Assigning Test Priority to Modules Using Code-Content and Bug History

Ermira Daka, and Egzon Mustafa

***Abstract*—Regression testing is a process that is repeated after every change in the program. Prioritization of test cases is an important process during regression test execution. Nowadays, there exist several techniques that decide which of the test cases will run first as per their priority levels, while increasing the probability of finding bugs earlier in the test life cycle. However, sometimes algorithms used to select important test cases may stop searching in local minima while missing the rest of the tests that might be important for a given change. To address this limitation further, we propose a domain-specific model that assigns testing priority to classes in applications based on developers' judgments for priority. Moreover, our technique which takes into consideration applications' code content and bug history, relates these features to overall class priority for testing. In the end, we test the proposed approach with a new (unknown) dataset of 20 instances. The predicted results are compared with developers' priority score and saw that this metric can prioritize correctly 70% of classes under test.**

***Index Terms*—Regression testing, Machine learning, Test prioritization, Bug history, Code complexity.**

I. Introduction

With the development of various applications, today people have easier communication, faster services, and lower costs both financially and in terms of time. Producing a software application is a process that passes through several stages otherwise known as Software Development Life Cycle (SDLC). However, to produce an application that has the quality and meets users' expectations, one of the important processes that should be given special attention, is software testing. In principle, no application can be considered a zero-bug application, but through testing, a level of confidence can be achieved regarding how efficient and functional an application is. Alternatively, testing can also be defined as a set of activities and procedures that must be undertaken to evaluate certain aspects of an application [1].

During all stages of software development, the application undergoes modifications. These modifications may be due to new features, change in the existing functions, or even removing part of functions not useful at that point. All these changes increase the possibility of introducing new bugs in the system, which requires testing and making sure that the modifications applied have not affected the functional parts of the software. This practice is also known as regression testing (RT).

RT is a costly process that needs to be repeated in each change, which means the amount of testing that needs to be performed each time increases. One of the methods of reducing testing load is tests prioritization. Test prioritization is a technique that first runs important tests, while less important ones are ignored [2] if conditions are so. However, since search algorithms that are used to select tests may be stuck on local minima, this process has its limitations, too.

In this paper, we will try to give another perspective on RT, where we will propose a novel technique that instead of ranging tests, will select classes that need to be tested based on their internal code content, and bug history. For this, we design a domain-specific model that prioritizes classes or modules based on the developer's judgments. Furthermore, we present:

- An overview of the phases used to generate and test a model for class priority assignment. (Section III)
- Two case studies taken from an industrial company and their code content and bug history data. Furthermore, we introduce a new technique that can parse code content or its number of statements and methods. (Section III-A)
- A classification model that can predict test priority for a given application under development. (Section III-B)
- A comparison between model priority prediction, and developer priority assignments for classes under test. (Section III-B1)

With collected data for both applications, we create a dataset of 79 instances with four attributes such as *number of methods*, *number of statements*, *number of issues within time,* and *priority to be tested.* The model is further trained using the Random Tree algorithm which gave the best precision and recall result with a value of 0.59. Our technique that can assign priority from 1 (very important to be tested) to 5 (less important to be tested), succeed in predicting priority for 70% modules. Although there are 6 incorrectly prioritized classes, the range of incorrectness is 1 for four classes, and 2 for two classes, while none of the instances have a large distance between what developers said and what the model predicts.

II. Related Work

Software testing is a process that exists at every stage of development. The idea of testing is that an application designed and developed based on a set of requirements, should meet its expectations, and achieve the purpose for which it was

Manuscript received December 23, 2021; revised March 22, 2022. Date of publication April 19, 2022. Date of current version April 19, 2022.

E. Daka and E. Mustafa are with the Department of Computer Science and Engineering, University for Business and Technology, Prishtina, 10000 Kosovo (e-mails: ermira.daka@ubt-uni.net, em30102@ubt-uni.net).

