# Unit Testing Effort Prioritization Using Combined Datasets and Deep Learning: A Cross-Systems Validation

Fadel TOURE

Department of Mathematics and Computer Science,
University of Quebec at Trois-Rivières,
Trois-Rivières, Québec, Canada.
Fadel.Toure@uqtr.ca

Mourad BADRI
Department of Mathematics and Computer Science,
University of Quebec at Trois-Rivières,
Trois-Rivières, Québec, Canada.
Mourad.Badri@uqtr.ca

***Abstract*— Unit testing plays a crucial role in object-oriented software quality assurance. Software testing is often conducted under tight time and resource constraints. Hence, testers do not usually cover all software classes. Testing needs to be prioritized and testing effort to be focused on critical components. The research we present in this paper is part of the development of a collaborative decision support tool allowing the developers' community to pool their unit testing experiences when selecting the candidate classes for unit tests. To achieve this, we proposed in our previous work a unit tests prioritization approach based on software information histories and software metrics. The goal is to suggest classes to be tested by building a classifier that matches the testers selection. Several machine learning classifiers have been previously considered. The current paper explores the deep neural network models with more software source code metrics including explicitly and implicitly tested classes. The training datasets that have been combined are from different systems. So, we considered metrics ranks. Using a cross systems validation technique, obtained results strongly suggest that deep neural network-based classifiers correctly reflect the tester's selections and could thus help in decision support during the selection of candidate classes for unit tests.**

***Key words*— *Tests Prioritization; Unit Tests; Source Code Metrics; Deep Neural Network; Deep Learning; Machine Learning Classifiers.***

## I. INTRODUCTION

Software testing plays a crucial role in software quality assurance. Unit testing is one of the main phases of the testing process where each software class is individually tested using dedicated 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. The unit tests aim at early reveal faults in software classes. In the case of large-scale OO software systems, because of resource limitations and tight time constraints, the unit testing efforts are often focused. Testers usually select a limited set of software classes for which they write dedicated unit test classes. Knowing that it is often not realistic to equally test all software classes, it becomes important for testers to target the most critical and fault-prone classes. However, the task is not obvious and requires a deep analysis of software. These issues belong to the family of tests prioritization topics. Several existing approaches try to prioritize test suites execution in order to discover the maximum of faults quickly, while others try upstream to focus the developer efforts on suitable classes to be tested. This paper focus on how to automatically target suitable candidate classes for unit tests. The long-term goal is to build a collaborative tool for the developers' community. That tool will collect source code metrics and classes unit test information from different projects in order to improve a unique cloud-hosted classifier performance to match the testers' selection of unit tests candidate classes. For new systems under development, the tool could suggest, after collecting some specific source code metrics, a set of candidate classes for unit tests. Due to the large source code diversity and increasing amount of data that the tool will face, we considered using deep neural network models trained on combined systems' datasets to explore how accurate the classifiers could match the testers' selection.

Many OO metrics, related to internal software class attributes have been proposed in literature [1, 2]. Some of them have already been recently used to predict unit testability of classes in OO software systems [3-10] by analyzing various existing open source Java software systems for which Junit [11] test cases were developed and are accessible in public repositories. For all systems, authors [3-10] found that only a subset of classes have dedicated unit test classes written by developers. In previous work [9, 12], we focused on how the selection of the candidate classes for unit tests was made by testers. Multivariate Logistic Regression, Naive Bayes, Random Forest and K-Nearest Neighbours classifiers have been used to automate the selection of candidate classes for unit tests. They have been validated within systems and between systems using Cross Systems Validation (CSV) and Leave One System Out Validation (LOSOV). The latter validation technique implied the use of combined datasets extracted from different systems.

Based on deep neural network models [13], the current work includes more source code metrics to capture various characteristics that we believe are determinative for a software class to be considered as a good candidate for unit tests from testers' point of view. We also included two ways of labelling tested classes according to the existence of dedicated unit test classes and to the actual unit testing coverage.

The paper is organized as follows. Section II presents some related works. Section III addresses the OO software

DOI reference number: 10.18293/SEKE2020-150

metrics we used for this study. Section IV describes the data collection procedure and the considered systems. Section V presents the empirical study we conducted and the results obtained with the related discussions. Section VI reports the main threats to validity relatively to our empirical experimentations. Finally, Section VII concludes the paper, summarizes the contributions of this work and outlines several directions for future investigations.

## II. Related Work

Test case prioritization has been widely discussed in the context of regression testing. Various techniques have been proposed in the literature and used different leverages. We can distinguish: (1) coverage rates based techniques, (2) software history information based techniques, and (3) risk analysis based techniques.

