Using the Normalized Levenshtein Distance to Analyze Relationship between Faults and Local Variables with Confusing Names: A further Investigation

Carmine Gravino⋆ Alessandra Orsi⋆ Michele Risi⋆

⋆ Department of Computer Science
University of Salerno, Italy
{gravino, mrisi}@unisa.it

Abstract

This paper exploits further uses of NLD (Normalized Levenshtein Distance), proposed in a recent study, to quantify the level of confusion of variables with the aim of verifying if they can provide indications about the presence of faults. We provide further evidence that fault prediction models based on the considered NLD measures can provide accurate estimations.

1 Introduction

This paper presents a further investigation about the use of the Normalized Levenshtein Distance (NLD) proposed by Tashima *et al* [1]. NLD allows to quantify the string similarity between local variables by exploiting the Levenshtein distance. In particular, it determines the minimum number of modifications in the characters to change one string to another string. NLD allows to quantify the confusion of local variables [1] and verify if the presence of not easily distinguishable variables in a method can suggest that the method is fault prone. In addition, we propose further uses of NLD: NLD_1, which represents the sum of all the NLD values calculated for each pair of variables declared in a method; NLD_2, which represents the average of all the NLD values for each pair of local variables; and NLD_3, which is defined as the sum of all the NLD values for each pair of local variables multiplied by the number of LOCs of the method.

To assess all the considered confusing measures, we have performed an empirical study by considering the same software systems employed in [1]. The results confirm and extend the ones of previous study about the relationships between the presence of faults and local variables with confusing names and how fault prediction models (built using the Random Forest) based on the considered distances can provide accurate estimations.

Organization of the paper: In Section 2 we summarize related work and recall the definition of NLD [1]. The design and the results of the performed empirical study are presented in Section 3 and Section 4, respectively. Conclusion concludes the paper.

2 Background

2.1 Related work

Software fault prediction has been widely investigated, aiming at identifying source code information that can help to accurately predict the presence of faults (e.g., [2] [3]). A few papers have investigated how the choices of developers when naming local variables can impact software quality. We can start mentioning the indications provided by Kernighan and Pike [4], who state that local variables have restricted role being used in a limited scope, thus it is unnecessary to use long and descriptive names for these identifiers. Some years later, Lawrie *et al.* [5] have performed a quantitative survey to investigate the impact of the variable composition on software comprehension. In their analysis they take into account three types of identifiers: a fully-word, abbreviated word, and a single character. The analysis of results reveals that understandability of identifiers decreases from full-words to single-character words. However, no significant difference can be highlighted between the use of full-words and abbreviated identifiers in terms of source code comprehension. More recently, a large-scale experiment performed by Scanniello *et al.* [6] has achieved similar results. They conducted a qualitative study to understand how identifier names either abbreviated or full-word the values, impact on fault fixing. Furthermore, it seems that even if variables with long names can help to better understand their use, the overall source code readability can be reduced [7]. Another study by Aman *et al.* [8] has also revealed that long local variable names are change-prone.

Binkley *et al.* [9] performed a study to compare the impact on the program comprehension when programmers use different naming styles. They compared the use of the camel case against the use of the snake case. The results show that the camel case improves the source code comprehension for developers at the beginning of their career while there is no significant difference for expert developers.

Regarding the relationship between the naming style and the presence of faults, the findings of a study performed by Kawamoto and Mizuno [10] reveal that the source code results to be fault-prone when classes contain long identifier names. With the aim of showing good practices when naming the identifiers, Butler *et al.* [11] have defined and as-

DOI reference number: 10.18293/SEKE2021-124

sessed 12 naming rules. The results of the performed analysis show that the use of identifiers not following the proposed rules increases the presence of faults.

Differently from the above mentioned contributions, Tashima *et al.* [1] have recently focused their attention on pairs of local variables with similar and confusing names. The aim of their investigation is to verify the relationships between the presence of such confusing variables and the fault-proneness at method level.

