Code Reusability Estimation based on Static Analysis Metrics

Konstantinos Samaras-Tsakiris , Ioanna Apostolina
Electrical and Computer Engineering Deptartment
Aristotle University Of Thessaloniki
Thessaloniki, Greece
Email: {kisamara , igaposto}@ece.auth.gr

Abstract—**The idea of reusing software components has been present in software engineering for several decades. This idea becomes more and more popular nowadays that open source software repositories are a common fact. So far, several tools have been developed to assess the software quality of these repositories which depends on fixed metric thresholds for defining the ground truth. In this work we present an organized approach that relates quality of software components with static analysis metrics and it estimates the reusability of software components in popular GitHub repositories. Our methodology includes two models: a one-class classifier, used to exclude all low quality code find inside classes of our 137 popular repositories and a neural network that computes a reusability score for each class of each repository. Evaluations shows that our approach can be effective and give sufficient results in the context of reuse of software components.**

Keywords—source code quality, reusability, static analysis, user-perceived quality, neural network, one-class classification

I. Introduction

The numerous open source projects that exist in online repositories such as GitHub have been used as a tool to reduce the software cost and time. The majority of software systems are now being developed to a certain degree from an assembly of already existing reusable components. In order to assess these open source software components effectively, we should measure their *reusability* .

Quality is a complex concept: because it means different things to different people, it is highly context-dependent [2]. There was a common need for standardization of *quality* so international standard ISO/IEC 25010:2011 created. This international standard defines that the quality of a software component can be measured with eight quality characteristics to ensure *Functional Suitability*, *Usability*, *Performance Efficiency*, *Portability*, *Operability*, *Security*, *Compatibility*, and *Maintainability* [3]. However from a user's perspective quality is referred to the reusability to a software component and it is related with only four of the above characteristics: *Functional Suitability*, *Usability*, *Maintainability and Portability* [4].

The attempt to measure software quality in an appropriate manner has led to the development of various metrics that we will discuss later. However it is difficult and inconsistent to define quality based on thresholds for these metrics; in any case, this approach requires not an easy task and it is usually performed by an expert [5]. This method is not flexible enough to describe in an effective way major software components. To fill this gap, a method that uses user-perceived quality as a measure of the reusability of a software component was introduced a year ago and had good evaluation results as we will describe in the next chapter [6].

In this paper, we appraise the code quality in terms of reusability of a software component through the use of static analysis metrics. We hypothesize that popularity from a user's point of view is a measure of reusability of a software component. In other words, we use the popularity of these projects as the ground truth. Based on this hypothesis, we design and create a system based on the four software quality characteristics: *Functional Suitability*, *Maintainability*, *Usability and Portability* with the simultaneous use of a set of static analysis metrics. Estimation of the reusability of a software component becomes possible. Our system consists of two main models: a one-class classifier trained with a support vector machine (SVM) that determines if a class belongs to the quality range this system has been trained in, and an artificial neural network (ANN) that estimates the reusability of software components given the static analysis metrics. The analysis and the models are built with the use of static analysis metrics referring to classes. Thus the system is based upon finding the reusability of each class of a repository and as such infer the reusability of the overall repository.

II. Research Overview

Previous years various software metrics have been proposed from research for measuring the quality of software components [7, 8] but this problem remains not an easy one. This task requires an exact specification of software metrics and it is usually accomplished by an expert. Nevertheless, the help of an expert is not always available. For this reason, various methods have been proposed.

Estimating quality characteristics has been a popular research topic for many years. Many methods have been developed about software metrics thresholds definition. Nevertheless, most of them do not be effective when it comes to real world scenarios. A common practice is the adaptable quality estimation with the design of models [9]. One other practice is by analyzing the results of numerous software metrics that are computed. Ferreira et al. have proposed a method that uses the value of the computed metrics with probability distribution of them and indicate the bounds of metrics [10]. Another common approach is the design of quality evaluation systems which target to a single quality characteristic. Kumar proposes an SVM -based classifier and in this ways builds a reusability estimation system for software components[11]. One other method works efficient is this of Papamichail et al. which given static analysis metrics and using the popularity of software repositories create a system wich combines two models: a one- class classifier and an artificial neural network that estimates the quality of the code [6].