Fault detection techniques focus on targeting the most fault prone components using, in practice, fault exposure factors as a proxy. Factors are estimated using different ways from the software artifacts. The results obtained by Rothermel et *al*. [11] and Yu and Lau [12] indicated that this approach improves the fault detection rates.

The coverage-based techniques run the test suites that cover most modified software artefacts during regression testing. Several machine learning algorithms (Naïve Bayes, Genetic Algorithms) are used to derive a prioritization approach. The investigations [14-16] results showed that coverage-based techniques also lead to fault detection rate improvement.

The history-based prioritization collects previous regression testing assets and current changes information of the same system in order to prioritize the new given test suites. Thus, the technique is unsuitable for the first regression testing of software. Kim and Porter [17] used the historical execution data to prioritize test cases for regression tests, while Lin et *al*. [18] investigated the weight of used information between two versions of history-based prioritization techniques. The results indicated that the history-based prioritization provides a better fault detection rate. Carlson et *al*. [19] mixed history and coverage-based techniques using a clustering based prioritization technique.

Lachmann et *al*. [20] introduced a test case prioritization technique for system-level regression testing based on supervised machine learning. The approach considers black-box metadata, such as test cases history, as well as natural language test case descriptions for prioritizing. They used the SVM Rank machine learning algorithms and evaluate their approach on 2 subject systems. The results outperform a test case order given by a test expert.

Spieker et *al*. [21] proposed the *Retecs* approach, a method for automatically learning test case selection and prioritization in continuous integration with the goal to minimize the round-trip time between code commits and developer feedback on failed test cases. The approach uses reinforcement learning. The Empirical study shows that reinforcement learning enables fruitful automatic adaptive test case selection and prioritization.

The history and machine learning based techniques prioritize test suites in a regression testing context. Some other techniques allow, upstream, the prioritization of components to be tested. They aim to optimize the testing efforts distribution by targeting the most fault prone components. Shihab et *al*. [22] explored the prioritization for unit testing phase in the context of legacy systems. Our previous papers [12] proposed machine learning approaches that aim to suggest candidate classes for unit tests. We used 2 classifiers trained on the dataset formed by source code metrics and labelled by tested/not tested, to build classifiers that match the candidate classes for unit tests. After applying cross systems validation techniques, our results indicated that for a given system, the ability of a classifier, to correctly suggest the candidate classes for unit tests ( more than 70% of accuracy). Furthermore, we considered more machine learning algorithms and we focused on affinities between the systems used as training and testing datasets during the cross systems validation. We wanted to determine whether some systems make better training sets for suggesting other specific systems unit test candidate classes. The result showed that the datasets of large systems could be only used to suggest large systems unit test candidate classes, while classifiers trained on small systems fail to suggest the candidate classes for unit tests on large and small systems. In the same study [12], we focused on the ability of combined datasets to suggest candidate classes for unit tests. After applying the leave one system out validation technique, the result show that more than 70% of candidate classes selected by testers were well predicted in the case of large size systems.

The current paper investigates deep neural network classifiers trained on combined datasets as predictor models for unit tests candidate classes selection. Combining different systems as a single training dataset presents several advantages such as diversity of observations and their amount. Indeed, our long-term objective is to build a collaborative IDE plugin, based on unit tests information and some specific metrics to support the unit tests prioritization. Hence, the plugin will collect source code metrics and test information from various software systems. Under such conditions, the ability of learning from combined datasets is of great importance. Combining training datasets may, however, lead to metric dimensionality issues. Indeed, from the tester point of view, a class with a given metric value may be considered as a good candidate or not depending on the metric values of the other classes of the system. The following section presents the software metrics we used in our study.

## III. Software Metrics

This section presents the considered OO source code metrics. We expanded the previous dataset metrics used in [9, 12] by including more source code attributes. The selected metrics are being adopted by practitioners. Several studies have shown that the considered metrics are related to testability [3-8], maintainability [23-26], and fault proneness [27-29]. The set of metrics is related to inheritance, coupling, complexity and size software attributes. We computed them using the Borland Together (http://www.borland.com).

***Depth of Inheritance Tree***: DIT metric is the maximum inheritance path from the given class to the root class. ***Coupling Between Objects***: The CBO metric counts for a given class, the number of other classes to which it is coupled and vice versa. ***Fan Out***: The FOUT metric counts the number of other classes referenced by a given class. ***Fan IN***: The FIN metric counts the number of other classes that reference to a given class. ***Weighted Methods per Class***: The WMC metric gives the sum of the complexities of the methods of a given class, where each method is weighted by its cyclomatic complexity [27]. Only methods specified in the

class are considered. ***Response For Class:*** The RFC metric measures the class's complexity in terms of method invocations. It sums the number of methods defined in a given class and the number of distinct method invocation made by that method. ***Lines of Code per Class***: The SLOC metric counts for a given class, its number of source lines of code.

