Assessing the User-Perceived Quality of Source Code Components using Static Analysis Metrics

Valasia Dimaridou, Alexandros-Charalampos Kyprianidis, Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis

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

Abstract. Nowadays, developers tend to adopt a component-based software engineering approach, reusing own implementations and/or resorting to third-party source code. This practice is in principle cost-effective, however it may also lead to low quality software products, if the components to be reused exhibit low quality. Thus, several approaches have been developed to measure the quality of software components. Most of them, however, rely on the aid of experts for defining target quality scores and deriving metric thresholds, leading to results that are context-dependent and subjective. In this work, we build a mechanism that employs static analysis metrics extracted from GitHub projects and defines a target quality score based on repositories' stars and forks, which indicate their adoption/acceptance by developers. Upon removing outliers with a one-class classifier, we employ Principal Feature Analysis and examine the semantics among metrics to provide an analysis on five axes for source code components (classes or packages): complexity, coupling, size, degree of inheritance, and quality of documentation. Neural networks are thus applied to estimate the final quality score given metrics from these axes. Preliminary evaluation indicates that our approach effectively estimates software quality at both class and package levels.

Keywords: Code Quality, Static Analysis Metrics, User-Perceived Quality, Principal Feature Analysis

1 Introduction

The continuously increasing need for software applications in practically every domain, and the introduction of online open-source repositories have led to the establishment of an agile, component-based software engineering paradigm. The need for reusing existing (own or third-party) source code, either in the form of software libraries or simply by applying copy-paste-integrate practices has become more eminent than ever, since it can greatly reduce the time and cost of software development [19]. In this context, developers often need to spend considerable time and effort to integrate components and ensure high performance. And still, this may lead to failures, since the reused code may not satisfy

basic functional or non-functional requirements. Thus, the quality assessment of reusable components poses a major challenge for the research community.

An important aspect of this challenge is the fact that quality is context-dependent and may mean different things to different people [17]. Hence, a standardized approach for measuring quality has been proposed in the latest ISO/IEC 25010:2011 [10], which defines a model with eight quality characteristics: Functional Suitability, Usability, Maintainability, Portability, Reliability, Performance and Efficiency, Security and Compatibility, out of which the first four are usually assessed using static analysis and evaluated intuitively by developers. To accommodate reuse, developers usually structure their source code (or assess third-party code) so that it is modular, exhibits loose coupling and high cohesion, and provides information hiding and separation of concerns [16].

Current research efforts assess the quality of software components using static analysis metrics [4, 22, 12, 23], such as the known CK metrics [3]. Although these efforts can be effective for the assessment of a quality characteristic (e.g. [re]usability, maintainability or security), 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. Moreover, the approaches that are based on metric thresholds, whether defined manually [4, 12, 23] or derived automatically using a model [24], are usually constrained by the lack of objective ground truth values for software quality. As a result, these approaches typically resort to expert help, which may be subjective, case-specific or even unavailable [2]. An interesting alternative is proposed by Papamichail et al. [15] that employ user-perceived quality as a measure of the quality of a software component.

In this work, we employ the concepts defined in [15] and build upon the work originated from [5], which performs analysis only at class level, in order to build a mechanism that associates the extent to which a software component (class or package) is adopted/preferred by developers. We define a ground truth score for the user-perceived quality of components based on popularity-related information extracted from their GitHub repos, in the form of stars and forks. Then, at each level, we employ a one-class classifier and build a model based on static analysis metrics extracted from a set of popular GitHub projects. By using Principal Feature Analysis and examining the semantics among metrics, we provide the developer with not only a quality score, but also a comprehensive analysis on five axes for the source code of a component, including scores on its complexity, coupling, size, degree of inheritance, and the quality of its documentation. Finally, for each level, we construct five Neural Networks models, one for each of these code properties, and aggregate their output to provide an overall quality scoring mechanism at class and package level, respectively.

The rest of this paper is organized as follows. Section 2 provides background information on static analysis metrics and reviews current approaches on quality estimation. Section 3 describes our benchmark dataset and designs a scoring mechanism for the quality of source code components. The constructed models are shown in Section 4, while Section 5 evaluates the performance of our system. Finally, Section 6 concludes this paper and provides insight for further research.

2 Related Work

According to [14], research on software quality is as old as software development. As software penetrates everyday life, assessing quality has become a major challenge. This is reflected in the various approaches proposed by current literature that aspire to assess quality in a quantified manner. Most of these approaches make use of static analysis metrics in order to train quality estimation models [18, 12]. Estimating quality through static analysis metrics is a non-trivial task, as it often requires determining quality thresholds [4], which is usually performed by experts who manually examine the source code [8]. However, the manual examination of source code, especially for large complex projects that change on a regular basis, is not always feasible due to constraints in time and resources. Moreover, expert help may be subjective and highly context-specific.

Other approaches may require multiple parameters for constructing quality evaluation models [2], which are again highly dependent on the scope of the source code and are easily affected by subjective judgment. Thus, a common practice involves deriving metric thresholds by applying machine learning techniques on a benchmark repository. Ferreira et al. [6] propose a methodology for estimating thresholds by fitting the values of metrics into probability distributions, while [1] follow a weight-based approach to derive thresholds by applying statistical analysis on the metrics values. Other approaches involve deriving thresholds using bootstrapping [7] and ROC curve analysis [20]. Still, these approaches are subject to the projects selected for the benchmark repository.

An interesting approach that refrains from the need to use certain metrics thresholds and proposes a fully automated quality evaluation methodology is that of Papamichail et al. [15]. The authors design a system that reflects the extent to which a software component is of high quality as perceived by developers. The proposed system makes use of crowdsourcing information (the popularity of software projects) and a large set of static analysis metrics, in order to provide a single quality score, which is computed using two models: a one-class-classifier used to identify high quality code and a neural network that translates the values of the static analysis metrics into quantified quality estimations.

Although the aforementionedy approaches can be effective for certain cases, their applicability in real-world scenarios is limited. The use of predefined thresholds [4, 8] results in the creation of models unable to cover the versatility of today's software, and thus applies only to restricted scenarios. On the other hand, systems that overcome threshold issues by proposing automated quality evaluation methodologies [15] often involve preprocessing steps (such as feature extraction) or regression models that lead to a quality score which is not interpretable. As a result, the developer is provided with no specific information on the targeted changes to apply in order to improve source code quality.

Extending previous work [5], we have built a generic source code quality estimation mechanism able to provide a quality score at both class and package levels, which reflects the extent to which a component could/should be adopted by developers. Our system refrains from expert-based knowledge and employs a large set of static analysis metrics and crowdsourcing information from GitHub

stars and forks in order to train five quality estimation models for each level, each one targeting a different property of source code. The individual scores are then combined to produce a final quality score that is fully interpretable and provides necessary information towards the axes that require improvement. By further analyzing the correlation and the semantics of the metrics for each axis, we are able to identify similar behaviors and thus select the ones that accumulate the most valuable information, while at the same time describing the characteristics of the source code component under examination.

3 Defining Quality

In this section, we quantify quality as perceived by developers using information from GitHub stars and forks as ground truth. In addition, our analysis describes how the different categories of source code metrics are related to major quality characteristics as defined in ISO/IEC 25010:2011 [10].

3.1 Benchmark Dataset

Our dataset consists of a large set of static analysis metrics calculated for 102 repositories, selected from the 100 most starred and the 100 most forked GitHub Java projects. The projects were sorted in descending order of stars and subsequently forks, and were selected to cover more than 100,000 classes and 7,300 projects. Certain statistics of the benchmark dataset are shown in Table 1.

