Prioritizing Unit Testing Effort Using Software Metrics and Machine Learning Classifiers

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. Unfortunately, software testing is often conducted under severe pressure due to limited resources and tight time constraints. Therefore, testing efforts have to be focused, particularly on critical classes. As a consequence, testers do not usually cover all software classes. Prioritizing unit testing effort is a crucial task. We previously investigated a unit testing prioritization approach based on software information histories. We analyzed different attributes of ten open-source Java software systems tested using the JUnit framework. We used machine learning classifiers (Multivariate Logistic Regression and Naïve Bayes) to obtain, for each system, a set of classes to be tested. The obtained sets of candidate classes have been compared to the sets of classes for which JUnit test cases have been actually developed by testers. The cross system validation (CSV) technique results showed, among others, that the sets of candidate classes suggested by machine learning classifiers properly reflect the testers' selection. In this paper, we extend our previous work by investigating more classifiers and using leave one system out validation (LOSOV) technique. This LOSOV technique uses a combination of training datasets from different systems. The obtained results indicate that: (1) the new classifiers correctly suggest classes to be tested, and (2) tested classes are particularly well predicted in the case of large-size systems.**

Key words— Tests Prioritization; Unit Tests; Source Code Metrics; Machine Learning Classifiers.

I. Introduction

Unit testing is one of the main phases of the testing process where each software unit is 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. The main goal is to 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 tests. Hence, it is important to target the most critical and fault-prone ones. However, the task is not obvious and requires a deep analysis of software. In this paper, we focus on how to automatically target suitable classes, candidates to unit testing. Our approach relies on classifiers algorithms trained on different unit tests information and source code metrics collected from different software systems.

DOI reference number: 1018293/SEKE2018-146

A large number of OO metrics, related to different OO internal class attributes, have been proposed in literature [1, 2]. Some of these metrics have already been used in recent years to predict unit testability of classes in OO software systems [3-9]. The authors noticed that, for each of the analyzed systems, unit test cases have been developed only for a subset of classes. In our previous work [10], we tried to understand and determine, using different source code attributes, which criteria have been considered during the classes' selection. We investigated, using Multivariate Logistic Regression (MLR) and Naive Bayesian (NB) classifiers, how to automate and improve the selection of classes on which unit testing effort has to be focussed using source code metrics. In the current study, we considered two more well-known classifiers (K-Nearest Neighbors (KNN) and Random Forest (RF) algorithms). We also used the Cross System Validation (CSV) and the Leave One System Out Validation (LOSOV) techniques. The LOSOV technique allows, in particular, combining data from various systems as training datasets. The goal was to determine, firstly, the affinities that may exist between prediction dataset systems and training dataset systems according to their characteristics (such as size, type, category, etc.) and, secondly, to investigate to what extent the combined datasets could improve or degrade the learners' prediction levels.

The rest of the paper is organized as follows. Section 2 presents some related works. Section 3 presents the OO software metrics we used in the study. Section 4 describes the data collection procedure. Section 5 presents the empirical study that we conducted. Section 6 focuses on the main threats to validity related to our empirical experimentations. Finally, Section 7 concludes the paper, summarizes the contributions of this work and outlines several directions for future investigations.

II. Related Work

Many researchers have proposed different tests prioritization techniques in the literature, particularly in the context of regression testing. The proposed techniques are based on various criteria such as fault detection, coverage rates, software history information, and risk analysis.

In fault detection based techniques, the main goal is to run test cases that target the most fault prone components. These techniques use different factors of fault exposure as proxies, which can be estimated in different ways from the software artifacts. These approaches have been proposed, among others, by Rothermel et *al.* [11] and Yu and Lau [12]. Results showed that these techniques improve the fault detection rates.

In coverage based techniques, the main goal is to run test suites that cover most modified software artefacts during regression testing. The authors [13-15] used Naïve Bayes, Genetic Algorithms and different levels of granularity to implement their prioritization approaches. Results showed that coverage based techniques also lead to fault detection rate improvement. Rothermel et *al*. [11] compared nine test case prioritization techniques based on random prioritization, coverage prioritization and fault detection prioritization. Obtained results provide insights into the trade-offs among various techniques for test cases prioritization.

