Code Evaluation

Brouzos Rafael
Electrical and Computer Engineering
Aristotele University of Thessaloniki
Thessaloniki, Greece
mprouzos@ece.auth.gr, rnm1816@gmail.com

***Abstract*— This assignment is about source code quality estimation with various classification methods. A metric is created and a lot of model in R to try to predict the quality of the source code from its static analysis metrics.**

Keywords—source code quality; reusability; (keywords separated by semicolon)

I. Introduction

Considering the development of the open source code repositories, nowadays, it is easy to find good or bad source code everywhere. There are also a lot of applications for automatic code generation and code testing.

There are many ways to share a project on public (or on your team) on internet and the open source applications are increasing with high rate. There are also a lot of code styles to develop a project and a lot of ways to analyze source code. The main way to evaluate a project is 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*.

Our main target is to evaluate each project based on its reusability but we will produce a various oriented for all of our estimations. We are going to explain each of the metric later in this paper.

II. Research Overview

We are going to create a model that predicts the quality of a source code based on its static metrics obtained from the static code testing [3].

We have a dataset, with a collection taken from gitHub that contains statistics from 137 source code repositories. All of them are linked with their Packages, Classes and Methods. We have the stars and the forks for each project and a big amount of static metrics for each package, method and class.

First of all we need to evaluate the source code with a metric (score function) for each part of one repository (method, package or class). Then we need to predict this score for every part of a repository from a model. This model should work for every method, class or package of a repository. Finally we will try to produce the final evaluation for the full repository based on its partial evaluations.

As we can see in paper [2] the stars and the forks of a repository are correlated. But it is also easy to understand that each of those metrics has its own meaning and we will combine both. So we have a correlation between the but we will use both cause everyone of them has its special information for the code evaluation.

III. Evaluation

A. Evaluation Methodology

Before developing our model we need a metric for the source code evaluation. We will try to approach a more complete metric that contains the most of criteria we have given above.

The first assumption made is that the strongest metric for reusability, portability and compatibility we have will be based on the forks of the project. On the other hand, the most complete metric we have are the stars of the project that shows more efficient in reliability, performance and functional suitability evaluation.

We should combine these 2 metric to a hybrid metric that is also able to evaluate a part of a repository. We need to share the stars and the forks of the repository to each method, class or package that belongs to this repository.

It is important to keep in mind the size of the part of a repository. In this work we will assume that the stars and forks of each repository are shared to its parts depending on the size of the part and the size of the repository.

So we will use a metric with 2 main factors, one for the forks and one for the stars. But we will share the stars to every part of a repository. Finally our metric will have 4 factors.

$$SCORE = \frac{stars \cdot LOC}{sum\ (LOC)} + \frac{stars}{parts} + \frac{forks \cdot LOC}{sum\ (LOC)} + \frac{forks}{parts} \qquad (1)$$

Equation (1) presents the first score function for our evaluation. Stars and forks are the stars and the forks for the whole repository that the evaluated part belongs to. Part, are the similar parts to the evaluated part that they are also belong to the same repository. LOC is the Lines of code statistic for the current evaluated part. Sum(LOC) is the summary of the Lines of code for every similar part in the same repository.

For example, for evaluating a Class of a repository we take the sum(LOC) as the summary of LOC of every Class in the current repository, the LOC as the lines of code of the evaluated Class, the part as the number of Classes in the current same repository, the forks and the stars of the whole repository.

The 1st and the 3rd factors share the stars and the forks depending on the size of the part. The 2nd and the 4th factors share the stars and the forks depending on the number of parts is broken the repository.

The whole assumption is based on a really simple way to share the stars and the forks. We assume that the stars and the forks are shared evenly to every part of a repository and we just give an extra emphasis to the biggest parts as they make more things.

The factors containing stars are oriented better to the reliability function suitability and performance awards. The factors containing forks are oriented better to the compatibility, portability, maintainability and usability awards.

Finally we have a simple and complete score function for the evaluation of the source code object. Our estimation is strongly based to its simplicity.

B. Quality and static metrics

In this moment it is interesting to present the diagrams that show the relationship of some static metrics with the stars or the forks of some randomly chosen repositories. These will help as understand why we have chosen this metric.

