Applying readability evaluation techniques for software projects by using data mining on open source repositories.

Kagiafas Nikolaos
Kyparissis Odysseas
Electrical and Computer Engineering
Aristotle University of Thessaloniki
Thessaloniki, Greece
kagiafas@ece.auth.gr, odyskypa@ece.auth.gr

***Abstract*— Many researches have been conducted with a view to finding a good method of assigning grounds of readability in software projects. Readability or how easily someone can read a piece of code and understand it is not an arithmetically measurable factor but an abstract one. Previous papers and a continuously increasing number of programmers claim that code readability strongly correlates with its maintainability. Written pieces of codes can be reused and modified with less effort, when someone understands them quickly. In this way, we present a new method of informing programmers about the readability level of their code. A lot of previous works on this topic try to train models that are based on human assignments of readability for some code snippets, in order to predict the readability factor of other codes. We have a dataset available from open source repositories, which includes approximately 1 million Java functions. These codes have been selected in concern with the stars that have gained and the number of forks on Github environment. This huge number of codes cannot be read and evaluated by humans due to the tremendous effort it needs. For this reason, we use a number of metrics and violation indexes that helps us assign a level of readability in each and every code of the dataset.**

I. Introduction

Readability of code is an essential attribute that concerns software developers [1],[2],[3]. The more readable a code is, the more maintainable and reusable it can be. Imagine a group of programmers who work for the same company and need to cooperate to design a software application. For the best of their communication, it is advisable to write codes that can be easily understood. In this way, a person in the group aiming to expand or improve a code written by a previous coworker of his will perform faster and more efficiently on the project.

One major issue with readability evaluation techniques is the struggle to measure something that holds personal perception [1]. Moreover, it is extremely difficult to collect the personal point of view of programmers for a large amount of programming functions. It would require large-scale surveys, multiple human raters and professional statistical analysis, in order to obtain important information from them [1]. Nonetheless, there are some measures, which can provide us with useful information if they are used correctly. These metrics can be divided into smaller groups, which include complexity, coupling, documentation and size measures of the written code. In addition to these metrics, there are some other tools, which can help us evaluate a piece of code. The one we used is PMD code violations, which count Java code errors and bugs that belong to categories, such as basic, brace, clone implementation, controversial, design, finalizer rules, etc[5].

We believe that it is important to split the dataset into small and big codes, on the grounds that the process of readability assignment in these two groups differs significantly. For instance, smaller functions with one to six lines are probably setters and getters and thus it is easy to be read and understood. On the contrary, bigger codes with 7 to 200 lines tend to be more complex, so we need to take into consideration a larger number of metrics, in order to label a code as more or less readable. According to this assumption, we selected different features for these two groups.

In the following sections, we present our methodology, the theoretical basis, the design and the structure of our system, the models we used and trained, as well as their evaluation and performance summary. Finally, we mention some changes that can be made, so that the system becomes more efficient and robust.

II. Research Overview

As it was stated above, the available dataset we make use of, consists of 1004589 Java functions with 47 metrics and a total of 193 rule violations calculated on them. First of all, for the sake of right data processing and analysis, we omit the records that contain missing values in some of their columns. The remaining 942736 functions need to be split into the two previously mentioned groups. Finding the appropriate percentile values, we see that a percentage of 57% of the dataset are codes with 6 or less total lines (TLOC) and between 57% and 99.9% exist functions with 7 to 200 total lines (TLOC). We should mention that we do not take into consideration bigger codes, due to the fact that a function with 1000 or more lines is over-complicated for the purpose of our research. These functions are considered as outliers. So for our remaining, gathered data, we begin with a simple, yet general research question:

Research Question 1: How can we select the best combination of metrics, in terms of readability, for each group separately?

