Using Deep Learning Classifiers to Identify Candidate Classes for Unit Testing in Object-Oriented Systems.

Wyao Matcha
Software Engineering Research Laboratory
Department of Mathematics and Computer Science
University of Quebec, Trois- Rivieres, Quebec, Canada
Wyao.Matcha@uqtr.ca

Fadel Touré
Software Engineering Research Laboratory
Department of Mathematics and Computer Science
University of Quebec, Trois- Rivieres, Quebec, Canada
Fadel.Touré@uqtr.ca

Mourad Badri
Software Engineering Research Laboratory
Department of Mathematics and Computer Science
University of Quebec, Trois- Rivieres, Quebec, Canada
Mourad.Badri@uqtr.ca

Linda Badri
Software Engineering Research Laboratory
Department of Mathematics and Computer Science
University of Quebec, Trois- Rivieres, Quebec, Canada
Linda.Badri@uqtr.ca

Abstract — **This paper aims at investigating the use of deep learning to suggest candidate classes to be tested rigorously during unit testing. The approach is based on software unit testing information history and source code metrics. We conducted our experiments using data collected from five (5) successive versions of the open source Java Apache software system ANT. For each version, we collected various source code metrics from the source code of the Java classes. We then extracted testing coverage measures for software classes for which dedicated JUnit test classes have been developed. We considered instruction and method level coverage granularities. Based on the different datasets collected, we trained several deep neural network models. We validated the constructed classifiers using Cross Version Validation technique. The obtained results strongly support the viability of our approach with an average accuracy greater than 87%.**

Keywords- Unit Testing, Tests Prioritization, Source Code Metrics, Testing Coverage Measures, Machine Learning, Deep Learning.

I. Introduction

Testing is the stage of software development where developers (testers) assess the conformity of the system developed (under development) with specifications [1]. Testing plays a crucial role in software quality assurance. It is, however, a time and resource-consuming process. Unit testing is one of the main phases of the testing process where each software unit is early and individually tested using dedicated unit test cases. In object-oriented (OO) software systems, units are software classes and testers usually write a dedicated unit test class for each software class they decided to test [2]. The main goal of unit testing is to early reveal the faults of software classes. In the case of large-scale OO software systems, because of resources limitation and time constraints, it is difficult, even unrealistic, to test rigorously all the classes [3]. The unit testing efforts are often focused. Testers (developers) usually prioritize unit tests by selecting and focussing the unit testing effort on a limited set of software classes (most critical) for which they write dedicated unit tests. Tests case prioritization is a method for prioritizing and planning test cases [3, 5]. This technique is used to run higher priority test cases (focus on the most critical components) to minimize time, cost and effort during the software testing process [3-6].

In this paper, we focus on unit testing of classes and particularly on how to automatically suggest suitable classes for unit testing using deep learning and information on both unit testing history and source code metrics. Several research related to test cases prioritization [7] using different OO metrics (The Chidamber and Kemerer metric suite in particular [8]) have been proposed in the literature. Some of these metrics, related to different software class attributes, have already been used in recent years to predict unit testability of classes [9-13]. These studies analyzed in particular various open source Java software systems and the corresponding JUnit test classes. One of the observations that have been made in these studies is that unit test cases were written only for a subset of classes [11-13].

Currently, many software repositories are available and can be used (among others) to analyze and predict software quality. Based on these repositories, can we extract valuable information that can be used to help software testers in prioritizing unit tests? Is it possible to automate this process? The main goal of our research is to propose an approach to identifying suitable classes for unit testing using deep learning techniques [14,15] (artificial neural networks). The approach is based on software unit testing information history and source code metrics. We conducted our experiments using data collected from five (5) successive versions of the

DOI reference number: 10.18293/SEKE2020-147

open source Java Apache software system ANT [16]. For each version, we first collected various source code metrics from the source code of the Java classes. We then extracted testing coverage measures for software classes for which dedicated JUnit [17] test classes have been developed. We considered instruction and method level coverage granularities. With the collected data, we trained several deep neural network models. We validated the constructed classifiers using Cross Version Validation technique. The obtained results strongly support the viability of our approach with an average accuracy greater than 87%.