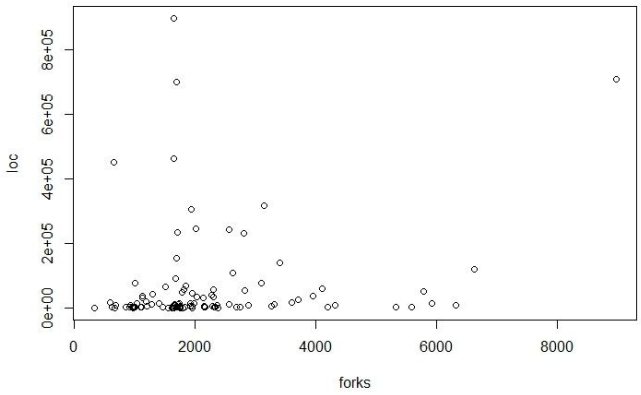

Figure 1. The relationship between forks and LOC. We notice that normally for very big LOC we have bad fork values. But in low Loc side we have weak relationship of these 2 variables. That makes sense cause bigger code mean less maintainability and reusability.

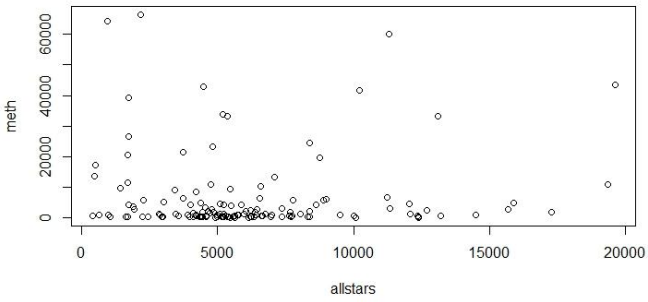

Figure 2 shows the relationship between stars and number of methods of a repository. This graph shows that the correlation is not so strong and the stars are shared almost evenly.

More experiments like above have made to finally help find the metric for the code evaluation but those experiment are not relative to this paper. Finally our assumption for the evenly shared stars and forks to all its parts seems to be representative. The biggest parts (with the most lines of code) are punished in our score function from the low star and fork rate of their repositories so we don't need to interfere on this.

C. Quality and static metrics

Our metric for package (example experiment) evaluation has the following statistics:

Max:	16393.55
Min:	1.561677
Average:	158.1342
Variance:	309132.8

It is easy to see that this metric has a very big range, very big variance and some very high or low values, far away from our average. So we will use a smooth factor to limit the score and to fix the statistic of the metric. We will use ln function to keep the Monotonicity. Actually we don't have score values under 1 so our new function will be positive and this is an advantage too. The final metric is:

$$\text{SCORE} = \ln\left(\frac{stars \cdot LOC}{sum\ (LOC)} + \frac{stars}{parts} + \frac{forks \cdot LOC}{sum\ (LOC)} + \frac{forks}{parts}\right) \quad (2)$$

With all values as explained above, offset = 0 and the smooth factor placed. The new statistics of the score function (metric) are:

Max:	9.704643
Min:	0.44576
Average:	3.353869
Variance:	3.444365

We will mention here that we have plenty of negative values in our metric result for method evaluation (result inside

the logarithm is lower than 1) and some for class evaluation too. So we could easily override this problem adding an offset in formula (2). Min value of the metric is about -3.15 for method evaluation and we could use an offset of 3.5. If we will add 3.5 to every metric result the statistics will be really similar with the statistics of the package evaluation score function. But our models are going to work for negative score values too, just with some changes in our secondary parameters. This is helpful because we can keep the same evaluation logic for each case. So we are not going to use any offset.