The history based prioritization uses information from previous regression tests of the same software system and current modification information in order to prioritize the new given test suites. This makes the prioritization technique unsuitable for the first regression testing of software. Kim and Porter [16] used the historical execution data to prioritize test cases for regression tests, while Lin et *al*. [17] investigated the weight of used information between two versions of history based prioritization techniques. The different results indicated that the history based prioritization provides a better fault detection rate.

Carlson et *al*. [18] mixed history and coverage based techniques using a clustering based prioritization technique. They improved the effectiveness of test cases prioritization techniques. Elbaum et *al*. [19] analyzed the conditions under which techniques are relevant. The obtained results provide insights and conditions into which types of prioritization techniques are or are not appropriate under specific testing scenarios.

Some other techniques allow, upstream, the prioritization of components to be tested. The goal is to optimize the testing efforts distribution by targeting the most fault prone components. Boehm and Basili [20] proposed a Pareto distribution in which 80% of all defects within software are found in 20% of the modules. Ray and Mohapatra [21] rely on that Pareto distribution to address the question of components prioritization. Shihab et *al*. [22] explored the prioritization for unit testing phase in the context of legacy systems.

Ray and Mohapatra' approach [21] ignores the history of the software, whereas the approach of Shihab et *al*. [22] is not suitable for new software. Moreover, neither approach takes advantage of the large amount of information available in the public open source repositories. In [10], we proposed the prioritization of unit test candidate classes for OO software systems. We conjectured that testers generally rely on classes' characteristics captured by source code metrics, in order to select the components to test. Thus, we proposed an approach that takes advantage of different software testers experiences and software class attributes, in order to prioritize classes (candidates) to be tested. With the same systems, the same software metrics, more learning algorithms and LOSOV technique, the current study focusses on the systems dataset affinities and the effect of merged training datasets on learner prediction performances. The long-term objective is to build a collaborative IDE plugin, based on tests information history and some specific metrics that support the tests prioritization decisions with suitable machine learning techniques.

III. SOFTWARE METRICS

We present, in this section, the OO source code metrics we selected for the empirical study. These metrics have received considerable attention from researchers and are also being increasingly adopted by practitioners as testability [3-9,23], maintainability [24-26], and fault proneness [26-31] indicators. These metrics have been computed using Borland Together IDE (http://www.borland.com). We also included the well-known source lines of code metric.

- *Coupling Between Objects*: The CBO metric counts for a given class, the number of other classes to which it is coupled and vice versa.
- *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.
- *Lines Of Code per class*: The LOC metric counts for a given class its number of source lines of code.

IV. DATA COLLECTION

A. *Data* collection procedure

The selected systems have been developed by different teams in Java language and tested using the JUnit framework. JUnit (http://www.junit.org/) is a simple framework for writing and running automated unit tests for Java classes. 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 need to execute its test class C_t by calling JUnit's test runner tool. JUnit will report how many of the test methods in C_t succeeded, and how many failed.

We used the prefix/suffix linking approach, as other authors [4, 23, 32-33], to match each software class to its JUnit test class (es). Indeed, developers usually name the JUnit dedicated test classes by prefixing or suffixing the name of software class under the test by "Test" or "TestCase". We assign the modality 1 to the set of *tested classes* and the modality 0 to the remaining classes, referred as *untested classes*.

B. *Selected Systems*

We extracted information from the repositories of 10 open source OO software systems that were developed in Java. For each system, only a subset of classes has been tested using the JUnit framework. We present in the following the selected systems, from small size to large-size systems.

- IO (https://commons.apache.org/proper/commons-io/): Commons IO is a library of utilities for developing Input/Output functionalities. It is developed by Apache Software Foundation (ASF).
- MATH (http://commons.apache.org/proper/commons-math/): Commons MATH is a library of lightweight, self-contained mathematics and statistics.
- JODA (http://joda-time.sourceforge.net/): JODA-Time is the de facto standard library for advanced date and time in Java.
- DBU (http://dbunit.sourceforge.net/): 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 (http://wiki.apache.org/logging-log4j/): Log4j is a fast and flexible framework for logging applications debugging messages.
- JFC (http://www.jfree.org/jfreechart/): JFreechart is a free chart library for Java platform.