Table 1. Dataset Statistics [5]

Statistics	**Dataset**
Total Number of Projects	102
Total Number of Packages	7,372
Total Number of Classes	100,233
Total Number of Methods	584,856
Total Lines of Code	7,985,385

We compute a large set of static analysis metrics that cover the source code properties of complexity, coupling, documentation, inheritance, and size. Current literature [11, 9] indicates that these properties are directly related to the characteristics of Functional Suitability, Usability, Maintainability, and Portability, as defined by ISO/IEC 25010:2011 [10]. The metrics that were computed using SourceMeter [21] are shown in Table 2. In our previous work [5], the metrics were computed at class level, except for McCC that was computed at method level and then averaged to obtain a value for the class. For this extended work the metrics were computed at a package level, except for the metrics that are available only at class level. These metrics were initially calculated at class level and the median of each one was enumerated to obtain values for the packages.

Table 2. Overview of Static Metrics and their Applicability on Different Levels

Static Analysis Metrics			Compute Levels	
Type	Name	Description	Class	Package
Comp-lexity	NL	Nesting Level	×	
	NLE	Nesting Level Else-If	×	
	WMC	Weighted Methods per Class	×	
Coupling	CBO	Coupling Between Object classes	×	
	CBOI	CBO Inverse	×	
	NII	Number of Incoming Invocations	×	
	NOI	Number of Outgoing Invocations	×	
	RFC	Response set For Class	×	
Cohesion	LCOM5	Lack of Cohesion in Methods 5	×	
Documentation	AD	API Documentation	×	
	CD	Comment Density	×	×
	CLOC	Comment Lines of Code	×	×
	DLOC	Documentation Lines of Code	×	
	PDA	Public Documented API	×	×
	PUA	Public Undocumented API	×	×
	TAD	Total API Documentation		×
	TCD	Total Comment Density	×	×
	TCLOC	Total Comment Lines of Code	×	×
	TPDA	Total Public Documented API		×
	TPUA	Total Public Undocumented API		×
Inheritance	DIT	Depth of Inheritance Tree	×	
	NOA	Number of Ancestors	×	
	NOC	Number of Children	×	
	NOD	Number of Descendants	×	
	NOP	Number of Parents	×	
Size	{L}LOC	{Logical} Lines of Code	×	×
	N{A,G,M,S}	Number of {Attributes, Getters, Methods, Setters}	×	×
	N{CL,EN,IN,P}	Number of {Classes, Enums, Interfaces, Packages}		×
	NL{A,G,M,S}	Number of Local {Attributes, Getters, Methods, Setters}	×	
	NLP{A,M}	Number of Local Public {Attributes, Methods}	×	
	NP{A,M}	Number of Public {Attributes, Methods}	×	×
	NOS	Number of Statements	×	
	T{L}LOC	Total {Logical} Lines of Code	×	×
	TNP{CL,EN,IN}	Total Number of Public {Classes, Enums, Interfaces}		×
	TN{CL,DI,EN,FI}	Total Number of {Classes, Directories, Enums, Files}		×

3.2 Quality Score Formulation

As already mentioned, we use GitHub stars and forks as ground truth information towards quantifying quality as perceived by developers. According to our initial hypothesis, the number of stars can be used as a measure of the popularity for a software project, while the number of forks as a measure of its reusability. We make use of this information in order to define our target variable and consequently build a quality scoring mechanism. Towards this direction, we aim to define a quality score for every class and every package included in the dataset.

Given, however, that the number of stars and forks refer to repository level, they are not directly suited for defining a score that reflects the quality of each class or package, individually. Obviously, equally splitting the quality score computed at repository level among all classes or packages is not optimal, as every component has a different significance in terms of functionality and thus must be rated as an independent entity. Consequently, in an effort to build a scoring mechanism that is as objective as possible, we propose a methodology that involves the values of static analysis metrics for modeling the significance of each source code component (class or package) included in a given repository.

The quality score for every software component (class or package) of the dataset is defined using the following equations:

$$S_{stars}(i,j) = (1 + NPM(j)) \cdot \frac{Stars(i)}{N_{components}(i)} \tag{1}$$

$$S_{forks}(i,j) = (1 + AD(j) + NM(j)) \cdot \frac{Forks(i)}{N_{components}(i)} \tag{2}$$

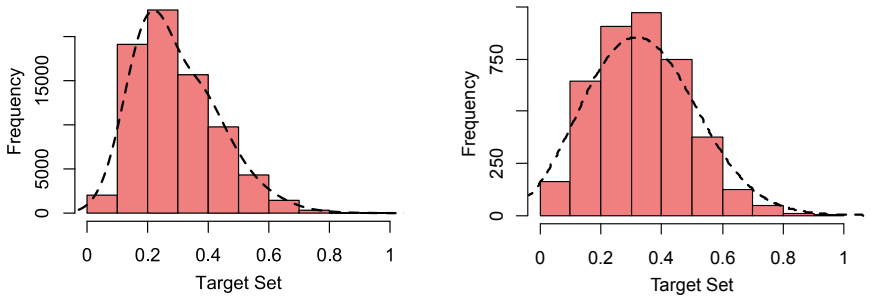
$$Q_{score}(i,j) = \log(S_{stars}(i,j) + S_{forks}(i,j)) \tag{3}$$

where $S_{stars}(i,j)$ and $S_{forks}(i,j)$ represent the quality scores for the j-th source code component (class or package) contained in the i-th repository, based on the number of GitHub stars and forks, respectively. $N_{components}(i)$ corresponds to the number of source code components (classes or packages) contained in the i-th repository, while $Stars(i)$ and $Forks(i)$ refer to the number of its GitHub stars and forks, respectively. Finally, $Q_{score}(i,j)$ is the overall quality score computed for the j-th source code component (class or package) contained in the i-th repository.

Our target set also involves the values of three metrics as a measure of the significance for every individual class or package contained in a given repository. Different significance implies different contribution to the number of GitHub stars and forks of the repository and thus different quality scores. $NPM(j)$ is used to measure the degree to which the j-th class (or package) contributes to the number of stars of the repository, as it refers to the number of methods and thus the different functionalities exposed by the class (or package). As for the contribution at the number of forks, we use $AD(j)$, which refers to the ratio of documented public methods, and $NM(j)$, which refers to the number of methods of the j-th class (or package), and therefore can be used as a measure of its functionalities. Note that the provided functionalities pose a stronger criterion

for determining the reusability score of a source code component compared to the documentation ratio, which contributes more as the number of methods approaches to zero. Lastly, as seen in equation (3), the logarithmic scale is applied as a smoothing factor for the diversity in the number of classes and packages among different repositories. This smoothing factor is crucial, since this diversity does not reflect the true quality difference among the repositories.

Figure 1 illustrates the distribution of the quality score (target set) for the benchmark dataset classes and packages. Figure 1a refers to classes, while Figure 1b refers to packages. The majority of instances for both distributions are accumulated in the interval [0.1, 0.5] and their frequency is decreasing as the score reaches 1. This is expected, since the distributions of the ratings (stars or forks) provided by developers typically exhibit few extreme values.



(a) (b)

Fig. 1. Distribution of the computed Quality Score at (a) Class and (b) Package level

4 System Design

In this section we design our system for quality estimation based on static analysis metrics. We split the dataset of the previous section into two sets, one for training and one for testing. The training set includes 90 repositories with 91531 classes distributed within 6632 packages and the test set includes 12 repositories with 8702 classes distributed within 738 packages. For the training, we used all available static analysis metrics except for those used for constructing the target variable. In specific, AD, NPM, NM, and NCL were used only for the preprocessing stage and then excluded from the models training to avoid skewing the results. In addition, any components with missing metric values are removed (e.g. empty class files or package files containing no classes); hence the updated training set contains 5599 packages with 88180 class files and the updated test set contains 556 packages with 7998 class files.