Digital Object Identifier (DOI): 10.24138/jcomss-2021-0185

created. However, testing is costly in terms of time and effort. After each change that is done in an application, testing is repeatedly done, and this is called regression testing. The main purpose of regression testing is to prove that certain developments, the implementation of new functionality, or the fixing of a preliminary problem have not caused problems in the existing functionalities of the software application. According to the IEEE standard, RT is defined in this form: "Selective retesting of a system or component to verify that the modifications made have not caused adverse effects and that the system or component still complies with the specified requirements." [3]
Mathematically, RT is defined as follow:

- Let T be the set of test cases for application P
- Let P' be the new version of the application P after changes are applied.
- Select $T' \subset T$, a set of test cases to execute on P'

Each time that RT is applied in an application, the total cost of it will be increased if we take into consideration the new and existing tests that need to be executed. Therefore, to normalize these costs, test cases are selected according to their priority, or the most important test is placed at the beginning of the execution process [4], [5]. The main reason behind this is to achieve test coverage as soon as possible or to find failing tests at the beginning [6].

One perspective that may affect test prioritization is the way developers do testing. It is well known that automated testing decrease the effort and time of testing in general, however, developers still spend a good amount of time on writing tests [7]. Taking this into consideration, test prioritization is listed as an important process in both manual and automated testing, and there are proposed various techniques that help in sequencing the tests that need to run based on the priority required [6].

Today there exist several test prioritization techniques, such as:

- *Coverage-Based* approach which aims to prioritize or select test cases based on the coverage that they have on the application under test. This means that tests with higher code coverage have higher priority [8]. Most prioritization techniques are based on code coverage [9], however, coverage alone is seen to not be enough [10]. Thus there are proposed other criteria, too.
- *Requirement-based* is another approach that is based on the initial requirements on which the software application is developed. During this process, the user is not focused on code [11]–[13], but more on customer priority, changes in the requirement, implementation complexity [14].
- *Risk-based* is an approach that is mainly based on the aspects of risk which derive from the requirements of the software application and also applies to those applications where risk is a priority. Amland [15] defines risk as the probability of an error that can occur and the cost of the error if we are in the production phase.
- *Search-based* is a test case prioritization approach that proves to find cost-effective problem solutions [16]. Some of the key and most popular search-based access algorithms are : *Greddy, Additional Greedy, Genetic algorithms*.
- *Fault-based* is a test case prioritization approach that is based on the potential that a module or component may have to fail [17].
- *History-based* is another approach that is based on the previous test execution history. Kim and Porter in their work had a hypothesis saying that history-based test prioritization, can reduce the testing cost, and increase the effectiveness of regression testing [18].

Besides all the listed test prioritization techniques, different articles use code complexity as a criterion for test selection. Software Complexity Metrics measure the cost of software development, maintenance, and usage [19], meanwhile, they are closely related to error distribution in the current code [9]. Afzal et al. [20] in their work, propose a new approach that uses path complexity and branch coverage to prioritize test cases. This approach outperforms the existing branch coverage-based approach in terms of APFD (Average Percentage of Faults Detected) up to 18% on average.

The complexity-based prioritization technique assumes that complex code contains more bugs. Since systems in use may become complex over time, there has been work where researchers measure the complexity of software in the previous several years. Hence, estimates show that existing software systems achieved to receive 40% to 70% of total expenses [10]. Heard et.al. [21] proposed an approach that computes the testing importance of each module by using fault proneness and importance of the module from system and user perspectives. Their work became even stronger with the proposed metric(Average Percentage of fault-affected Modules Cleared per test case) which measures the effectiveness of various prioritization techniques.

Based on these findings, we started to work on this article assuming that complex code is more difficult to change, is more likely to contain faults, and is difficult to be integrated. Therefore, we took code internal content and bug history as two criteria to further proceed with an approach of module prioritization for RT.

