User-Perceived Source Code Quality Estimation based on Static Analysis Metrics

Michail Papamichail, Themistoklis Diamantopoulos and Andreas Symeonidis
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
Intelligent Systems & Software Engineering Labgroup, Information Processing Laboratory
Thessaloniki, Greece
Email: {mpapamic, thdiaman}@issel.ee.auth.gr, asymeon@eng.auth.gr

Abstract—**The popularity of open source software repositories and the highly adopted paradigm of software reuse have led to the development of several tools that aspire to assess the quality of source code. However, most software quality estimation tools, even the ones using adaptable models, depend on fixed metric thresholds for defining the ground truth. In this work we argue that the popularity of software components, as perceived by developers, can be considered as an indicator of software quality. We present a generic methodology that relates quality with source code metrics and estimates the quality of software components residing in popular GitHub repositories. Our methodology employs two models: a one-class classifier, used to rule out low quality code, and a neural network, that computes a quality score for each software component. Preliminary evaluation indicates that our approach can be effective for identifying high quality software components in the context of reuse.**

Keywords—Software quality estimation, static analysis, user-perceived quality, neural networks, one-class classification

I. Introduction

The exploitation of numerous open source software projects residing in online software repositories, such as GitHub and Sourceforge, has greatly facilitated the process of rapid software prototyping. The deluge of open and available source code has raised the need to determine whether a software component is suitable for reuse, a problem which is closely related to the *quality* of the component.

Quality is a complex concept and highly context-dependent due to the fact that it means different things to different people [1]. Therefore, the need for standardization arises to create a common reference among the scientific community. Towards this direction, the international standard ISO/IEC 25010:2011 [2] defines a quality model that consists of eight quality characteristics: *Functional Suitability*, *Reliability*, *Performance* and *Efficiency*, *Usability*, *Maintainability*, *Security*, *Compatibility* and *Portability*. From a developer perspective, the extent to which a software component is reusable, namely its *reusability*, is related to four of these quality characteristics: Functional Suitability, Usability, Maintainability, and Portability [3, 4].

Attempting to measure software quality in a quantified manner has led to the proposal of various metrics. Some of them measure aspects of the software development process [6] or quantify human resources [7], while others are applied on source code, such as the widely-known CK metrics [5]. In a generic context, research efforts towards estimating software quality are based on source code metrics [8], [9]. In those cases, the problem that arises is how to determine the thresholds of these metrics in order to decide whether the source code abides by specific quality criteria. Defining these thresholds is a non-trivial task, usually performed by an expert [10], and thus it is highly improbable that the process would be flexible enough to effectively describe diverse classes of software components. As a result, certain approaches focus on designing models that provide adaptable quality estimation for most components [12]. Still, such systems are usually limited within certain quality thresholds, imposed by some ground truth (possibly expert-defined) quality value, which may not be objective enough for all cases.

In this work, we mitigate the need for such an imposed ground truth quality value, by employing *user-perceived quality* as a measure of the reusability of a software component. From a developer perspective, we are able to associate user-perceived quality with the extent to which a software component is adopted by developers, thus, arguing that the popularity of a component can be a measure of its reusability. We provide a proof-of-concept for this hypothesis and further investigate the relationship between popularity and user-perceived quality by using a dataset of source code components along with their popularity as determined by their rating on GitHub. Based on this argument, we design and implement a methodology that covers the four software quality characteristics related to user-perceived quality by using a large set of static analysis metrics (73 in total). Given these metrics and using the popularity of GitHub projects as ground truth, we can estimate the quality of a source code component. Our system employs two models: a one-class classifier trained using support vector machines (SVMs) that determines whether a source code component falls below a minimum quality threshold, and an artificial neural network (ANN) that estimates the quality of the code given the static analysis metrics.

The rest of this paper is organized as follows. Section II reviews the current state-of-the-art in quality estimation using software metrics. Section III illustrates the formation of our dataset, assessing also the relationship between popularity and user-perceived quality in the context of reusability. Section IV describes the data-mining models, the one-class classifier and the ANN, and illustrates the process of translating static analysis metrics into quality estimation. Sections V and VI assess our methodology and discuss any threats to validity, respectively, while Section VII concludes the paper and provides insight for further research.

II. Related Work

Estimating quality characteristics is a well-known problem, which has drawn the attention of researchers for many years. Although several software metrics have been proposed to measure quality [5], determining the thresholds is not a trivial problem. Furthermore, in cases where thresholds can be defined [16], these usually cover very restricted scenarios.

Identifying software metrics thresholds usually requires quality experts to examine the code and assess its quality. Since, however, expert help is not always available or in some cases not feasible, several approaches have been proposed. A common practice involves deriving metrics thresholds by analyzing the values of software metrics computed for various software projects. Towards this direction, Ferreira et al. [11] determine metrics thresholds by fitting the values of the computed metrics into probability distributions. The distribution that best fits the values of each metric indicates the bounds of the accepted values.