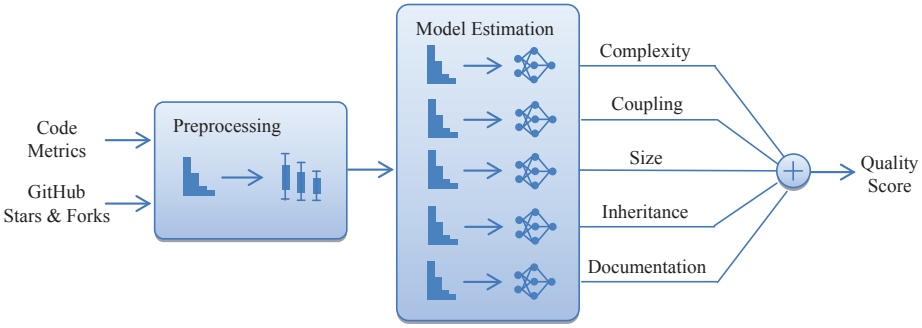


Fig. 2. Overview of the Quality Estimation Methodology [5]

4.1 System Overview

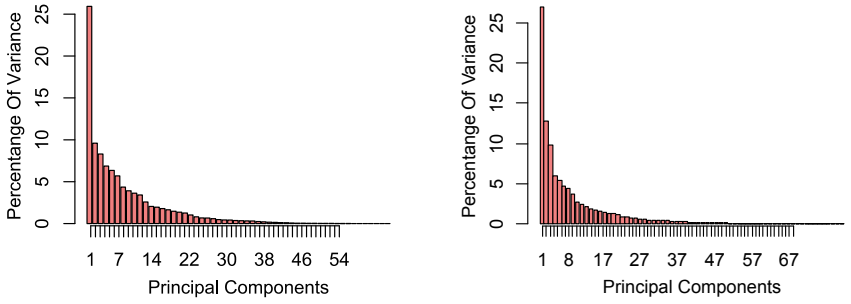
Our system is shown in Figure 3. The input is given in the form of static analysis metrics, while the stars and forks of the GitHub repositories are required only for the training of the system. As a result, the developer can provide a set of classes or packages (or a full project), and receive a comprehensible quality analysis as output. Our methodology involves three stages: the preprocessing stage, the metrics selection stage, and the model estimation stage. During preprocessing, the target set is constructed using the analysis of Section 3, and the dataset is cleaned of duplicates and outliers. Metrics selection determines which metrics will be used for each metric category, and model estimation involves training 5 models, one for each category. The stages are analyzed in the following paragraphs.

4.2 Data Preprocessing

The preprocessing stage is used to eliminate potential outliers from the dataset and thus make sure that the models are trained as effectively as possible. To do so, we developed a one-class classifier for each level (class/package) using *Support Vector Machines (SVM)* and trained it using metrics that were selected by means of *Principal Feature Analysis (PFA)*.

At first, the dataset is given as input in two PFA models which refer to classes and packages, respectively. Each model performs *Principal Component Analysis (PCA)* to extract the most informative *principal components (PCs)* from all metrics applicable at each level. In the case of classes, we have 54 metrics, while in the case of packages, we have 68. According to our methodology, we keep the first 12 principal components, preserving 82.8% of the information in the case of classes and 82.91% in the case of packages. Figure 3 depicts the percentage of variance for each principal component. Figure 3a refers to class level, while figure 3b refers to package level. We follow a methodology similar to that of [13] in order to select the features that shall be kept. The transformation matrix

generated by each PCA includes values for the participation of each metric in each principal component.



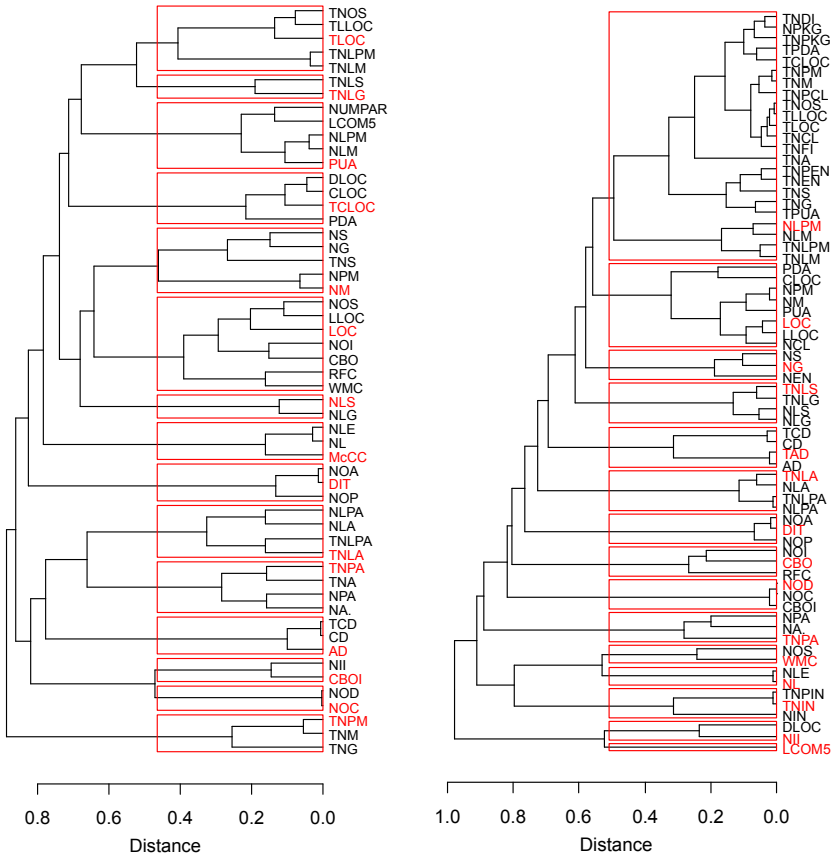
(a) (b)

Fig. 3. Variance of Principal Components at (a) Class and (b) Package level

We first cluster this matrix using hierarchical clustering and then select a metric from each cluster. Given that different metrics may have similar trends (e.g. McCabe Complexity with Lines of Code), complete linkage was selected to avoid large heterogeneous clusters. The dendrograms of the clustering for both classes and packages is shown in Figure 4. Figure 4a refers to classes, while Figure 4b refers to packages.

The dendrograms reveal interesting associations among the metrics. The clusters correspond to categories of metrics which are largely similar, such as the metrics of the local class attributes, which include their number (NLA), the number of the public ones (NLPA), and the respective totals (TNLPA and TNLA) that refer to all classes in the file. In both class and package levels, our clustering reveals that keeping one of these metrics results in minimum information loss. Thus, in this case we keep only TNLA. The selection of the kept metric from each cluster in both cases (in red in Figure 4) was performed by manual examination to end up with a metrics set that conforms to the current state-of-the-practice. An alternative would be to select the metric which is closest to a centroid computed as the Euclidean mean of the cluster metrics.

After having selected the most representative metrics for each case, the next step is to remove any outliers. Towards this direction, we use two SVM one-class classifiers for this task, each applicable at a different level. The classifiers use a radial basis function (RBF) kernel, with *gamma* and *nu* set to 0.01 and 0.1 respectively, and the training error tolerance is set to 0.01. Given that our dataset contains popular high quality source code, outliers in our case are actually low quality classes or packages. These are discarded since the models of Figure 2 are trained on high quality source code. As an indicative assessment of our classifier, we use the code violations data described in Section 3.



(a) (b)

Fig. 4. Dendrogram of Metrics Clustering at (a) Class and (b) Package level

In total, the one-class classifiers ruled out 8815 classes corresponding to 9.99% of the training set and 559 packages corresponding to 9.98% of the training set. We compare the mean number of violations for these rejected classes/packages and for the classes/packages that were accepted, for 8 categories of violations. The results, which are shown in Table 3, indicate that our classifier successfully rules out low quality source code, as the number of violations for both the rejected classes and packages is clearly higher than that of the accepted.

