze Projects
a
Extract Metrics General Distribution
b
Calculate Reusability Scores
c
Construct Models
d
Construct Metrics Profiles
Reusability Score per Property
e
Final Reusability Score
f

Figure 5: Methodology Flow Chart.

generic model, relies in the fact that our primary design principle was to provide interpretable results that can lead to certain actionable recommendations towards improving the reusability degree of the component under evaluation.

Finally, having built our models and given that each metric belongs to a certain property, we apply a hierarchical evaluation approach in order to compute the final reusability score for the component under evaluation. This hierarchical approach involves calculating a score for each respective property by aggregating the scores of all metrics that belong to this property (step e) and aggregate the scores at property level into the final reusability score (step f). Of course, given the fact that each metric has a different significance and thus contributes differently in the reusability degree of software, we use weighted average as our aggregation formula. The weight for each metric is the correlation coefficient between its values and the reuse rate.

4.2. *Binning Strategy*

As already mentioned, we use the values of each static analysis metric along with the calculated reuse rate of software components in order to create a profile that translates the values of the respective metric into a reusability score. In order to examine, and thus model, the influence of each metric on the reuse rate of software components, we first extract its distribution at both class and package levels. Given that these values are distributed differently among the different projects of the benchmark dataset, we use the values from all packages (or classes) included in the dataset for which the calculated reuse rate values are non-zero, so as to formulate this generic distribution, and then determine the optimal bin specifications (number of bins and bin size) that result in minimum information loss.

Figure 6 depicts the histogram calculated for the values of the Weighted Methods per Class (WMC) metric for the classes that belong to three different projects. Each color (blue, red, and green) corresponds to a different project. The differences among the three distributions are expected since each project has its individual characteristics in terms of functionality, scope, contributors etc., which are reflected in the source code. For each metric we average over its values for all source code components (classes or packages), which allows us to produce a generic distribution that reflects our entire benchmark dataset.

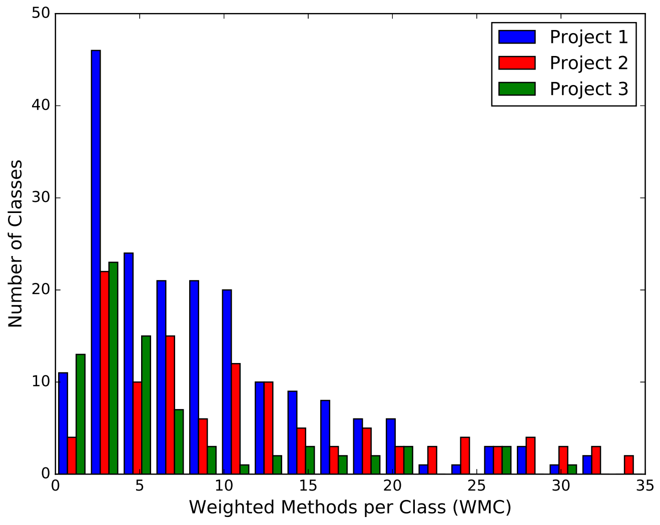


Figure 6: Distribution of the Weighted Method per Class (WMC) metric values for three different projects.

Figure 7 depicts the generic distribution of the values for the Weighted Methods per Class (WMC) metric, produced using the above methodology. We employ the Scott formula [50] for selecting the appropriate bin size, which asymptotically minimizes the integrated mean squared error and represents a global error measure of a histogram estimate. The bin width is given by the following formula:

$$BinWidth = 3.49 \cdot \hat{\sigma} \cdot n^{-1/3} \tag{1}$$

where $\hat{\sigma}$ is an estimate of the standard deviation and n is the size of the data sample.

4.3. *Reusability Modeling*

Upon having extracted the generic distribution of the values of each static analysis metric, we use the generated bins in order to construct a set of data instances that translate the values of each metric into a reusability score. Towards this direction, we use the reuse rate given by the AGORA code search engine. Our methodology that translates the values of each metric (e.g.

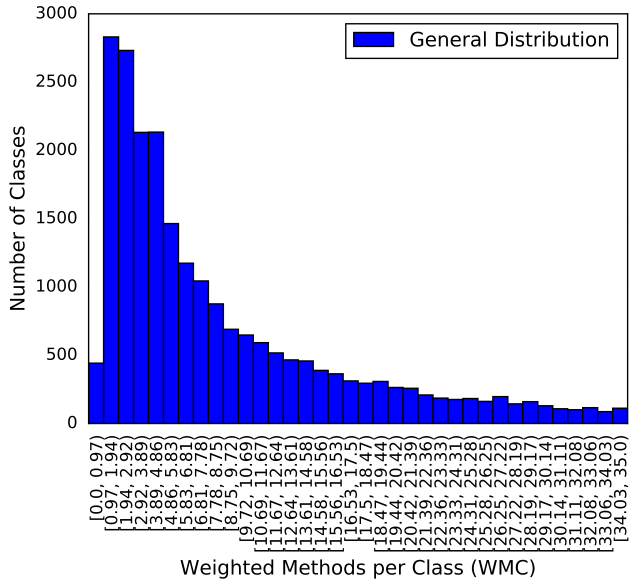


Figure 7: The generic distribution of the Weighted Method per Class (WMC) metric values.

WMC at class level) to a reusability score involves the following three steps:

1. For each generated bin (using the binning strategy described in the previous subsection), we select all components (classes or packages) for which the values of the metric lie in the bin interval.
2. We assign a reusability score to each bin (i.e. at the bin center), which is computed as the sum of the reuse rates of the components of the bins that were selected in the previous step. Once all bins are assigned with a reusability score, we normalize these scores so that they are all in the interval [0, 1].
3. We apply polynomial regression on the set of data instances produced in the previous steps, which have the form $[BinCenter, ReusabilityScore]$. The result of the regression is a reusability evaluation model that returns a reusability score given the value of the metric (continuous input/output).