II. Related Work

Touré et *al.* [7] investigated an approach based on software information history to support the prioritization of classes to be tested. They have analyzed different class attributes of ten Java open source software systems for which JUnit unit test classes were developed. Using different techniques, they first characterized the classes for which JUnit test classes were developed by the testers. Secondly, they built two classifiers using OO metrics and unit tests information collected from the selected systems. The classifiers provide, for each software, a set of classes on which the unit testing efforts must be focused. The sets of obtained candidate classes were compared to the sets of classes for which JUnit test classes were developed by the testers. The results have shown that: (1) average values of the metrics of the classes tested are very different from average values of the metrics of the other classes (2) there is a significant relationship between the fact that a JUnit test class was developed for a class and its source code attributes, and (3) the sets of classes suggested by the classifiers correctly reflect the selection of testers.

Mirarab et *al.* [18] used Bayesian networks to create a unified model based on information provided by the CK metrics suite [8], changes and testing coverage rates. The defined approach optimizes coverage and improves the fault detection rate compared to the random planning of test scenarios.

Helge Spieker et *al.* [19] present Retecs (Reinforcement Learning for Automatic Test Cases Prioritization and Selection in Continuous Integration), which is a method for selecting and prioritizing test cases based on machine learning. It is used in cases of continuous integration with the aim of minimizing the round-trip delay between code validation and developer comments in the event of failure of the test cases. The Retecs method uses reinforcement learning to select and prioritize test cases according to their duration, the last execution history and failure. In an environment where new test cases are created and obsolete test cases are deleted, the Retecs method learns to prioritize error-prone tests using a reward function and build on previous cycles of integration. By applying Retecs on data extracted from three industrial case studies, Helge Spieker et *al*. have shown for the first time that reinforcement learning allows fruitful selection and automatic prioritization in continuous integration and regression tests.

III. Data Collection

For our investigations, we carried out our experiments on Apache ANT software system, which is a command-line tool used to control the execution processes described in mutually dependent XML files (build files) [16]. The versions 1.3, 1.4, 1.5, 1.6 and 1.7 of ANT system have been extracted to carry on our experiments.

A. Data Collection Tools

The source code of ANT’s considered versions has been grabbed from GitHub [20] repository. Under IntelliJ [21] IDE, we used Code Mr [22], Cover [23] plugins as well as the JUnit Framework [17] to run the unit test suites and collect source code metrics and testing coverage measures.

B. Source Code Metrics

Four our study, we considered six (6) source code metrics, five from the well-known metrics suite proposed by Chidamber and Kemerer [8] and the widely used SLOC metric. These metrics capture various OO attributes such as coupling, inheritance, cohesion, complexity and size. They have received particular attention in empirical software engineering research and are computed by Code Mr plugin according to the following definitions:

Coupling metrics: (1) CBO (Coupling between objects) calculates the number of classes to which a class is linked and vice versa.

Inheritance metrics: (2) DIT: (Depth of Inheritance) counts the number of classes between the measured class and the root of its inheritance hierarchy. (3) NOC: (Number of Children) calculates the number of subclasses that inherit from the measured class.

Cohesion metrics: (4) LCOM: (Lack of cohesion in Methods) assesses the lack of cohesion in a class. It counts the number of methods pairs whose similarity is 0 minus the number of method pairs whose similarity is different from zero.

Complexity metrics: (5) WMC: (Weighted Method Complexity) sums up the cyclical complexities of all the methods of the measured class. (6) RFC: (Response for classes) calculates the number of possible methods that can be called in response to the method invocation of the measured class.

Size metrics: (7) SLOC: (Source Lines of Codes) calculates the number of lines of code in measured class.