## IV. DATA COLLECTION

### A. Selected Systems

The source codes of 10 open source OO software systems developed in Java have been extracted from public repositories and described below. For each system, only a subset of classes has been tested using JUnit framework.

***IO*** [1] is a library of utilities for developing input/output functionalities. It is developed by Apache Software Foundation. ***MATH*** [1] is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language. ***JODA*** [2] is the de facto standard library for advanced date and time in Java. It provides a quality replacement for the Java date and time classes. The design supports multiple calendar systems, while still providing a simple API.
***DBU*** [3] (DbUnit) is a JUnit extension (also usable with Ant) used in database-driven projects that, among others, put a database into a known state between test runs. ***LOG4J*** [1] is a fast and flexible framework for logging applications debugging messages. ***JFC*** [4] (JFreeChart) is a free chart library for Java platform. ***IVY*** [1] is an agile dependency manager characterized by flexibility, simplicity and tight integration with Apache Ant. ***LUCENE*** [1] is a high-performance, full-featured text search engine library. It is a suitable technology for applications requiring full-text search. ***ANT*** [1] is a Java library and command-line tool that drives processes described in build files as target and extension points dependent upon each other. ***POI*** [1] is an APIs for manipulating various file formats based upon the Office Open XML standards and Microsoft's OLE2. It can read and write MS Excel files using Java.

### B. Unit Test Data Collection Procedure

The selected systems have been tested using the JUnit framework. JUnit [11] is a framework for writing and running automated unit tests for Java classes. JUnit gives testers some support so that they can write the test cases more conveniently. A typical usage of JUnit is to test each class $C_s$ of the software by means of a dedicated test class $C_t$. To actually test a class $C_s$, we execute its test class $C_t$ by calling JUnit's test runner tool. JUnit report how many of the test methods in $C_t$ succeeded, and how many failed.

In [12], we used the prefix/suffix linking approach, as other authors [4, 10, 30], to link each software class to its dedicated JUnit test class if exists. Linked classes are referred as E-TESTED classes. Furthermore, we considered, the level of JUnit Coverage (JUC) score computed by Borland Together Tool to take transitively tested classes into account. Indeed, in [8, 9, 12], we noted that some of software classes were tested by transitive method invocations during unit tests.

Table 1: Percent of tested classes

| | MATH | IO | JODA | DBU | LOG4J |
|---|---|---|---|---|---|
| % I-TESTED | 84.04% | 81% | 76.62% | 46.70% | 40.26% |
| % E-TESTED | 61.7% | 66% | 37.81% | 40.1% | 19.5% |
| | JFC | IVY | LUCENE | ANT | POI |
| % I-TESTED | 66.26% | 61.68% | 52.52% | 16.89% | 67.73% |
| % E-TESTED | 55.50% | 15.62% | 18.54% | 16.89% | 28,00% |

The JUC score is based on unit test class invocation, representing for each class the percent of software lines of code covered by the set of unit test classes. Classes with a JUC score greater than 0 are referred as I-TESTED classes. Table 1 summarizes the distribution of E-TESTED and I-TESTED classes.

### Descriptive Statistics

Table 2 summarizes the statistics of selected metrics for the 10 systems ordered by increasing sizes in terms of the number of classes.