Figure 8 illustrates the results of the reusability modeling for the Weighted Methods per Class (WMC) metric at class level. Each bin has been assigned with a reusability score based on the reuse rate of the classes contained in the benchmark dataset (see the aforementioned steps 1 and 2). The red line depicts the fitted curve, calculated using polynomial regression (degree = 7), while the green line refers to the normalized fitted curve. The normalized fitted curve is used in order to ensure that the reusability scores will fall into the interval [0, 1]. The normalized score for each bin is given by the following equation:

$$Normalized_{Score} = \frac{Score - min_{Score}}{max_{Score} - min_{Score}} \tag{2}$$

where min_{Score} and max_{Score} refer to the minimum and the maximum scores, respectively.

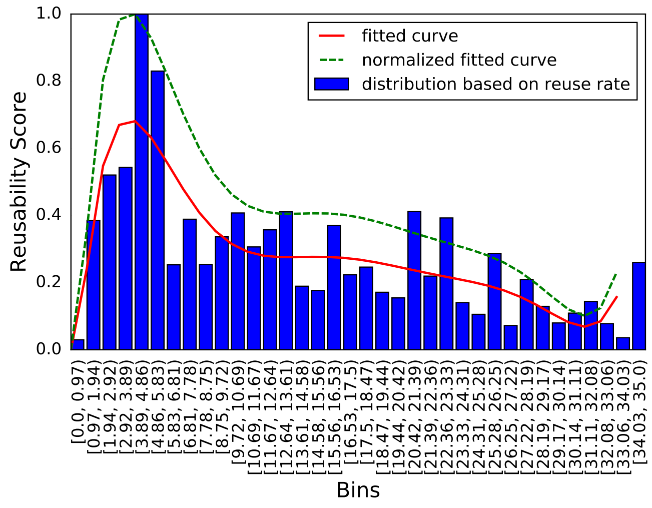


Figure 8: Overview of the fitting procedure regarding the Weighted Methods per Class (WMC) metric at class level.

The degree of the polynomial for each metric was determined using the elbow method on the *Root-Mean-Square-Error (RMSE)*. This ensures that the constructed models are effective and able to provide reasonable reusability estimates, while we avoid overfitting. Given the actual scores $y_i, y_2, \ldots, y_n$ and the predicted scores $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$, the RMSE is calculated as follows:

$$RMSE = \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (\hat{y}_i - y_i)^2} \tag{3}$$

where N is the number of samples in the dataset. The RMSE and the Mean Absolute Error (MAE) of the polynomial regression models for all metrics at class and package levels are shown in Table 4. Up until this point, we have produced individual models, each receiving as input the value of a static analysis metric and producing a reusability degree according to this metric. The next step is to produce a reusability score for each source code property (size, complexity, etc.). To this end, for each property, we aggregate the reusability scores of the related metrics. During this process, we need to take into account that each metric has a different level of significance in the formulation of the reuse rate of each software component. We quantify this significance by computing the correlation coefficient between the values of each metric and the reuse rate. As a result, the reusability score for each source code property is given by the following equation:

$$Score_{Property} = \frac{\sum_{i=1}^{N} w(i) \cdot Score_{metric(i)}}{\sum_{i=1}^{N} w(i)} \tag{4}$$

where N is the total number of metrics that belong to the property, $Score_{metric(i)}$ is the reusability score calculated for the $i-th$ metric and $w(i)$ is the Pearson correlation coefficient between the values of the $i-th$ metric and the $ReuseRate$.

Table 4 presents the calculated coefficients (weights) for all static analysis metrics at both class and package levels. Our design choice for using weighted average for aggregation (instead

Table 4: Polynomial regression results

Metric	Level	RMSE	MAE	Weight	Metric	Level	RMSE	MAE	Weight
LCOM5	class	0.124	0.23	0.87	LLOC	class	0.086	0.237	0.73
NL	class	0.02	0.124	0.79	TLLOC	class	0.073	0.217	0.68
NLE	class	0.165	0.262	0.88	TNA	class	0.004	0.048	0.9
WMC	class	0.036	0.144	0.59	NG	class	0.009	0.068	0.79
CBO	class	0.044	0.175	0.15	TNG	class	0.006	0.059	0.82
CBOI	class	0.088	0.194	0.58	TNM	class	0.109	0.262	0.63
NII	class	0.013	0.095	0.93	TNOS	class	0.018	0.09	0.78
NOI	class	0.088	0.152	0.88	TNPM	class	0.057	0.179	0.77
RFC	class	0.068	0.17	0.73	AD	package	0.061	0.178	0.67
AD	class	0.076	0.175	0.38	CD	package	0.124	0.28	0.33
CD	class	0.078	0.234	0.81	TCD	package	0.175	0.326	0.33
TCD	class	0.086	0.246	0.84	TCLOC	package	0.378	0.568	0.07
TCLOC	class	0.098	0.248	0.36	PDA	package	0.118	0.294	0.36
CLOC	class	0.148	0.334	0.38	TPDA	package	0.043	0.148	0.61
DLOC	class	0.208	0.405	0.28	LOC	package	0.187	0.348	0.55
PDA	class	0.244	0.421	0.52	LLOC	package	0.052	0.185	0.62
DIT	class	0.339	0.512	0.75	TLLOC	package	0.007	0.056	0.84
LOC	class	0.051	0.188	0.6	NCL	package	0.038	0.13	0.53

of simple average) originates from our data-driven approach; we refrain from taking the arbitrary decision that all metrics contribute equally in the reusability degree of software components, rather we evaluate the importance of each metric based on the correlation of its values with the calculated reuse rate. In this context, differences in the importance of metrics are expected since each metric has a different scope, which is reflected in its calculation formula. For instance, the fact that the DLOC metric appears to have less significance than CD is reasonable from a quality perspective, given that the percentage of documented lines of code (expressed by CD) within a class incorporates more information as compared to the absolute lines of documentation (expressed by DLOC).

At the end of the process, we compute the final reusability score of a source code component (class of package) by aggregating the scores calculated for each source code property using average as our aggregation formula. As a result, the final reusability score is given by the following equation:

$$Score_{Final} = \frac{1}{N} \sum_{i=1}^{N} Score_{Property}(i) \tag{5}$$

where N is the total number of the evaluated properties (six for the case of classes and two for the case of packages and $Score_{Property}(i)$ is the reusability score calculated for the $i-th$ property.