The answer to that question can be extracted from the use of PCA algorithm into the two groups. For big codes (TLOC 7-200) we found the first eight principal components (86.5% of the information contained in the dataset) and we checked in their eigenvectors which metrics have the most important participation. In this way, the most critical metrics are:

1. LLOC: Logical Lines of Code
2. TCD: Total Comment Density
3. CLLC: Clone Logical Line Coverage
4. CI: Clone Instances
5. NL: Nesting Level
6. HTRP: Halstead Time Required to Program
7. NUMPAR: Number of Parameters
8. NII: Number of Incoming Invocations

So, we keep these metrics for our analysis and then we compute the correlation matrix, shown in Figure 1, between all the features in the group of big codes. Our goal is to avoid the use of metrics, strongly correlated with each other, in view of getting rid of overlapping information. As it is presented in Figure 1, CLLC and CI are closely connected. For this reason, we decide to replace CI with McCC (McCabe's Cyclomatic Complexity), due to the fact that complexity is frequently found in bigger codes. Thus, following this expert-based approach, we believe that this variable will provide us with useful information for our system design. Furthermore, we are trying to use as small number of features as possible aiming to avoid the curse of dimensionality, so we need to choose either NUMPAR or NII. We pick NII, because we believe that in bigger codes this attribute is more significant and it does not correlate with any other feature. Besides that, we are calculating the sum of violations per line of code (VPTLOC) for each function and use it as our last attribute.

Concerning the small codes, we observe that the previous method is not so effective for this group, because many of the extracted principal components are not so essential. The majority of them are setters and getters with two to four lines (TLOC). They are so small that understandability in them is much easier and can be predicted by using only the total comment density (TCD) of the given snippet. The higher the density, the more readable a small code is. Apart from this, similarly to the previous group, we use as an additional attribute the overall violations per line of code (VPTLOC). In this way, we have an overall view of the snippet's quality. As a consequence, the assignment of readability level is much more effective.

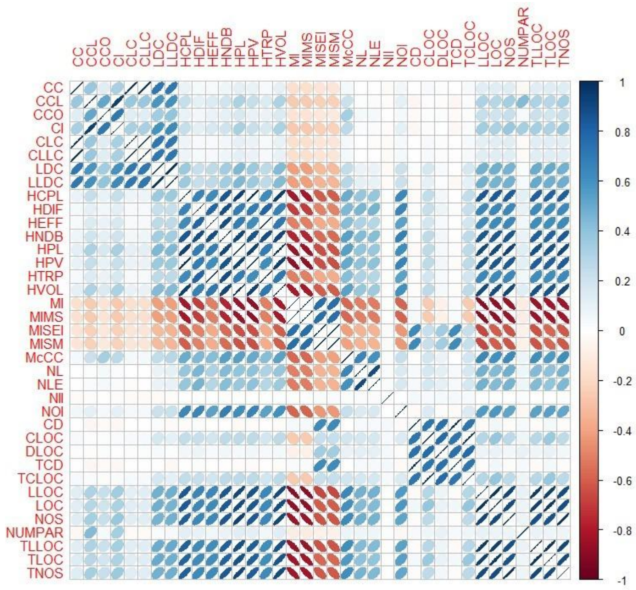

Fig. 1: Correlation matrix display using a set of little ellipses that provide a generalized depiction of the strength and sign of the correlation between the metrics of the big codes dataset.

After the initial processing and selection of the useful metrics, we continue with the normalization of the data. Data normalization is a most important technique for better clustering quality. In this stage of the data manipulation, it is crucial that we use the most appropriate algorithm for clustering among a set of multiple known ones. K-means algorithm and three types of hierarchical clustering methods take part into this type of comparison. To achieve that, we sample the dataset without replacement, picking up randomly 25000 functions. This is unavoidable, because of the calculation of distance matrices on the purpose of computing the silhouette value. However, being able to compare the values of silhouette requires the beforehand decision on the number of clusters, which are going to be created.

Research Question 2: Which is the optimal number of clusters and which is the most appropriate clustering method?