Table 2 Descriptive statistics

| Syst | Obs | Stat | FIN | CBO | DIT | LOC | RFC | FOU | WMC |
|---|---|---|---|---|---|---|---|---|---|
| MATH | 94 | Min. | 0 | 0 | 1 | 2 | 13 | 0 | 0 |
| | | Max | 13 | 18 | 6 | 660 | 119 | 12 | 174 |
| | | Sum | 275 | 306 | 195 | 7779 | 3717 | 194 | 1824 |
| | | μ | 2.93 | 3.26 | 2.07 | 82.76 | 39.54 | 2.06 | 19.40 |
| | | σ | 2.47 | 3.72 | 1.11 | 97.60 | 18.64 | 2.46 | 25.12 |
| IO | 100 | Min. | 0 | 0 | 1 | 7 | 17 | 0 | 1 |
| | | Max | 14 | 39 | 5 | 968 | 202 | 21 | 250 |
| | | Sum | 323 | 405 | 214 | 7604 | 3782 | 254 | 1817 |
| | | μ | 3.23 | 4.05 | 2.14 | 76.04 | 37.82 | 2.54 | 18.17 |
| | | σ | 4.07 | 5.70 | 1.01 | 121.56 | 24.79 | 3.27 | 31.75 |
| Syst | Obs | Stat | FIN | CBO | DIT | LOC | RFC | FOUT | WMC |
| JODA | 201 | Min. | 0 | 0 | 1 | 5 | 11 | 0 | 1 |
| | | Max | 106 | 36 | 6 | 1760 | 287 | 22 | 176 |
| | | Sum | 2116 | 1596 | 447 | 31339 | 17857 | 1089 | 6269 |
| | | μ | 10.53 | 7.94 | 2.22 | 155.92 | 88.84 | 5.42 | 31.19 |
| | | σ | 16.12 | 6.44 | 1.28 | 210.97 | 64.21 | 4.78 | 30.55 |
| DBU | 212 | Min. | 0 | 0 | 1 | 4 | 11 | 0 | 1 |
| | | Max | 28 | 24 | 6 | 488 | 95 | 19 | 61 |
| | | Sum | 517 | 1316 | 452 | 12187 | 6827 | 901 | 1989 |
| | | μ | 2.43 | 6.18 | 2.13 | 57.22 | 32.05 | 4.23 | 9.34 |
| | | σ | 3.44 | 5.32 | 1.22 | 60.55 | 14.54 | 3.94 | 9.45 |
| LOG4J | 231 | Min. | 0 | 0 | 1 | 5 | 11 | 0 | 1 |
| | | Max | 72 | 107 | 7 | 1103 | 632 | 47 | 207 |
| | | Sum | 966 | 1698 | 467 | 20150 | 15879 | 1088 | 3694 |
| | | μ | 4.18 | 7.35 | 2.02 | 87.23 | 68.74 | 4.71 | 15.99 |
| | | σ | 9.29 | 10.12 | 1.30 | 130.42 | 105.75 | 5.93 | 25.70 |
| Syst | Obs | Stat | FIN | CBO | DIT | LOC | RFC | FOUT | WMC |
| JFC | 409 | Min. | 0 | 0 | 1 | 4 | 11 | 0 | 0 |
| | | Max | 55 | 101 | 7 | 2041 | 677 | 56 | 470 |
| | | Sum | 2583 | 4861 | 967 | 67481 | 50628 | 3253 | 13428 |
| | | μ | 6.28 | 11.83 | 2.36 | 164.19 | 123.18 | 7.91 | 32.67 |
| | | σ | 8.99 | 14.07 | 1.40 | 228.06 | 148.28 | 9.43 | 46.73 |
| IVY | 608 | Min. | 0 | 0 | 0 | 2 | 1 | 0 | 0 |
| | | Max | 103 | 92 | 6 | 1039 | 458 | 46 | 231 |
| | | Sum | 2239 | 5205 | 1037 | 50080 | 35274 | 3419 | 9664 |
| | | μ | 3.68 | 370.03 | 1.71 | 219.60 | 58.02 | 5.62 | 15.84 |
| | | σ | 7.89 | 11.74 | 1.31 | 141.80 | 61.67 | 7.33 | 27.38 |
| LUCENE | 615 | Min. | 0 | 0 | 1 | 1 | 11 | 0 | 0 |
| | | Max | 63 | 55 | 6 | 2644 | 433 | 46 | 557 |
| | | Sum | 2860 | 3793 | 1212 | 56108 | 23724 | 2872 | 10803 |
| | | μ | 4.65 | 6.17 | 1.97 | 91.23 | 38.58 | 4.67 | 17.57 |
| | | σ | 7.18 | 7.24 | 1.06 | 192.87 | 34.61 | 5.49 | 35.70 |
| ANT | 663 | Min. | 0 | 0 | 0 | 1 | 11 | 0 | 0 |
| | | Max | 300 | 41 | 6 | 1252 | 550 | 30 | 245 |
| | | Sum | 3228 | 4613 | 1563 | 63548 | 36282 | 3294 | 12034 |
| | | μ | 4.87 | 6.96 | 2.36 | 95.85 | 54.72 | 4.97 | 18.15 |
| | | σ | 16.87 | 7.25 | 1.28 | 132.92 | 46.25 | 5.41 | 24.17 |
| Syst | Obs | Stat | FIN | CBO | DIT | LOC | RFC | FOUT | WMC |
| POI | 1382 | Min. | 0 | 0 | 1 | 2 | 11 | 0 | 0 |
| | | Max. | 189 | 168 | 7 | 1686 | 642 | 62 | 374 |
| | | Sum | 5733 | 9660 | 2899 | 130185 | 66574 | 5924 | 23810 |
| | | μ | 4.15 | 6.99 | 2.10 | 94.20 | 48.17 | 4.29 | 17.23 |
| | | σ | 9.51 | 10.78 | 1.24 | 154.28 | 58.44 | 6.27 | 28.32 |