Further examining the problem of software quality estimation, several research efforts have been directed towards constructing systems that provide a quality score. Cai et al. [12] proposed the generic quality assessment tool ComPARE, which makes use of several software metrics in order to define quality models. The evaluation is based on the software metrics values computed by the tool and the user defined model parameters. Another approach is proposed by Washizaki et al. [9], who suggests a metrics-based framework that categorizes the main software quality characteristics defined in ISO9126-1 into sub characteristics, each of which is assigned with a metric. After that, it computes a score for every characteristic based on the values of the software metrics, where the desired intervals are determined using benchmarking.

Furthermore, there are also approaches related to building quality evaluation systems targeting a single quality characteristic. Baggen et al. [14] overview the SIG approach on measuring the maintainability of source code which uses a standardized maintainability measurement model based on the ISO/IEC 9126. Hegedus et al. [15] employ static analysis to build a probabilistic maintainability evaluation model. The proposed model is also based on the ISO/IEC 9126 and involves experts in order to define the metrics weights. Lastly, another approach by A. Kumar [19] employs an SVM-based classifier so as to build a code reusability evaluation system.

Although the approaches discussed in the previous paragraphs can be effective for certain cases, their applicability in real-world scenarios is limited. At first, approaches using predefined thresholds [16], even with the guidance of an expert, cannot extend their scope beyond the specific class and the characteristics of the software for which they have been designed. Automated quality evaluation systems appear to overcome these issues [9, 14, 15, 17, 19]. However, they are also still confined by the ground truth knowledge of an expert for training the model, and thus do not rate the quality of source code according to the diverse needs of the developers. As a result, several academic and commercial approaches [12, 13] resort to providing detailed reports of quality metrics, leaving configurations to developers or experts. These systems also do not offer a single output measure that covers the characteristics of user-perceived quality.

In this work, we build a generic source code quality estimation system able to provide a single quality rating based on a wide set of static analysis metrics that correspond to all four quality characteristics related to user-perceived quality, as defined in Section I. Our system does not use predefined metric thresholds, while it also refrains from expert representations of quality, since these definitions are not generic enough to cover the user perception of quality. Instead, we use popularity as the ground truth of quality. Given that popular components have a high rate of reuse [18], by intuition popularity can be a measure of high quality. Thus, we employ this concept in order to construct a metrics-based quality system that provides a user-perceived quality score, while at the same time providing measurable information for the values of metrics.

III. Correlation of Static Analysis Metrics and Popularity

In this section, we discuss the potential connection between quality and popularity and form a dataset that shall be used to create our quality model (more on the constructed model in Section IV). The dataset consists of the values of 73 static analysis metrics computed for the Java files of the 100 most popular Java repositories of GitHub (24.930 files in total), where popularity is determined by the number of stargazers.

A. Popularity and Quality of Source Code

As already noted, the formulation of our target set relies on the number of stars for every repository, which reflects its popularity. To support this claim, we computed the correlation between the number of stars and the number of forks in order to examine how popularity is correlated with reuse. As illustrated in Figure 1, there is a strong positive correlation between the two metrics. In specific, the value of the correlation coefficient is 0.68.

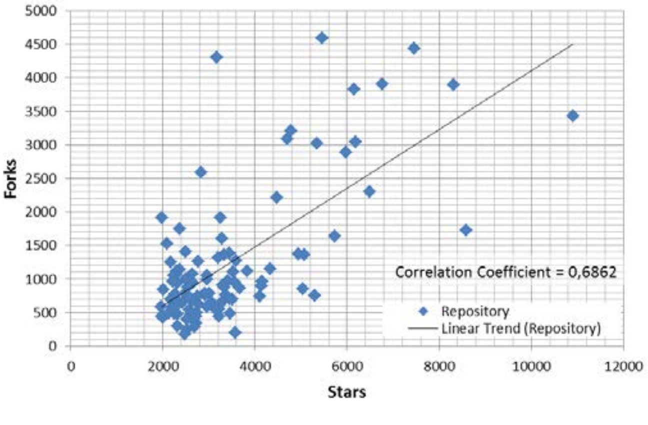


Fig. 1. Stars – Forks Diagram

In an attempt to further validate our intuition that popular repositories offer high quality (and thus reusable) code, we performed various checks related to good coding practices using the PMD tool [22]. Table I highlights the percentage of files found to have violations regarding good coding practices (rules) that belong to seventeen different categories (rulesets).

TABLE I. PMD ANALYSIS RESULTS

PMD Ruleset	Percentage (%) of code files containing violations	
	Violations Priority 1	*Violations Priority 2*
Unused Code	0.0%	0.0%
Basic	0.015%	0.337%
Braces	0.0%	0.0%
Comments	0.0%	0.0%
Naming	14.11%	0.45%
Clone	0.0%	0.16%
Code Size	0.0%	0.0%
Controversial	1.75%	1.58%
Coupling	0.0%	0.0%
Design	3.37%	3.9%
Empty	0.0%	0.0%
Finalizers	0.0%	0.0%
Optimizations	0.0%	0.0%
Strict Exception	4.99%	0.0%
Strings	0.0%	0.06%
Unnecessary	0.0%	0.0%

Those rulesets cover a wide range of coding standards such as naming conventions, dead code analysis, optimizations etc. (the full reference of the rules can be found in [22]). Rules are categorized into five priority levels (Priority 1, 2, 3, 4, and 5) where priority level 1 indicates the most severe violations, while level 5 refers to minor issues. Back to our analysis, it is obvious that the great majority of the analyzed files contain only few violations regarding priority 1 and priority 2 rules. The fact that the naming ruleset appears to have higher values (still below 15%) is anticipated, since it refers to guidelines which are highly project specific. Consequently, based on PMD results, we can safely conclude that our dataset consists of source code files that exceed a certain quality threshold.