Although most of these proposals do not work properly in real-world scenarios as we referred earlier. Firstly, they are limited within certain quality thresholds; something that shows a limitation to the extent of being objective for all classes [9]. Automated system from the other side need the knowledge of an expert for training the model so we cannot use them every time that we need to evaluate the quality of a software component. Furthermore, these systems do not offer a single output measurement the user-perceived quality.

In this work, we build a system that estimates the code reusability and provides a single metric based on a set of static analysis metrics referred to classes, related to user-perceived quality and specifically referred to classes of each repository . Given that popularity of components has a strong relationship with reuse [12], we consider that popularity means high quality of components. In other words, a repository residing in GitHub which is rated with many stars and forks possibly is a high quality project and reusable one. Based on this idea, we build a system that provides a reusability score for each repository and information about the values of metrics of classes. From a user-perspective, we chose class level to built our system because methods are too fine-grained to assess reusability, and packages too coarse.

III. SCORING CLASSES OF REPOSITORIES

In this section, we discuss the target of our dataset and we build a metric based on the number of stars and forks.

At first, we have to define the reason that lead us to the creation of a target of our dataset that is based both on the number of forks and stars for each repository To support thic claim, we computed the correlation between stars and forks of a repository. As illustrated in figure 2, there is no strong correlation between stars and forks. Actually the value of the correlation between the two metrics is 0.474.

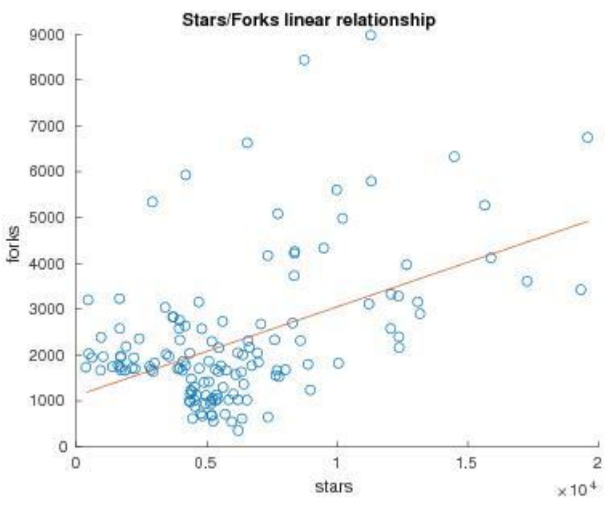


Fig. 1. Stars - Forks Diagram

The dataset consists of 60 static analysis metrics computed for the most 137 GitHub repositories .Nevertheless, the number of stars and forks is not enough and a score is required per class of each repository and reflects its reusability individually. We should note that the number of scores and stars cannot be equally distributed among the classes of a repository as the classes have not the same significance and impact for the reuse of a repository. The score for a class is given by two terms: one that includes star rating of a repository and accounts also the significance of the class as an independent entity based on stars and the second term that includes forks rating of the repository and the significance of the class based on forks. Finally the score for a class of a repository is given by the following equation:

$$F_{score}(i,j) = 0.4 * s + 0.6 * f \ (1)$$

Equation (1) that we wrote above consists of 2 main terms. The first used term of this equation is equal to:

$$s = \log_2\left(\left(1 + CBOI(i)\right)\left(1 + \frac{R_{stars}(j)}{n_{classes}(j)}\right)\right)$$

and the second term f is:

$$f = \log_2\left(\left(1 + NPM(i)\right)\left(1 + \frac{R_{forks}(j)}{n_{classes}(j)}\right)\right)$$

where $F_{score}(i,j)$ represents the target score for class i which is included in repository j, $CBOI(i)$ is the Coupling Between Object classes Inverse metric for class i that is to say the direct use of the class i from other classes, $R_{stars}(j)$ represents the number of stars for repository j, $R_{forks}(j)$ represents the number of forks, $n_{classes}(j)$ shows the number of classes into the j repository and $NPM(i)$ is the Number Of Public Methods metric for class i.