For instance, the classes rejected by the classifier are typically complex since they each have on average approximately one complexity violation; on the other hand, the number of complexity violations for the accepted classes is minimal. Furthermore, on average each rejected class has more than 8 size violations (e.g. large method bodies), whereas accepted classes have approximately 1.

Table 3. Mean Number of Violations of Accepted and Rejected Components

Violation Types	Mean Number of Violations			
	Classes		Packages	
	Accepted	Rejected	Accepted	Rejected
WarningInfo	18.5276	83.0935	376.3813	4106.3309
Clone	4.3106	20.9365	2.9785	10.7513
Cohesion	0.3225	0.7893	0.2980	0.6556
Complexity	0.0976	1.2456	0.0907	0.9320
Coupling	0.1767	1.5702	0.2350	1.2486
Documentation	12.5367	49.9751	13.9128	37.2039
Inheritance	0.0697	0.4696	0.0439	0.2280
Size	1.0134	8.1069	1.2812	5.6296

4.3 Models Preprocessing

Before model construction, we use PFA to select the most important metrics for each of the five metric categories: complexity metrics, coupling metrics, size metrics, inheritance metrics, and documentation metrics. As opposed to data preprocessing, PFA is now used separately per category of metrics. We also perform discretization on the float variables (TCD, NUMPAR, McCC) and on the target variable and remove any duplicates in order to reduce the size of the dataset and thus improve the training of the models.

Analysis at Class Level

Complexity Model The dataset has four complexity metrics: NL, NLE, WMC, and McCC. Using PCA and keeping the first 2 PCs (84.49% of the information), the features are split in 3 clusters. Figure 5a shows the correlation of the metrics with the first two PCs, with the selected metrics (NL, WMC, and McCC) in red.

Coupling Model The coupling metrics are CBO, CBOI, NOI, NII, and RFC. By keeping the first 2 PCs (84.95% of the information), we were able to select three of them, i.e. CBO, NII, and RFC, so as to train the ANN. Figure 5b shows the metrics in the first two PCs, with the selected metrics in red.

Documentation Model The dataset includes five documentation metrics (CD, CLOC, DLOC, TCLOC, TCD), out of which DLOC, TCLOC, and TCD were found to effectively cover almost all valuable information (2 principal components with 98.73% of the information). Figure 5c depicts the correlation of the metrics with the kept components, with the selected metrics in red.

Inheritance Model For the inheritance metrics (DIT, NOA, NOC, NOD, NOP), the PFA resulted in 2 PCs and two metrics, DIT and NOC, for 96.59% of the information. Figure 5d shows the correlation of the metrics with the PCs, with the selected metrics in red.

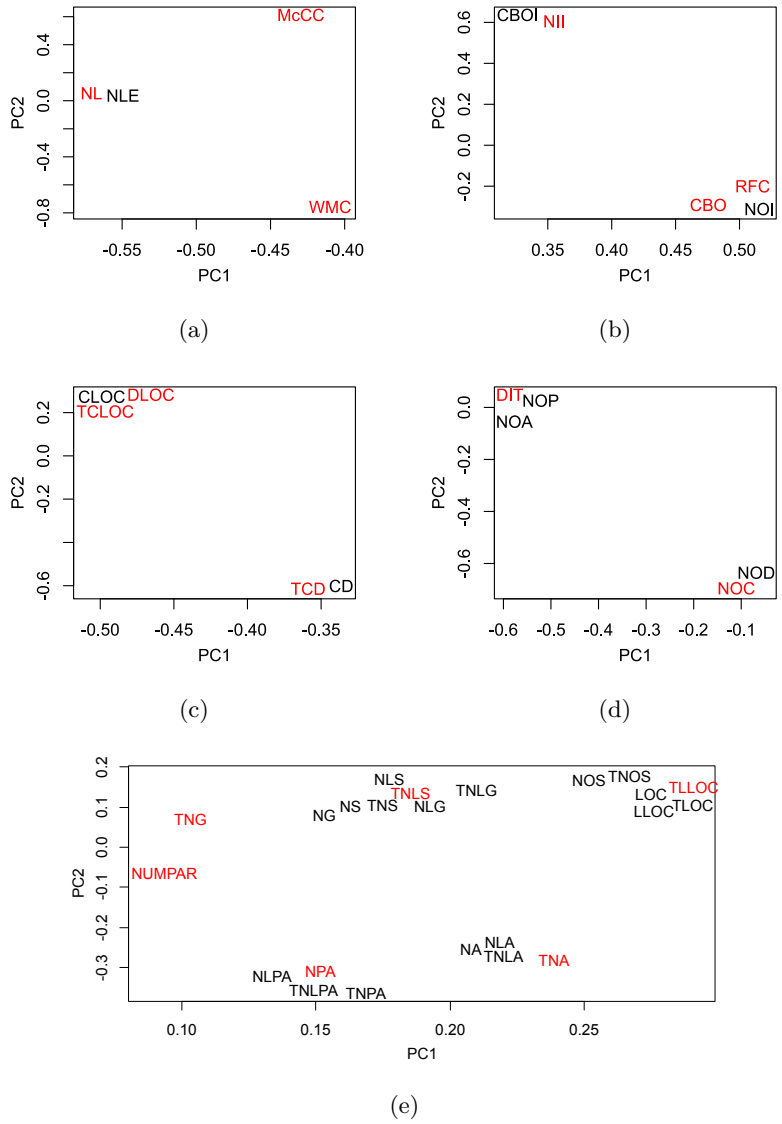


Fig. 5. Visualization of the top 2 PCs at Class Level for (a) Complexity, (b) Coupling, (c) Documentation, (d) Inheritance and (e) Size property [5]

Size Model The PCA for the size metrics indicated that almost all information, 83.65%, is represented by the first 6 PCs, while the first 2 (i.e. 53.80% of the variance) are visualized in Figure 5e. Upon clustering, we select NPA, TLLOC, TNA, TNG, TNLS, and NUMPAR in order to cover most information.

Analysis at Package Level

Complexity Model The dataset has three complexity metrics: WMC, NL and NLA. After using PCA and keeping the first two PCs (98.53% of the information), the metrics are split in 2 clusters. Figure 6a depicts the correlation of the metrics with the PCs, with the selected metrics (NL and WMC) in red.