III. The Proposed Approach

Our new approach for testing priority, is based on two industrial case studies or two software applications developed using C# programming language. Both of them are produced by the same company and are used by real users. Applications are delivered to us with their source code, bug history, and testing priority assigned to each class that they contain. However, to collect even more information for their code content, we implemented a parser that reads through every class in the application and counts the number of statements and methods that they have (Phase 1 in Figure 1). This process, will give us a general understanding of code to be prioritized, and shows its correlation with bug history and priority assigned by developers. After collecting all four parameters (number of issues, number of statements, number of methods, and testing priority), in the second phase (Phase 2 in Figure 1), we created a dataset of 99 instances where 79 are used for training, and 20 are used for testing the model.

In the last phase (Phase3 in Figure 1) we use supervised machine learning to construct a predictive model of test priority from those features. To predict the testing priority of a new class, we used 20 new instances and their feature values and apply the learned model. In this paper, we use a simple Linear Regression learner [22] and Random Tree classifiers, although in principle other learners are also applicable (e.g., multilayer perceptron [22]). However, with Random Tree classifier, the resulting model (which consists of weightings for individual features) is quick, and can easily be interpreted.
In this section, we describe how we collected the data to learn this model, the features of classes under the test we considered, and the machine learning algorithms that we applied to create our final model. Moreover, we applied the trained model to a test dataset and compared the results with developers' priorities for each class.

A. Phase1 - Code Review and Bug History Collection

To have real data and do a study on them, we agreed with a local software development company containing 51 employees, 12 of which were software developers. This mid-size company, was asked to provide us with some applications that were representative of their overall development history, implemented using the same principles (especially bug tracking and project management system), within a similar time frame, and for our local market. Hence, they offered two applications named *iTms*, and *Real Estate* which will be subject to the process of class prioritization for testing. *iTms* is more complex in terms of internal structure that it has, while *Real Estate* is simpler (less code amount). For both applications the necessary data have been collected which will contribute to the process of evaluation and completion of this work. The components that were needed to complete the paper include the project solution (all application classes) and the development history of each module within the software application (total number of problems for each component).

Firstly, to parse the inner structure of the given applications, we implemented a console program in C# programming language, through which a single solution is read at the same time. This means that each class of an application under parsing is filtered based on the content of their code. Code content analysis is based on two main parameters which are:

- Number of methods,
- Number of statements.

Therefore, each class's source code in the applications' folder is read, and their number of statements and methods are counted. While parsing methods in the class, their type is checked (protected, private, and public), too. Also, the control over the number of methods does not take into account the interface and other classes that may be within the parent class. Regarding the number of statements, we took into consideration all lines in the class that contain an action. This may include declaring variables, assigning values, calling methods, looping, and branching expressions.

TABLE I
ITMS AND REAL ESTATE CLASS PRIORITIZATION LIST.

Application	*iTms*	*Real Estate*
Number of Classes	90	9
Priority 1	3	1
Priority 2	1	1
Priority 3	2	0
Priority 4	10	1
Priority 5	74	6

Finally, to rank the classes based on the number of methods and statements, we assigned an internal 'priority', which will show the internal structure of a given class in the application. Thus, the sum of total methods and statements for each class is taken and divided into 5 ranges. The difference from this division also determines the priority number that a certain class has within the software application under testing. Respectively, the classes with the largest number of methods and statements belong to priority 1, then for the other classes, the priorities are divided depending on the other number of methods and statements together, until the end where the classes with the smallest number of methods and statements have priority 5.

In Table I, there is presented code content analyses for each applications. From 90 constituent classes of *iTms* application, 3 classes have priority 1, only 1 class has priority 2, 2 classes have priority 3, 10 classes have priority 4 and 74 classes have priority 5, respectively. Hence, from 9 constituent classes of *Real Estate* application, 1 class has priority 1, 1 class has priority 2, 0 class has priority 3, 1 class has priority 4 and 6 classes have priority 5.