2.2 Normalized Levenshtein Distance

The motivation of Tashima *et al.* [1] is that the presence of local variables with high similarity names implies the possibility to confuse their use in the source code. To this aim, the Levenshtein distance is used to evaluate how much two names are confusing. The Levenshtein string edit Distance (LD) algorithm is one of most important models for string matching [12]. This edit distance is defined as the minimum number of insert, delete, and replace operations required to transform a source string x into a target string y. The approach assumes that insert and delete operations have cost 1, while the replacement has cost 2 (it is equivalent to a sequence of delete and insert operations). However, to evaluate more precisely the degree of confusion between two strings (local variables' names) s_1 and s_2, Tashima *et al.* propose a normalized LD, which is computed by dividing the distance by a factor that depends on the length of the considered local variables:

$$NLD(s_1, s_2) = \frac{LD(s_1, s_2)}{max\{\lambda(s_1), \lambda(s_2)\}}$$

where $LD(s_1, s_2)$ is the Levenshtein distance between the two strings s_1 and s_2, and the function λ computes the length of the corresponding string. In particular, Tashima *et al.* adopted the following definition:

$$NLD_T(s_1, s_2) = \min_{\forall s_1, s_2 \in M, s_1 \neq s_2} (NLD(s_1, s_2))$$

3 Study design

We have formulated the following research question:

RQ *Can information on variables with confusing names help to predict the presence of faults?*

To answer RQ we have built different prediction models based on the considered distance measures and assessed their accuracy in prediction. We also decided to build a prediction model exploiting the Line of Code (LOC) metric as independent variable, to verify whether the predictions achieved with NLD based measures are better than those obtained using only LOC.

3.1 Exploited NLD based measures

We considered further uses of NLD proposed in [1], starting from two considerations: *Why lower confusion values are excluded? Can the use of other software size measures (like the size of the module being analyzed) improve the effectiveness of NLD?* To this aim, we consider three further NLD based measures:

- NLD$_1$: is a "cumulative" measure computed by simply adding all the values of NLD for each pair of variables (i.e., we performed all the possible permutations of the identifiers defined in a method):

  $$NLD_1 = \sum_{\forall s_1, s_2 \in M, s_1 \neq s_2} NLD(s_1, s_2)$$

  where s_1 and s_2 can be all the local variables defined in a method M of the analyzed software class.

- NLD$_2$: it is based on NLD$_1$. In particular, the cumulative value of all the obtained distances is normalized by a factor depending on the number of all the local variables in the method M: NLD_1/n, where n is the overall number of local variables defined in M.

- NLD$_3$: it is obtained by multiplying the NLD$_2$ value by the number of LOCs present in the method under consideration (i.e., LOC_M): NLD_2 * LOC_M

3.2 Datasets

We considered the same five open source projects employed by Tashima *et al.* [1] for different reasons. We were interested in analyzing software implemented in Java and managed with Git in order to identify useful information such as the presence of faults. And more important, we selected the same software since our aim was to further assess the accuracy of NLD based measures (including NLD$_T$) given that NLD$_T$ provided good results on these software as reported in the original work of Tashima *et al.* [1]. In particular, the systems are: Apache Tomcat v. 9.0.12, Birt v. 4.8.0, Eclipse JDT User Interface v. 4.10.0, Eclipse Platform User Interface v. 4.10.0, Eclipse SWT v. 4.9.

In order to conduct the study, it was necessary to collect data from different sources. In particular, the collection of information to calculate the confusing measures, i.e., NLD$_T$, NLD$_1$, NLD$_2$, and NLD$_3$, was computed by analyzing the local variables of the methods of the source code of the considered projects. To this aim, we exploited a parser written in Java that makes use of the Eclipse JDT core library to extract information on methods and their local variables. We computed the values for NLD for each pair of local variables and then the values of each NLD$_i$ ($i \in \{T, 1, 2, 3\}$) as described above. Then, we add data about the presence of faults for each method by exploiting information from Promise repository [13], also used by

Tashima *et al.* but for different versions of the software projects, and by manually analyzing information provided in Git. This was the only strategy to adopt since the versions of the five projects we considered are among the most recents and are not the same as in the previous study by Tashima *et al.* [1]. Thus, the fault recovery was made by making an intersection at method and class levels between the datasets used by Tashima *et al.*, containing also the faults, and those used in our study. Whenever there was a correspondence of modules between the old and the new versions of a given project, the fault related to that module in our dataset for the project was added. In particular, in our analysis NLD_T, NLD_1, NLD_2, NLD_3 have been used as independent variable while Fault (the presence of fault, i.e., 1, or not, i.e., 0) as dependent variable.

3.3 Prediction models and accuracy evaluation

To build our fault prediction models, we employed the Random Forest that is a popular method for various machine learning tasks. It exploits a classifier as specified above, however it constructs more classification trees instead of a single tree [14]. As for its implementation, we exploited the tool Weka that offers widely used estimation techniques [15]. In particular, for our analysis we used the Classify module by selecting: *i)* all the parameters necessary for the construction of the model, *ii)* the Random Forest algorithm, *iii)* the independent variable (i.e., a measure of confusion), *iv)* the dependent variable (Fault) and the type of the validation method to be used.