To begin with the answer of this question, we should mention that we used single, average, complete hierarchical clustering and k-means. A method of selecting the best number of clusters using k-means is doing the SSE (Sum of Squares Error) plot, shown in Figure 2, for values of K between 2 and 10. The mathematical formula for SSE is:

$$\mathrm{SSE} = \sum_{i=1}^{K} \sum_{x \in C_i} \|x - m_i\|^2$$

where K is the number of clusters, C_i the i-th cluster with centroid m_i and x the data points.

The equivalent process is followed in hierarchical clustering by calculating the mean value of silhouette for each and every value of K (K = 2…7). Apart from the fact that the value of silhouette is smaller in hierarchical methods, we observe, by doing the silhouette plot that partition of the data is not so optimal, because, for 3 clusters, for example, the algorithm attaches the majority of the data in one cluster, some other in the second one and only 1 or 2 of them in the third one. For this reason, we decide to use k-means for the data clustering with k = 3, as shown in Figure 2. The mean silhouette value using this method is 0.61 for the sampled data. We followed the same process for small codes and we have similar results, as shown in Figure 3. It is important to mention that we use a special edition of k-means, the k-means++, to make sure that the optimal centers are selected.

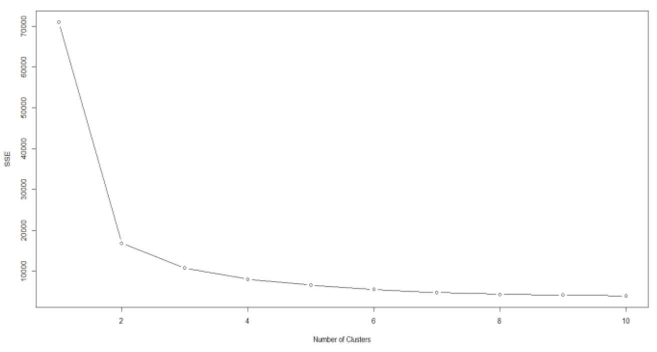

Fig. 2: Plot of SSE which help us decide the number of clusters using k-means algorithm for big codes.

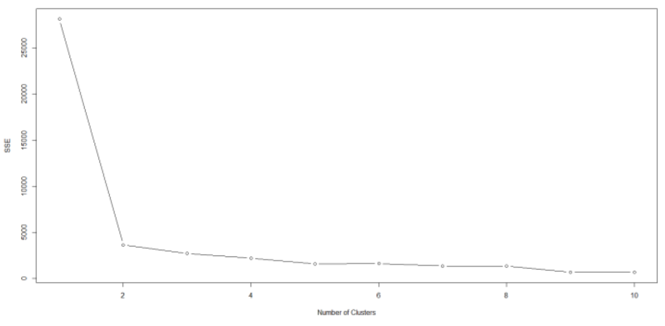

Fig. 3: Plot of SSE which help us decide the number of clusters using k-means algorithm for small codes.

After having selected the clustering algorithm and the optimal number of clusters, we proceed with their creation. Then, we have to assign a level of readability to each one of them.

Research Question 3: How can we apply readability levels in each one cluster that has been created?

At first, we compute the mean value of every feature for all the codes in each individual cluster. We assign the "High" readability level to the cluster that has the smallest mean value in the violations per total lines of codes (VPTLOC) feature. "Low" level is given to the cluster with the biggest mean value of VPTLOC and "Medium" to the remaining one. At that moment, we check if the other metrics in each cluster follow the same pattern as the pre-assigned readability level. For instance, we expect that the mean value of TCD (Total Comment Density) for the cluster with level "High" will not be smaller than the mean value of TCD for clusters with levels "Medium" and "Low". Below, we present two tables for each one of the data segments (small and big), in which it is depicted which features follow the same logic.