1 https://apache.org/

2 http://joda-time.sourceforge.net/

3 http://dbunit.sourceforge.net/

4 http://www.jfree.org/jfreechart/

The number of lines of code varies from 7,779 lines spread over 94 software classes for MATH system, to more than 130,185 lines of code over 1,382 software classes for POI system. The number of classes and their cyclomatic complexity follow the same trend. Following the descriptive statistics, we grouped the systems into 4 categories relatively to their size in order to better interpret the results: (1) the small-size systems, about 100 software classes such as IO and MATH, (2) the medium-size systems around 200 classes such as LOG4J, DBU and JODA, (3) the large-size systems, between 400 and just over 600 classes such as LUCENE, IVY, ANT and JFC, and (4) the very large-size systems over than 1,000 software classes such as POI.

The average cyclomatic complexity varies widely between systems with similar sizes as for JODA and DBU systems. Indeed, these systems present similar number of classes (around 200) but quite a different average of cyclomatic complexity (31.19 vs. 9.34). We made the same observations for LUCENE and JFC systems.

The DIT metric varies from 1 to 7 in all systems when it average is about 2 for the majority of systems. DIT has the lowest variance values compared to other metrics. The minimum average value of DIT is observed for IVY (1.71) and the maximum average value for JFC and ANT (2.36). A very deep inheritance tree may indicate a bad design while shallow inheritance reflects the lack of code reusability.

JODA software has the highest average value of FIN (10.53) while JFC got the highest average value of FOUT (7.91) and RFC (123.18).

## V. Empirical Analysis

### A. Research question

The machine learning models in [12] have been trained on combined datasets formed by source code metrics and unit tests information of different systems. With 70% of correct classifications, the generated classifiers well suggested the candidate classes for unit tests as long as the targeted systems was large enough. That result has been mainly explained by the probably missing of strategies when testing small software systems. With more source code metrics, our current work test different deep neural network topologies to improve the results we observed previously. The main research question is:

*Can deep neural network-based classifier better fit the candidate classes selected by testers for unit testing?*

The main goal remains to use metric information in order to support unit tests prioritization decisions. Our research question allows to validate whether a deep neural network model can produce good classifiers that fit the testers selection of candidate classes for unit tests. The empirical study we conducted is based on combined training datasets from which the system under analysis has been excluded, a technique we referred as Leave One System Out Validation.

### B. Deep Neural Network

Deep neural network is a family of Artificial Neural Network (ANN) that contains more than one hidden layer. When well trained (Deep learning), it allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how ANN should change its internal parameters that are used to compute the representation in each layer, from the representation in the previous layer [13].

In deep neural network models, the layers configuration may strongly impact the performances of classifiers. Unfortunately, there is no systematic approach that may determine the right layers topology for a given dataset. Hence, we adopted the try and error strategy to find the suitable architecture for our datasets.

### C. Leave One System Out Validation LOSOV

The LOSOV consists of combining the datasets of different $S_i$ systems excluding $S_j$ to form a unique training dataset for the neural network model. The generated classifier is tested on the remaining $S_j$ system. After many tries, following layers topology has been set for the deep neural network model.

*The input layer:* We managed a dataset that contains 7 properties formed by the selected metrics which lead us to set 7 neurons on the entry layer.

*The hidden layers:* The hidden layers organization result from multiple tries/error, and the best results was obtained when setting 6 layers of 175 neurons each of them activated with *relu* function. With fewer neurons, the model trends to misclassify the large and the very large systems, while more neurons conduct to overfitting issues. We tried different compressing topologies by gradually reducing the number of cells along the layers, from entry toward the output layer. The results were inconclusive. We also increased/decreased the number of layers and combined them with different epoch numbers but misclassifications and overfitting issues still persisted.

*The output layer:* The output layer is composed of 2 neurons to match our binary classification problem. The layer uses *softmax* activation function.

We also found, after many tries, that 350 epochs gave the best results. Increasing that number leads to overfitting with totally unbalanced confusion matrix (classifier tends to suggest all software classes or none of them as candidates for unit tests), while reducing it produces misclassifications.

### D. Results & Discussion

We considered both the E-TESTED and I-TESTED unit test perspectives. Table 3 summarizes the results we got by generated classifiers with 350 epochs. On each row that represents evaluated system, LOSVO approach validates the classifier obtained from the dataset composed of all remaining systems by testing it on that system. The *accuracy* column indicates the accuracy percentage, while the *conf. matrix* column holds the confusion matrix produced by the classifier.