Equation (1) consists of two factors. The first one is where the +1 unit assigns a base score to all the classes and normalizes the CBOI metric, ensuring the term is >1 of a class in case it's value is smaller than one. This reasoning applies to the 2nd term as well, ensuring that their product is always >1. Thus, the logarithm is bounded below to 0. The logarithm plays a smoothing role and better distributes the targetset's values in the range 0-10. This is necessary due to the exponential distribution of the metrics and the wildly varying number of classes among the analyzed repositories. In fact, as is shown in figure 2, the target set's distribution falls rapidly towards 10.

Although the presented equation thus guarantees a lower bound of 0 for the target set, there is only experimental evidence that values approaching and above 10 are rare. Thus with minimal distortion the target set applies an extra rule that clips the rare score that is >10 to the maximum value of 10.

The second factor shows the number of stars of the repository j which account for every class. We add as previous 1 a unit for the normalization of the fraction of number of stars to the number of classes of the repository. The logarithmic function is used as a smoothing factor of the diversity between the numerous classes into different software repositories. This is a very important function because of the fact that other repositories could have only 200 classes and other could have more than 3000. In other words it's a way to have representative results.

The same logic applies to the second term "f" of equation (1).

Finally, the histogram of our target shown (Figure 2) that it is a Gaussian Distribution and values taken are in the interval [0, 10].

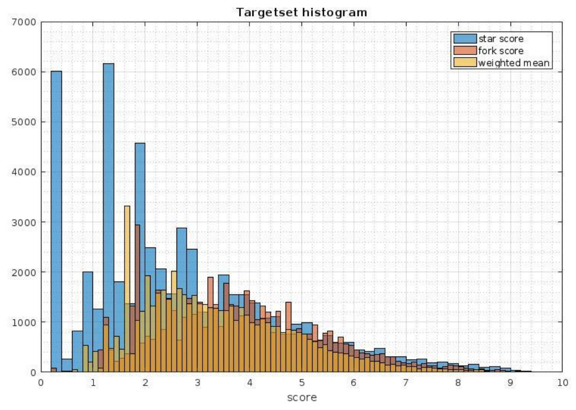


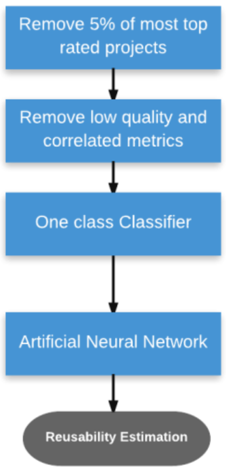
Fig. 2. Histogram of the Target Set

IV. System Design

This section presented the analysis of the provided dataset and gives a thorough description of our reusability estimation system.

A. *System Overview*

In this effort to estimate quality we use a set of static analysis metrics for every class of our dataset so as to train two models: a one class-classifier and a artificial neural network. Before training the models, the dataset is preprocessed and both low-quality observations and uninformative metrics are eliminated.


Remove 5% of most top rated projects
Remove low quality and correlated metrics
One class Classifier
Artificial Neural Network
Reusability Estimation
Fig.3. Overview of the Reusability Estimation System

B. *Metrics initial preprocessing*

That said, the first preprocessing step caters to the needs of the targetset; the 5% most-starred and most-forked repositories are eliminated. According to figure 1, most repositories cluster in an area at the bottom-left of the graph without linear correlation, so the most-starred and most-forked ones can be considered outliers. After all, this is already a curated list of the highest stars and forks. For these repositories the argument stands that their popularity might be influenced by external factors other than code quality. In retrospect, the ANN model was much better able to fit the remaining data, thus corroborating this hypothesis.

The next step is to remove low quality classes out of our dataset via handpicked filters. First action taken is removing the classes that have Total Number of Statements TNOS < 5. Furthermore, we exclude from our dataset the classes where the Coupling between Object classes (CBO) metric is zero, which means that these classes are not used directly any other class and consequently not reusable. The last filter removes the classes that do not have any public methods or public attributes .