B. Metrics

In an effort to build a generic quality estimation system that covers the four software quality characteristics related with user-perceived quality, we resort in computing a large set of static analysis metrics. This exhaustive analysis is the state of the practice approach followed by most popular static analysis tools. The computation was performed using Google Codepro Analytix [21]. Table II summarizes the computed metrics and their association with the four software quality characteristics. Further investigating the selected set of static analysis metrics, we conclude that they successfully cover various source code properties such as *size* (Lines of Code, Number of Methods etc.), *coupling* (Instability, Afferent and Efferent Couplings etc.), *complexity* (Average Cyclomatic Complexity etc.) and *comprehensibility* (Comments Ratio, Number of Comments etc.). Those high-level properties are highly related to the four software quality characteristics at hand.

C. Scoring source code files

Concerning the target of our dataset, we built a metric based on the number of stars for each repository. Note, however, that the number of stars for every repository is not enough as our analysis requires a score per source code file to reflect its quality individually. Furthermore, the star score of a repository cannot be distributed equally among the source code files, since each source code file of a repository does not have the same significance in terms of functionality and thus the same impact on the reuse rate of the repository. As a result, the score for each source code file must include the star rating of the repository, and account also for its significance as an independent entity. In an effort to achieve this as objectively as possible, we perform dependency analysis in every repository in order to identify the significance of each source code file. Consequently, the score for a source code file is given by the following equation:

$$F_{score}(i,j) = log\left(\frac{R_{stars}(j)}{n_{files}(j)} + \frac{dep(i)}{n_{files}(j)} \cdot R_{stars}(j)\right) \quad (1)$$

where $F_{score}(i,j)$ represents the target (score) for source code file i which is included in repository j, $R_{stars}(j)$ represents the number of stars of repository j, $dep(i)$ is the number of files included in the repository that depend on file i and $n_{files}(j)$ is the number of source code files included in repository j.

Equation (1) consists of two terms. The first term, $R_{stars}(j)/n_{files}(j)$, assigns a base score to all files in the same repository based on the star rating. The second term, $\left(dep(i)/n_{files}(j)\right) \cdot R_{stars}(j)$, incorporates the significance of the source code file inside the repository by adding value to the file proportional to the percentage of the source code files included in the repository that depend on that file. Lastly, the logarithm functions as a smoothing factor of the diversity in the number of source code files among different repositories. This smoothing factor is crucial, since in the top Java projects of GitHub there are repositories with more than 5000 files and others with less than 100 files. This diversity does not reflect the true quality difference among the repositories.

In addition, based on equation (1), which involves the score for each file in a repository, we are able to estimate the number of stars for the repository. The estimation is given by the following set of equations:

$$R_{stars}(i,j) = \frac{e^{F_{score}(i,j)} \cdot n_{files}(j)}{1 + dep(i)} \quad (2)$$

$$R_{stars}(j) = \sum_{i=1}^{n_{files}(j)} \frac{dep(i)}{dep_{total}(j)} \cdot R_{stars}(i,j) \quad (3)$$

In equation (2), $R_{stars}(i,j)$ represents the estimated number of stars for repository j based on the quality score of file i, $F_{score}(i,j)$ represents the quality score of file i of repository j, $n_{files}(j)$ represents the number of files of repository j and $dep(i)$ represents the number of files inside the repository that depend on file i. In equation (3), $R_{stars}(j)$ represents the final estimation for the number of stars of the repository j and $dep_{total}(j)$ represents the total number of dependencies calculated for all files that belong to repository j.

The above set of equations gives the estimated number of stars for a repository using the quality scores computed for each one of its source code files. At first, the quality score of every source code file that belongs to the repository can be used to estimate its individual star rating (see equation (2)). Consequently, there are different estimations depending on the

TABLE II. OVERVIEW OF STATIC ANALYSIS METRICS AND ASSOSIATION WITH QUALITY CHARACTERISTICS

Static Analysis Metrics		**Software Quality Characteristic**			
Name	***Description***	**Functional Suitability**	**Usability**	**Maintainability**	**Portability**
Abstractness	The percentage of abstract classes		X	X	
Average Block Depth	The average number of nested blocks		X	X	
Average Cyclomatic Complexity	Average value of McCabe's complexity [20]		X	X	
Average Depth of Inheritance Hierarchy	Average depth of the types defined in the source file		X	X	
Average Lines of Code Per Method	Average lines of code per method			X	
Average Number of Constructors Per Type	Average number of constructors per type	X			
Average Number of Fields Per Type	Average number of fiels per type	X			
Average Number of Methods Per Type	The average number of methods	X			
Average Number of Subtypes	The average number of subtypes		X		
Comments Ratio	Number of Comments / Lines of Code		X		
Distance	Abstractness + Instability - 1			X	X
Afferent Couplings (Ca)	Number of objects that depend on the source file			X	X
Efferent Couplings (Ce)	Number of objects that depend on other source files			X	X
Instability	Ce / (Ce + Ca)				X
Lines of Code	Lines of code			X	
Number of Characters	Number of characters			X	
Number of Comments	{end-of-line, multi-line, javadoc}		X		
Number of Constructors	{public, protected, package, private}	X			
Number of Fields	{instance, static}, {public, protected, package, private}	X			
Number of Lines	Number of lines with comments included			X	
Number of Methods	{instance, static}, {public, protected, package, private}			X	
Number of Packages	{compilation units, class files}			X	
Number of Semicolons	Number of semicolons			X	