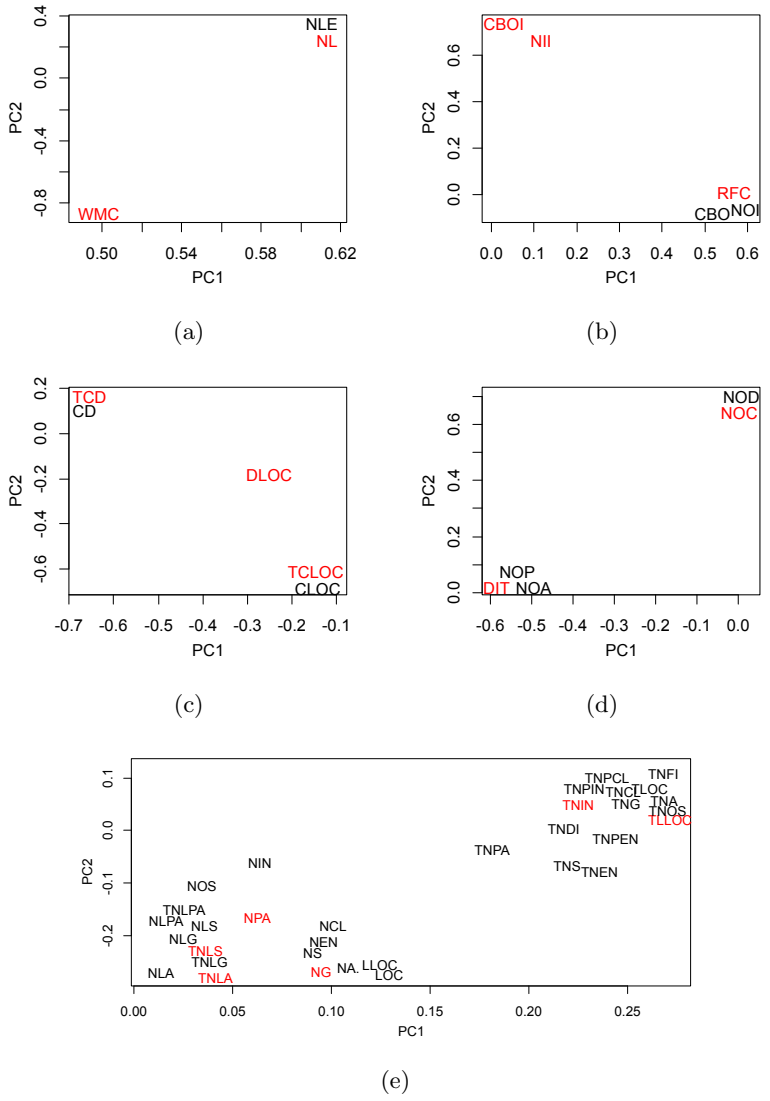


Fig. 6. Visualization of the top 2 PCs at Package Level for (a) Complexity, (b) Coupling, (c) Documentation, (d) Inheritance and (e) Size property

Coupling Model Regarding the coupling metrics, which for the dataset are CBO, CBOI, NOI, NII, and RFC, three of them were found to effectively cover most of the valuable information. In this case the first three principal components were kept, which correspond to 90.29% of the information. The correlation of each metric with the first two PCs is shown in figure 6b, with the selected metrics (CBOI,NII and RFC) in red.

Documentation Model For the documentation model, upon using PCA and keeping the first two PCs (86.13% of the information), we split the metrics in 3 clusters and keep TCD, DLOC and TCLOC as the most representative metrics. Figure 6c shows the correlation of the metrics with the PCs, with the selected metrics in red.

Inheritance Model The inheritance dataset initially consists of DIT, NOA, NOC, NOD and NOP. By applying PCA, 2 PCs were kept (93.06% of the information). The process of selecting metrics resulted in 2 clusters, of which NOC and DIT were selected as the figure 6d depicts.

Size Model The PCA for this category indicated that the 83.57% of the information is successfully represented by the 6 first principal components. Thus, as Figure 6e visualizes, NG, TNIN, TLLOC, NPA, TNLA and TNLS were selected out of 33 size metrics of the original dataset.

4.4 Models Validation

We train five Artificial Neural Network (ANN) models for each level (class and package), each one of them corresponding to one of the five metric properties. All networks have one input, one hidden, and one output layer, while the number of nodes for each layer and each network is shown in Table 4.

Table 4. Neural Network Architecture for each Metrics Category

Metrics Category	**Class**		**Package**	
	Input Nodes	**Hidden Nodes**	**Input Nodes**	**Hidden Nodes**
Complexity	3	1	2	2
Coupling	3	2	3	3
Documentation	3	2	3	3
Inheritance	2	2	2	2
Size	6	4	6	4

10-fold cross-validation was performed to assess the effectiveness of the selected architectures. The validation error for each of the 10 folds and for each of the five models is shown in Figure 7.

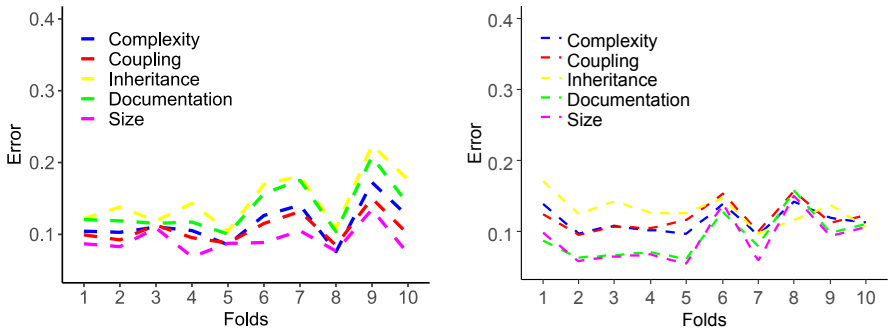


Fig. 7. 10-fold Cross-Validation Error for the 5 ANNs referring to (a) Class level and (b) Package level

Upon validating the architectures that were selected for our neural networks, in the following paragraphs, we describe our methodology for training our models.

4.5 Models Construction

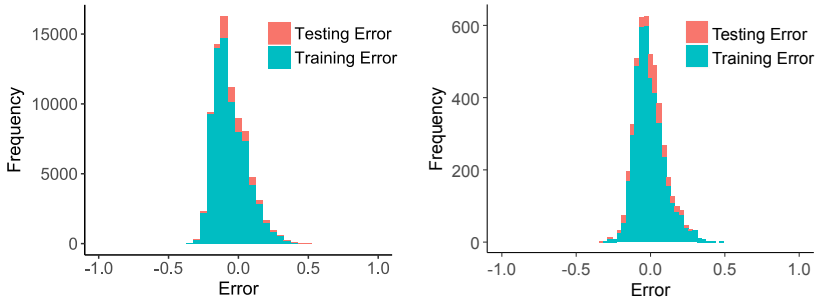
The model construction stage involves the training of five ANN models for each level (class and package) using the architectures defined in the previous subsection. For each level, every model provides a quality score regarding a specific metrics category, and all the scores are then aggregated to provide a final quality score for a given component. Although simply using the mean of the metrics is reasonable, we use weights to effectively cover the requirements of each individual developer. For instance, a developer may be more inclined towards finding a well-documented component even if it is somewhat complex. In this case, the weights of complexity and documentation could be adapted accordingly.

The default weight values for the models applicable at each level are set according to the correlations between the metrics of each ANN and the respective target score. Thus, for the complexity score, we first compute the correlation of each metric with the target score (as defined in Section 3), and then calculate the mean of the absolutes of these values. The weights for the other categories are computed accordingly and all values are normalized so that their sum is equal to 1. The computed weights for the models of each level are shown in Table 5, while the final score is calculated by multiplying the individual scores with the respective weights and computing their sum. Class level weights seem to be more evenly distributed than package level weights. Interestingly, package level weights for complexity, coupling, and inheritance are lower than those of documentation and size, possibly owing to the fact that the latter categories include only metrics computed directly at package level (and not aggregated from class level metrics).

Table 5. Quality Score Aggregation Weights

Metrics Category	Aggregation Weights	
	Class Level	Package Level
Complexity	0.207	0.192
Coupling	0.210	0.148
Documentation	0.197	0.322
Inheritance	0.177	0.043
Size	0.208	0.298

Figure 8 depicts the error distributions for the training and test sets of the aggregated model at both levels (class and package), while the mean error percentages are in Table 6.



(a) (b)

Fig. 8. Error Histograms for the Aggregated model at (a) Class and (b) Package level

The ANNs are trained effectively, as their error rates are low and concentrate mostly around 0. The differences in the distributions between the training and test sets are also minimal, indicating that both models avoided overfitting.

Table 6. Mean Error Percentages of the ANN models

Metrics Category	Error at Class Level		Error at Package Level	
	Training	Testing	Training	Testing
Complexity	10.44%	9.55%	11.20%	9.99%
Coupling	10.13%	8.73%	10.81%	10.08%
Documentation	11.13%	10.22%	7.62%	9.52%
Inheritance	13.62%	12.04%	12.15%	10.98%
Size	9.15%	8.73%	7.15%	9.21%
Final	11.35%	8.79%	7.86%	8.43%

5 Evaluation

5.1 One-Class Classifier Evaluation

Each one-class classifier (one for each level) is evaluated on the test set using the code violations data described in Section 3. Regarding the class level, our classifier ruled out 1594 classes corresponding to 19.93% of the classes, while for the package level, our classifier ruled out 89 packages corresponding to 16% of the packages. The mean number of violations for the rejected and the accepted classes and packages are shown in Table 7, for all the 8 categories of violations.