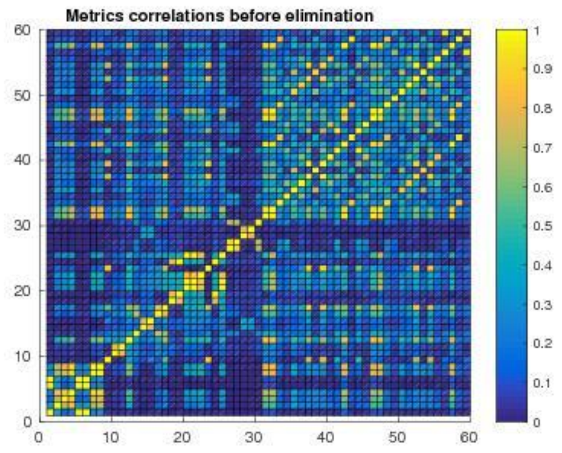


Fig.4. Correlation of static analysis metrics before elimination

At this point, a large issue remains: many metrics are highly correlated, as figure 4 shows. The highly correlated pairs provide redundant information and make the estimation task more difficult; thus, from each pair 1 metric was eliminated. In most cases the reason was obvious: Total Number of Public Methods ought to be correlated with Total Number of Local Public Methods. In other cases not so much and both metrics were kept. After this cleanup the remaining correlations are shown in figure 5.

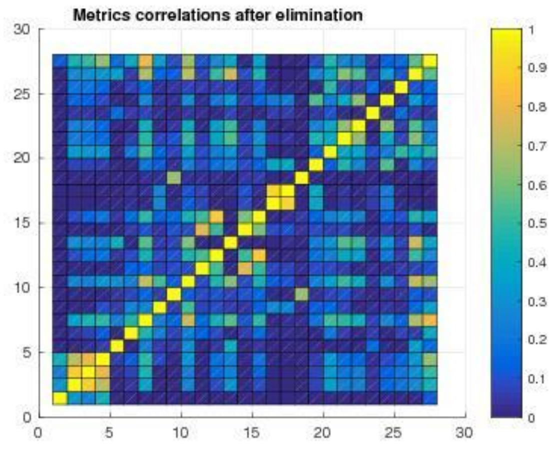


Fig. 5. Correlation of static analysis metrics after elimination

C. Acceptance Classifier

We use an one-class classifier to remove the code code with metrics that differ greatly from the training data, to avoid having the ANN extrapolate. At first, in the effort to train this model we use principal component analysis (PCA) to select the metrics with the maximum effect on the decision taken. As these metrics have the greatest impact to our dataset the one-class acceptance classifier is trained with them.

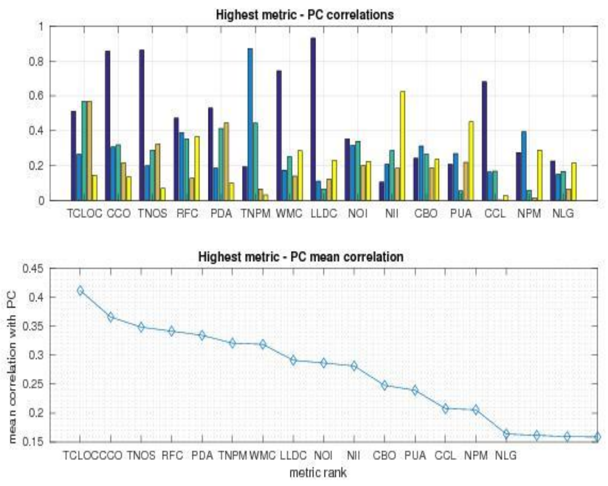


Fig. 6. Static Analysis Metrics - Top 5 Component correlation diagram

The Gaussian radial basis kernel function is used to train the one-class classifier and the training assumes that 5% of the so-far accepted dataset should be considered an outlier. This number is arbitrary, but aims to provide a robust decision boundary.. It is important to refer that the selection parameter nu used for the support vector machine is equal to 0.2 and the scale of the kernel is equal to 0.49.

D. Scorer: Artificial Neural Network Model

The regression model that estimates the targetset based on the metrics of any classes that have reached this far in the datapath is an ANN. In fact, two architectures with promising results are presented here and many more were explored without success. The first is a simple one-layer network and the second has 2 layers, with the first being an autoencoder.

The first artificial neural network is a one-layer feedforward network with sigmoid input and output neurons. trained with the Levenberg-Marquardt algorithm(LMA) [13]. The input consists of the remaining set of 28 static analysis metrics after preprocessing while the output is the reusability score. (Section III).