number of files included in the repository. According to the initial hypothesis, every file within a repository has a different significance which is evaluated based on the number of files that depend on it. Thus, the different estimations are handled accordingly for the computation of the final estimation which is the weighted average of all estimations. The weights are computed using the files dependencies (see equation (3)).

IV. ESTIMATING CODE QUALITY

This section presents the analysis of the selected dataset and illustrates our source code quality estimation system.

A. *System Overview*

In this work, we employ the set of static analysis metrics (73 in total) described in subsection III-B computed for every source code file of the dataset so as to train two models: a one-class classifier that determines whether a source code file falls below a lowest quality threshold (thus it should be excluded from further analysis), and an artificial neural network which estimates a value for source code quality.

Figure 2 depicts the flow of our quality estimation system. For every source code file, our system performs static analysis to compute the metrics defined in subsection III-B, and checks for code practice violations of several rulesets (see Table I). A subset of those metrics (see subsection IV-B) is used by the one-class classifier which either accepts or rejects the file as input for the neural network. For the accepted source code files, the system employs the neural network to estimate a quality metric (a value that belongs in the interval [0, 1]), while the rejected ones are prompt for retest after structural revision.

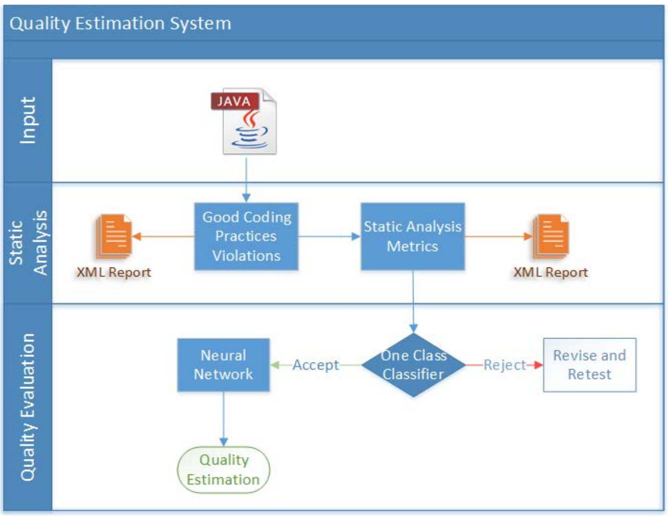
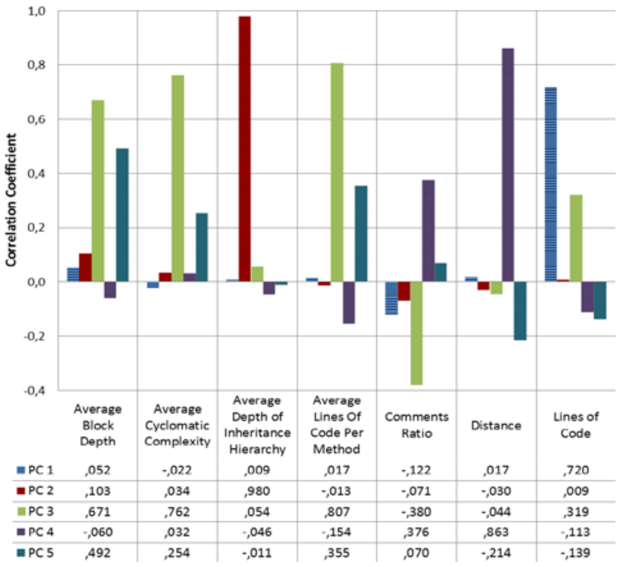


Fig. 2. Overview of the Quality Estimation System

B. Support Vector Machines One-Class Classifier

The one-class classifier is used to remove any low quality files because our model (see subsection IV-C) is applicable on the area of high quality source code. In an effort to train a classifier able to distinguish the source code files of acceptable quality, we had to carefully select the attributes of our training procedure. We performed principal component analysis to select the attributes with the maximum effect on the decision.



