Some Notes on Software Diversification and Test Amplification Using Machine Learning Methods

César Soto-Valero
Email: cesarsotovalero@gmail.com

***Abstract*—The application of machine learning methods has proven to be a successful approach for managing a wide variety of computer science problems. The aim of this technical report is to present some ideas related to the analysis of source code and software systems using machine learning techniques. In particular, we focus our study on its applications to (1) software diversification and (2) automatic test amplification. Due to the nature of this two problems, machine learning methods are suitable tools to leverage its development. We review part of the existing literature and discuss new ideas regarding these issues, which could serve as a starting point towards further research on both fields.**

***Keywords*—machine learning, software diversification, automatic test amplification.**

I. Introduction

Machine learning refers to the detection of meaningful patterns in data [1]. For many real scenarios, due to the complexity of the patterns that need to be detected, a human programmer cannot provide an explicit, fine detailed specification of how such tasks should be executed. Taking example from intelligent beings, many of our skills are acquired or refined through learning from our experience (rather than following explicit instructions given to us). Machine learning methods are concerned with endowing programs with the ability to "learn" and adapt from the environment.

Machine learning provides the technical basis of data mining, in which has been widely used to extract information from the raw data present in databases [2]. The goal is to construct models that adapt to the changes in the system and infer valuable information from data for some specific purpose. Thus, it involves the implementation of algorithms that can obtain useful knowledge from data, even structured data, without relying on static rules–based programming.

The kind of knowledge obtained can be used for prediction, explanation, or understanding of the data. The learning process can be guided in three different ways: supervised, semi-supervised and unsupervised [3]. Supervised learning deals with approximating a target function from labeled examples (e.g., lazy learning, decision trees, bayesian learning, neural networks, support verctor machines). Unsupervised learning attempts to learn patterns and associations from a set of objects that do not have attached class labels (e.g., clustering, association rules). Semi-supervised learning consists in learning from a combination of labeled and unlabeled examples (e.g., expectation–maximization with generative mixture models, self–training, co–training, transductive support vector machines).

In the past couple of decades it has become a common tool in almost any task that requires information extraction from large datasets. Machine learning algorithms have proven to be of great practical value in a variety of application domains such as natural language processing, pattern recognition in images or surveillance videos, detection of web spam, genetics and genomics, etc. The field of software engineering turns out to be a fertile ground where many software development and maintenance tasks could be formulated as learning problems and approached in terms of learning algorithms [4].

Machine learning algorithms can give us insights into software processes and products, such as what software modules are most likely to contain bugs [5], what amount of effort is likely to be required to develop new software projects [6], what commits are most likely to induce crashes [7], how the productivity of a company changes over time, how to improve productivity [8], etc.

We see a wealth of opportunities in this research area. Recently, the study of software diversification has emerged as an active research field in software engineering [9]. In this report, we address the problem of diversifying software systems and present some ideas about it using machine learning paradigms. Furthermore, we study the application of machine learning techniques to the problem of the automatic generation of testing cases [10] that aims to enhance the coverage of tests cases with respect to a well-defined engineering goal.

This technical report is organized as follows. Section II reviews the problem of software diversification and discusses some ideas regarding to the application of machine learning concepts in this area. Section III summarizes the automatic test amplification procedure and gives some research insights to the use of machine learning in the search-based test amplification field. Finally, Section IV concludes the report and outlines some issues for future research.

II. Software Diversification

During the traditional flow of software development, all instances (or clones) of a software are commonly deployed using the same code and logic. Accordingly, an attack that works successfully for one of its instances also works on all. Thus, the robustness and security of the systems is affected because of the feasibility of replication of an attack in any of the instances. Software diversification provides a viable solution

to manage this issue, representing a natural evolution to the traditional software construction and deployment paradigms.

Software diversification consists in the modification of software in order to create many instances of the same software, but with different implementations, while providing identical functionality [9]. Software diversification aims to increase the adaptability and robustness of the whole software system, making it more secure and resistant to attacks or perturbations in the environment.

When a software system is diversified, since the generated instances contain different code and logic, an attack on one instance is not guaranteed to work on another instance. This can slow the spread of attacks and mislead attackers, increasing the security of the software system and preventing them for exploiting massively. For instance, diverse computing environments decrease the chances for a successful worm or botnet attacks [11], as both types of attack rely on uniform and compatible environments in order to replicate (worm) or initiate (botnet). Diversification can therefore be viewed as an obfuscation technique for increasing the robustness and adaptability of software systems. Its main goal is to promote adaptive capacities in the face of unforeseen structural and environmental variations.