70% of the data samples were used as training set, 15 % as testing set and 15% for validation. The mean square error was used as performance action, over 139 epochs. Figure 7 shows the error histogram of the model. The relative MSE is also calculated: 16.4% for training, 17.1% for test data.

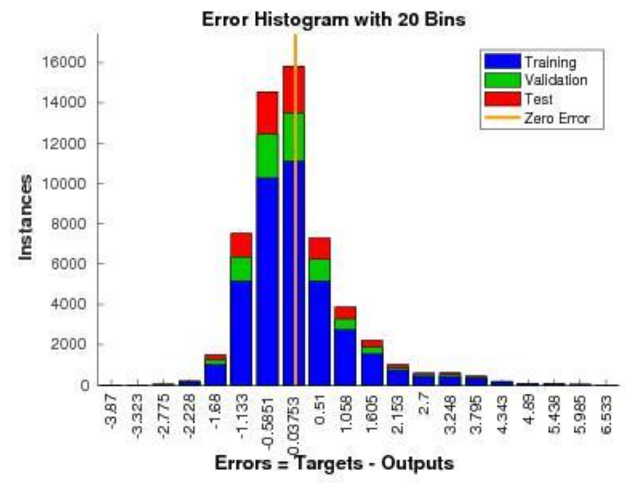


Fig. 7. Error histogram of mean square error

Figure 8 shows no overfit.

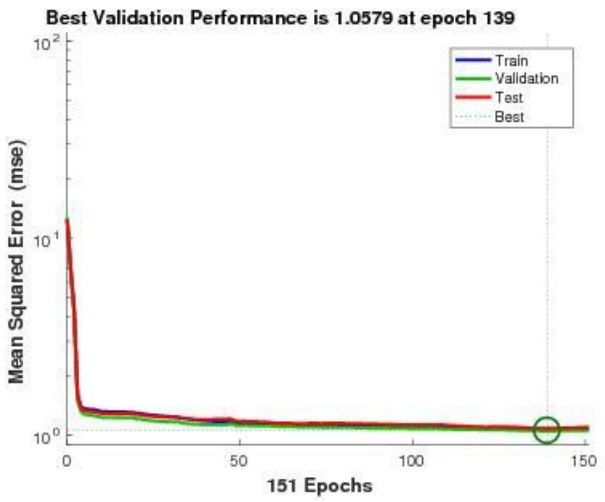


Fig. 8. Mean Square Error Performance

Figure 9 plot the output versus the target, showing their linear regression.. Ideally, if output and target were equal, this would be the dashed line. The current fit is the colored line. An important observation is that there is a bigger spread towards the best scores, which are consistently underestimated.

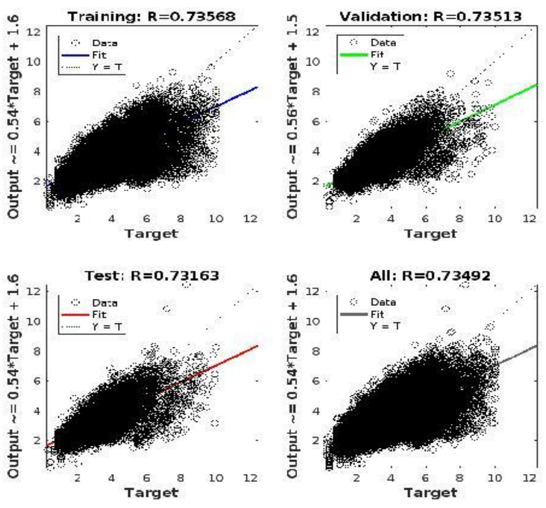


Fig. 9. Regression of ANN model

E. Artificial Neural Network Model: A different approach

The second artificial network is more interesting. As stated, it has 2 layers: a sparse autoencoder and a regression layer. The autoencoder plays a role similar to an extended nonlinear PCA. By enforcing a mean sparsity level of 2 neurons per class out of 50, it transforms the correlated and mangled input of the static analysis metrics into a more orderly fashion. The regression layer can then solve an easier problem by assigning values to the sparse activations.

The training of the 2 layers is mostly independent. The autoencoder is trained in an unsupervised manner. It has 2 layers, an encoder and a decoder, and attempts to recreate its input at its output, while the encoder provides the transformed features. The training stops after a predifined number of iterations. The sparsity constraint was enforced with a large regularization weight, so the trained error remains quite large. The regression layer is trained with Levenberg-Marquardt backpropagation (like network 1), with the autoencoder features as input and the targetset as output.