Another possible solution to the difference between

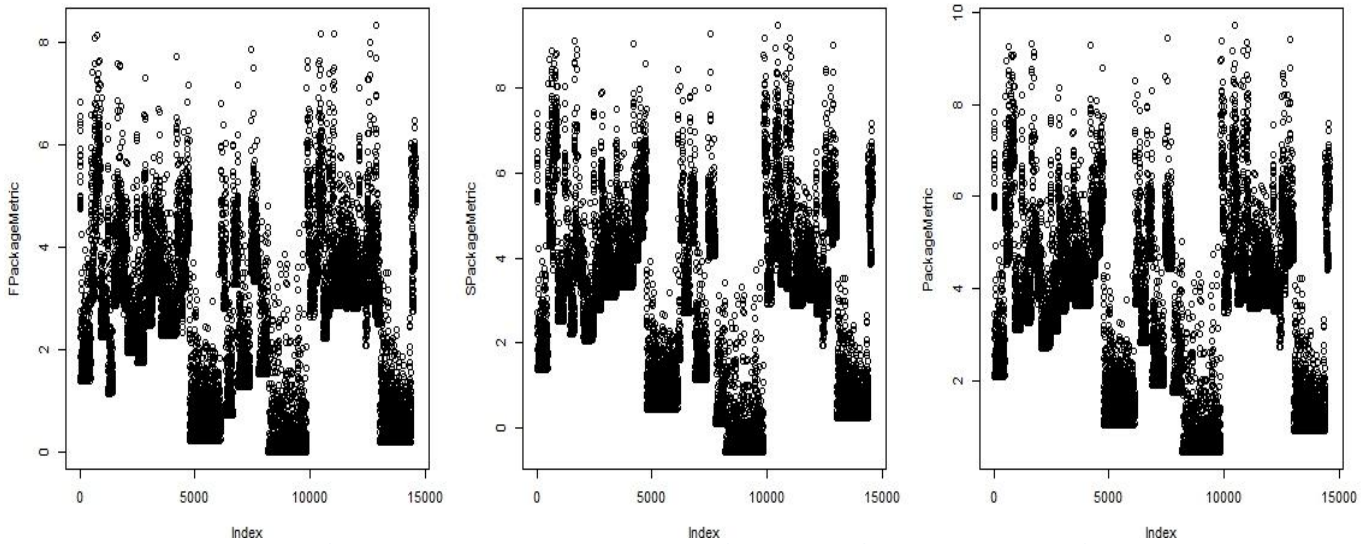

Figure 3. The Metric results for the package evaluation only with a) Fork factors, b) Star factors and with c) every factor.

method, class and package score metrics would be a scale of our metric. We will use a scale later and it could drive our results to similar statistics but the resolution will be decreased (from [0,10] to [-3,3] for example). So we will present a scaled metric as an alternative solution.

Before ending our conversation about the metric we will check something last. There is a correlation between Stars and Forks as we mentioned. Why don't we take only the fork factors or only the star factors in the score's formula?

If we take the score function (2), every time we will take the best evaluation of the code, because we give extra points for forks and extra points for stars. This will do our metric results more competitive. This means that with formula (2), every source code will raise its evaluation (for example from the case that we had only star factors). This is clear in figure 3 above. This figure shows that the main job is the same in every metric. But with the full metric (forks and stars) we get the highest possible evaluation every time. The full metric looks like a combination of both behaviors and this turns the results to better statistics. We have better evaluation for more source code files and we can evaluate more strictly later. We also avoid negative values (we have a small quantity of negative values if we don't take the full metric). The metric's mean and max values are close in every case. Formula (2) suits to our needs but we have a disadvantage. We cannot reproduce the number of Stars and Forks only from our metric. This is because we cannot take 2 values from one equation. However we are going to override this problem. We will turn the Stars and the Forks from the beginning to a score list and we will work only with this list as a class value.

IV. System Design

A. System Overview

Our goal is to guess the score of a source code file with its static analysis metrics. We can assume that we divide the source code files in 2 categories, the Very Low quality ones and the others. As we can see in this paper [2] we can use the one class classifier for this reason before our main classification. We will design a system with 2 Steps, one categorical and one numeric classification. We will use the same components of the static analysis Metrics for both Steps.

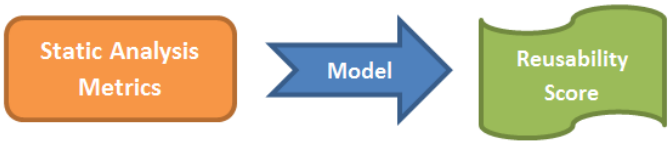

Figure 4. Main idea of classification.

Fig. 1 shows a very general overview of our system. We are going to check closer our Model in this section, as we have seen about the 2 metrics before.

We will use a categorical class variable in the first step to predict if a source code file is a very low quality one. A file is considered as a very low quality file if its score is lower than a threshold, decided from the quantiles of our Score function. We decide to eliminate the 5-10% of the source code files before Step 2. So we can evaluate the rest of files more

strictly. The threshold is calculated as we explained and it is different from package to class and method evaluation.