Secondly, as part of the data-gathering phase, we have requested company bug history data for both applications. To collect the information that we exactly needed, we requested to have only 'bugs' or issues that caused a test case to fail (e.g., error in the code, incomplete user requirement) and is registered as change. We did not collect changes in the code that were due to additional user requirements. Bug history was obtained from the whole development period and was downloaded from an existing internal tool that the company uses for managing application development and troubleshooting. However, to apply companies' confidentiality rules, we did not have the opportunity to obtain data regarding the time distribution of bugs during the software development lifecycle. In Table II and Table III, we have listed *iTms* and *Real Estate* applications' bug history, respectively. Moreover, there are listed the number of problems for each module/component, their priority which is again assigned by developers, and the code content details (number of methods and number of statements) extracted from our parser mentioned in Phase1.

In Figure 2 we visually demonstrated all the classes under study, the number of methods and statements that they contain, and the number of issues that they had until that point. As shown in the figure, in most cases, when the class has more statements and methods, the higher the number of issues for it. However, there are seen edge cases, too, where even though the module is small, it might have issues.

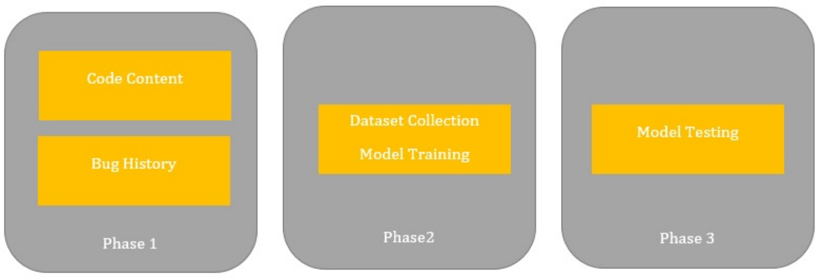

Fig. 1. Three phases that are used to create the model for class prioritization during regression testing.

TABLE II
FIRST 32 ITMS CLASSES, THEIR CODE CONTENT, ISSUE HISTORY, AND THE PRIORITY GIVEN BY DEVELOPMENT TEAM.

Controllers/Modules	*Module Priority*	*No. Issues*	*No. of Methods*	*No. of Statements*
AccidentReportAssessmentController.cs	4	15	14	19
AccountController.cs	1	100	195	372
AdminSettingsController.cs	3	66	13	16
AlertsController.cs	3	30	34	45
AppointmentsController.cs	4	8	6	4
AssetsController.cs	3	16	13	43
ATMS_Authorize.cs	3	5	10	7
AuthorizeUserAttribute.cs	3	45	53	118
BranchesController.cs	2	88	49	139
BudgetPeriodsController.cs	5	2	9	11
BusinessController.cs	4	11	8	18
ChartsController.cs	3	19	7	0
ClientAssetsController.cs	3	32	13	43
CommunicationTypesController.cs	5	11	8	3
ConfigurationTypesController.cs	5	21	9	3
ContractsController.cs	3	55	12	6
CustomersController.cs	2	136	9	3
DashboardController.cs	2	61	71	124
DocumentsController.cs	3	31	8	3
DocumentsTypesController.cs	5	6	8	3
DynamicLibrary.cs	5	18	30	176
ElementsController.cs	4	7	11	9
EmployeesController.cs	2	111	44	69
EncryptUploadFilesController.cs	5	0	1	0
ErrorsController.cs	5	22	6	3
ExpenseCategoryController.cs	5	4	14	8
ExpenseDepartmentController.cs	5	3	15	8
ExpenseInternalController.cs	4	15	16	7
ExpensePaymentMethodController.cs	4	0	11	6
ExpensesController.cs	1	420	108	171
ExpenseStatusController.cs	5	0	14	9
FilesController.cs	2	95	78	67

TABLE III
REAL ESTATE CLASSES, THEIR CODE CONTENT, ISSUE HISTORY, AND THE PRIORITY GIVEN BY DEVELOPMENT TEAM.