TABLE I: DESCRIPTIVE STATISTICS

	MATH			**JFC**		
	CBO	***LOC***	***WMC***	***CBO***	***LOC***	***WMC***
Obs.	94	94	94	411	411	411
Min.	0	2	0	0	4	0
Max.	18	660	174	101	2041	470
Sum	306	7779	1824	4861	67481	13428
µ	3.255	82.755	19.404	11.827	164.187	32.672
σ	3.716	97.601	25.121	14.066	228.056	46.73
Cv	1.141	1.179	1.295	1.189	1.389	1.43
	IO			**IVY**		
	CBO	***LOC***	***WMC***	***CBO***	***LOC***	***WMC***
Obs.	100	100	100	610	610	610
Min.	0	7	1	0	2	0
Max.	39	968	250	92	1039	231
Sum	405	7604	1817	5205	50080	9664
µ	4.05	76.04	18.17	8.533	82.098	15.843
σ	5.702	121.565	31.751	11.743	141.801	27.38
Cv	1.408	1.599	1.747	1.376	1.727	1.728
	JODA			**LUCENE**		
	CBO	***LOC***	***WMC***	***CBO***	***LOC***	***WMC***
Obs.	201	201	201	615	615	615
Min.	0	5	1	0	1	0
Max.	36	1760	176	55	2644	557
Sum	1596	31339	6269	3793	56108	10803
µ	7.94	155.915	31.189	6.167	91.233	17.566
σ	6.443	210.974	30.553	7.243	192.874	35.704
Cv	0.811	1.353	0.98	1.174	2.114	2.033
	DBU			**ANT**		
	CBO	***LOC***	***WMC***	***CBO***	***LOC***	***WMC***
Obs.	213	213	213	663	663	663
Min.	0	4	1	0	1	0
Max.	24	488	61	41	1252	245
Sum	1316	12187	1989	4613	63548	12034
µ	6.178	57.216	9.338	6.958	95.849	18.151
σ	5.319	60.546	9.451	7.25	132.915	24.168
Cv	0.861	1.058	1.012	1.042	1.387	1.332
	LOG4J			**POI**		
	CBO	***LOC***	***WMC***	***CBO***	***LOC***	***WMC***
Obs.	231	231	231	1382	1382	1382
Min.	0	5	1	0	2	0
Max.	107	1103	207	168	1686	374
Sum	1698	20150	3694	9660	130185	23810
µ	7.351	87.229	15.991	6.99	94.2	17.229
σ	10.119	130.419	25.7	10.782	154.282	28.319
Cv	1.377	1.495	1.607	1.543	1.638	1.644

- IVY (http://ant.apache.org/ivy/): IVY is a simple and flexible agile dependency manager tightly integrated with Apache Ant.
- LUCENE (http://lucene.apache.org/): LUCENE is a high-performance, full-featured text search engine library suitable for applications requiring full-text search.
- ANT (http://www.apache.org/): ANT is a Java library and command-line tool that drives processes described in build files as target.
- POI (http://poi.apache.org/): POI is a Java APIs for manipulating various file formats based upon the Office Open XML standards and Microsoft's OLE2.

C. Descriptive Statistics

Table I summarizes the statistics of selected metrics of all systems. It shows that the considered systems are of different sizes. The number of lines of code varies from 7,600 lines spread over 100 software classes (IO), to more than 130,185 lines of code over 1,382 software classes (POI). Table I also suggests 4 groups of systems according to their size: (1) the small-size systems, about 100 classes (IO and MATH), (2) the medium-size systems around 200 classes (LOG4J, DBU and JODA), (3) the large-size systems, between 400 and 600 classes (LUCENE, IVY, ANT and JFC), and (4) the very large-size systems over than 1,000 software classes (POI).

The average cyclomatic complexity varies widely between systems with similar sizes. Indeed, the medium-size systems, JODA and DBU, have a quite different average of cyclomatic complexity (9.34 vs 31.18). Similar trend is observed for LUCENE and JFC systems. In the dataset, each row has a binary attribute TESTED taking modalities 1 or 0 indicating whether it is a *tested class* or *untested class*.