	Average Block Depth	Average Cyclomatic Complexity	Average Depth of Inheritance Hierarchy	Average Lines Of Code Per Method	Comments Ratio	Distance	Lines of Code
PC 1	,052	-,022	,009	,017	-,122	,017	,720
PC 2	,103	,034	,980	-,013	-,071	-,030	,009
PC 3	,671	,762	,054	,807	-,380	-,044	,319
PC 4	-,060	,032	-,046	-,154	,376	,863	-,113
PC 5	,492	,254	-,011	,355	,070	-,214	-,139

Fig. 3. Static Analysis metrics – Top 5 Components correlation diagram

Twenty of the principal components include the 86.57% of the information of the 73 static analysis metrics. Thus, upon setting this threshold, we examined the correlation of every metric with the principal components in order to select the most significant metrics. The results (see Figure 3) showed that the metrics with the greatest contribution in the formation of the principal components are Average Block Depth, Average Cyclomatic Complexity, Average Depth of Inheritance Hierarchy, Average Line of Codes Per Method, Comments Ratio, Distance, and Lines Of Code. Thus, these metrics were used in the training procedure of the one-class classifier.

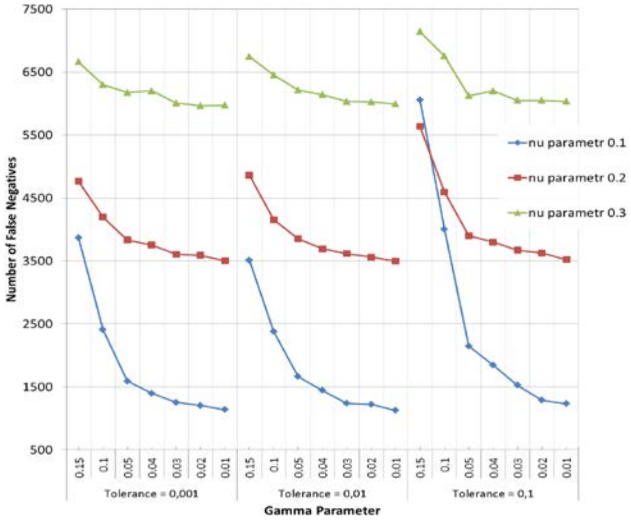


Fig. 4. One-Class Classifier Training Parameters Selection Diagram

The Gaussian radial basis kernel function was used for the one-class classifier. Figure 4 illustrates the selection procedure for the parameters (nu, tolerance and gamma) of the classifier. The selection nu parameter, which reflects the upper threshold of the outliers and the lower threshold of the support vectors, is crucial for the number of false-negatives. On the other hand, tolerance seems to have little impact, while the impact of gamma parameter increases once the value of nu parameter decreases. The optimal combination, i.e. the one found to have minimum false-positives (1124 in total), is the one with values 0.1, 0.01, and 0.01 for nu, gamma, and tolerance respectively.

C. Artificial Neural Networks Model

Our ANN is a two-layer feedforward network with sigmoid hidden input and output neurons. The Levenberg-Marquardt algorithm (LMA) [23] is used as the training function for adjusting the weights and the biases. The input consists of the values of the 73 static analysis metrics, while the output is the source code quality score. The training procedure involves the metrics values of 23.806 source code files that exhibited quality values higher than the threshold determined by the one-class classifier (see subsection IV-B). The desired output (target) is a quality score (see subsection III-C).

Seventy (70%) of the data samples were used for training, while the remaining samples were equally split into sets for validation (15%) and testing (15%). The mean square error (mse) was used as the termination criterion, over the 64 epochs. By further examining the train, validation and test curve, the mean square error is gradually decreasing and has similar decrease rate in all three, thus no overfitting occurs.

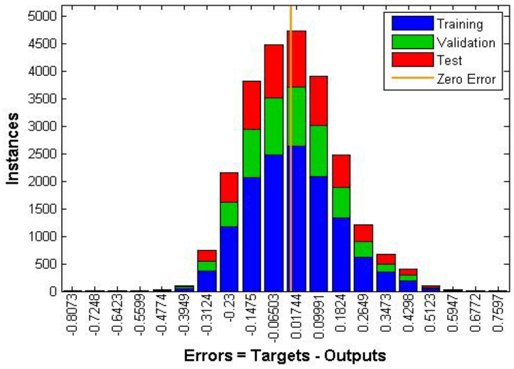
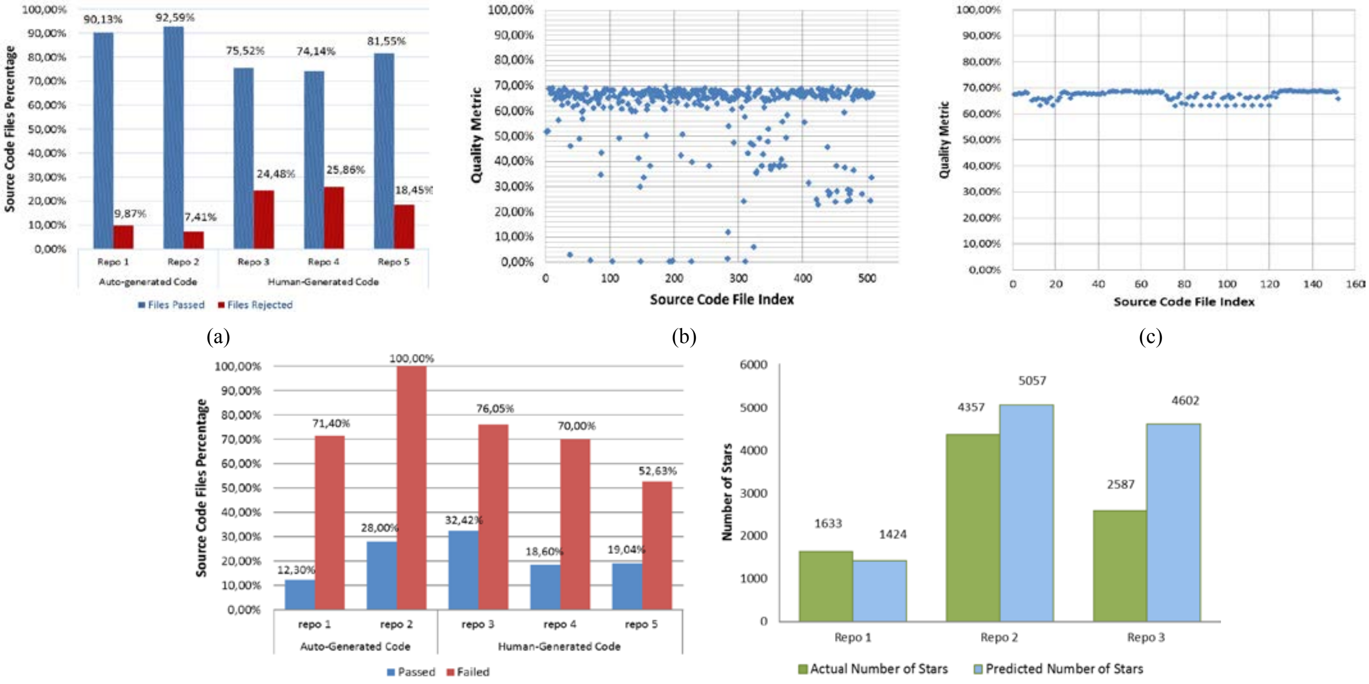