Controllers/Modules	*Module Priority*	*No. Issues*	*No. of Methods*	*No. of Statements*
AccountController.cs	2	15	28	20
BaseController.cs	3	0	0	0
ForRentController.cs	1	24	2	3
ForSaleController.cs	1	35	2	3
HomeController.cs	3	10	3	0
ManageController.cs	2	22	23	10
RealEstateController.cs	1	7	3	5
UserProfileController.cs	2	17	5	7
UserProfilePersonalController.cs	2	8	5	2

B. Phase 2 - Model Feature Extraction and Training

During this phase, we prepared a training data set for a model that can further predict the priority for a class to be tested during the Regression Testing phase. Therefore, we took both of these applications which in total have 99 classes, and listed 4 different features for each of them. However, to have a set of test data for later phases of model testing, we used only 79 instances as training data, and keep 20 for

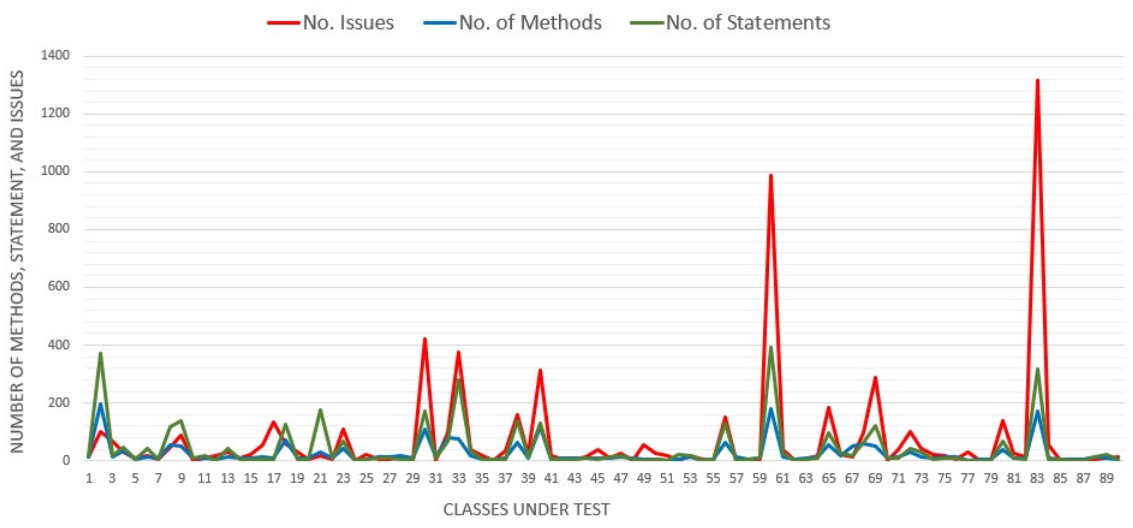

Fig. 2. Comparison of number of issues with number of statements and number of methods for each class.

TABLE IV
MODEL TRAINING, AND THEIR PERFORMANCE VALUES USING WEKA.

Algorithm	*Correlation coefficient*
Linear Regression	0.45
Random Tree	0.59

the testing purpose. Model attributes are in fact the features that we measured during the study, or *number_of_issues*, *number_of_methods*, *number_of_statements*, and the final attribute which is the *class* or *priority_score*.

1) Application of Machine Learning for Module Prioritization for Testing: Using the data collected for both case studies, we have applied Weka [23] Machine Learning (ML) tool for test priority model learning. We applied our dataset with different learners available in Weka, however, for the sake of result interpretation we choose to proceed with Linear Regression, and then a classifier such as Random Tree. After executing each of these algorithms we listed the results for both learners in Table IV.

Each algorithm is selected to train the model with the 10-fold cross-validation option. It enables the data set to be randomly divided into 10 sub-examples, where a single sub-sample is stored as validity data for model testing, and the remaining 9 sub-samples are used as training data. This process is repeated 10 times (folds), with each of the 10 sub-examples used exactly once as validation data.

As it is seen from Table IV, the Random Tree classifier has a higher correlation coefficient compared to Linear Regression, even though the last one is simpler to be interpreted in terms of feature importance in testing priority.

C. Phase 3 - Model Testing

Following the trained model with the Random Tree algorithm, the next step was to test our dataset of 20 instances that we saved in the beginning. As we already have the testing priority for each of the 20 classes, we saved those numbers for comparison between what our model predicted and what developers said is important to be tested.