At this stage the stacked network almost provides acceptable performance. The last step is a combined training phase for both layers together with Levenberg-Marquardt backpropagation. Because the autoencoder is quite large it is a time-consuming task and was not followed through to perfection for the requirements of this project. Still, the results are similar in accuracy to the 1st network with no signs of overfitting and potential for further training.

Given the networks’ better performance for the lower values of the target set, an attempt was made to split the problem in 2 phases: first, a classification between “low”, ”medium” and “high” reusability, then a different regression model for each to provide a finer answer. Though the classification task was assumed, due to its relative coarseness, to be much easier, the network failed spectacularly. The results are shown in the below figures 10, 11.

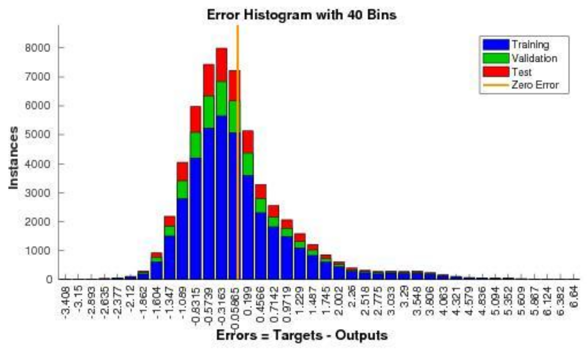


Fig. 10. Error histogram of mean square error

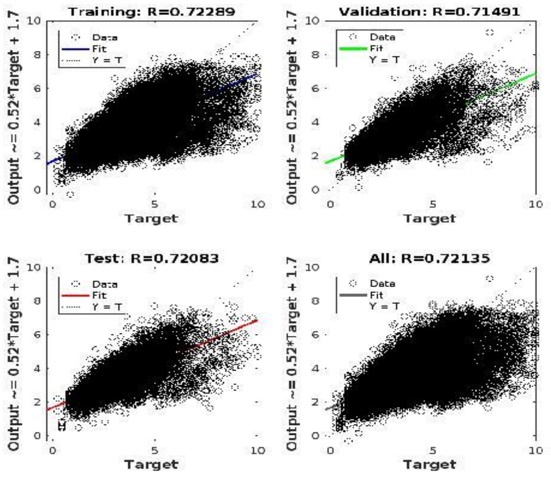


Fig. 11. Regression of ANN model

V. Evaluation

This section discusses the evaluation of our system. The neural network scorers perform a supervised learning task, thus a figure of merit was shown already for their regression performance, which is acceptable and suffices for a coarse estimation and could be improved with further tuning.. The acceptance classifier and the entire system are evaluated here.

A. *Evaluation Methodology*

At first we check the probability density functions of violations as shown when data are accepted from our SVM and the instance that they are rejected. This method help us to investigate if SVM rejects good or bad quality code. Then we will examine the whole system through the use of a new unknown repositories. The evaluation shows that the one-class classifier rejects with high probability classes with outlying metrics and the neural network.

B. *Evaluation Results*

Figure 12 below shows the different probability distributions of rule violations for the repositories that have been accepted or rejected by the SVM. It is clear that the probability of serious violations is higher among the rejected repositories, thus it can be argued that the SVM actually protects against low-quality code. All the data this system has been trained on should be considered relatively high quality, since they come from the most-starred repositories, and so the scorer models wouldn’t be accurate in estimating low-quality code.

The performance of the entire system, including the decision of the targetset, is evaluated with 2 repositories that were not used for training and were created automatically with the S-CASE tool. Because the code is not handwritten, the quality is expected to be always acceptable and the reusability, if not high, then at least consistent, without great variability. This hypothesis is partially confirmed in figure 13.