5. Evaluation

In this section we evaluate our methodology, in order to measure its effectiveness for assessing the reusability of source code components. To this end, we performed a multi-faceted evaluation following a bottom-up strategy. At first, we manually examine certain components that have been assessed by our system and exhibit high and low reusability scores. This ensures that the reusability scores provided by our methodology are not only interpretable, but they also conform to reasonable code quality characteristics. Secondly, we evaluate the accuracy of our methodology against a different approach that also employs metrics towards evaluating the reusability of software components and also against a different modelling strategy, which involves using all metrics in order to construct a single regression model for evaluating the reusability degree of software. This comparison shall provide interesting insight about the accuracy of our methodology. Finally, we apply our modeling methodology on the benchmark dataset in order to validate it and ensure that there are no biases in our models. To do so, we examine the distribution of the overall reusability score at both class and package levels, as well as the distributions of the scores for each source code property.

5.1. Example Reusability Evaluation - Case Study

As already noted, in the context of our evaluation, we manually examined the static analysis metrics of sample classes and packages in order to check whether they align with the estimation of our system. Table 5 provides an overview of the computed static analysis metrics for representative examples of classes and packages with different reusability scores. The table contains static analysis metrics for two classes and two packages that received both high and low reusability scores. The first class has a score of 89.1% and the second has 6.58%, while the first package has a score of 89.3% and the second has 9.68%.

Table 5: Overview of the reusability scores for classes and packages that received both high and low reusability scores

Metric				**Classes**		**Packages**	
Property	**Name**	**Min**	**Max**	**Class with high score** *(0.891 or 89.1%)*	**Class with low score** *(0.066 or 6.6%)*	**Package with high score** *(0.893 or 89.3%)*	**Package with low score** *(0.097 or 9.7%)*
Cohesion	LCOM5	0	15	0	12	—	—
Cohesion Scores				***100.00%***	***0.00%***	—	—
Complexity	NL	0	22	1	17	—	—
	NLE	0	9	1	6	—	—
	WMC	0	741	12	421	—	—
Complexity Scores				***87.43%***	***9.18%***	—	—
Coupling	CBO	0	87	2	45	—	—
	CBOI	0	110	1	35	—	—
	NII	0	576	1	346	—	—
	NOI	0	223	0	123	—	—
	RFC	0	343	3	130	—	—
Coupling Scores				***93.74%***	***11.69%***	—	—
Documentation	AD	0%	100%	100%	4.12%	100%	8.46%
	CD	0%	82.3%	31.03%	2.32%	57.85%	7.49%
	TCD	0%	82.3%	31.03%	2.32%	57.85%	7.49%
	CLOC	0	1,487	45	17	—	—
	TCLOC	0	1,521	45	17	277	274
	DLOC	0	1,188	45	17	—	—
	PDA	0	78	3	5	—	—
	TPDA	2	4573	—	—	11	17
Documentation Scores				***89.67%***	***7.48%***	***87.34%***	***8.46%***
Inheritance	DIT	0	5	1	5	—	—
Inheritance Scores				***81.10%***	***1.06%***	—	—
Size	LOC	3	4,078	145	732	480	3657
	LLOC	3	2,944	99	697	391	2984
	TLLOC	3	3,029	99	697	391	2984
	TLOC	36	9,847	—	—	480	3657
	TNA	0	167	10	95	65	162
	NG	0	101	3	2	—	—
	TNG	0	162	3	2	—	—
	TNM	0	500	6	21	—	—
	TNOS	0	1,845	36	634	—	—
	TNPM	0	309	5	3	—	—
	NCL	0	85	—	—	5	11
Size Scores				***87.47%***	***10.08%***	***91.28%***	***10.91%***

Examining the values of the metrics, we may note that the reusability estimation in all four cases is reasonable from a quality perspective. Concerning the class that received high reusability estimation, it appears to be very cohesive as the LCOM5 (Lack of Cohesion in Methods 5) metric, which refers to the number of cohesive classes that a non-cohesive class should be split, is 0. From a complexity perspective, the class appears to be very well structured, which is denoted by the low values of the nesting level (NL and NLE) metrics, along with the low value (12) of the weighted methods per class (WMC) metric. The latter is computed as the sum of the McCabes Cyclomatic Complexity (McCC) values of the class' local methods. As a result, a high score for the complexity property is expected. Furthermore, this class appears to be very well documented (the value of the API documentation percentage is 100%), which indicates that all of its public methods are documented and thus is suitable for reuse. The proper level of documentation also originates from the value of the comments density (CD) metric according to which there is almost one line of comments every three lines of code. On top of the above, given the values of the inheritance-related and the coupling-related metrics, the class appears to be very well decoupled. This fact is obvious from the low values of the CBO and DIT metrics, which refer to the number of directly used other classes by the class and the length of the path that leads from the class to its farthest ancestor, respectively. Lastly, the value of its Lines of

Code (LOC) metric combined with the number of its methods (NM), which are highly correlated with the degree of understandability and thus the reusability, are typical. As a result, the reusability score for this class is quite rational and complies with the interpretation of the static analysis metrics from a quality perspective.

On the other hand, the class that received low score appears to have very low cohesion (reflected in the high value of the LCOM5 metric), which suggests that the class should be split in more classes in order to regain high cohesion. In addition, given the high values of the complexity metrics and the coupling metrics, the class appears to be poorly structured as it is rather complex and highly coupled with other classes. As expected, this has a negative impact on the reusability estimation, which is reflected on both the complexity score (9.18%) and the coupling score (11.69%). Moreover, upon examining the values of the metrics that quantify the degree of documentation and the size of the class, we can see that the class is not properly documented as only 1 out of the 23 public methods are documented (AD values is 4.12%) and its size is above average (LOC value is 732). As a result of the aforementioned findings, the understandability degree of the class and thus its reusability is low. This is of course reflected on the value of the overall reusability score (6.58%) as well as on the scores for each source code property.