C. Descriptive statistics

The descriptive statistics on the different versions of the ANT system [13] show that the number of classes increases from 1093 for version 1.3 to 1145 classes for version 1.7. The average complexity (WMC) and source lines of code (SLOC) vary slightly from one version to another (around 19 for WMC and 156 for SLOC). This small variation is also observed for the other metrics average values. The great variability of metrics within the same version indicated by the standard deviations (σ), reflects the variability level of the software classes characteristics.

TABLE 1:Descriptive statistics of the used source code metrics - ANT

Vers.	obs.	Stat	CBO	DIT	LCOM	SLOC	NOC	RFC	WMC
V1.3	1096	Min	0	1	0	4	0	1	1
		Max	672	7	30	1899	143	326	236
		Mean	10.97	2.4	1.85	156.64	0.67	26.06	19.74
		σ	1111.2	1.82	5.36	49022.6	27.09	1130	864.26
V1.4	1102	Min	0	1	0	6	0	1	1
		Max	674	7	30	2204	143	321	235
		Mean	10.84	2.39	1.84	155.65	0.67	25.78	19.51
		σ	1104.3	1.82	5.35	50129.7	26.97	1106.4	841.08
V1.5	1103	Min	0	1	0	6	0	1	1
		Max	674	7	30	2204	143	321	235
		Mean	10.84	2.39	1.84	155.73	0.67	25.79	19.52
		σ	1104.3	1.82	5.35	50304.1	26.97	1108.1	843.18
V1.6	1140	Min	0	1	0	6	0	1	1
		Max	694	7	30	2214	145	321	252
		Mean	10.8	2.38	1.82	155.01	0.66	25.63	19.44
		σ	1123	1.82	5.18	50743.9	27.72	1096.2	840.85
V1.7	1143	Min	0	1	0	4	0	0	1
		Max	694	7	30	2214	145	321	255
		Mean	10.81	2.39	1.83	157.33	0.66	25.65	19.45
		σ	1125.1	1.83	5.25	53739.4	27.71	1100.1	843.81

D. *Empirical Analysis*

We have collected the source code of the five different versions of ANT [16] considered for the study, grabbed from GitHub repository [20]. We followed the steps described below for each version:

Step 1: Extracting the considered source code metrics using Code Mr [22] plugin.
Step 2: Running unit test suites and collecting the testing coverage rates by using Cover [23] plugin and JUnit [17] framework.
Step 3: Filtering outlier observations, which are mainly OO artefacts with 0 complexity (interfaces, some of abstract classes, enumerations and constants collecting classes).
Step 4: Computing the rank of each value of the metrics. This step is motivated by the fact that the classifier will be trained on several versions of different sizes. Ranks will help the classifiers to mitigate the metrics' values.
Step 5: Labelling testing coverage data. We have labelled each class with a binary value, 1 or 0, depending on whether its unit testing coverage reaches a given threshold percent or not. 50%, 30% and 0% thresholds have been considered. For example, by setting the threshold at 50%, all the classes having a testing coverage greater or equal to 50% are labelled as 1 (considered as tested), and the rest of classes is labelled as 0 (considered as not tested).

For each class, the attributes formed by the source code metrics, the associated rank values as well as the binarized coverage rates, form a labelled observation. With the collected data, our goal is to train a deep neural network model to build classifiers that could automatically predict (suggest) the label of each observation. In addition, we wanted to investigate if a classifier built from the dataset of version n, correctly predicts the level of testing coverage in the successive n + 1 version.

TABLE 2: Distribution of tested classes - method granularity

Versions	Obs	0%	30%	50%
13	1096	179	108	81
14	1102	187	114	82
15	1103	193	117	85
16	1140	193	116	85
17	1143	196	119	87

TABLE 3: Distribution of tested classes - instruction granularity

Versions	Obs	0%	30%	50%
13	1096	179	98	60
14	1102	187	102	61
15	1103	193	105	65
16	1140	193	103	62
17	1143	196	106	64

IV. EXPERIMENTAL METHOD

A. *Neural networks*

Neural networks belong to the family of machine learning classifier models [24, 25]. The building blocks for neural networks are artificial neurons [26]. These are simple computational units that have weighted input signals and produce an output signal using an activation function [27].