Fig. 5. Error Histogram of the Neural Network

Figure 5 presents the error histogram of the ANNs model. The results show that the trained model is able to successfully estimate the desired output with mean error of about 14%. Further examining the histogram, 54.3% of the outputs exhibit error below 10%. In addition, the error is evenly distributed among the training, validation and test sets, indicating that the model is not overfitted. Most error bins have values closer to 0, while only about 10% of the output has high error values (above 30%). Inspecting the repositories containing source code files with high error values, we concluded that the error originates from the special characteristics of each repository. For example, a very popular repository containing quite a few source code files having no dependencies (e.g. a graphics library containing the implementation of different animations) tends to be overestimated.



(a) (b) (c)

(d) (e)

Fig. 6. Evaluation results of our source code quality estimation model. Diagrams (a) and (b) illustrate the quality score for every source code file included in human-generated and auto-generated projects respectively. Diagram (c) presents the percentages of the accepted and the rejected files by the one-class classifier for the repositories analyzed. Diagram (d) illustrates the percentages of the accepted and the rejected source code files that contained severe PMD violations, i.e. violations of rules with priorities 1 and 2. Finally, diagram (f) illustrates the predicted number of stars for each repository in comparison with the actual stars.

V. Evaluation

This section discusses the evaluation of our system on a set of diverse projects. We employ two practices for validating our results. At first, we check all files quantitatively using PMD to determine whether the models are effective enough for scoring (or approving) files with different violations. Secondly, we manually examine a representative sample of files and their metrics to qualitatively assess the effectiveness of the models.

A. Experimental Setup

Our evaluation targets on two axes: the system's ability to distinguish and estimate the quality of source code files, and the accuracy of predicting the popularity of Java repositories given their files. We analyzed 5 java projects, out of which 3 were retrieved from GitHub and 2 were auto-generated using the tools of S-CASE[1]. The auto-generated projects are the RESTful services Webmarks and WSAT. These services were selected to assess whether our system provides correct estimates for high quality code, given that they follow all code conventions and are architecturally and functionally complete. As for the evaluation of the system's capability to predict the popularity of a repository, 3 random GitHub repositories older than 6 months were analyzed and the results were compared to the actual number of stars. Table III presents the repositories used for assessing the effectiveness of our system. The last three repositories were used in the popularity prediction procedure. The total number of files checked during the evaluation process was 829 containing 101.257 lines of code.

TABLE III. Repositories used for the System Evaluation

Project/Repository Name	Project Information		
	Type	*Number of Files*	*LOC*
core	Human generated	103	7.377
dempsy	Human generated	116	15.004
android-mobile	Human generated	290	49.050
Webmarks[2]	Auto generated	27	1.452
WSAT[2]	Auto generated	127	8.322
Android View Animations	Human generated	69	1.426
emojicon	Human generated	19	3.386
Jsoup	Human generated	78	15.240

B. Results

Figure 6 (a) illustrates the results of the one-class classifier. The training procedure included the challenge of building a model neither too strict nor too lenient. The results showed that the percentage of the rejected files varies from 7.41% to 25.86% which indicates that the behavior of the model is balanced between rejecting all files that slightly differ from the ones contained in the dataset and accepting files with entirely different quality characteristics. The above conclusion is also derived by examining the metrics values of the rejected files, the vast majority of which contained metrics with extreme values. Concerning the difference between auto-generated and human-generated code, as expected, the percentage of the rejected human-generated files is clearly higher.