We have, now, to evaluate the rest of the files in Step2 and we will use our score metric as a numeric class variable here. We want to create a classifier that predicts the score metric of each file, for each type of files (packages are our main type but the system works for all the types).

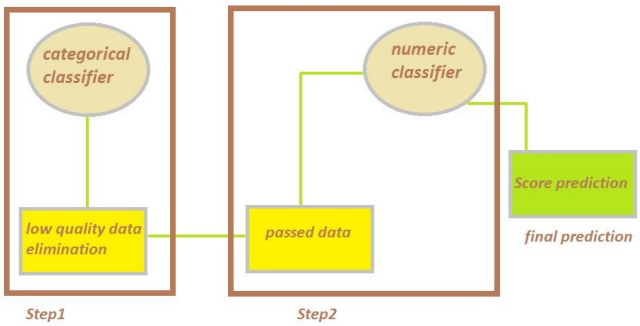

Figure 4. System overview, with its 2 steps until the final prediction.

In the Figure 4 above you can see the overview of the system, its 2 classifiers and the data way until the final prediction of the score function.

B. Data Preprocessing

We need to create a dataset and a testset. We get randomly a number of projects and we mix their parts (packages, methods classes) to get a dataset and a testset randomly again from our project. Every set contains records with their Metric. The records are not depended on the project index in the original dataset.

First, we check for duplicates in our dataset in the attributes of methods, classes and packages. We do the same for missing values everywhere. These 2 checks are easy in R and we don't need to analyze more on this paper. Our dataset is clean from duplicates and missing values. We also make a check for possible human faults in our dataset, checking that every LOC (lines of code) value is positive number.

If we check our metric results (for example) for every package for outliers we will notice that we get outliers only close to 10 (maximum value of our metric is 9.704643) after the smooth factor application. Those values are the best rated source code files but maybe we need to drop them. For methods and classes we get more outliers and we can replace their values with another metric (f.e. the mean) or simply drop them because our dataset is very big. We need also to eliminate the very bad rated source code files as a first step to evaluate more strictly in the next step. As we can see in this paper [2] we can use the one class classifier for this reason before our main classification. So there is no need to do it now (in the preprocess section).

Our data have plenty of constant components that we need to drop because we are going to use PCA with scale for better estimation. We apply PCA to the non-constant numeric data and we can see that the information tends to be in the first 15 to 25 components. We will make one assumption here for method, class and package evaluation and we get 18 principal components for each case. We can see the information of those 18 components in each case in the table 1 below.

In the case of classes we have about 9% lost information but we are going to continue like this because we want the same PCA dimension for all the parts.

Finally, we can start our system, after dropping outliers. In the case of Packages (our main case in this paper) we don't have big problem with outliers. For the other types it is recommended to clean the dataset from outliers.

TABLE 1

PCA and information (18 components)		
Parts	***Info. Obtained %***	***Info. Lost %***
Methods	98.876	1.124
Packages	99.161	0.839
Classes	90.911	9.089

C. Model Construction

For Step1 we created a binary class variable ("ok" or "Low") and we use a **Decision Tree** and a **SVM** classifier for this reason to predict and eliminate Low quality files from Step2. We are going to check their results in the results sections. We just mention here that accuracy is not our main goal in this step. We just want to eliminate Low quality files without false eliminations (of good scored files). So we care about higher precision and fewer false eliminations. We can reach these results by decreasing accuracy and recall.

For the **Decision Tree** model we created a model and we pruned this model until the CP of the model reached its min value. For the **SVM** model we made a parametric analysis on gamma values in a script separately and we obtained very similar values of gamma for every type of files and close to 0.4.

In each case we have evaluated our models by their recall values and their false eliminations. The **Tree** tends to give better results for Methods and Classes and the **SVM** tends to give better results for Packages. Our main work (preprocess is "Package oriented") is based on Packages so in the most cases we use the **SVM** classifier for step1.

For step2 we will return back to our score metric class variable. We want a classifier that predicts a score value for the files that passed from step1. We will use a **Decision Tree**, a **KNN**, 2 **ANNs** and a **SVM** classifier for this reason and we will compare their results.

For each case of the step2 classifiers we have done a parametric analysis and we obtained the best parameters for the original metric and the scaled (with center) metric. We have based our models on decreasing the MSE metric. MSE is

the main evaluation metric for our test set classification results that we will compare later.