V. Empirical Analysis

A. Research questions

The current study focusses on the systems' dataset affinities and the effect of mixing datasets on classifiers' performance. We tried to respond to the following research questions:

- RQ1- Can other well-known machine learning algorithms correctly predict the testers' selections?
- RQ2- To what extent can we mix dataset histories of different systems to predict testers' selections for a given new system.

B. Goals

The goal of our first research question (RQ1) is to validate our previous results using other machine learning algorithms, to compare their prediction performances, and to determine whether they depend (or not) on the type or size

TABLE II: CROSS SYSTEM VALIDATIONS

		MATH	IO	JODA	DBU	LOG4J	JFC	IVY	LUCENE	ANT	POI
MATH	***LR***	**0.745**	0.590	0.338	0.404	0.160	0.414	0.188	0.184	0.193	0.295
	NB	**0.723**	0.620	0.478	0.521	0.411	0.606	0.400	0.405	0.363	0.418
	KNN	**0.734**	0.64	0.527	0.563	0.416	0.431	0.53	0.48	0.431	0.446
	RF	**0.926**	0.58	0.338	0.469	0.32	0.397	0.382	0.358	0.398	0.391
IO	***LR***	0.617	**0.740**	0.458	0.624	0.329	0.623	0.418	0.403	0.359	0.395
	NB	0.628	**0.710**	0.413	0.667	0.494	0.652	0.454	0.433	0.403	0.505
	KNN	0.585	**0.79**	0.418	0.592	0.364	0.275	0.334	0.304	0.275	0.378
	RF	0.649	**0.94**	0.388	0.493	0.394	0.305	0.301	0.366	0.276	0.373
JODA	***LR***	0.415	0.390	**0.711**	0.620	**0.779**	0.620	**0.814**	**0.807**	**0.796**	**0.721**
	NB	0.457	0.420	0.692	0.620	**0.771**	0.681	**0.799**	**0.800**	**0.742**	**0.721**
	KNN	0.468	0.39	**0.841**	0.446	0.632	0.698	0.692	**0.706**	0.698	0.633
	RF	0.447	0.43	**0.95**	0.582	0.684	0.656	**0.729**	0.62	0.649	0.656
DBU	***LR***	0.457	0.490	0.622	0.671	0.645	0.664	**0.743**	**0.725**	0.677	0.681
	NB	0.489	0.600	0.448	**0.756**	0.589	0.684	0.617	0.571	0.511	0.627
	KNN	0.543	0.57	0.453	**0.85**	0.563	0.538	0.671	0.558	0.538	0.567
	RF	0.489	0.49	0.453	**0.967**	0.563	0.561	0.666	0.58	0.555	0.588
LOG4J	***LR***	0.436	0.440	0.647	0.554	**0.840**	0.577	**0.818**	**0.816**	**0.804**	**0.728**
	NB	0.468	0.490	0.637	0.629	**0.758**	0.672	**0.783**	**0.774**	0.692	**0.744**
	KNN	0.394	0.44	0.612	0.577	**0.883**	**0.768**	**0.803**	**0.78**	**0.768**	**0.719**
	RF	0.415	0.47	0.602	0.559	**0.961**	**0.724**	**0.78**	**0.735**	**0.719**	**0.716**
JFC	***LR***	0.511	0.500	0.602	0.601	**0.736**	0.698	**0.766**	**0.771**	0.697	**0.724**
	NB	0.511	0.530	0.597	0.685	**0.701**	0.689	**0.730**	**0.736**	0.632	**0.707**
	KNN	0.404	0.38	0.632	0.592	**0.775**	**0.855**	**0.794**	**0.782**	**0.855**	0.69
	RF	0.383	0.37	0.667	0.559	**0.749**	**0.952**	**0.77**	**0.766**	**0.958**	0.686
IVY	***LR***	0.383	0.340	0.622	0.596	**0.801**	0.511	**0.822**	**0.820**	**0.828**	**0.726**
	NB	0.479	0.480	0.667	0.634	**0.749**	0.696	**0.786**	**0.772**	**0.707**	**0.728**
	KNN	0.404	0.4	0.657	0.592	**0.775**	**0.786**	**0.893**	**0.784**	**0.786**	**0.726**
	RF	0.415	0.42	0.642	0.629	**0.766**	**0.736**	**0.964**	**0.777**	**0.741**	**0.708**
LUCENE	***LR***	0.394	0.360	0.657	0.596	**0.814**	0.533	**0.823**	**0.820**	**0.706**	**0.730**
	NB	0.468	0.500	0.662	0.615	**0.753**	0.689	**0.784**	**0.777**	**0.706**	**0.736**
	KNN	0.394	0.45	0.612	0.592	**0.805**	**0.789**	**0.803**	**0.863**	**0.789**	**0.711**
	RF	0.426	0.42	0.592	0.554	**0.753**	**0.753**	**0.798**	**0.964**	**0.765**	**0.705**
ANT	***LR***	0.404	0.360	0.692	0.596	**0.823**	0.513	**0.820**	**0.816**	**0.833**	**0.728**
	NB	0.394	0.390	0.692	0.629	**0.779**	0.628	**0.814**	**0.813**	**0.778**	**0.735**
	KNN	0.404	0.38	0.632	0.592	**0.775**	**0.855**	**0.794**	**0.782**	**0.855**	0.69
	RF	0.404	0.4	0.647	0.573	**0.762**	**0.964**	**0.788**	**0.753**	**0.965**	0.696
POI	***LR***	0.415	0.370	0.682	0.596	**0.823**	0.582	**0.822**	**0.816**	**0.837**	**0.728**
	NB	0.511	0.540	0.527	**0.709**	**0.714**	0.681	**0.725**	**0.712**	0.605	**0.718**
	KNN	0.489	0.46	0.572	0.568	**0.745**	**0.7**	**0.78**	**0.746**	**0.7**	**0.846**
	RF	0.5	0.49	0.617	0.592	**0.736**	0.698	**0.76**	**0.756**	0.697	**0.94**