Similar conclusions can be drawn by examining the metric values at package level for the packages that received high and low reusability score. In the case of packages, the reusability estimation involves only two properties, *size* and *documentation*, as the rest of the static analysis metrics are only computed at class level. As derived by examining the metrics at file level, the conclusions are similar. The package that received high score appears to be very well documented (AD value is 100%) and also has typical size as it contains five classes with 480 lines of code in total. These values make it suitable for reuse, which is confirmed by the final reusability score (89.3%). On the contrary, the package that received low score is poorly documented and its LOC value is considerably above average.

Given the above findings, our reusability estimation methodology appears to be able to effectively translate the values of static analysis metrics into an interpretable reusability score. Furthermore, our system provides an analysis of the reusability degree of software components from six different perspectives, each corresponding to a different source code property.

5.2. Accuracy of Reusability Evaluation

In order to perform a versatile evaluation for our reusability evaluation approach in terms of providing effective and accurate reusability scores, we further analyzed the results of our methodology using the actual reuse rate of software components as our point of reference. In this context and in an effort to perform a thorough analysis, we do not only provide results regarding our models, but we also compare our reusability evaluation against a different reusability assessment methodology developed by Fazal et al. [49]. According with this methodology, which is applicable at class level, reusability is decoupled into six attributes; *flexibility*, *understandability*, *portability*, *Scope coverage*, *maintainability*, and *variability*. The reusability score at class level is given by the following series of equations:

$$\begin{aligned} Reusability = {} & 0.16 \times Flexibility + \\ & 0.16 \times Understandability + \\ & 0.16 \times Portability + \\ & 0.16 \times ScopeCoverage + \\ & 0.16 \times Maintainability + \\ & 0.16 \times Variability \end{aligned} \tag{6}$$

$$Flexibility = 1 - [(0.5 \times CBO) + (0.5 \times LCOM)] \tag{7}$$

$$\begin{aligned} Understandability = {} & 1 - 0.25 \times CBO \\ & - 0.25 \times LCOM \\ & - 0.25 \times CD \\ & - 0.125 \times LOC \\ & - 0.125 \times NM \end{aligned} \tag{8}$$

$$Portability = 1 - DIT \tag{9}$$

$$ScopeCoverage = \frac{NM}{TNM} \tag{10}$$

$$Maintainability = 0.5 \times McCC + 0.5 \times MI \tag{11}$$

$$Variability = 0.5 \times \frac{NOC}{NCL} + 0.5 \times ScopeCoverage \tag{12}$$

In the above equations, *McCC* refers to the McCabe Cyclomatic Complexity, *MI* refers to Maintainability Index, while *NOC* refers to the number of children. The rest of the metrics are described in Table 2. It is worth noting that the values of the static analysis metrics are normalized.

Our choice to compare our methodology against this approach originates from the fact that the two approaches have common design principles. At first, both approaches follow a hierarchical evaluation strategy and decompose reusability in a set of properties that correspond to the primary evaluation axes. In addition, both approaches employ static analysis metrics and link each one of them with the corresponding reusability-related property. Finally, both approaches construct an aggregation mechanism for calculating the final reusability score. Except for the fact that our approach uses more metrics, the main difference of the two approaches relies in the fact that our approach uses the reuse rate as ground truth information and thus provides data-driven reusability evaluation, while the one of Fazal et al. uses an empirical approach which involves experts and surveys in order to validate the results. Furthermore, trying to further evaluate our design choice of constructing a hierarchical reusability evaluation strategy, which involves training one model for each metric and aggregating the results into a final reusability score, we also compared our results against a different modeling strategy. Instead of training one model for each metric, this strategy involves training one regression model that uses the values of all metrics in order to directly predict the final reusability score. Ridge Regression [51] was used for training the regression model, while the dedicated regularized least-squares routine was selected as the solver function [52].

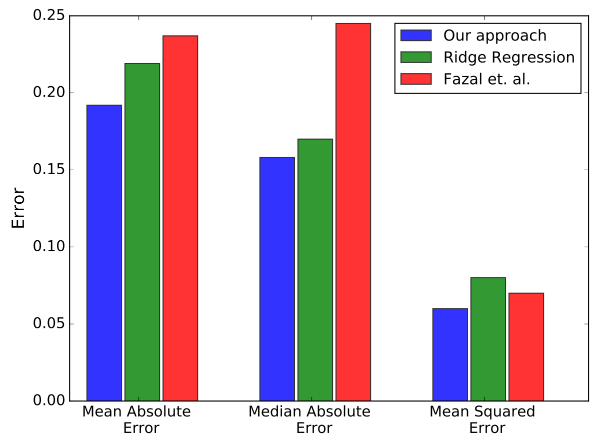


Figure 9: Comparative evaluation results.

Figure 9 provides an overview of the prediction errors of our reusability evaluation models against the two aforementioned approaches as reflected in the values of three metrics: a) the Mean Absolute Error, b) the Median Absolute Error, and c) the Mean Squared Error. The error values refer to the deviation of the predicted reusability score (using each one of the three approaches) from the actual reuse rate (normalized). As shown in the figure, our approach provides more accurate reusability evaluation than the other two approaches, which is reflected in the values of all three metrics. The approach of Fazal et al. appears to exhibit the highest values in both mean absolute error and median absolute error. The same conclusions are derived by Figure 10, which illustrates the distribution of the error values in the three approaches. Given the distribution of the error values regarding our approach (both in the original models and in the regression model), it is obvious that they are gradually decreasing as they diverge from zero. On the contrary, in the case of Fazal et al., there is a strong left-sided skewness, which originates from the fact that various components tend to be receive higher reusability scores than expected.

One could argue that the accuracy of the Ridge Regression modeling approach seems to be comparable to our modeling strategy thus there is no evident added value. However, the proposed approach exhibits certain advantages that lie in several directions. First of all, having one model that accounts for the values of all metrics in order to provide a single reusability score does not support the interpretability of the reusability score and thus does not enable certain actionable recommendations. In contrast, our hierarchical approach provides information regarding all the intermediate levels (metrics and properties) that influence the reusability degree of a software component and thus enables the direct identification of the source code characteristics that need improvement towards enhancing reusability. Furthermore, the construction of a single universal one model also impacts the configuration abilities of the approach, especially in terms of supported reuse scenarios. Ridge regression computes the reusability score of a certain component taking into account all metrics regardless of the preferences of the developer. In this context, it does not provide flexibility to the developer or the quality expert to ignore specific metrics when calculating the reusability score (could be different per case). This is easily tackled by our approach. Thus, we argue that our modeling strategy is more expressive and more configurable than ridge regression.