To verify whether or not the obtained fault estimations are useful predictions of the actual faults we exploited *10*-fold cross validation [16] with $k = 10$, which requires the splitting of the dataset in $k-1$ training sets and 1 validation test for k times. Each time the training set is employed to define the estimation models, with the selected estimation techniques, while the corresponding validation test is used to validate the predictions obtained with the built models.

To evaluate the accuracy of the fault predictions, we employed F-measure defined as the weighted harmonic mean of the Precision and Recall [17]. Since the fault estimations have been computed on a dependent variable representing two classes with a very different number of observations, the Matthews correlation coefficient (MCC) can be generally adopted to measure the quality of a binary classifier. MCC represents a correlation coefficient between the observed and predicted binary classifications [18]. The MCC measure ranges from *+1* for a perfect classifier through *0* for a random classifier to *-1* for a weak classifier.

3.4 Threats to Validity

Some threats could affect the validity of our analysis. We considered five software projects developed in Java, and so the number and type of software can introduce a bias with respect to external validity. Thus, further investigations with different type of software projects and a greater number of projects should be carried out. However, to mitigate this threat we considered software projects whose information are publicly available and employed in previous investigations. Regarding the collection of information we employed Eclipse JDT to analyze source files to calculate the confusing measures, by analyzing the local variables of the methods of the source code of the considered projects. Eclipse JDT is a widely used tool for accomplishing such kind of work. As for the collection of fault data, possible threat is related to the fact that the fault recovery was made by making a intersection at method and class level between the datasets of the Promise used by Tashima et al. [1], containing also the faults, and those used in our study.

Table 1. Prediction accuracy achieved by the built fault estimation models

System	Employed measure	Correctly classified instances (%)	Incorrectly classified instances (%)	F-measure	MCC
Apache Tomcat	NLD_1	75	25	0.75	0.503
	NLD_2	77.5	22.5	0.77	0.548
	NLD_3	72.5	27.5	0.72	0.441
	NLD_T	85	15	**0.85**	**0.698**
	LOC	52.5	47.5	0.53	0.055
Birt	NLD_1	74.97	25.03	0.71	0.153
	NLD_2	76.28	23.72	0.76	0.033
	NLD_3	73.7	26.3	**0.72**	**0.179**
	NLD_T	76.70	23.30	0.68	0.118
	LOC	76.2	23.8	0.69	0.115
JDT ui	NLD_1	65.77	34.23	0.60	0.148
	NLD_2	65.94	34.06	0.57	0.127
	NLD_3	67.01	32.99	**0.64**	**0.206**
	NLD_T	65.83	34.17	0.61	0.159
	LOC	64.1	35.9	0.51	0.043
Platform ui	NLD_1	62.1	37.9	0.51	0.041
	NLD_2	62.91	37.09	0.50	0.034
	NLD_3	66.7	33.3	**0.66**	**0.266**
	NLD_T	64	36	0.55	0.122
	LOC	66.1	33.9	0.63	0.214
SWT	NLD_1	69.6	30.4	0.7	0.392
	NLD_2	56.65	43.35	0.56	0.118
	NLD_3	72.2	27.8	**0.72**	**0.440**
	NLD_T	54.9	45.1	0.52	0.071
	LOC	62.9	37.1	0.63	0.251

Other threats can regard the data analysis performed. As for the technique applied to obtain the prediction model we exploited Random Forrest since it is widely used for classification problems similar to ours. Furthermore, it was also used in the original work by Tashima et al. [1]. As for the assessment of the achieved fault predictions, other measures could be used, such as accuracy, however F-measure and MCC are widely employed in studies similar to ours.

4 Results

First of all, for two of the confusing measures, i.e., NLD_1 and NLD_3, we have observed a specific relationship between the presence of faults and local variables with confusing names, namely, as the value of confusing measures increases (i.e., the distance between local variables increase and so they are less confusing) the value of the fault rate increases as well. For the other two considered confusing measures we cannot provide a clear trend.

Table 1 reports the results in terms of Correctly classified instances (%), Incorrectly classified instances (%), F-measure, and MCC for each software systems and the built estimation models (i.e., based on NLD_1, NLD_2, NLD_3, NLD_T, and LOC) obtained by averaging the results of the *10*-fold cross validation as designed in Section 3.3.