of considered applications (affinities). In the experiment that we conducted to answer RQ1, the training datasets are composed of 1 system tests information history at the time. Thus, we could determine potential prediction affinities between different systems which could lead to introduce more affinity parameters that may be related to the systems type, or size, in order to determine the suitable kind of training dataset for a given new system. To address the second research question (RQ2), we conducted an empirical study that uses mixed training datasets from different systems, builds a classifier and tries to determine whether the datasets mixing can improve or degrade the predictions obtained in the previous experiment.

C. Classifiers and cross-system validation

We used machine learning classifiers trained on a system's test information data to provide a set of classes to be tested for other software systems. We added to the MLR and NB classifiers previously considered, the well-known K-KNN and RF algorithms to build prediction models from the datasets. KNN is an intuitive and fast algorithm that classifies observations according to their similarity. KNN is particularly suitable for pattern recognition. RF behaves well when irrelevant features are present or these features have skewed distributions. The larger the training dataset the more accurate the classifier. It makes RF particularly interesting when the training dataset is a combined information of different systems.

1) Cross System Validation

In the CSV technique, datasets collected from each of the 10 systems are used in turn as training set and the derived classifiers are cross-validated on each of the 9 remaining systems. In such approach, training dataset may be small depending on the considered system, hence the usefulness of NB classifiers which can perform on small datasets.

Table II presents the performances of models derived from MLR, NB, KNN and RF classifiers. In each table, the *cell(i,j)* shows the accuracy (1-error) of the 4 classifiers built from training dataset of the system on row *i*, and tested on the dataset of the system on column *j*. The diagonal *cells (k, k)* hold the adjustment (1 – optimistic error) of classifiers on the dataset of system *k*. We considered the models with accuracy values greater than 0.70 (error < 0.30) as good classifiers. In Table II, systems are sorted from the smallest one to the largest one in terms of number of classes.

The results show that the very small-size systems (MATH and IO) form bad training datasets and are not well predicted. All non-adjustment prediction rates related to both of the systems are smaller than 70%. Three reasons may explain these results:

- The small size of the systems. IO (with 100 classes) and MATH (with 125 classes) are the smallest systems in our datasets. The dataset they form may not be large enough to build good classifiers.
- The use of very specific criteria when selecting classes to be tested leading to over-fitting problems. This hypothesis is supported by the optimistic prediction rates observed on both systems' diagonal cells.