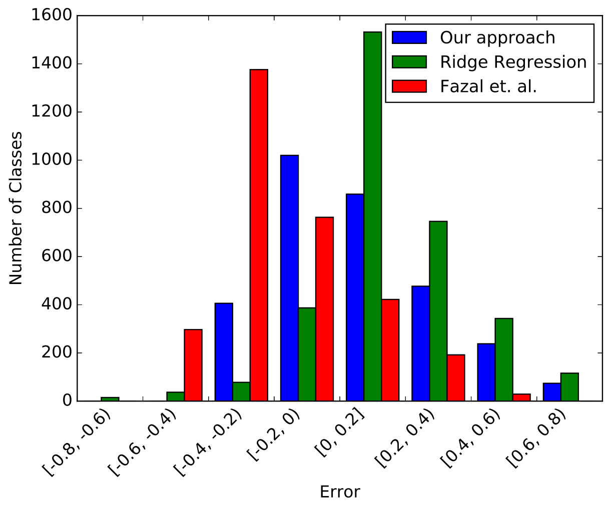


Figure 10: Error distribution of the reusability scores.

Finally, in an effort to further validate the accuracy of our reusability estimation methodology, we analyzed four different GitHub projects that offer the same functionalities. The selected projects, which are multimedia tools that offer image view and edit capabilities, differ in terms of popularity and reuse as reflected in the values of the number of GitHub stars and forks. Table 6 presents the reusability scores along with the number of GitHub stars and forks for each one of the analyzed projects. In order for the results to be comparable and eliminate any biases that originate from the fact that the projects differ in terms of size, each score corresponds to the mean reusability values of all components (classes or packages) that contain at least one public method. Given the provided results, it is obvious that the scores at both class and package levels are in line with the number of GitHub stars and forks.

Table 6: Reusability scores for Libraries that offer same functionality

Project Index	GitHub Stars	GitHub Forks	Score at Package Level	Score at Class Level
#1	5,360	840	57.8%	60.3%
#2	5,430	1,707	69.1%	65.8%
#3	138	66	45.4%	41.7%
#4	15	8	28.4%	22.9%

#1: davemorrissey/subsampling-scale-image-view

#2: daimajia/AndroidImageSlider

#3: chiuki/android-swipe-image-viewer

#4: hangox/CircleImageView

5.3. Distribution of Reusability Score

Further assessing the validity of our reusability scoring approach in order to identify any existing biases, we apply our methodology on the benchmark dataset. Figures 11 and 12 depict the distributions of the reusability scores for all projects at class and package level, respectively. As expected, the score at both levels of granularity follows a distribution similar to a normal distribution and the majority of instances are accumulated evenly around 0.5. In addition, the range of the calculated scores at both levels is higher that 0.9 (0.946 for classes and 0.954 for packages). As a result, we can conclude that our modeling approach effectively determines the degree to which a class or a package is reusable. For the score at package level, we observe a small left-sided skewness. After manual inspection of the classes with scores in [0:15; 0:25], we may conclude that they appear to contain little valuable information (e.g. most of them have LOC < 20 and comments density > 95%) and thus are given low score.

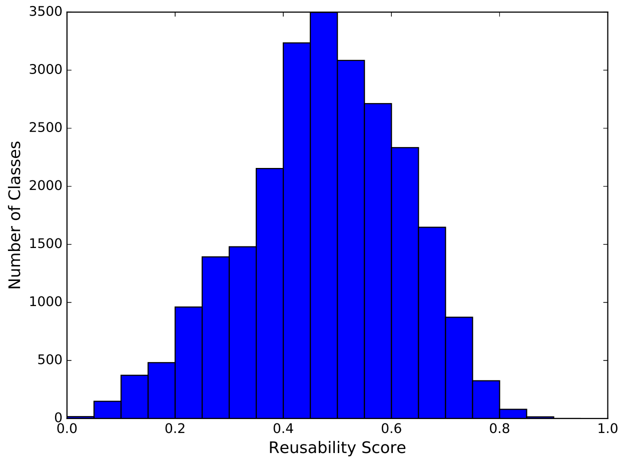


Figure 11: Overview of the fitting procedure regarding the Weighted Methods per Class (WMC) metric at class level.

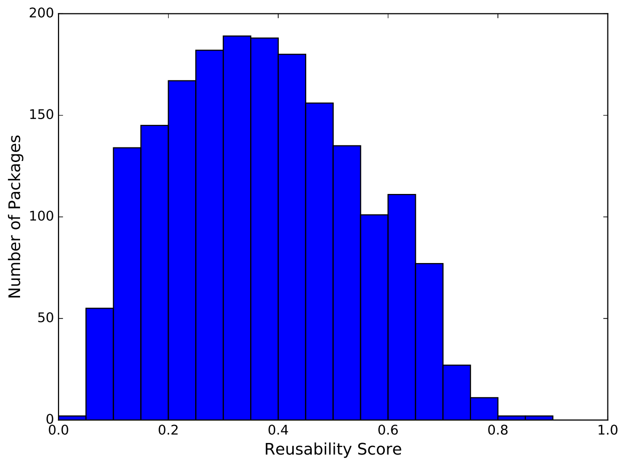


Figure 12: Overview of the fitting procedure regarding the Weighted Methods per Class (WMC) metric at class level.

Figures 13 and 14 depict how the individual quality scores of the source code properties (dashed lines) are aggregated into one final reusability score (solid black line) at class and package level, respectively. The x-axis refers to the classes (or packages) index and the y-axis refers to the reusability score. The indexes of the source code components in both figures are sorted in ascending order of quality score. As for the classes, the final score depends on the scores that were computed for six source code properties (*Complexity*, *Coupling*, *Cohesion*, *Inheritance*, *Documentation*, *Size*), while in the case of packages the final score relies on the scores of two properties (*Documentation*, *Size*).