In Table V, we can see that after applying the test data (test data set), the model we have trained has managed to make the correct prediction in 70% of instances or 14 out of 20 classes. Moreover, the results show that for each case where our model miss-predicted the priority, the error rate is small taking into account that the interval of priorities is between one and five. In four classified instances, the difference between the actual and the predicted value is a unit (instances 2, 6, 15, and 18) or small, or two units (instances 8, and12) medium. This tells that, our model can predict how important is a class to be tested, as well as a software developer aware of code history.

Finally, even though the agreement between the model and developers seems to be substantial (70%), still there is enough space for model accuracy improvement. According to actual studies, to improve the prediction accuracy, researchers tend to increase their data-sets, improve data-points (detect outliers), or combine predictors [24]–[26]. For our study even though we considered increasing the overall data-set, obtaining data from industry is not always possible due to many reasons (e.g., confidentiality, projects belonging to different development time-frame, projects belonging to different teams). Therefore, we continued to base it on the projects that are produced in the same period, using the same issue tracking and project management system and developed by the same team. Hence, we concluded with a result that can be improved, but which seems to be a general issue for most of the prediction metrics, and keep to be a continuous open area that requires work [27]–[29].

IV. CONCLUSIONS

Testing software applications is a very important process, but also costly because it requires sufficient resources for the efficient implementation of this flow. In general, we can say that the prioritization of the test case enables cost reduction in regression testing or general software development. To address

TABLE V
PRIORITY MODEL RESULT PREDICTION. EACH INSTANCE CONTAINS ITS REAL PRIORITY SCORE (ACTUAL VALUE), AND PREDICTED SCORE WITH RANDOM TREE ALGORITHM.

Algorithm	*Instance number*	*Actual value*	*Predicted value*
Random Tree	1	5	5
	2	3	4
	3	2	2
	4	3	3
	5	3	3
	6	3	4
	7	4	4
	8	3	1
	9	5	5
	10	5	5
	11	2	2
	12	2	4
	13	5	5
	14	1	1
	15	2	3
	16	5	5
	17	5	5
	18	4	5
	19	5	5
	20	4	4

this problem, we have considered two industrial applications and built a predictive model that assigns testing priority to classes under development, based on data on how developers prioritize them. The features that we used to train the model are all methods and statements in the code, extracted using an implemented parser, bug history and testing priority, extracted using information given from software developers. We have applied this model to the new test dataset, and compared its results with developers' priority assignments. Our technique to prioritize classes for further regression testing improves the quality of the product by prioritizing features that need testing and decreasing the time and effort required for quality assurance. In summary, we proposed a domain-specific model of testing priority based on code complexity and bug history. We found that our model agrees with developers in 70% of cases about which modules in the application are important to be tested.

REFERENCES

[1] I. Burnstein, *Practical Software Testing: A Process-Oriented Approach*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[2] Y. Lou, J. Chen, L. Zhang, and D. Hao, "Chapter one - a survey on regression test-case prioritization," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 113, pp. 1–46. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245818300615

[3] IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std., 1990.

[4] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing Verification and Reliability*, vol. 22, no. 2, pp. 67 – 120, Mar. 2012.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[6] A. G. Malishevsky, "The general prioritization framework," *Comput. Sci. J. Moldova*, vol. 24, no. 2, pp. 192–201, 2016. [Online]. Available: http://www.math.md/publications/csjm/issues/v24-n2/12191/

[7] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211.

[8] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, and S. Jalali, "Incorporating fault-proneness estimations into coverage-based test case prioritization methods," *Information and Software Technology*, vol. 121, p. 106269, May 2020. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2020.106269

[9] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: an empirical study," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, 1999, pp. 179–188.