	"Low"	**"Medium"**	**"High"**
CLLC	✔	✔	✔
HTRP	✔	✔	✔
LLOC	✔	✔	✔
McCC	✔	×	×
NII	✔	✔	✔
NL	×	✔	×
TCD	×	×	✔

Table 1: Checking if metrics are following the pattern in each cluster of big codes data segment.

	"Low"	**"Medium"**	**"High"**
TCD	✔	✔	✔

Table 2: Checking the pattern of TCD in each cluster of small codes data segment.

As shown in Table 1, we can verify that the majority of the metrics follow the desired pattern in each one of the clusters. Only NL (Nesting Level), McCC (McCabe's Cyclomatic Complexity) and TCD (Total Comment Density) present a slight difference from the normal behavior. That is quite inevitable, because it is very difficult for all the characteristics to follow the ideal pattern. What is more, as it is demonstrated in Table 2, TCD's pattern is fully compatible with the logic that we want it to follow.

When all the above process is completed, we can continue with the training of all the appropriate models as well as their evaluation and performance summary. We are now ready to begin with the design and implementation of our system.

III. System Design

A. *System Overview*

To begin with, as shown in Figure 4, we take into consideration some specific metrics and the sum of overall violations generated by PMD source code analyzer [5]. We use these features to cluster the initial data into different groups based on their readability, as mentioned before. Then, we are constructing classification models, which we are evaluating and comparing, in order to test their performance and reliability.

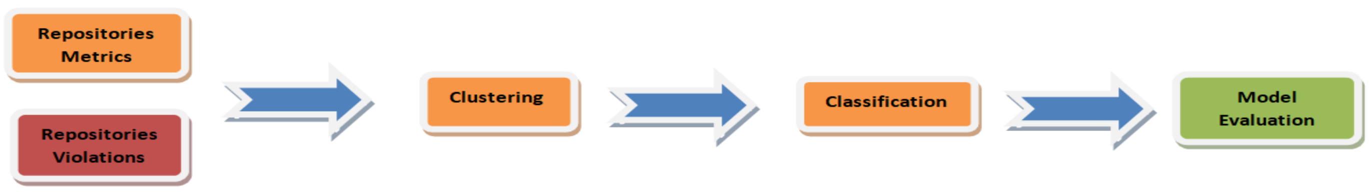

Fig. 4: System overview and structure.

B. *Data Preprocessing*

Data preprocessing includes four stages:

- Removing missing (NA) values from the dataset.
- Splitting the data into two groups, small and big functions.
- Generating a new metric called "violations per total lines of code" (VPTLOC) for each code of the given dataset.
- Data normalization.

C. *Clustering*

Clustering process consists of the following four stages:

- Selection of the optimal number of clusters.
- Comparing the methods by using silhouette value.
- Selecting the most suitable method, which is k-means++.
- Assigning levels of readability to each cluster.

D. *Model Construction*

Reaching the classification stage of our research, we made use of several models, so that we test them in terms of predictability and performance. Specifically, we began by training a decision tree, we continued with a naive Bayes model with Laplace Smoothing, a neural network without any hidden layers and finally knn algorithm. We also tried to apply SVM technique, but we did not manage to reach to a final result, due to the large number of dimensions in the big code dataset.

Using a decision tree as an initial approach is a quite effective technique so far, due to the high accuracy that we receive during its testing stage. However, we cannot be sure for its overall usability, on the grounds that there is the danger of overtraining. Nevertheless, it is a very simple and fast algorithm that gives satisfying results. Under no circumstances should it be ignored.

With regard to the Bayesian model, we used the data as they are initially, without normalizing them, otherwise the algorithm did not converge. We used Laplace smoothing, in order to avoid zero probabilities that may have been occurred in our dataset. Those probabilities might not have been zero in reality but negligible and thus the Bayesian model would not have been executed properly.