Table 7. Number of Violations of Accepted and Rejected Components

Violation Types	**Mean Number of Violations**			
	Classes		**Packages**	
	Rejected	**Accepted**	**Rejected**	**Accepted**
WarningInfo	57.6481	17.4574	1278.4831	312.3640
Clone	18.8338	4.1953	13.2359	2.4935
Cohesion	0.5922	0.3003	0.4831	0.2987
Complexity	1.5772	0.0963	1.3033	0.0985
Coupling	1.4737	0.2099	0.9494	0.2109
Documentation	26.2083	11.4102	23.9213	12.5620
Inheritance	1.2516	0.2854	0.5112	0.1113
Size	7.7114	0.9599	6.0505	1.2751

5.2 Quality Estimation Evaluation

Class Level Although the error rates of our system are quite low (see Figure 8), we also have to assess whether its estimations are reasonable from a quality perspective. This type of evaluation requires examining the metric values, and studying their influence on the quality scores. To do so, we use a project as a case study. The selected project, MPAndroidChart, was chosen at random as the results are actually similar for all projects. For each of the 195 class files of the project, we applied our methodology to construct the five scores corresponding to the source code properties and aggregated them for the final quality score.

We use Parallel Coordinates Plots combined with Boxplots to examine how quality scores are affected by the static analysis metrics (Figures 9a to 9f). For each category, we first calculate the quartiles for the score and construct the Boxplot. After that, we split the data instances (metrics values) in four intervals according to their quality score: $[min, q1)$, $[q1, med)$, $[med, q3)$, $[q3, max]$, where min and max are the minimum and maximum score values, med is the median value, and $q1$ and $q3$ are the first and third quartiles, respectively. Each line represents the mean values of the metrics for a specific interval. For example,

the blue line refers to instances with scores in the $[q3, max]$ interval. The line is constructed by the mean values of the metrics NL, $McCC$, WMC and the mean quality score in this interval, which are 1.88, 1.79, 44.08, and 0.43 respectively. The red, orange, and cyan lines are constructed similarly using the instances with scores in the $[min, q1)$, $[q1,)$, and $[med, q3)$ intervals, respectively.

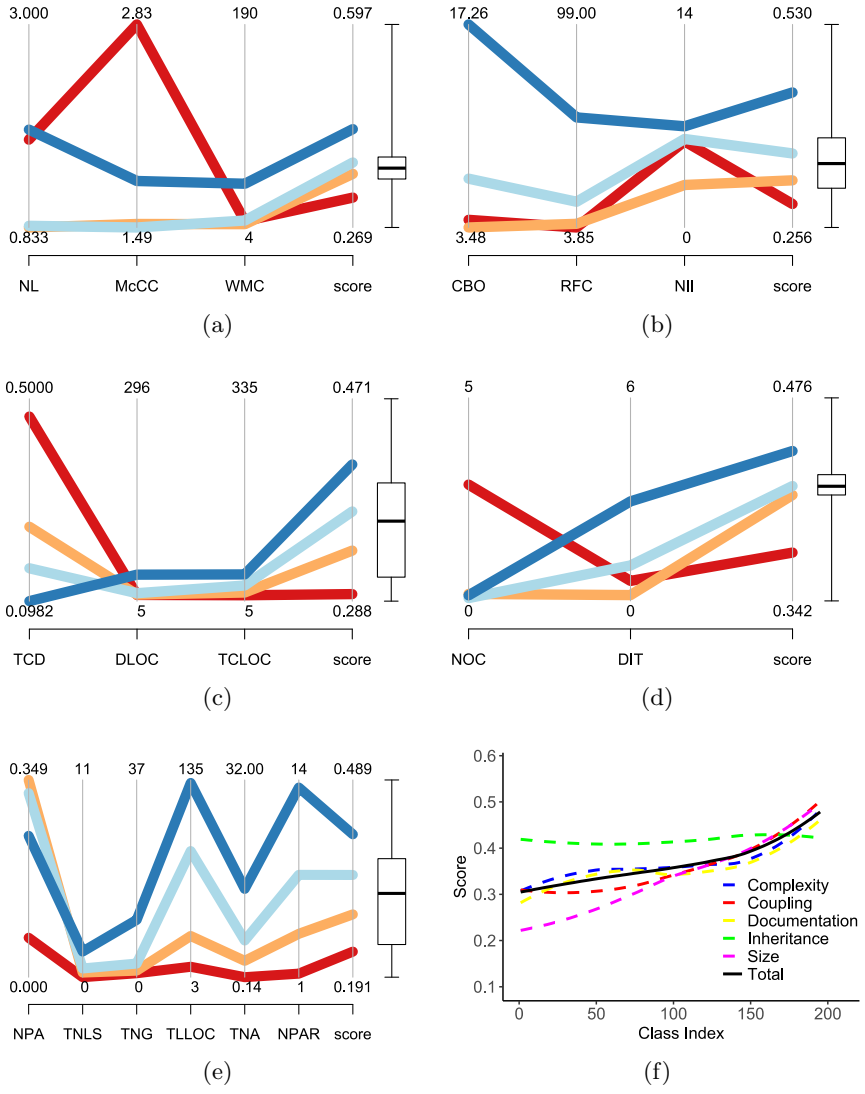


Fig. 9. Parallel Coordinates Plots at Class level for the Score generated from (a) the Complexity Model, (b) the Coupling Model, (c) the Documentation Model, (d) the Inheritance Model, (e) the Size Model, and (f) plot showing the Score Aggregation [5]

Figure 9a refers to the complexity model. This plot results in the identification of two dominant trends that influence the score. At first, $McCC$ appears to be crucial for the final score. High values of the metric result in low score, while low ones lead to high score. This is expected since complex classes are prone to containing bugs and overall imply low quality code. Secondly, the metrics WMC and NL do not seem to correlate with the score individually; however they affect it when combined. Low WMC values combined with high NL values result in low quality scores, which is also quite rational given that more complex classes with multiple nested levels are highly probable to exhibit low quality.

Figures 9b and 9c refer to the coupling and the documentation models, respectively. Concerning coupling, the dominant metric for determining the score appears to be RFC. High values denote that the classes include many different methods and thus many different functionalities, resulting in high quality score. As for the documentation model, the plot indicates that classes with high comment density (TCD) and low number of documentation lines ($DLOC$) are given a low quality score. This is expected as this combination probably denotes that the class does not follow the Java documentation guidelines, i.e. it uses comments instead of Javadoc.

Figures 9d and 9e refer to the inheritance and size models, respectively. DIT appears to greatly influence the score generated by the inheritance model, as its values are proportional to those of the score. This is expected as higher values indicate that the class is more independent as it relies mostly on its ancestors, and thus it is more reusable. Although higher DIT values may lead to increased complexity, the values in this case are within acceptable levels, thus the score is not negatively affected.

As for the size model, the quality score appears to be mainly influenced by the values of $TLLOC$, TNA and $NUMPAR$. These metrics reflect the amount of valuable information included in the class by measuring the lines of code and the number of attributes and parameters. Classes with moderate size and many attributes or parameters seem to receive high quality scores. This is expected as attributes/parameters usually correspond to different functionalities. Additionally, a moderately sized class is common to contain considerable amount of valuable information while not being very complex.