Neurons are arranged into networks of neurons. A row of neurons is called a layer. Neural networks may have multiple layers. The first layer, called input or visible layer, takes input from the dataset. The following layers are referred as hidden layers. They are responsible for compressing and building internal representation of datasets. The last layer, also called output layer, is responsible for classifying the output on the required classes.

Deep neural networks refer to networks containing more than one hidden layer. Training those layers requires large amount of data and specific techniques that prevent vanishing gradient issue [25].

The neural net training process relies on forward propagation technique that consists of feeding input values to the neural network and getting an output (predicted value). On each training epoch, error is computed using back-propagation technique [28].

B. *Construction and Architecture of the Neural Network*

Our deep learning architectures (ANN) are built using Python programing language, under TensorFlow [29] and Keras [30] frameworks supported by Pandas [31] library for data manipulation. We carried out several tests before coming to the following configuration that produced the obtained results.

Our input layer contains fourteen (14) neurons to match the 14 characteristics (7 for source code metrics plus 7 for rank values associated with each metric value). Thirteen (13) hidden layers follow the input layer containing 169 neurons each. And finally, 2 neurons compose the output layer to match our binary classification problem. We used ReLu as activation function for input and hidden layers. We relied on "Adam" as optimizer and the mean square as loss function. The model has been evaluated using confusion matrix.

C. *Classifier Validation*

We validated our classifier using Cross Version Validation (CVV) technique inspired by classical cross validation techniques. CVV approach consists of training the neural network model on dataset of a version V of the ANT system, then validating the obtain classifier on the dataset of the successive version V+1.

V. RESULTS AND INTERPRETATIONS

The CVV technique has validated the suggestions of classes to be tested for a given version, made by a classifier trained on the previous version. The following results were obtained by this validation technique applied to the different considered versions of ANT.

TABLE 4: CVV results on Method granularity level

	CM50	**CM30**	**CM0**
13->13	93.20%	91.77%	88.28%
13->14	92.97%	91.19%	87.99%
14->14	94.04%	90.84%	88.79%
14->15	93.77%	90.57%	88.26%
15->15	93.51%	92.62%	89.06%
15->16	93.64%	92.35%	88.57%
16->16	94.67%	94.07%	89.86%
16->17	94.25%	93.56%	89.70%

From Table 4 (CVV results on method granularity level), it can be seen that:

The threshold of 50% produces accuracy scores that are greater than 92%. The lowest accuracy is obtained for a validation on version 1.4 (92.97%) and the highest is obtained for a validation on version 1.7 (94.25%).

For a threshold of 30%, the obtained accuracy scores are greater than 90%. The lowest accuracy is obtained while validating on version 1.5 (90.57%) and the highest for the validation on version 1.7 (93.56%).

For a threshold of 0%, the accuracy scores obtained are greater than 87%. The lowest accuracy is obtained when validation is done on version 1.4 (87.99%) and the highest for validation on version 1.7(89.70%).

After all validations, we can notice the trends of the accuracy to increase. Indeed, the highest accuracy scores obtained are noticed for a threshold set at 50%. This indicates that there exists a relationship between classes metrics and labelled classes (classes labelled 1 or 0). Other results obtained with threshold sets at 30% and 0% are good but show that their classifiers are not as accurate as the one obtained with the threshold set at 50%.

From these results, we can conclude that by using method granularity level, we are able to predict correctly candidate classes for unit testing.