We immediately remarked that: (1) the candidate classes for unit tests of larger systems are better predicted with better accuracy compared to our previous works, and (2) the I-TESTED point of view leads to better suggestion results in terms of the number of correctly predicted systems. The relationship between systems' size and classifiers' performances is not surprising but follows the trends we previously observed using other classifiers models. The explanation may come from the lack of strategy when testing

small systems. It may also be related to the training dataset scale. The largest system (POI) predication is weak according to E-TESTED point of view (about 63.6%). Removing POI from the combined dataset may unbalance the training dataset and could explain the weakness of the prediction accuracy.

Table 3: LOSVO trained on metric values

| | E-TESTED, Value Only | | | I-TESTED, Value Only | | |
|---|---|---|---|---|---|---|
| | Accuracy | Conf. Matrix | | Accuracy | Conf. Matrix | |
| MATH | 38.30% | 28 | 8 | 64.89% | 5 | 10 |
| | | 50 | 8 | | 23 | 56 |
| IO | 52.00% | 30 | 4 | 68.00% | 15 | 4 |
| | | 44 | 22 | | 28 | 53 |
| JODA | **73.13%** | 97 | 28 | **78.61%** | 34 | 13 |
| | | 26 | 50 | | 30 | 124 |
| DBU | 63.21% | 115 | 12 | **80.66%** | 84 | 29 |
| | | 66 | 19 | | 12 | 87 |
| LOG4J | **86.15%** | 175 | 12 | **78.79%** | 106 | 32 |
| | | 20 | 24 | | 17 | 76 |
| JFC | **83.37%** | 168 | 14 | **82.15%** | 114 | 24 |
| | | 53 | 173 | | 49 | 222 |
| IVY | **88.49%** | 472 | 41 | **85.20%** | 188 | 45 |
| | | 29 | 66 | | 45 | 330 |
| LUCENE | **80.98%** | 436 | 65 | **84.55%** | 243 | 49 |
| | | 52 | 62 | | 46 | 277 |
| ANT | **86.12%** | 497 | 54 | **78.73%** | 422 | 129 |
| | | 38 | 74 | | 12 | 100 |
| POI | 63.6% | 725 | 270 | **78.22%** | 431 | 84 |
| | | 233 | 154 | | 217 | 650 |

When considering the I-TESTED point of view, the candidate classes for unit tests are better predicted by classifiers. 8 systems over 10 (against 6 over 10 for E-TESTED) have an accuracy greater than 70%. The associated confusion matrices ensure us that the classifiers are not suggesting no class or all classes (at the same time) as candidate classes for unit tests. Indeed, we faced that situation when using shallow neural networks or when we increased the number of training stages epochs during our investigations.

When deepening our investigations and reviewing the descriptive statistics, we understood that some characteristics of class attributes relatively to other classes in the same software system may have an impact on developer decision to select or not that class as a candidate for unit tests. The raw values of source code metrics considered alone are not sufficient for our classifiers to correctly match the tester selections. Indeed, a WMC score of 50 (for example) may be important when a developer tests a system for which the average class complexity (WMC) is much smaller than 50 which lead him to write an explicit test class for that software class. On the other hand, a class with the same complexity score may be considered as little complex by the developer when it belongs to a large system in which average complexity of classes is largely higher than 50. We thus need an attribute that captures the metric values for a class relatively to the other classes within the same system. When combining datasets, that attribute will mitigate it corresponding source code metrics. The ranks of metric scores are good candidates. In the following steps, we tested whether including metric ranks could improve the results of Table 2. We computed the rank of each metric's value of each class inside each system. Ranks have been included to the datasets. All new datasets contain 14 attributes that constrains us to review our neural network topology.

*The Input layer:* With the new datasets of 14 properties formed by the metrics and their ranks. We set the number of neurons in the entry layer to 14.

*The Hidden layers:* We set the number of hidden layers to 13. Now, each layer contains 350 neurons. We kept the *relu* as activation function.

*The Output layer:* The output remains unchanged. Its 2 neurons still match our binary classification problem. They are activated with *softmax* function. The number of epochs has also been doubled to 750 to prevent misclassification.

Table 4 summarizes the results. We immediately remark that all predictions highly improved compared to Table 3. The large systems are suggested with more than 99% of correctness in both perspectives. MATH results with E-TESTED point of view, slightly improved but remains the only system under 70% of correctness. For several systems (JODA, DBU, LOG4J and LUCENE) all tested classes have been found by the classifier without any false positive or false negative classification. We reached 100% of correctness.