[1] http://www.scasefp7.eu/

[2] http://s-case.github.io/

TABLE IV. REVIEW OF STATIC ANALYSIS METRICS OF SOURCE CODE FILES WITH DIFFERENT QUALITY EVALUATION

Static Analysis Metrics		***Source Code File Failed One-Class Classifier***		***File with quality estimation 0.115 (11.5%)***		***File with quality estimation 0.42 (42%)***		***File with quality estimation 0.712 (71.2%)***	
Abstractness		0%		100%		100%		0.0%	
Afferent Couplings		0		0		0		0	
Average Block Depth	minimum	0.86	0.0	0.0	0.0	1.11	0.0	0.95	0.0
	maximum		3.0		0.0		2.0		2.0
Average Cyclomatic Complexity	minimum	**6.81**	1.0	1.0	1.0	1.37	1.0	1.13	1.0
	maximum		**24.0**		1.0		2.0		3.0
Average Depth Of Inheritance Hierarchy	minimum	2.0	2.0	1.0	1.0	1.0	1.0	1.14	1.0
	maximum		2.0		1.0		1.0		2.0
Average Lines Of Code Per Method	minimum	22.09	13	1.0	1.0	4.5	1	7.86	1
	maximum		31		1.0		10		20
Average Number of Constructors Per Type	minimum	1	1	0.0	0	1.0	1	0	0
	maximum		1		0		1		0
Average Number of Fields Per Type	minimum	18	18	0.0	0	1.0	1	0.42	0
	maximum		18		0		1		3
Average Number of Methods Per Type	minimum	10	10	2.0	2	7.0	7	3.14	1
	maximum		10		2		7		16
Comments Ratio		16.1%		**60%**		2.4%		2.2%	
Distance		0.0		0.0		1.0		0.0	
Efferent Couplings		1		0		1		1	
Instability		1.00		0.00		1.00		1.00	
Lines Of Code		285		**5**		41		178	
Number Of Methods		10		2		7		22	
Number Of Lines		431		22		59		197	
Number of Comments	end of line	46	35	3	0	1	0	4	3
	multi line		11		0		0		1
	javadoc		0		3		1		0

Figures 6 (b) and 6 (c) illustrate the score that has been computed by the ANNs model for human-generated and auto-generated code respectively. Comparing the two figures, it is obvious that the variance of the quality metric is much higher in the case of human-generated code than in the auto-generated code. This is not surprising especially in cases of repositories with many contributors (as the ones examined), that may have diverse coding styles and possibly even different levels of expertise. On the other hand, files generated from the same tools share similar quality characteristics, which are reflected in the values of the static analysis metrics and thus are expected to have similar quality scores.

We next validated the results of the one-class-classifier using PMD. Figure 6 (d) illustrates the percentage of files that contain severe violations (violations of PMD rules with priority 1 and 2), among the ones accepted and rejected by the one-class-classifier. It is obvious that in all five repositories, the percentage of the source code files containing severe violations is much higher among the rejected than in the accepted ones.

Further assessing the validity of the results we manually examined the static analysis metrics of sample source code files in order to check whether they are aligned with the quality estimation. Table IV provides an overview of the static analysis metrics computed for four representative examples of source code files with different quality estimation. According to the one-class classifier, the first failed to be considered as part of the high quality source codes while the other three passed. Each of these three files has received a different quality score; the first got 11.5% (low rating), the second got 42% (medium rating) and the third got 71.2% (high rating). Examining the static analysis metrics, it is obvious that the rejected file appears to have methods of very high complexity (24), which contradicts the complexity of the dataset (94% of the dataset files have no method of complexity greater than 10). Similarly, the file that received low quality estimation seems to contain very little valuable information. Although the comments ratio metric value (60%) appears to be satisfactory, its combination with the actual lines of code metric, which is only 5, reflects this absence. On the other hand, the values of the static analysis metrics of the file that received high quality estimation seem to follow the ones observed in the dataset. Lastly, there are also cases where reviewing the static analysis metrics in order to explain the quality estimation value is very difficult. This is more or less expected since the quality estimation is a result of the combination of all metrics and their correlations. A representative example is the file rated with 42% quality rating.

Upon evaluating our system, we also assess its ability to predict the popularity of source code repositories. Figure 6 (e) illustrates the predicted and the actual number of stars for three random repositories. The predicted popularity value lies quite near the actual values for repositories 1 and 2. For repository 3, the results are also encouraging, given that it contains a large number of independent source code files. Several of those files have high quality scores, thus the predicted popularity of the repository is driven to a slightly higher value.

To sum up, having evaluated our system for the validity of its results, we ended up having a system able to correspond to two goals: the first is the reliable determination of the area of high quality source code based on the values of static analysis metrics and the second is the formation of a generic user-perceived quality estimation model that incorporates elements of the four quality characteristics related to reusability.