[10] G. Rothermel, R. H. Untch, C. Chu and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[11] H. Srikanth, C. Hettiarachchi, and H. Do, "Requirements based test prioritization using risk factors," *Inf. Softw. Technol.*, vol. 69, no. C, p. 71–83, jan 2016. [Online]. Available: https://doi.org/10.1016/j.infsof.2015.09.002

[12] X. Wang and H. Zeng, "History-based dynamic test case prioritization for requirement properties in regression testing," in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, ser. CSED '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 41–47. [Online]. Available: https://doi.org/10.1145/2896941.2896949

[13] T. Ma, H. Zeng, and X. Wang, "Test case prioritization based on requirement correlations," in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2016, pp. 419–424.

[14] R. Kavitha, V. Kavitha, and N. Suresh Kumar, "Requirement based test case prioritization," in *2010 International Conference on Communication Control and Computing Technologies*, 2010, pp. 826–829.

[15] S. Amland, "Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study," *Journal of Systems and Software*, vol. 53, pp. 287–295, 09 2000.

[16] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[17] F. Farooq and A. Nadeem, "A fault based approach to test case prioritization," 12 2017, pp. 52–57.

[18] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 119–129. [Online]. Available: https://doi.org/10.1145/581339.581357

[19] M. J. Harrold, "Testing evolving software: Current practice and future promise," in *Proceeding of the 1st Annual India Software Engineering Conference, ISEC 2008, Hyderabad, India, February 19-22, 2008*, G. Shroff, P. Jalote, and S. K. Rajamani, Eds. ACM, 2008, pp. 3–4. [Online]. Available: https://doi.org/10.1145/1342211.1342213

[20] T. Afzal, A. Nadeem, M. Sindhu, and Q. uz Zaman, "Test case prioritization based on path complexity," in *2019 International Conference on Frontiers of Information Technology (FIT)*, 2019, pp. 363–3635.

[21] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 523–534.

[22] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed., ser. Morgan Kaufmann Series in Data Management Systems. Amsterdam: Morgan Kaufmann, 2011. [Online]. Available: http://www.sciencedirect.com/science/book/9780123748560

[23] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, and I. H. Witten, *Weka: A machine learning workbench for data mining.* Berlin: Springer, 2005, pp. 1305–1314. [Online]. Available: http://researchcommons.waikato.ac.nz/handle/10289/1497

[24] W. Li, W. Mo, X. Zhang, J. Squiers, Y. Lu, E. Sellke, W. Fan, M. DiMaio, and J. Thatcher, "Outlier detection and removal improves accuracy of machine learning approach to multispectral burn diagnostic imaging," *Journal of biomedical optics*, vol. 20, p. 121305, 12 2015.

[25] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction a large scale experiment on data vs. domain vs. process," 08 2009, pp. 91–100.

[26] F. Yucalar, A. Ozcift, E. Borandag, and D. Kilinc, "Multiple-classifiers in software quality engineering: Combining predictors to improve software fault prediction ability," *Engineering Science and Technology, an International Journal*, vol. 23, no. 4, pp. 938–950, 2020.

[27] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 107–118. [Online]. Available: https://doi.org/10.1145/2786805.2786838

[28] A. Amin, L. Grunske, and A. Colman, "An approach to software reliability prediction based on time series modeling," *Journal of Systems and Software*, vol. 86, p. 1923–1932, 07 2013.

[29] S. N. Umar *et al.*, "Software testing defect prediction model-a practical approach," *International Journal of Research in Engineering and Technology*, vol. 2, no. 5, pp. 741–745, 2013.

Ermira Daka is a lecturer in the University for Business and Technology (UBT) in Kosovo. She received the BSc. degree in Electrical and Computer Engineering in the University of Prishtina on 2006, MSc. degree in Software Design in the University of Lugano on 2010 and Ph.D. degrees in Software Testing and Verification in the University of Sheffield on 2018. Dr. Daka is author of many proceedings in the area of software testing. Her research interests include automated software testing, machine learning and its application in software test automation.

Egzon Mustafa received the BSc. degree in Computer Science and Engineering in the University of Business and Technology (UBT) - Kosovo, MSc. degree in Data Science in the University of Business and Technology (UBT) - Kosovo, in 2017 and 2021 respectively. His research interests include software testing, machine learning, data mining, big data, blockchain etc.