The third model we constructed is a neural network with three outputs, for the readability levels, and without any hidden layers, as shown in Figure 5 and in Figure 6. Neural networks execute only arithmetical operations, so categorical values cannot be used during their training procedure. To overcome this obstacle, we must transform our class attribute, which consists of categorical values, such as "Low", "Medium", "High". To achieve that we create three vectors, which correspond to each one of the pre-mentioned levels and they replace the class attribute value. For example, a function that has been characterized as low readable will be reassigned with the value of "1" only in the low readability vector and with "0" in medium and high ones. After the model has been trained, the predicted level of readability for a new record is the column with the maximum value. For instance, if the low readability column of this new function has the maximum value, it means that our model characterizes this code as low readable.

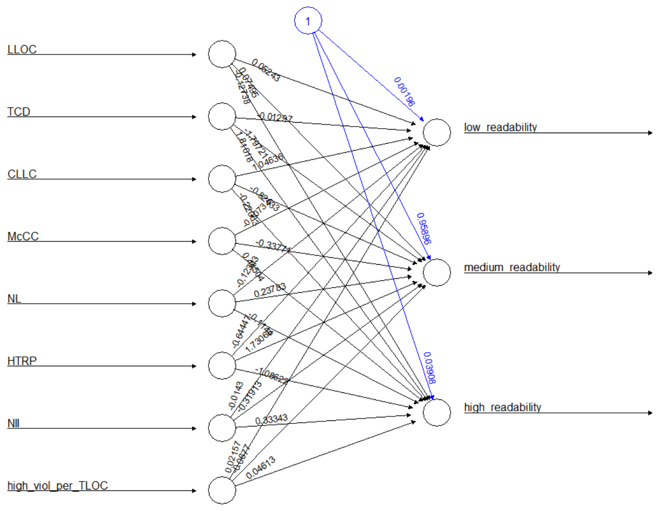

Fig. 5: The designed and trained neural network for the big codes data segment.

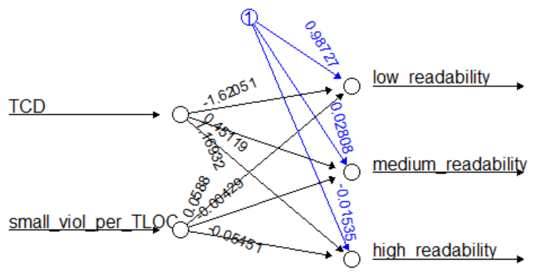

Fig.6: The designed and trained neural network for the small codes data segment.

Model	Accuracy	Low Precision	Low Recall	Low F-measure	Medium Precision	Medium Recall	Medium F-measure	High Precision	High Recall	High F-measure
Decision Tree	0.999038	0.998	0.998	0.998	0.999	1	1	0.999	0.998	0.999
Naïve-Bayes	0.957165	0.905	1	0.95	0.996	0.943	0.968	0.899	0.955	0.926
Neural Network	0.957863	0.983	0.976	0.98	0.944	0.996	0.97	0.985	0.79	0.877
Knn	0.997541	0.998	0.998	0.98	0.999	0.998	0.97	0.993	0.996	0.877

Table 3: Performance metrics of all models for big codes.

Finally, the last model, which has been implemented, is the one that uses the knn (k-nearest neighbors) algorithm. In this method, data normalization is obligatory for the proper model execution and prediction of the class attribute for future entries. A crucial problem that we dealt with, concerns the large number of ties which occurred during the prediction phase of the model. That's why we used seven neighbors for the big codes, instead of two or three. However, we did not manage to overcome the above mentioned difficulty in the small codes data segment.

IV. Evaluation

A. Evaluation Methodology

After the completion of the models' construction process, we draw from a set of performance measures to evaluate them. Specifically, we use accuracy, precision, recall, f-measure and we have computed the confusion matrix for each and every model.