- The lack of a unit test strategy in small-size systems. With limited software classes, the testers could cover a higher percent of classes without any strategy. Thus, the poor rules selection makes irrelevant the information captured by software metrics. Candidate classes become unpredictable.

The medium-size systems (DBU JODA and LOG4J) results are mitigated. LR and NB models based on JODA training set well predicted all the larger systems except JFC. JODA's testers may use common criteria that are also used when testing larger systems, while DBU testers seem to use a particular strategy. As a consequence, we obtained the same performances as for the small-size systems. The same previous reason may explain DBU's results. LOG4J results are similar to large-size systems.

The LOG4J system, the large-size and the very large-size systems form good training sets and are well predicted by classifiers trained on their datasets. The results may be explained by the adoption of an effective tests prioritization strategy by testers in order to target the critical classes. Indeed, in large-size systems, testers may adopt an effective strategy to carefully select the software classes to be tested. The adopted strategy may suggest large, complex and highly coupled classes as unit test candidates. Complexity and coupling attributes are captured by the LOC, WMC and CBO metrics. This may explain why the associated systems are well predicted and form good training datasets.

JFC, ANT and IVY are of standalone type systems while remaining systems are libraries. The results suggest no affinity based on the application type. This is supported by the fact that many libraries are not well predicted by other libraries datasets. The good predictions obtained between standalone apps seem to be related to their size affinity.

The cross-validation results, especially for the medium, large and very large-size systems, show that it is possible, based on only a combination of metrics, to construct classifier models from existing software datasets that automatically suggest, for another software system, a set of classes to be tested. It also indicates that small systems are unpredictable and do not form good training datasets. Furthermore, the most obvious affinity between systems is related to their size. The larger the systems, the better the training datasets and the prediction levels of learners.

2) Leave One System Out Validation

In Leave One System Out Validation (LOSOV) technique, each system will be used in turn as testing dataset to validate the classifiers derived from the training dataset formed by the 9 remaining systems. This approach uses large training datasets and combines different selection criteria if they exist. However, if one of the systems' testers randomly selects classes to be tested, it may impact the whole training dataset quality depending on that system size.

Table III presents the accuracy rates of the classifiers for LOSOV. The results confirm those obtained using the CSV and show that each of medium, large and very large size systems are well predicted by classifiers trained on merged dataset of 9 remaining systems. The best accuracy rates vary from 70.1% to 93.2%.

Some classifiers failed to correctly predict some systems: NB on JFC system, KNN on POI system and RF on POI system. The POI data test prediction results may be explained by its size. Indeed, leaving POI out from training datasets during the LOSOV may drastically reduce the training dataset (POI represents 30.7% of classes and 30.5% of tested classes).

TABLE III: LEAVE ONE SYSTEM OUT VALIDATIONS

	LR	**NB**	**KNN**	**RF**
MATH	0.404	0.457	0.447	0.468
IO	0.37	0.53	0.5	0.41
JODA	0.667	0.627	0.567	0.647
DBU	0.596	0.629	0.573	0.559
LOG4J	**0.818**	**0.736**	**0.71**	**0.701**
JFC	**0.834**	0.661	**0.846**	**0.92**
IVY	**0.84**	**0.796**	**0.76**	**0.747**
LUCENE	**0.82**	**0.746**	**0.732**	**0.702**
ANT	**0.834**	0.661	**0.839**	**0.932**
POI	**0.731**	**0.734**	0.681	0.679

The small-size systems are not well predicted since all classifiers' prediction rates are smaller than 70%. This result suggests that their testers used very specific criteria or uncaptured (by metrics) criteria, or may be no criteria during the selection candidate classes. This result is plausible for small systems, since testers could test all classes, they also may not need any particular strategy.

Compared with CSV results, LOSOV slightly improve and degrade the prediction levels in several cases. However, LOSOV takes advantage of being usable in real conditions, in the context of a collaborative tool that supports unit tests prioritization when different information history (metrics and tests data) are provided by different teams of developers.