Finally, Figure 9f illustrates how the individual quality scores (dashed lines) are aggregated into one final score (solid line), which represents the quality degree of the class as perceived by developers. The class indexes (project files) are sorted in descending order of quality score. The results for each score illustrate several interesting aspects of the project. For instance, it seems that the classes exhibit similar inheritance behavior throughout the project. On the other hand, the size quality score is diverse, as the project has classes with various size characteristics (e.g. small or large number of methods), and thus their score may be affected accordingly. Finally, the trends of the individual scores are in line with the final score, while their variance gradually decreases as the final score increases. This is expected as a class is typically of high quality if it exhibits acceptable metric values in several categories.

Package Level Following the same strategy as in the case of classes, we constructed Parallel Coordinates Plots combined with Boxplots towards examining the influence of the values of the static analysis metrics on the quality score. Figure 10 depicts the plots for each of the five source code properties under evaluation and the aggregated plot of the final quality score.

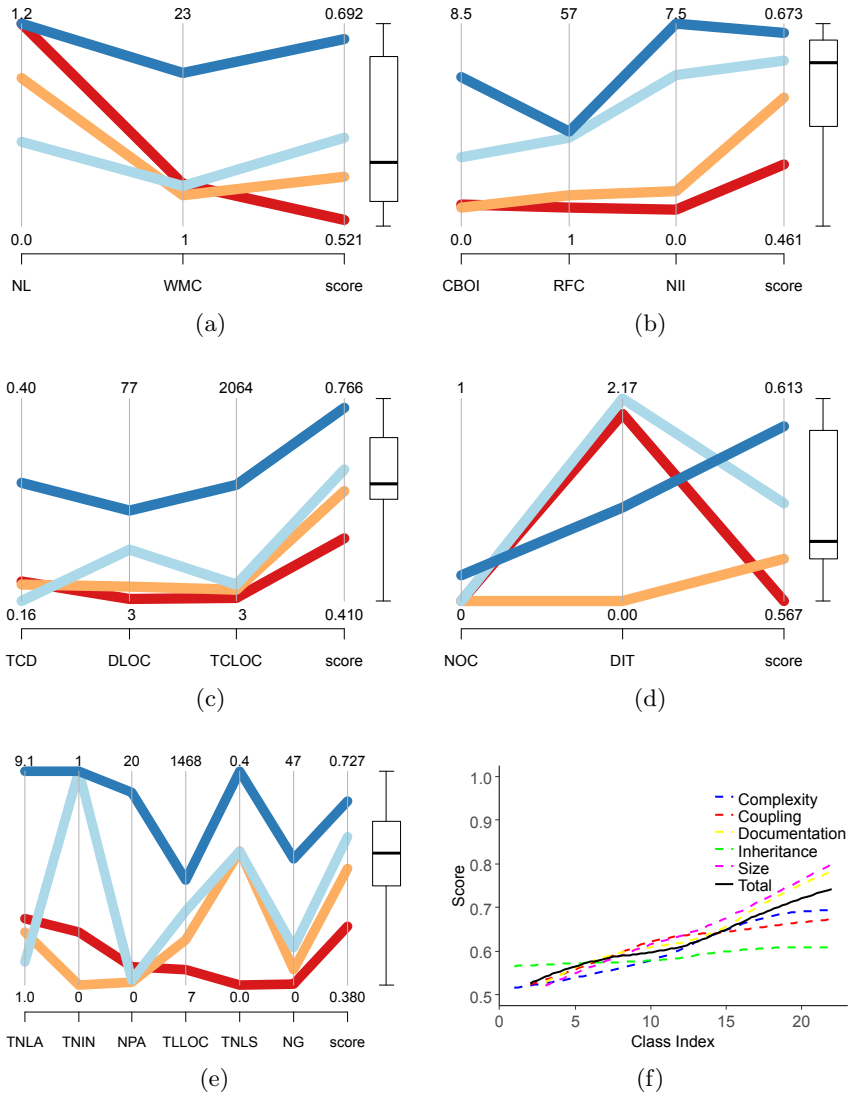


Fig. 10. Parallel Coordinates Plots at Package level for the Score generated from (a) the Complexity Model, (b) the Coupling Model, (c) the Documentation Model, (d) the Inheritance Model, (e) the Size Model, and (f) plot showing the Score Aggregation

At this point, it is worth noticing that only in the cases of size and documentation, the values of the static analysis metrics originate from the packages themselves, while for the other three models the values of the static analysis metrics originate from classes. As a result, the behaviors extracted in the cases of size and documentation are considered more accurate which originates from the fact that they do not accumulate noise due to aggregations. As already noted in subsection 3.1, the median was used as an aggregation mechanism, which is arguably an efficient measure as it is at least not easily influenced by extreme metrics' values.

Figure 10a refers to the complexity model. As it can be seen from the diagram, the outcome of the complexity score appears to be highly influenced by the values of WMC metric. High WMC values result in high score while lower values appear to have the opposite impact. Although this is not expected as higher complexity generally is interpreted as an negative characteristic, in this case, given the intervals of the complexity-related metrics, we can see that the project under evaluation appears to exhibit very low complexity. This is reflected in the intervals of both NL and WMC which are [0, 1.2] and [0, 23], respectively. Consequently, the extracted behaviour regarding the influence of WMC in the outcome of the final score can be considered logical as extremely low values of WMC (close to zero) indicate absence of valuable information and thus the score is expected to be low.

Figures 10b and 10c refer to the coupling and the documentation model, respectively. In the case of coupling, it is obvious that the values of the NII (Number of Incoming Invocations) metric appear to highly influence the outcome of the final score. High NII values result in high score, while low values appear to have a negative impact. This is expected as NII metric reflects the significance of a given package due to the fact that it measures the number of other components that call its functions. In addition, we can see that high values of CBOI (Coupling Between Objects Inverse) metric result in high coupling score which is totally expected as CBOI reflects how decoupled is a given component. As for the documentation model, it is obvious that the Total Comments Density (TCD) metric appears to influence the outcome of the final score. Moderate values (around 20%) appear to result in high scores which is logical considering the fact that those packages appear to have one line of comment for every five lines of code.

Figures 10d and 10e refer to the inheritance and the size model, respectively. As for the inheritance model, DIT metric values appear to greatly influence the generated score in a proportional manner. This is expected as higher DIT values indicate that a component is more independent as it relies mostly on its ancestors, and thus it is more reusable. It is worth noticing that although higher DIT values may lead to increased complexity, the values in this case are within acceptable levels, thus the score is not negatively affected. As for the size model, the packages that appear to have normal size as reflected in the values of TLLOC (Total Logical Lines Of Code) metric receive high score. On the other hand, the ones that appear to contain little information receive low score, as expected.

Finally, Figure 10f illustrates how the individual quality scores (dashed lines) are aggregated into one final score (solid line), which represents the quality degree of the package as perceived by developers. The package indexes are sorted in descending order of quality score. Similar to the case of classes, the trends of the individual scores are in line with the final score. The score that originates from the inheritance model exhibits the highest deviation from the final score, while the lowest deviation is the one of the scores originating from the size and the documentation models. This is expected as those two properties include metrics that are applicable directly at package level.

5.3 Example Quality Estimation

Further assessing the validity of our system, for each category we manually examine the values of the static analysis metrics of 20 sample components (10 classes and 10 packages) that received both high and low quality scores regarding each one of the five source code properties, respectively. The scores for these classes and packages are shown in Table 8. Note that the presented static analysis metrics refer to different classes and packages for each category. For the complexity model, the class that received low score appears to be much more complex than the one that received high score. This is reflected in the values of McCC and NL, as the low-scored class includes more complex methods (8.5 versus 2.3), while it also has more nesting levels (28 versus 4). The same applies for the packages that received high and low scores, respectively.