TABLE 5: CVV results on Instruction granularity level

	CLOC50	**CLOC30**	**CLOC0**
13->13	94.63%	93.47%	88.37%
13->14	94.48%	93.06%	87.28%
14->14	95.37%	94.22%	89.86%
14->15	95.02%	93.95%	89.32%
15->15	94.48%	94.57%	90.12%
15->16	94.67%	94.24%	89.69%
16->16	95.79%	94.76%	88.57%
16->17	95.62%	94.42%	88.58%

From Table 5 (CVV results on instruction granularity level), it can be seen that:

For a threshold set at 50%, the accuracy scores obtained are greater than 94%. The lowest accuracy is obtained for a validation on version 1.4 (94.48%) and the highest is obtained for a validation on version 1.7 (95.62%).

For a threshold set at 30%, the accuracy values obtained are greater than 93%. The lowest accuracy is obtained while validating on version 1.4 (93.06%) and the highest for validation on version 1.7 (94.42%).

For a threshold set at 0%, the accuracy values obtained are greater than 87%. The lowest accuracy is obtained for validation on version 1.4 (87.28%) and the highest for validation on version 1.6 (89.69%).

As we go from validation on version 1.3 to validation on version 1.7, we observed an increase of the accuracy from 87% to 89%.

As in the case of method granularity level, we obtained the highest accuracy scores for a threshold set at 50%. Hence a classifier obtained by training it on the dataset will be more efficient and accurate than the one obtained with another threshold. We also observe a tendency of the accuracy to increase during the test.

From these results, we can conclude that by using instruction granularity level, we are also able to predict correctly candidate classes for unit testing.

By comparing the results obtained for the different levels of granularities (methods and instruction), we see that we obtain high accuracy scores for the instruction level.

From these two tests, we can conclude that the choice of the level of granularity as well as the threshold set is crucial.

The CVV validation showed that it is possible to predict classes to be tested on a version developed or under development by a classifier trained on the previous version of the same system. Moreover, we can notice that we got higher accuracy rates for the instruction granularity level with a threshold of 50%.

VI. LIMITATIONS

The obtained results are quite significant with regard to the used dataset (information collected from five different versions of a same system) but should nevertheless be considered as exploratory. The selection of the classes to test explicitly (for which JUnit test classes have been developed) is in most cases left to the goodwill of developers. This can lead to partially tested classes and systems with few tested classes, which in turn can influence the results obtained in our various experiments.

The generalization of our results requires additional investigations including tests quality as well as the application's domain of the considered software systems, which can also impact the results and restrict their scope. In our analyzes, we limited ourselves to Java OO language. Even if Java is a reference language in OO programming, our results could be biased by this limitation.

We limited our investigations to a well-known case study system (ANT). The study should be replicated on more software systems in order to draw more general conclusions.

VII. CONCLUSION

In this work, the goal was to provide an approach that supports unit testing decision when selecting the software classes to be tested. For that, we used a classifier build from deep neural network model based on various class source code metrics and the corresponding unit testing coverage data. Two levels of testing coverage measures have been considered in combination. We validated the constructed classifier using cross version validation technique. By obtaining after various tests more than 87% of accuracy rates, we can conclude that the obtained results strongly support the viability of the approach.

These results open the possibility of using software metrics and the developers' experience (particularly in terms of unit testing development) in guiding the distribution of the overall unit testing effort. Hence using current big data analysis techniques (involving artificial intelligence algorithms), it is possible to develop cloud-based tools in order to build a new generation of tests prioritization tools and guidance integrated into software development environments.

REFERENCES

[1] B. Boehm, "A Spiral Model of Software Development and Enhancement", Proc. Int'l Workshop Software Process and Software Environments, ACM Press, 1985; also in ACM Software Eng. Notes, Aug. 1986, pp. 22-42.

[2] B. Boehm. "Software Engineering Economics". Prentice Hall, Englewood Cliffs, NJ, ISBN-10 : 0138221227, edition 01 oct 1981.

[3] S. Elbaum, A. G. Malishevsky and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies", IEEE Transactions Software Engineering, Vol. 28, No. 2, pp.159-182, 2002.