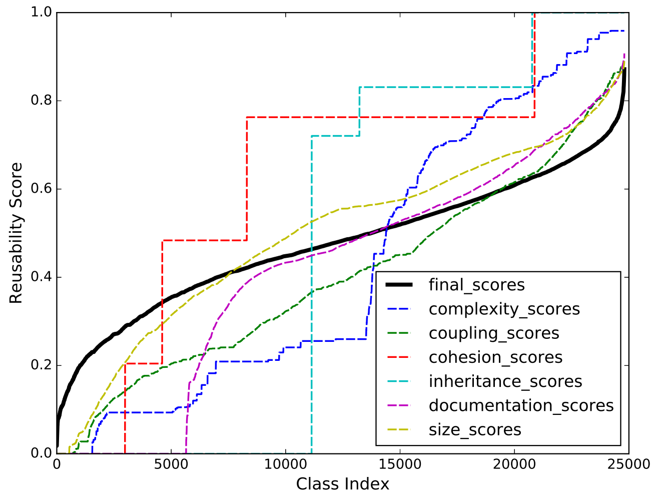


Figure 13: Overview of how the individual quality scores per property (dashed lines) are aggregated into the final score (solid line) at class level.

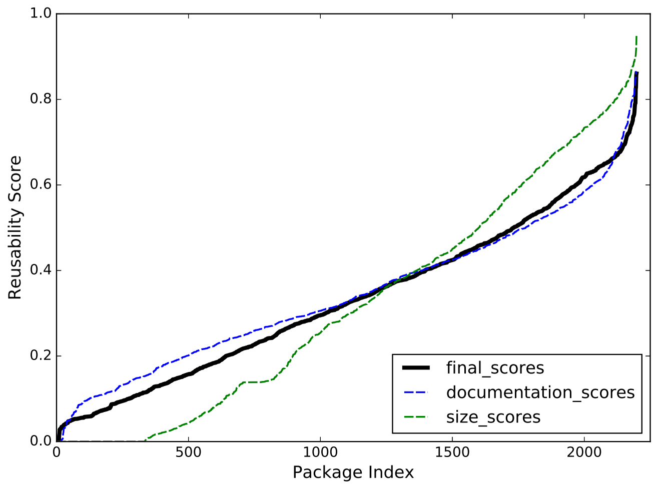


Figure 14: Overview of how the individual quality scores per property (dashed lines) are aggregated into the final score (solid line) at package level.

Upon examining the scores at class level, we may note that although each source code property appears to have its own behavior as its score values are distributed in a different manner, all properties seem to be in line with the final score. It addition, it is worth noticing that the increase behaviour of the values of the source code properties *cohesion* and *inheritance* reveals the existence of certain steps in the score. This is expected as both properties are linked, and thus evaluated, with only one metric (LCOM5 in the case of cohesion and DIT in the case of inheritance). On top of that, both LCOM5 and DIT metrics accept only distinct values which restricts the reusability evaluation result in certain ranges. This observation also explains the fact that *cohesion* and *inheritance* appear to have the highest deviation from the final score. Similar to case of classes, the trends of the individual scores at package level seem to follow the trend of the final score. The same results are derived by Figure 15, which illustrates how the individual quality scores of the complexity-related metrics (dashed lines) are aggregated into one complexity score (solid black line line).

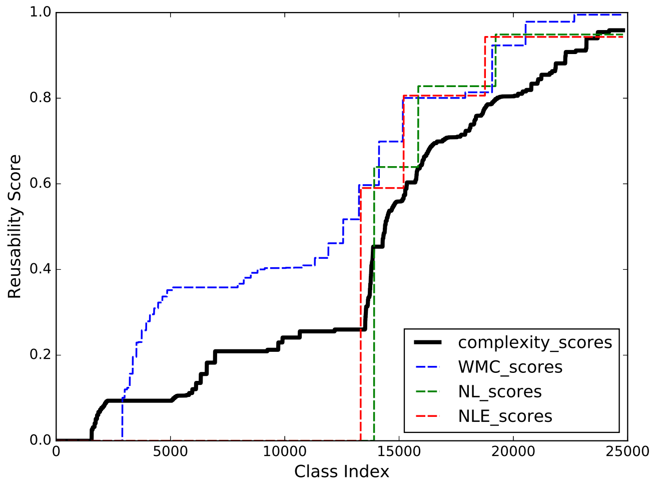


Figure 15: Overview of how the individual quality scores per complexity-related metric (dashed lines) are aggregated into the final complexity score (solid line) at class level.

6. Applicability of the Proposed Approach

One of our primary design principles was to construct a fully reproducible and expandable methodology towards evaluating the reusability degree of software components. In this section, we provide insights on how our methodology and/or the respective results can be exploited by the community.

First of all, given that our benchmark dataset is composed of a large set of general purpose libraries and covers a wide range of reuse scenarios, our reusability evaluation models can be used "as-is" towards evaluating and improving the reusability degree of software artifacts on the basis of certain source code properties. In this context, our models can be incorporated in code search engines and/or recommendation systems in order to optimize the ranking of the results based on the reusability of the proposed assets. Furthermore, even in the cases where reusability should be calculated based on specific metrics given the individual characteristics of the project under evaluation, our methodology is still applicable simply by changing the benchmark repository with a different one that includes projects that share the same characteristics with the one under evaluation. In this case, one can follow the steps described in section 3 in order to generate the new models. In addition, one can use the findings regarding the importance of the different static analysis metrics in the reusability degree of software components or enhance them by applying the described methodology on different domains and additional reuse scenarios. Finally, given the hierarchical approach of our methodology and the fact that we train one model for each metric, the proposed reusability evaluation approach provides several configuration abilities that can serve the individual needs of software projects. These configurations may target the aggregation layers by using different weights or the inclusion/exclusion of metrics and/or attributes that take part in the evaluation process.

7. Threats to Validity

The limitations and threats to validity of our reusability evaluation approach span along the following axes: a) limitations imposed by the use of the reuse rate as a measure of the extent to which a software component is reusable, b) choice of the benchmark dataset that consists of the most popular maven projects, and c) selection of static analysis metrics that constitute the information basis upon which our models are built.