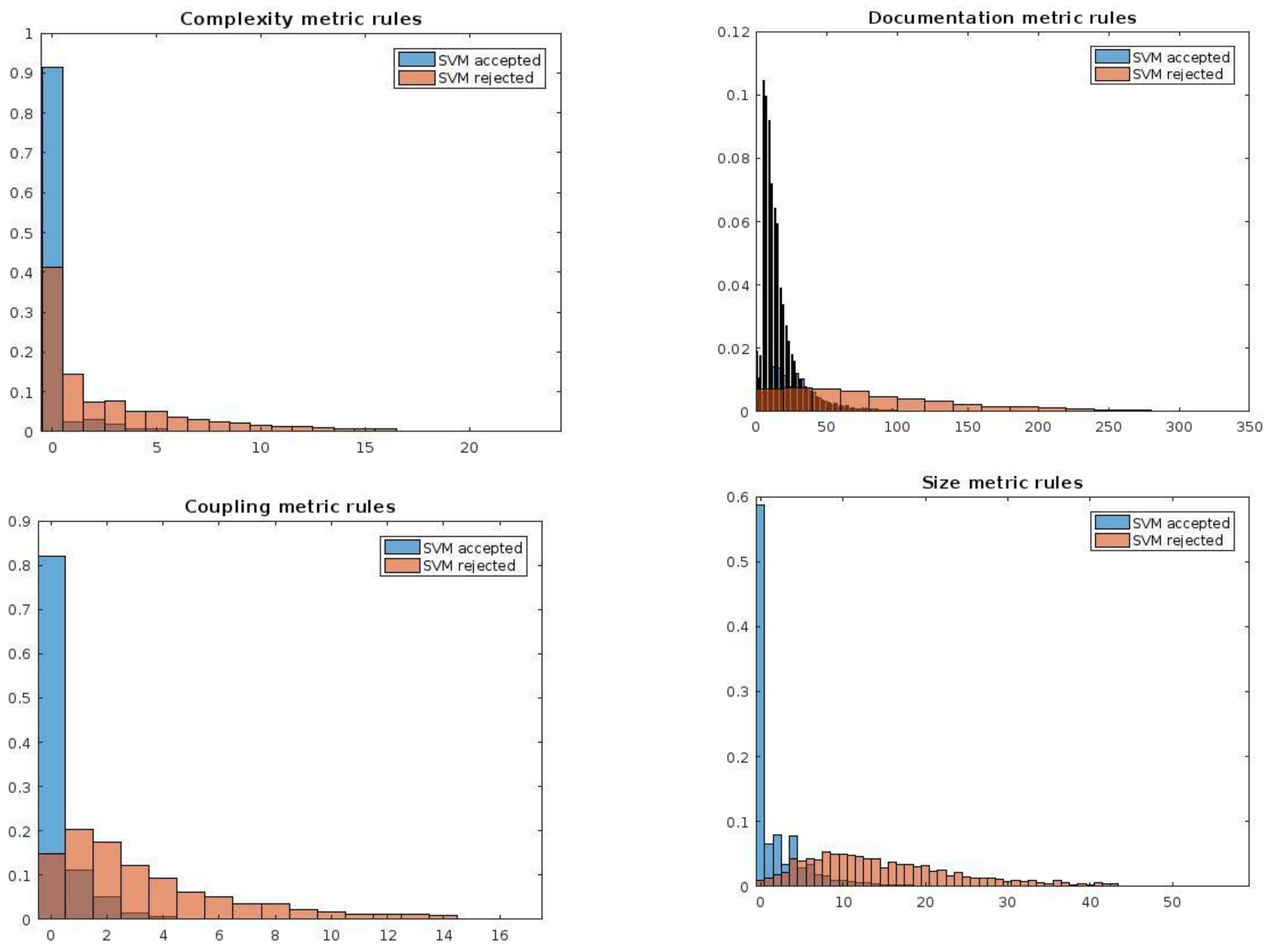


Fig. 12. Probability density functions of violation rules for SVM accepted and rejected data.