VI. THREATS TO VALIDITY

The study we presented in this paper 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 measuring methods and approaches have limitations that can restrict the generalization of certain conclusions.

The external validity threats are mainly related to the application domain of considered systems. Indeed, some analyzed systems are mathematical algorithms libraries (IO), while other systems have more complex architectures and involve many OO-technology specific artifacts such as inheritance and polymorphism (JFC). Hence, when a learning algorithm is trained on some types of systems, it could be well-adjusted when tested on the datasets of similar domain systems and not able to suggest good candidate classes for other types and domain systems.

The data we collected does not provide any information on tested classes selection criteria. It may be that, for some systems, tested classes were randomly selected particularly for the small size systems.

The main threat of construct validity lies in the technique we used to match JUnit test classes to software classes during the *tested classes* identification. Indeed, the remaining unpaired software classes that are tested by transitive method invocations are ignored by our approach.

VII. CONCLUSIONS AND FUTURE WORK

Ten open source (Java) software systems have been analyzed in this study and totalize more than 4400 software classes. The testers of each system developed dedicated unit test classes for a subset of classes using the JUnit Framework. In our previous investigations, we explored the possibility of explaining and reusing the selection criteria for different systems through three experiments using three source code metrics.

This study extends our previous work by including the

KNN and RF classifiers to the MLR and NB classifiers. In addition to the CSV, we used LOSOV validation technique, which merges training sets from different systems. The main objective was to know to what extents the combined information of different systems could be a good training set for the learners and to determine if there exist affinities between different datasets of systems' test information history. Results show that systems with more than a hundred classes have their *tested classed* generally well predicted by classifiers built from medium, large and very large systems training datasets. Furthermore, the obtained results suggest that all obtained classifiers could help to support unit tests prioritization with more than 70% of accurate predictions. The results of the experiments we conducted become particularly interesting knowing that effort prioritization is especially useful during large and complex systems testing. It demonstrates the viability of a unit tests prioritization automation technique that uses classifiers trained on merged software source code metrics with the unit tests information history. Grouping the systems according to their domains could improve our results. Since the proposed prioritization technique suggests a slightly different (30%) tested classes from those of the testers, it would be pertinent to analyze and compare their actual performance on covering faulty classes. This topic will be the next direction of our investigations.

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] Gupta V., Aggarwal K.K. and Singh Y., 2005, A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability, Journal of Computer Science, Vol. 1, No. 2, pp. 276-282.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Toure F., Badri M. and Lamontagne L., 2014. Towards a metrics 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Wong W., Horgan J., London S., and Agrawal, H., 1997. A study of effective regression in practice, Proceedings of the 8th International Symposium on Software Reliability Engineering, November, p. 230–238.

[14] Mirarab S. and Tahvildari L., 2007. A prioritization approach for software test cases on Bayesian networks, In FASE, LNCS 4422-0276, pages 276–290.

[15] Walcott K.R., Soffa M.L., Kapfhammer G.M. and Roos R.S., 2006. Time aware test suite prioritization, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006). ACM Press, New York, 1–12.

[16] 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.

[17] 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.

[18] 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.

[19] Elbaum S., Rothermel G., Kanduri S. and Malishevsky A.G., 2004. Selecting a cost-effective test case prioritization technique, Software Quality Control, 12(3):185–210.

[20] Boehm B. and Basili V. R., 2001. Software defect reduction top-10 list, Computer, vol. 34, no. 1, pp. 135–137.

[21] Ray M. and Mohapatra D.P., 2012. Prioritizing Program elements: A pretesting effort to improve software quality, International Scholarly Research Network, ISRN Software Engineering.

[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] 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.

[24] Li W., and Henry S., 1993. Object-Oriented Metrics that Predict Maintainability Journal of Systems and Software, vol. 23 no. 2 pp. 111-122.

[25] 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.

[26] 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.

[27] McCabe T. J., 1976. A Complexity Measure, IEEE Transactions on Software Engineering: 308–320.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Rompaey B. V. and Demeyer S., 2009. Establishing traceability links between unit test cases and units under test, in Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR '09), pp. 209–218.