[4] S. Elbaum, A. G. Malishevsky and G. Rothermel, "Prioritizing test cases for regression testing". Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA), Portland, OR, USA, 22-25 August 2000, pp. 102-12

[5] G. Rothermel, R.H Untch, C. Chu and M.J Harrold . "Test case prioritization: an empirical study", International Conference on Software Maintenance, Oxford, UK, pp. 179–188.,1999.

[6] J. Kim, and A. Porter , "A history-based test prioritization technique for regression testing in resource constrained environments", In Proceedings of the International Conference on Software Engineering, 2002.

[7] F. Toure., M. Badri and L. Lamontagne, "Investigating the Prioritization of Unit Testing Effort Using Software Metrics", In Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'17) Volume 1: ENASE, pages 69-80, 2017.

[8] Chidamber S.R. and Kemerer C.F., "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476–493, 1994.

[9] M. Bruntink , and A.V. Deursen,"Predicting Class Testability using Object-Oriented Metrics", 4th Int. Workshop on Source Code Analysis and Manipulation (SCAM), IEEE, 2004.

[10] V. Gupta, K. K. Aggarwal and Y. Singh, "A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability", Journal of Computer Science, Vol. 1, No. 2, 2005, pp. 276-282. doi:10.3844/jcssp.2005.276.282.

[11] M. Bruntink and A. Van Deursen, "Predicting class testability using object-oriented metrics", in Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '04), pp. 136–145, September 2004.

[12] M. Bruntink and A. van Deursen, "An empirical study of class testability", Journal of Systems and Software, vol. 79, no. 9, pp. 1219–1232, 2006.

[13] L. Badri, M. Badri, and F. Toure, "An empirical analysis of lack of cohesion metrics for predicting testability of classes", International Journal of Software Engineering and Its Applications, vol. 5, no. 2, 2011.

[14] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun. "What is the best multi-stage architecture for object recognition?" In International Conference on Computer Vision, pages 2146–2153. IEEE, 2009.

[15] Y. LeCun, K. Kavukcuoglu, and C. Farabet, Convolutional networks and applications in vision. In Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on, pages 253–256. IEEE, 2010.

[16] Apache ANT releases, https://github.com/apache/ant/releases, Visited in december 2019.

[17] JUnit Framework, https://junit.org/junit5/. Visited in december 2019.

[18] S. Mirarab, A. Hassouna, and L. Tahvildar, "Using Bayesian belief networks to predict change propagation in software systems" in Proceedings of the 15th IEEE International Conference on Program Comprehension, pages 177-188, 2007.

[19] H. Spieker, A. Gotlieb, D. Marijan and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration", Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, July 2017.

[20] ANT on Github Repository, https://github.com/apache/ant, Visited in december 2019.

[21] IntelliJ IDE, https://www.jetbrains.com/idea/, Visited in december 2019.

[22] Code Mr plugin, https://plugins.jetbrains.com/plugin/10811-codemr/, Visited in december 2019.

[23] Code-Coverage plugin, https://www.jetbrains.com/help/idea/code-coverage.html, Visited in december 2019.

[24] Artificial Intelligence and life in 2030, one-hundred-year study on artificial Intelligence, report of the 2015 Study panel, September 2016

[25] Y. LeCun, Y. Bengio, G, Hinton. "Deep learning" . Nature. 2015;521(7553):436-444. doi:10.1038/nature14539

[26] F. Rosenblatt, "The perceptron : A probabilistic model for information storage and Organization in the brain", in cognitive systems. Buffalo: Cornell Aeronautical Laboratory, Inc. Rep. No. VG-1196-G-1, 1958.

[27] M. Minsky, and S. Papert, “Perceptrons: An Introduction to Computational Geometry,” MIT Press, expanded edition, ISBN-10 : 0262631113, décembre 1987

[28] D. E. Rumelhart, G. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", Cognitive modeling 5.3 (1988):

[29] Tensorflow: https://www.tensorflow.org/

[30] Kera: https://keras.io/

[31] Panda: https://pandas.pydata.org