For the coupling model, the high-quality class has significantly higher NII and RFC values when compared to those of the low-quality class. This difference in the number of exposed functionalities is reflected in the quality score. The same applies for the inheritance model, where the class that received high score is a lot more independent (higher DIT) and thus reusable than the class with the low score. The same conclusions can be derived for the case of packages where it is worth noticing that the difference between the values of the coupling-related metrics between the high-scored and the low-scored package are smaller. This is a result of the fact that the described coupling metrics are only applicable at class level.

As for the inheritance score, it is obvious in both the cases of classes and packages that the higher degree of independence as reflected in the low values of NOC and NOP metrics results into high score. Finally, as for the documentation and size models, in both cases the low-quality components (both classes and packages) appear to have no valuable information. In the first case, this absence is obvious from the extreme value of comments density (TCD) combined with the minimal documentation (TCLOC). In the second case, the low-quality class and package contain only 10 and 40 logical lines of code (TLLOC), respectively, which indicates that they are of almost no value for the developers. On the other hand, the high-quality components seem to have more reasonable metrics values.

Table 8. Static Analysis Metrics per Property for 20 components (10 Classes and 10 Packages) with different Quality Scores.

Metrics		Classes		Packages	
Category	**Name**	**High Score (80–90%)**	**Low Score (10–15%)**	**High Score (80–90%)**	**Low Score (10–15%)**
Complexity	McCC	2.3	8.5	–	–
	WMC	273	51	12	6
	NL	4	28	2	4
Coupling	NII	88	0	21.5	4
	RFC	65	7	30	8
	CBO	5	35	3	14
Documentation	TCD	0.3	0.8	0.35	0.82
	DLOC	917	2	372	7
	TCLOC	1,019	19	2654	24
Inheritance	DIT	8	0	3	0
	NOC	1	16	1	9
	NOP	2	14	2	8
Size	NUMPAR	27	3	–	–
	NCL	–	–	9	1
	TNA	69	0	65	2
	NPA	36	0	29	0
	TLLOC	189	10	1214	40
	TNLS	13	2	78	4
	NM	9	1	98	1

5.4 Threats to Validity

The threats to the validity of our approach and our evaluation involve both its applicability to software projects and its usage by the developers. Concerning applicability, the dataset used is quite diverse; hence our methodology can be seamlessly applied to any software project for which static analysis metrics can be extracted. Concerning expected usage, developers would harness the quality estimation capabilities of our approach in order to assess the quality of their own or third-party software projects before (re)using them in their source code. Future work on this aspect may involve integrating our approach in a system for software component reuse, either as an online component search engine or as an IDE plugin.

6 Conclusions

Given the late adoption of a component-based software engineering paradigm, the need for estimating the quality of software components before reusing them (or before publishing one's components) is more eminent than ever. Although previous work on the area of designing quality estimation systems is broad, there is usually some reliance on expert help for model construction, which in turn may lead to context-dependent and subjective results. In this work, we employed information about the popularity of source code components to model their quality as perceived by developers, an idea originating from [15] that was found to be effective for estimating the quality of software classes [5].

We have proposed a component-based quality estimation approach, which we construct and evaluate using a dataset of source code components, at class and package level. Upon removing outliers using a one-class classifier, we apply Principal Feature Analysis techniques to effectively determine the most informative metrics lying in five categories: complexity, coupling, documentation, inheritance, and size metrics. The metrics are subsequently given to five neural networks that output quality scores. Our evaluation indicates that our system can be effective for estimating the quality of software components as well as for providing a comprehensive analysis on the aforementioned five source code quality axes.

Future work lies in several directions. At first, the design of our target variable can be further investigated for different scenarios and different application scopes. In addition, various feature selection techniques and models can be tested to improve on current results. Finally, we could assess the effectiveness of our methodology by means of a user study, and thus further validate our findings.

References

1. Alves, T.L., Ypma, C., Visser, J.: Deriving metric thresholds from benchmark data. In: IEEE International Conference on Software Maintenance (ICSM). pp. 1–10. IEEE (2010)
2. Cai, T., Lyu, M.R., Wong, K.F., Wong, M.: ComPARE: A generic quality assessment environment for component-based software systems. In: proceedings of the 2001 International Symposium on Information Systems and Engineering (2001)
3. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering 20(6), 476–493 (1994)
4. Diamantopoulos, T., Thomopoulos, K., Symeonidis, A.: QualBoa: reusability-aware recommendations of source code components. In: IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), 2016. pp. 488–491. IEEE (2016)
5. Dimaridou, V., Kyprianidis, A.C., Papamichail, M., Diamantopoulos, T., Symeonidis, A.: Towards modeling the user-perceived quality of source code using static analysis metrics. In: 12th International Conference on Software Technologies (ICSOFT). pp. 73–84. Madrid, Spain (2017)
6. Ferreira, K.A., Bigonha, M.A., Bigonha, R.S., Mendes, L.F., Almeida, H.C.: Identifying thresholds for object-oriented software metrics. Journal of Systems and Software 85(2), 244–257 (2012)

7. Foucault, M., Palyart, M., Falleri, J.R., Blanc, X.: Computing contextual metric thresholds. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1120–1125. ACM (2014)
8. Hegedűs, P., Bakota, T., Ladányi, G., Faragó, C., Ferenc, R.: A drill-down approach for measuring maintainability at source code element level. Electronic Communications of the EASST 60 (2013)
9. Heitlager, I., Kuipers, T., Visser, J.: A practical model for measuring maintainability. In: 6th International Conference on the Quality of Information and Communications Technology, 2007. QUATIC 2007. pp. 30–39. IEEE (2007)
10. ISO/IEC 25010:2011. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en (2011), [Online; accessed November 2017]
11. Kanellopoulos, Y., Antonellis, P., Antoniou, D., Makris, C., Theodoridis, E., Tjortjis, C., Tsirakis, N.: Code quality evaluation methodology using the iso/iec 9126 standard. CoRR 1(3), 17–36 (2010)
12. Le Goues, C., Weimer, W.: Measuring code quality to improve specification mining. IEEE Transactions on Software Engineering 38(1), 175–190 (2012)
13. Lu, Y., Cohen, I., Zhou, X.S., Tian, Q.: Feature selection using principal feature analysis. In: Proceedings of the 15th ACM international conference on Multimedia. pp. 301–304. ACM (2007)
14. Miguel, J.P., Mauricio, D., Rodríguez, G.: A review of software quality models for the evaluation of software products. arXiv preprint arXiv:1412.2977 (2014)
15. Papamichail, M., Diamantopoulos, T., Symeonidis, A.: User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics. In: IEEE International Conference on Software Quality, Reliability and Security (QRS), 2016. pp. 100–107. IEEE (2016)
16. Pfleeger, S.L., Atlee, J.M.: Software engineering: theory and practice. Pearson Education India (1998)
17. Pfleeger, S., Kitchenham, B.: Software quality: The elusive target. IEEE Software pp. 12–21 (1996)
18. Samoladas, I., Gousios, G., Spinellis, D., Stamelos, I.: The sqo-oss quality model: measurement based open source software evaluation. Open source development, communities and quality pp. 237–248 (2008)
19. Schmidt, C.: Agile Software Development Teams. Springer (2015)
20. Shatnawi, R., Li, W., Swain, J., Newman, T.: Finding software metrics threshold values using roc curves. Journal of Software: Evolution and Process 22(1), 1–16 (2010)
21. SourceMeter static analysis tool. https://www.sourcemeter.com/ (2017), [Online; accessed November 2017]
22. 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 8(1), 114–120 (2014)
23. Washizaki, H., Namiki, R., Fukuoka, T., Harada, Y., Watanabe, H.: A framework for measuring and evaluating program source code quality. In: PROFES. pp. 284–299. Springer (2007)
24. Zhong, S., Khoshgoftaar, T.M., Seliya, N.: Unsupervised Learning for Expert-Based Software Quality Estimation. In: HASE. pp. 149–155 (2004)