Some concise definitions of these performance metrics are presented below:

- Accuracy: it is the number of correct predictions made divided by the total number of predictions, multiplied by 100 to turn it into a percentage.
- Precision: it is the fraction of relevant instances among the retrieved instances.
- Recall: it is the fraction of the total amount of relevant instances that were actually retrieved.
- F-measure: it is the weighted harmonic mean of the precision and recall metrics.

Before we calculate the performance metrics, we shuffle the initial data and we split them into a training and a testing set. We follow this technique initially, because we want some measures for our metrics quickly and then, for better results, we execute 10-fold cross validation for more sophisticated model evaluation and comparison.

B. Evaluation Results

The evaluation methodology we mentioned in the previous section was implemented for each one of the models individually and the results are presented in Table 3. The results for small codes are quite similar and for this reason they can be omitted.

As we can observe from this table, decision tree model offers the best performance for all the metrics we have used. K nearest neighbors (Knn) algorithm races up in the second place of our model comparison, with highly satisfying results resembling those of the tree model. Although neural network and naive Bayes offer slightly worse prediction ability than the others, their performance cannot be ignored.

Nonetheless, following this evaluation technique for our model comparison and selection is not adequate. We should make a more complex partition of the dataset, in order to select training and testing data. If we implement this selection several times for different segments of the dataset, we can reach to some safer conclusions for our system performance. In this way, using 10-fold cross validation method can help us overcome this struggle.

Adopting this procedure in each one of the models, we extract mean values of accuracy for 10 different partitions of the dataset:

- Decision Tree: 0.998708
- Naive Bayes: 0.957442
- Neural Network: 0.95400
- Knn: 0.997855

At this point, a very significant question comes to mind: What can we do to enhance the performance and the prediction ability of our system? The answer to that question is presented in the next section.

V. System improvement

Despite the highly satisfying results, the fear of unsuccessful generalization of our system remains. To eliminate this fear there are some actions that can be made, which can lead to its enhancement.

First of all, selecting different repository metrics during the stage of data processing would be a vital factor for system improvement. Choosing different metrics than before, with determinant participation in the first principal components, without them being correlated, could help us achieve our goal. What is more, in our approach we took into consideration all of the violations that PMD source code analyzer generates[5]. For a more targeted confrontation of the readability assignment problem in a code segment, we could pick only some of the overall violations more wisely. For example, maybe some violations occur more frequently in small codes than in big ones and vice versa. In addition, a more professional parameter tuning of the models would lead to better and more precise results. Last but not least, making a survey in a group of programmers for assigning readability levels in some code segments may as well help us introduce to our model human perception of readability.

VI. CONCLUSIONS

In a nutshell, concerning code readability, as an abstract factor as it may seem, there are ways to characterize a code automatically, in terms of it, without any human intervention. In previous papers and surveys on this topic, the whole analysis was based from its beginning on human rating, with a view to classifying a given code snippet as more or less readable. From the above mentioned approach, a new system could be developed, so that it informs programmers instantly about how easily other coworkers of theirs can understand the code they are writing. In this way, programmers will be able to write more readable and more comprehensible codes in the future, which will lead to more maintainable and reusable programs. Furthermore, the results in which we reached were based on more technical metrics than human-perceptive ones, so their validity is still unclear. However, we do believe that a more theoretically well-founded model could be constructed, if more human-subject data were available, in order to make the algorithm resemble the human way of thinking.

References

[1] H. A. D. P. Posnett Daryl, A Simpler Model of Software Readability, Waikiki, Honolulu, HI, USA: ACM, 2011.

[2] W. W. R. Buse Raymond P. L., Learning a Metric for Code Readability, Piscataway, NJ, USA: IEEE Press, 2010.

[3] J. Dorn, A General Software Readability Model.

[4] M. L.-V. D. P. R. O. S. Scalabrino, Improving code readability models with textual features, 2016.

[5] «PMD,» [Ηλεκτρονικό]. Available: https://pmd.github.io/pmd-6.0.0/. [Πρόσβαση 23 February 2020].