We develop a pruned (to minimize CP) **Tree** model and as we know these kind of models help to evaluate the other types better. **Trees** are easily developed without hard tuning and they perform usually better than the half of the best case scenario. In our case the model act like usually and gives a prediction ratio close to the good side.

We also create 2 **ANN** models one from package "nnet" and one from package "neuralnet" in R. They have different parameters and they perform differently depending on the real class values. We have increased the performance of the 1st using 2 hidden neurons (1 hidden layer) and decay value (after caret tuning). For the other we tried to find values of parameters without any hidden layers because it was difficult to converge. Finally the 1st is one of the best classifiers we have developed as we will see in the result section later.

Another classifier used is a **KNN** model. We finally decided that for k=1 the model works better but still it doesn't look to fit our problem's needs.

For the end, we tried a **SVM** classifier after a parametric analysis on gamma values. We decide to use "radial" kernel and gamma close to 1.2 for every file type, because those values give the less MSE in our results. This model is one of the best in our results and especially for scaled Metric we have a good MSE. But the results are not always the same for every execution and sometimes we have a small variance in our MSE.

We are going now to see the results of each classifier and for each Step. In the most of cases we tuned our models to fit better in Package evaluation and after the **SVM** step1 elimination. But after tests we have seen that the models work normally and very similar for method and classes and also after **Tree** eliminations in step1.

V. Results

Firstly, we will present the Step 1 evaluation results and after that we will continue with Step 2. We will start with Package evaluation as this is our main work in this paper.

The result given are made from an experiment with 1000 randomly taken files of each type and from various projects as a training set and 279 randomly taken (different) files of each type and from various projects as a testing set. All the metric are calculated by the prediction on the testing set to avoid overfiting in our models.

DECISION TREE

	precision1	**recall1**	**f11**
Low	0.8260870	0.7307692	0.7755102
Ok	0.9726562	0.9841897	0.9783890

SVM

	precision2	**recall2**	**f12**
Low	0.750000	0.5769231	0.6521739
Ok	0.957529	0.9802372	0.9687500

As we can see we have high precision for both classifiers and we can say that we decreased the system's recall to avoid false eliminations. **SVM** tends to eliminate less files and avoids false eliminations better. The **TREE** goes better in the method and class evaluation.

After applying the eliminations from the **SVM** predictor we can continue with the Step 2 results.

TABLE 2 (UNSCALED METRIC)

Model	**Model's MSE performance (best parameters)**		
	Packages	***Methods***	***Classes***
ANN1	2.72	1.28	1.84
ANN2	1.87	1.3	1.67
TREE	2.15	1.45	1.87
KNN	2.88	1.86	2.23
SVM	1.95	1.23	1.61

TABLE 3 (SCALED METRIC)

Model	**Model's MSE performance (best parameters)**		
	Packages	***Methods***	***Classes***
ANN1	0.9	0.84	0.84
ANN2	0.69	0.76	0.72
TREE	0.75	0.93	0.86
KNN	0.95	1.14	1.01
SVM	0.7	0.86	0.73

Table 2 presents the results for unscaled metric and Table 3 for scaled. As we can see the MSE seems to be really better in the second case but actually the difference is not so big. In the first case an error is not so weighted because we score to a 10 ranged scale. In the second case we score to a close 5 scale and we should weight more the errors.

In every case we get the idea of the classification results for each type of file.

VI. Conclusions

As we have seen above **SVM** and **ANN** classifiers tends to give us the best results and TREES reached our estimation. All the models are tuned to reach the best MSE possible and could be developed with the given Source code. The Evaluations of the models are also available in a script. The code is well

commented to guide the user. The data maker and the preprocess are first, later the Binary class maker and for the finish are the models and the evaluation.

We tried also to develop an idea of clustering but it couldn't happen inside the deadline and we live this kind of code evaluation as a future work.

References

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

[2] 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.

[3] https://en.wikipedia.org/wiki/Static_program_analysis

[4] Vera Barstad, Morten Goodwin, Terje Gjøsæter "Predicting Source Code Quality with Static Analysis and Machine Learning" Faculty of Engineering and Science, University of Agder, Serviceboks 509, NO-4898 Grimstad, Norway, *NIK-2014 conference*