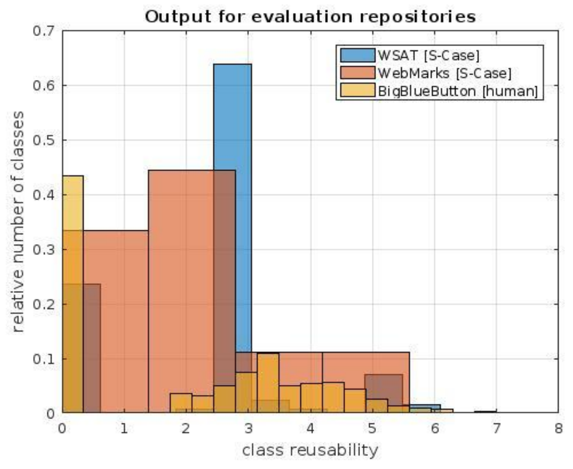


Fig. 13. Output of repositories used at evolution

VI. Conclusion and future work

A new approach to estimate the reusability of a class using static analysis metrics was proposed. The results can be generalized to the repository level by considering a weighted mean of the reusability of each class, based on its CBOI and NPM contributions. A direct inversion of the targetset formula would be preferable to this estimation, as it would relate directly to stars and forks and could serve as evaluation. It can only be inverted to a complicated function of the stars and forks with little intuitive meaning, which has been omitted in favor of the weighted mean.

The viability of this approach has been supported by a relatively thin evaluation, which however finds no evidence against it. The rejected classes show a tendency to violate rules, the regression errors are acceptable, and the hypothesis of small variation in reusability for the machine-generated code cannot be rejected.

Future work is needed to ameliorate the scorer network's performance. The autoencoder network that was employed in this work was not fully trained and should be pursued further.

An ensemble method featuring multiple models is also worth exploring, especially since the networks that were presented here were consistently underestimating the best-performing classes.

What might be of greater use however is improving the collected data. By gathering a greater diversity of repositories and not just the most highly rated a more general model could be achieved; also the improvement of the static analysis metrics by incorporating semantic meaning with deeper analysis of the Abstract Syntax Tree, such as the identification of cloned code segments with similar but not exactly the same function, is a broad field.

References

[1] IEEE Manuscript Templates for Conference Proceedings, online: http://www.ieee.org/conferences_events/conferences/publishing/templates.

[2] Kitchenham, Barbara, and Shari Lawrence Pfleeger. "Software quality: the elusive target [special issues section]." *IEEE software* 13.1 (1996): 12-21.

[3] "CISQ Code Quality Standarts." [Retrieved March 2017]. [Online]. Available: http://it-cisq.org/standards/

[4] Diamantopoulos, Themistoklis, Klearchos Thomopoulos, and Andreas Symeonidis. "QualBoa: reusability-aware recommendations of source code components." *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016.

[5] Zhong, Shi, Taghi M. Khoshgoftaar, and Naeem Seliya. "Unsupervised Learning for Expert-Based Software Quality Estimation." *HASE*. 2004.

[6] M. Papamichail, T. Diamantopoulos and A. Symeonidis, "User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics," *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Vienna, 2016, pp. 100-107.

[7] Padhy, Neelamadhab, Suresh Satapathy, and R. P. Singh. "Utility of an Object Oriented Reusability Metrics and Estimation Complexity." *Indian Journal of Science and Technology* 8.1 (2017).

[8] Ferreira, Kecia AM, et al. "Identifying thresholds for object-oriented software metrics." *Journal of Systems and Software* 85.2 (2012): 244-257.

[9] Cai T, Lyu MR, Wong KF, Wong M. ComPARE: A generic quality assessment environment for component-based software systems. InProceedings of the 2001 *International Symposium on Information Systems and Engineering 2001*.

[10] Ferreira, Kecia AM, et al. "Identifying thresholds for object-oriented software metrics." *Journal of Systems and Software* 85.2 (2012): 244-257.

[11] Kumar, Ajay. "Measuring Software reusability using SVM based classifier approach." *International Journal of Information Technology and Knowledge Management* 5.1 (2012): 205-209.

[12] Borges, Hudson, Andre Hora, and Marco Tulio Valente. "Predicting the popularity of github repositories." *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2016.

[13] Moré, Jorge J. "The Levenberg-Marquardt algorithm: implementation and theory." *Numerical analysis*. Springer Berlin Heidelberg, 1978. 105-116.

[14] Marquardt, Donald W. "An algorithm for least-squares estimation of nonlinear parameters." *Journal of the society for Industrial and Applied Mathematics* 11.2 (1963): 431-441.

[15] "S-Case: Software Done Simply." [Retrieved March 2017]. [Online]. Available: http://www.scasefp7.eu/