VI. Threats to Validity

One may argue that validity issues may arise regarding the selection of projects used in training and evaluation; however, specific actions were taken to minimize these threats. Concerning training, the dataset used comprises the top 100 GitHub repositories with respect to their number of stars, providing a basic criterion on popularity. Apart from that, static analysis using PMD was performed to ensure code quality. Concerning evaluation, we selected 8 projects independently in order to avoid any bias on the produced results. Three typical GitHub projects were used to assess the ability of our system to predict quality scores. Each of them has roughly 200-300 files with a lines-of-code-per-file ratio around 100, including also several extreme cases. They align with the size range of the vast majority of GitHub projects and can, thus, be considered representative. The three projects used for assessing predicted popularity also comprise a representative sample of mature libraries with solid user-base. Finally, the auto-generated projects illustrate the capability of our mechanism to generalize, since these projects follow the main conventions for RESTful web services and are built based on standard quality rules. This way we tried to ensure a fair assessment approach.

VII. Conclusion And Future Work

In this work we proposed a different source code quality estimation approach according to which user-perceived quality is employed as a measure of the reusability of a software component using code popularity as ground truth information. As derived from the evaluation on the main features of our system, both qualitatively and quantitatively, we conclude that it can be a valuable tool for estimating source code quality as perceived by developers.

Future work on our system lies in several directions. At first, we could further investigate the response of our model in different scenarios by expanding the dataset with more high quality repositories or more static analysis metrics and coding rules. In addition, another approach would be to expand the ground truth coverage by using more metrics (other than GitHub stars) targeting additional characteristics such as reusability. Finally, given that there are overlaps between metrics, we could apply feature selection techniques in order to drop overlapping metrics and provide a simpler but of the same accuracy user-perceived quality prediction model.

VIII. Acknowledgment

Parts of this work have been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission.

References

[1] B. Kitchenham and S. L. Pfleeger, "Software quality: The elusive target", IEEE Software, vol. 13, no. 1, pp. 12–21, January 1996.

[2] "CISQ Code Quality Standarts". [Retrieved June, 2016]. [Online]. Available: http://it-cisq.org/standards/

[3] T. Diamantopoulos, K. Thomopoulos, and A. Symeonidis. "QualBoa: reusability-aware recommendations of source code components.", in proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). ACM, pp. 488-491, 2016.

[4] Taibi, F. "Empirical Analysis of the Reusability of Object-Oriented Program Code in Open-Source Software," International Journal of Computer, Information, System and Control Engineering, vol. 8, no. 1, pp. 114 – 120, 2014.

[5] 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, June 1994.

[6] 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 ICSE, pp. 181–190, 2008.

[7] N. Nagappan , A. Zeller , T. Zimmermann , K. Herzig and B. Murphy, "Change Bursts as Defect Predictors", in proceedings of the IEEE 21st ISSRE, pp. 309-318, 2010.

[8] C. Le Goues and W. Weimer, "Measuring code quality to improve specification mining", in IEEE Transactions on Software Engineering (TSE), pp. 175 –190, 2012.

[9] H. Washizaki, R. Namiki, T. Fukuoka, Y. Harada, H. Watanabe, "A Framework for Measuring and Evaluating Program Source Code Quality", in proceedings of PROFES, vol. 4589, pp. 284-299, 2007.

[10] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, "Unsupervised Learning for Expert-Based Software Quality Estimation", in HASE, pp. 149-155, March 2004.

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

[12] T. Cai, M. Lyu, K.-F. Wong, and M. Wong, "ComPARE: A generic quality assessment environment for component-based software systems", in proceedings of the 2001 International Symposium on Information Systems and Engineering, 2001.

[13] "SonarQube platform". [Retrieved June, 2016]. [Online]. Available: http://www.sonarqube.org/

[14] R. Baggen, J. P. Correia , K. Schill and J. Visser, "Standardized code quality benchmarking for improving software maintainability", in Software Quality Journal, vol. 20, Issue 2 , pp. 287-307, June 2012.

[15] P. Hegedus, T. Bakota, G. Ladanyi, C. Farago and R. Ferenc, "A Drill-Down Approach for Measuring Maintainability at Source Code Element Level", in proceedings of the Seventh International Workshop on Software Quality and Maintainability, vol. 60, 2013.

[16] I. Samoladas, G. Gousios, D. Spinellis and I. Stamelos, "The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation", in proceedings of the series IFIP, vol. 275, pp. 237-248, 2008.

[17] N. Goyal and Er. D. Gupta, "Reusability Calculation of Object Oriented Software Model by Analyzing CK Metric", in IJARCET, vol. 3, Issue 7, July 2014.

[18] H. Borges, M. T. Valente, A. Hora and J. Coelho, "On the Popularity of GitHub Applications:A Preliminary Note".

[19] A. Kumar, "Measuring Software Reusability Using Svm Based Classifier Approach", in IJITM, vol. 5, no. 1, pp. 205-209, 2012.

[20] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design", in IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, 1994.

[21] Google Codepro Analytix". [Retrieved March, 2015]. [Online]. Available: https://developers.google.com/java-dev-tools/codepro/doc/

[22] "PMD". [Retrieved June, 2016]. [Online] Available: http://pmd.github.io/pmd-5.3.2/

[23] D. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters", in SIAM Journal on Applied Mathematics, pp. 431–441, 1963.