We can note that F-measure values range from 0.5 to 0.85. The greatest values were obtained with Apache Tomcat (which is smaller in size with respect to the others) while the worst values were obtained with Platform ui. A similar consideration can be provided for MCC values.

The NLD based measures used as independent variables in fault prediction models built with the Random Forest that allowed to obtain better predictions are NLD_1 and NLD_3. NLD_T provided results similar to NLD_1 and in one case (i.e., Apache Tomcat) provided better predictions than the others. For the other 4 systems the measure that allowed to obtain better predictions is NLD_3. Let us remember that NLD_1 is the sum up NLDs of all the pairs, thus as a long method or a complicated method tends to have more variables its NLD_1 tends to be larger. NLD_3 is NLD_2 multiplied by LOC, so greater LOC greater NLD_3. So, both NLD_1 and NLD_3 are influenced by the source code size and one can image that the size measure has a large influence on the defect prediction performance. However, we can note from Table 1 that the good results in terms of NLD_3 for the systems JDT ui and Apache Tomcat are mainly due to the contribution of NLD_2, which is not related to size, or to the interaction between size and the confusion of variable (i.e., NLD_2). In two cases (i.e., Birt and Platform ui) LOC seems to have contribute more to the results achieved in terms of NLD_3. Moreover, it is important to note that NLD_3 allowed to obtain better results than LOC for all the software systems. In particular, for JDT ui, SWT, and Apache Tomcat the predictions achieved with NLD_1 and NLD_3 are particularly better than those obtained with LOC. Thus, we can positively answer RQ2 because information on variables with confusing names can help to predict the presence of faults. However, given that the our study is conducted on source code developed in open-source projects, our answer is cautious though. Indeed, observe that predictions are more accurate on some systems (i.e., Apache Tomcat) than on others independently from the confusing measures used.

5 Conclusions and Future Work

Our results and those of the original work by Tashima *et al* [1] can provide evidence of the usefulness of knowledge of variables with confusing names to improve the quality of the source code. In the future we intend to further investigate the relationship between faults and local variables with confusing names by considering different datasets and other combinations of NLD based and source code measures.

Acknowledgment

The authors would like to thank Hirohisa Aman [1] for the suggestions provided for the identification of the case studies used in the empirical analysis.

References

[1] K. Tashima, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "Fault-prone java method analysis focusing on pair of local variables with confusing names," in *Procs. of the Euromicro Conf. on Softw. Eng. and Adv. Applications (SEAA)*, 2018, pp. 154–158.

[2] T. Menzies, A. Butcher, D. R. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Trans. on Softw. Eng.*, vol. 39, no. 6, pp. 822–834, 2013.

[3] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies, "Class level fault prediction using software clustering," in *Procs. of Intl. Conf. on Automated Softw. Eng. (ASE)*, 2013, pp. 640–645.

[4] B. W. Kernighan and R. Pike, *The Practice of Programming*. Addison-Wesley, 1999.

[5] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Procs of the IEEE Intl. Conf. on Program Compreh. (ICPC)*, 2006, pp. 3–12.

[6] G. Scanniello, M. Risi, P. Tramontana, and S. Romano, "Fixing faults in c and java source code: Abbreviated vs. full-word identifier names," *ACM Trans. on Softw. Eng. Meth.*, vol. 26, no. 2, pp. 6:1–6:43, 2017.

[7] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," *Science of Computer Programming*, vol. 74, no. 7, pp. 430–445, 2009.

[8] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of change-proneness in methods having local variables with long names and comments," in *Procs. of the ACM/IEEE Intl. Symp. on Emp. Softw. Eng. and Measurement (ESEM)*, 2015, pp. 1–4.

[9] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Emp. Softw. Eng.*, vol. 18, no. 2, pp. 219–276, 2013.

[10] K. Kawamoto and O. Mizuno, "Predicting fault-prone modules using the length of identifiers," in *Procs. of the Intl. Workshop on Emp. Softw. Eng. in Practice (IWESEP)*, 2012, pp. 30–34.

[11] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *Procs. of the Working Conf. on Reverse Eng. (WCRE)*, 2009, pp. 31–35.

[12] V. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Sov. Phys. Doklady*, vol. 10, pp. 707–710, 1966.

[13] T. Menzies, R. Krishna, and D. Pryor, "The promise repository of emp. softw. eng. data," 2015.

[14] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832–844, 1998.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[16] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann Publishers Inc., 2011.

[17] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Inf. Retrieval*. Addison-Wesley, 1999.

[18] B. W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure"*, vol. 405, no. 2, pp. 442–451, 1975.