Table 4 LOSVO trained on metric and rank values

| | E-TESTED, Value + Rank | | | I-TESTED, Value + Rank | | |
|---|---|---|---|---|---|---|
| | Accuracy | Conf. Matrix | | Accuracy | Conf. Matrix | |
| MATH | 52.13% | 14 | 22 | **76.60%** | 3 | 12 |
| | | 23 | 35 | | 60 | 69 |
| IO | **93.00%** | 30 | 4 | **99.00%** | 19 | 0 |
| | | 3 | 63 | | 1 | 80 |
| JODA | **100.00%** | 125 | 0 | **99.50%** | 46 | 1 |
| | | 0 | 76 | | 0 | 154 |
| DBU | **100.00%** | 127 | 0% | **100.00%** | 113 | 0 |
| | | 0% | 85 | | 0 | 99 |
| LOG4J | **100.00%** | 187 | 0 | **99.57%** | 137 | 1 |
| | | 0 | 44 | | 0 | 93 |
| JFC | **99.76%** | 181 | 1 | **99.51%** | 138 | 0 |
| | | 0 | 227 | | 2 | 269 |
| IVY | **99.84%** | 513 | 0 | **98.85%** | 226 | 7 |
| | | 1 | 94 | | 0 | 375 |
| LUCENE | **100.00%** | 501 | 0 | **99.19%** | 289 | 3 |
| | | 0 | 114 | | 2 | 321 |
| ANT | **99.85%** | 550 | 1 | **99.85%** | 551 | 0 |
| | | 0 | 112 | | 1 | 111 |
| POI | **99.78%** | 994 | 1 | **99.35%** | 510 | 5 |
| | | 2 | 385 | | 4 | 863 |

The results we obtained in Tables 3 and 4 strongly support our hypothesis. It's possible to build a prediction classifier based on deep neural network and trained on combined datasets composed by different software systems that correctly suggest classes to be tested. "Correctly" means matching the real testers' selection. E-TESTED and I-TESTED points of view have no impact when we included the ranks values in the datasets. Let's recall that our long-term goal was to build an IDE plugin tool that automatically collects source code metrics of systems under development in order to suggest a set of classes to be tested. The plugin's classifiers would be trained from datasets of various systems. Under such conditions, it was important for us to explore in the current work, the suggestion capability of classifiers trained on such a mixed dataset.

## VI. THREATS TO VALIDITY

Obtained results are suggestive and the study we presented was performed on 10 open-source systems containing almost a half million lines of code (453K). The sample is large enough to allow obtaining significant results, but the experimental approaches may present limitations that could restrict the generalization of certain conclusions. Indeed, all systems we used are developed using Java language and tested using JUnit framework. Java and JUnit are popular in the developers' community. The obtained results may not be generalizable to other unit testing frameworks or programming languages. More investigations are required to rule on the issue. Furthermore, it would be

interesting to know, in such a condition, whether mixing dataset from systems built using different languages and unit framework could improve or degrade the results. The neural network topology we identified matches very well the analyzed group of systems. Changing the number of systems and their categories may degrade obtained results. Replicating the study on more systems could help to draw more general neural network topology that fits unit test decision support.

## VII. Conclusions and Future Work

Ten open source software systems have been analyzed in this study which totals more than 4400 classes. The testers of each system developed dedicated unit test classes for a subset of classes using the Junit Framework. We explored the possibility of deep neural network models to correctly match developers' selections of the candidate classes for unit tests. To achieve our investigations, we considered explicitly and implicitly tested classes. With the combination of the 10 datasets formed by the considered systems, we tested various deep neural network topologies that we validated using Leave One System Out Validation technique. The objective was to know to what extents the combined information of different systems could be a usable training dataset for deep neural network-based classifiers. Results show that it was possible to correctly match the candidate classes for unit tests proposed by testers. Furthermore, the results indicated that all systems could be well predicted with more than 93% of accuracy. These results are particularly interesting since the long-term goal of our work is to build a collaborative plugin tool that suggests the set of the candidate classes for unit tests by learning from different systems information history. Our next challenge will be to validate this approach using different unit testing frameworks under different programming language before developing the plugin tool.