Our approach employs the reuse rate of software components (classes and packages) as ground truth information towards evaluating the degree to which a software component is reusable. In this context and given the nature of reuse rate, which measures how often a software components has been reused, there are also other factors that may influence the final outcome, such as the offered functionality, current trends, and the popularity of the provider. In an effort to minimize the impact of those limitations, we resort in analyzing the most popular projects included in the maven registry. Maven registry is composed of a very large set of projects and thus ensures that there are multiple options regarding each reuse scenario, all of them equally "easy-to-use" from a functional perspective. As a result, the introduced biases are limited to the ones that originate from existing trends in the software development ecosystem. Note, however, that we select the most popular libraries of the maven repository, which at least ensures that the effect of these trends will not be too diverse among projects.

Another limitation that occurs from our methodology is the fact that the ability of our models to generalize totally depends on the benchmark dataset and the selected metrics that quantify the properties of the source code. In our case and in an effort to cover a wide series of development scenarios, we chose maven that mainly contains general purpose libraries and thus contains source code that is subject to be reused by a wide audience. In addition, in order to cover different needs and scopes, we chose metrics that provide source code analysis on multiple axes. As a result in cases when the reuse scenario involves a certain domain with certain characteristics that are reflected in the source code and do not comply with the general reuse principles, our methodology requires the use of a different benchmark dataset and/or additional metrics. This benchmark along with the corresponding metrics should be carefully selected in order to comply with the desired behavior.

8. Conclusions and Future work

As already noted, contemporary software engineering practices rely more and more on reusing existing components, especially in the context of open-source development. In this aspect, the new challenge that arises is to effectively determine the reusability of source code components. And although there are several efforts towards this direction, most of them either rely on expert help or employ arbitrary ground truth datasets, thus resulting in context-dependent and subjective results.

In this work, we proposed a novel software reusability evaluation approach based on the hypothesis that the extent to which a software component is reusable is associated with the way that it is perceived by developers. Towards this direction, we analyzed the one hundred most popular projects included in the maven registry and used the capabilities of the code search engine AGORA, in order to calculate the reuse rate at class and package level and thus formulate our ground truth. Furthermore, we designed a methodology for modeling the impact of the values of various static analysis metrics on the reuse rate of software components. We associated each metric with a certain source code property and followed a correlation-based approach in order to calculate a reusability score for each property. Each score corresponds to the extent to which a software component is reusable given its property-related characteristics. Finally, upon evaluating our system on several axes, our findings indicate that our methodology can be effective for estimating the reusability degree of classes and packages as perceived by developers.

Future work on our methodology can be performed in multiple directions. At first, we intend to extend our modeling approach so as to evaluate the reusability degree of software at method level and thus expand its applicability in more reuse scenarios. Furthermore, in an effort to further expand our ground truth and thus the effectiveness of our models, we can also incorporate additional metrics that originate from the software development process and from online repositories (i.e. number of GitHub stars/forks and/or process metrics such as the update frequency or the number of related issues). An interesting idea would also be the investigation of an adaptive reusability score in a domain-specific context. Moreover, we could add more static analysis metrics in the analysis in order to cover additional aspects of the source code or better quantify the ones that are already covered. A representative example of such metrics can be the ones proposed by QMOOD model [15]. Finally, we could assess the effectiveness of our approach with a user study, under some realistic reuse scenarios, and thus further validate our findings.

References

[1] C. Schmidt, Agile Software Development Teams, Springer, 2015.

[2] M. Robillard, R. Walker, T. Zimmermann, Recommendation Systems for Software Engineering, IEEE Softw. 27 (4) (2010) 80–86.

[3] S. Thummalapenta, T. Xie, PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web, in: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 204–213.

[4] N. Sahavechaphan, K. Claypool, XSnippet: Mining for Sample Code, SIGPLAN Not. 41 (10) (2006) 413–430.

[5] O. Hummel, W. Janjic, C. Atkinson, Code Conjurer: Pulling Reusable Software out of Thin Air, IEEE Softw. 25 (5) (2008) 45–52.

[6] T. Diamantopoulos, G. Karagiannopoulos, A. Symeonidis, Codecatch: Extracting source code snippets from online sources, in: IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), Gothenburg, Sweden, 2018, pp. 21–27.

[7] S. Pfleeger, B. Kitchenham, Software quality: The elusive target, IEEE Software (1996) 12–21.

[8] ISO/IEC 25010:2011, `https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en`, [Online; accessed November 2018] (2011).

[9] S. L. Pfleeger, J. M. Atlee, Software engineering: theory and practice, Pearson Education India, 1998.

[10] A. P. Singh, P. Tomar, Estimation of Component Reusability through Reusability Metrics, International Journal of Computer, Electrical, Automation, Control and Information Engineering 8 (11) (2014) 1965–1972.

[11] P. S. Sandhu, H. Singh, A reusability evaluation model for OO-based software components, International Journal of Computer Science 1 (4) (2006) 259–264.

[12] T. Diamantopoulos, K. Thomopoulos, A. Symeonidis, QualBoa: reusability-aware recommendations of source code components, in: IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), 2016, IEEE, 2016, pp. 488–491.

[13] F. Taibi, Empirical Analysis of the Reusability of Object-Oriented Program Code in Open-Source Software, International Journal of Computer, Information, System and Control Engineering 8 (1) (2014) 114–120.

[14] C. Le Goues, W. Weimer, Measuring code quality to improve specification mining, IEEE Transactions on Software Engineering 38 (1) (2012) 175–190.

[15] H. Washizaki, R. Namiki, T. Fukuoka, Y. Harada, H. Watanabe, A framework for measuring and evaluating program source code quality, in: PROFES, Springer, 2007, pp. 284–299.

[16] S. Zhong, T. M. Khoshgoftaar, N. Seliya, Unsupervised Learning for Expert-Based Software Quality Estimation., in: HASE, 2004, pp. 149–155.

[17] A. Kaur, H. Monga, M. Kaur, P. S. Sandhu, Identification and performance evaluation of reusable software components based neural network, International Journal of Research in Engineering and Technology 1 (2) (2012) 100–104.