## References


1. Chidamber S.R. and Kemerer C.F., 1994. A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476–493.
2. Henderson-Sellers B. 1996. Object-Oriented Metrics Measures of Complexity, Prentice-Hall, Upper Saddle River.
3. Bruntink M. and Van Deursen A. 2006. An Empirical Study into Class Testability, Journal of Systems and Software, Vol. 79, No. 9, pp. 1219-1232.
4. Badri L., Badri M. and Toure F., 2010. Exploring Empirically the Relationship between Lack of Cohesion and Testability in Object-Oriented Systems, JSEA Eds., Advances in Software Engineering, Communications in Computer and Information Science, Vol. 117, Springer, Berlin.
5. Badri M. and Toure F., 2011. Empirical analysis for investigating the effect of control flow dependencies on testability of classes, in Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering SEKE.
6. Badri M. and Toure F. 2012. Empirical analysis of object oriented design metrics for predicting unit testing effort of classes, Journal of Software Engineering and Applications (JSEA), Vol. 5 No. 7, pp.513-526.
7. Toure F., Badri M. and Lamontagne L., 2014. Towards a metric suite for JUnit Test Cases. In Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE Vancouver, Canada. Knowledge Systems Institute Graduate School, USA pp 115–120.
8. Toure F., Badri M. and Lamontagne L., 2014. A metrics suite for JUnit test code: a multiple case study on open source software, Journal of Software Engineering Research and Development, Springer, 2:14.
9. Toure F., Badri M. and Lamontagne L., 2017. 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.
10. Bruntink M., and Deursen A.V., 2004. Predicting Class Testability using Object-Oriented Metrics, 4th Int. Workshop on Source Code Analysis and Manipulation (SCAM), IEEE.
11. JUnit Framework, https://junit.org/junit5/. Visited in December 2019.
12. Toure F., and Badri M., 2018. Prioritizing Unit Testing Effort Using Software Metrics and Machine Learning Classifiers, In Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering, SEKE 2018 DOI:10.18293/SEKE2018-146
13. LeCun Y, Bengio Y, and Hinton G., 2015. Deep learning. Nature. 2015, 521(7553):436-444. doi:10.1038/nature14539.
14. Rothermel G., Untch R.H., Chu C. and Harrold M.J., 1999. Test case prioritization: an empirical study, International Conference on Software Maintenance, Oxford, UK, pp. 179–188.
15. Yu Y. T. and Lau M. F., 2012. Fault-based test suite prioritization for specification-based testing, Information and Software Technology Volume 54, Issue 2, Pages 179–202.
16. Mirarab S. and Tahvildari L., 2007. A prioritization approach for software test cases on Bayesian networks, In FASE, LNCS 4422-0276, pages 276–290.
17. Kim J. and Porter A., 2002. A history-based test prioritization technique for regression testing in resource constrained environments, In Proceedings of the International Conference on Software Engineering.
18. Lin C.T., Chen C.D., Tsai C.S. and Kapfhammer G. M., 2013. History-based Test Case Prioritization with Software Version Awareness, 18th International Conference on Engineering of Complex Computer Systems.
19. Carlson R., Do H., and Denton A., 2011. A clustering approach to improving test case prioritization: An industrial case study, Software Maintenance, 27th IEEE International Conference, ICSM, pp. 382-391.
20. Lachmann R., Schulze S., Nieke M., Seidl C. and Schaefer I., 2016 System-Level Test Case Prioritization Using Machine Learning, 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), Anaheim, CA, 2016, pp. 361-368.
21. Spieker H., Gotlieb A., Marijan D. and Mossige M., 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.
22. Shihaby E., Jiangy Z. M., Adamsy B., Ahmed E. Hassany A. and Bowermanx R., 2010. Prioritizing the Creation of Unit Tests in Legacy Software Systems, Softw. Pract. Exper., 00:1–22.
23. Li W., and Henry S., 1993. Object-Oriented Metrics that Predict Maintainability Journal of Systems and Software, vol. 23 no. 2 pp. 111-122.
24. Dagpinar M., and Jahnke J., 2003. Predicting maintainability with object-oriented metrics – an empirical comparison, Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), IEEE Computer Society, pp. 155–164.
25. Zhou Y., and Leung H., 2007. Predicting object-oriented software maintainability using multivariate adaptive regression splines, Journal of Systems and Software, Volume 80, Issue 8, August 2007, Pages 1349-1361, ISSN 0164-1212.
26. Basili V.R., Briand L.C. and Melo W.L., 1996. A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering. vol. 22, no. 10, pp. 751-761.
27. Aggarwal K.K., Singh Y., Kaur A., and Malhotra R., 2009. Empirical Analysis for Investigating the Effect of Object-Oriented Metrics on Fault Proneness: A Replicated Case Study, Software Process Improvement and Practice, vol. 14, no. 1, pp. 39-62.
28. Shatnawi R., 2010. A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems, IEEE Transactions On Software Engineering, Vol. 36, No. 2.
29. Zhou Y. and Leung H., 2006. Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults, IEEE Transaction Software Engineering, vol. 32, no. 10, pp. 771-789.
30. Mockus A., Nagappan N. and Dinh-Trong T. T., 2009. Test coverage and post-verification defects: a multiple case study, in proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 291– 301.