[18] S. Manhas, R. Vashisht, P. S. Sandhu, N. Neeru, Reusability Evaluation Model for Procedure-Based Software Systems, International Journal of Computer and Electrical Engineering 2 (6).

[19] A. Kumar, Measuring Software reusability using SVM based classifier approach, International Journal of Information Technology and Knowledge Management 5 (1) (2012) 205–209.

[20] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.

[21] T. Cai, M. R. Lyu, K.-F. Wong, M. Wong, ComPARE: A generic quality assessment environment for component-based software systems, in: Intern. Symposium on Information Systems and Engineering, 2001.

[22] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, T. Gyimóthy, A probabilistic software quality model, in: 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 243–252.

[23] J. P. Correia, J. Visser, Benchmarking technical quality of software products, in: 2008 15th Working Conference on Reverse Engineering, 2008, pp. 297–300.

[24] R. Baggen, J. P. Correia, K. Schill, J. Visser, Standardized code quality benchmarking for improving software maintainability, Software Quality Journal 20 (2) (2012) 287–307.

[25] P. Oliveira, M. T. Valente, F. P. Lima, Extracting relative thresholds for source code metrics, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 254–263.

[26] T. L. Alves, C. Ypma, J. Visser, Deriving metric thresholds from benchmark data, in: 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–10.

[27] M. Papamichail, T. Diamantopoulos, A. L. Symeonidis, User-perceived source code quality estimation based on static analysis metrics, in: 2016 IEEE International Conference on Software Quality, Reliability and Se-

curity (QRS), Vienna, Austria, 2016, pp. 100–107.

[28] V. Dimaridou, A.-C. Kyprianidis, M. Papamichail, T. Diamantopoulos, A. Symeonidis, Towards modeling the user-perceived quality of source code using static analysis metrics, in: 12th International Conference on Software Technologies (ICSOFT), Madrid, Spain, 2017, pp. 73–84.

[29] V. Dimaridou, A.-C. Kyprianidis, M. Papamichail, T. Diamantopoulos, A. Symeonidis, Assessing the user-perceived quality of source code components using static analysis metrics, in: Communications in Computer and Information Science (CCIS), Vol. 868, 2018, pp. 3–27.

[30] M. Papamichail, T. Diamantopoulos, I. Chrysovergis, P. Samlidis, A. Symeonidis, User-perceived reusability estimation based on analysis of software repositories, in: 2018 IEEE International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), Campobasso, Italy, 2018, pp. 49–54.

[31] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, M. Irlbeck, On the extent and nature of software reuse in open source java projects, in: Proceedings of the 12th International Conference on Top Productivity Through Software Reuse, ICSR'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 207–222.

[32] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, A. E. Hassan, A large-scale empirical study on software reuse in mobile apps, IEEE software 31 (2) (2014) 78–86.

[33] G. Gui, P. D. Scott, Ranking reusability of software components using coupling metrics, Journal of Systems and Software 80 (9) (2007) 1450–1459.

[34] O. P. Rotaru, M. Dobre, Reusability metrics for software components, in: The 3rd ACS/IEEE International Conference on Computer Systems and Applications, IEEE, 2005, p. 24.

[35] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on software engineering 28 (1) (2002) 4–17.

[36] M. G. Siavvas, K. C. Chatzidimitriou, A. L. Symeonidis, Qatch-an adaptive framework for software product quality assessment, Expert Systems with Applications 86 (2017) 350–366.

[37] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, H. C. Almeida, Identifying thresholds for object-oriented software metrics, Journal of Systems and Software 85 (2) (2012) 244–257.

[38] T. L. Alves, C. Ypma, J. Visser, Deriving metric thresholds from benchmark data, in: IEEE International Conference on Software Maintenance (ICSM), IEEE, 2010, pp. 1–10.

[39] P. Oliveira, M. T. Valente, F. P. Lima, Extracting relative thresholds for source code metrics, in: IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, IEEE, 2014, pp. 254–263.

[40] T. G. Bay, K. Pauls, Reuse Frequency as Metric for Component Assessment, Tech. rep., ETH, Department of Computer Science, Zurich, technical Reports D-INFK (2004).

[41] J. Davies, D. M. German, M. W. Godfrey, A. Hindle, Software bertillonage: Finding the provenance of an entity, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, ACM, New York, NY, USA, 2011, pp. 183–192.

[42] K. Inoue, Y. Sasaki, P. Xia, Y. Manabe, Where does this code come from and where does it go?-integrated code history tracker for open source systems, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, 2012, pp. 331–341.

[43] R. Abdalkareem, E. Shihab, J. Rilling, On code reuse from stackoverflow: An exploratory study on android apps, Information and Software Technology 88 (2017) 148–158.

[44] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, K. Inoue, Identifying source code reuse across repositories using lcs-based source code similarity, in: IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2014, pp. 305–314.

[45] K. Aggarwal, A. Hindle, E. Stroulia, Co-evolution of Project Documentation and Popularity Within Github, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, New York, NY, USA, 2014, pp. 360–363.

[46] S. Weber, J. Luo, What Makes an Open Source Code Popular on GitHub?, in: 2014 IEEE International Conference on Data Mining Workshop, ICDMW, 2014, pp. 851–855.

[47] H. Borges, A. Hora, M. T. Valente, Understanding the Factors That Impact the Popularity of GitHub Repositories, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), ICSME, 2016, pp. 334–344.

[48] N. S. Gill, S. Sikka, Inheritance hierarchy based reuse & reusability metrics in oosd, International Journal on Computer Science and Engineering 3 (6) (2011) 2300–2309.

[49] A. K. M. Fazal-e Amin, A. Oxley, Reusability assessment of open source components for software product lines, International Journal on New Computer Architectures and Their Applications (IJNCAA) 1 (3) (2011) 519–533.

[50] D. W. Scott, On optimal and data-based histograms, Biometrika 66 (3) (1979) 605–610.

[51] A. E. Hoerl, R. W. Kennard, Ridge regression: Biased estimation for nonorthogonal problems, Technometrics 12 (1) (1970) 55–67.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.