The first experiments with software diversification investigated its advantages for fault tolerance [12], [13]. More recently, source code randomization has gain attention for cyber security [14], [15]. Baudry et al. [9] extended the concept of software diversity to a wide range of facets. For instance, natural software diversification is perceived as a form of software diversity that could emerge spontaneously during development and is more common in open source communities [16]. On the other hand, automated software diversification consists of techniques for artificially and automatically synthesizing the diversity [17].

Many recent works have experimented with the integration of multiple forms of diversity in the software systems in order to obtain benefits from several forms of protection. The idea is to use biological evolutionary mechanisms with the aim of facilitate the emergence of multiple forms of software diversity in collaborative adaptive systems, through automatic transformation and evolution [18]. For example, [19] et al. aim to break the application monoculture of web applications by promoting multi-tier diversification, combining natural and automatic diversity, in a realistic web–based architectural setting. In this scenario, the expected outcome is a set of software evolution and maintenance methods that spontaneously sustain diversity in collaborative adaptive systems.

One of the current problems of software diversification is determining better ways to explore the space of transformations. In addition, there is no clear consensus of how to measure and evaluate the quality of the diversity for different scenarios [20], [21]. Test suites dispose diversification inside different regions of programs in very unequal ways (i.e., diversification has different performances on a statement that is covered by one hundred test case than on a statement that is covered by a single test case [22]). Furthermore, diversification has a direct impact on distribution and maintenance. For example, when the binary code of an application must be signed by a third-party, the production of millions of diverse variants becomes a challenge. Another example is dump trace analysis or incremental updates. This will require accurate traceability of variants and reversible code transformations, as well as new forms of code analysis for automatic patching.

The application of machine learning methods represents an attractive approach to handle some of these concerns in software diversification. For example, extending the space of transformations through the search of more suitable variants using some heuristic mechanism, clustering functionalities that are best related each other in order to achieve a more specific diversification of the system, or analyzing the behavior of the natural diversity in software repositories. In the following, we address in more details some of these approaches.

A. *Generation of synthetic diversity*

Recent work focus on the automatic synthesis of software instances in order to maximize the potential impact of diversification for system's resilience. For example, Feldt [23] used genetic programming to automatically synthesize variants of an aircraft controller in order to achieve a better response to failure diversity, Rinard et al. [24], [25] developed unsound program transformations that support the runtime production of diversity and handle changes in quality of service, Forrest et al. [26] have explored genetic programming for automatic bug fixing and neutral mutation [27].

Another interesting approach in the generation of synthetic diversity is the work of Baudry et al. [28]. They create "sosie" programs, which are variants of software that exhibits the same functionality, passing the same test suit but computationally diverse in control statements or data flow. The generation of these sosies is based on the transformation of the original program through statement deletion, addition or replacement operators.

From the perspective of machine learning, software instances are perceived as sources of information from which to learn. In some manner, diversified software versions could be handled as machine learning instances. This approach broadens the vision of software as an entity from which useful knowledge can be gained.

The problem resembles a representation task: source code is unambiguous and highly structured. There have been several efforts on the field of code mining and code analysis, such as code representation using abstract syntax trees (AST) [29], control flow graph (CFG) [30], or even as XML format [31]. The purpose is to explain how the code instructions are composing into a higher–level meaning, which results in useful software engineering tools that help for code construction and maintenance.

The generation of synthetic software versions using machine learning methods has several advantages over other arbitrary randomization approaches. These techniques aim to maximize the potential impact of diversification on the system's resilience. This is a first step towards the more general goal

of developing machine learning methods that learn through the use of some code representation. Additionally, software diversification comprises a new modality of machine learning mechanisms with different characteristics compared to images and natural language processing. Models based on source code analysis and software transformation fall into a new branch of methods that have interesting parallels to traditional processing images and natural language.

An interesting approach to match software diversification with machine learning could be the way of generating synthetic instances. Many supervised machine learning applications present problems when learning from imbalanced datasets. The SMOTE algorithm, proposed by Chawla et al. [32], is a popular method of over–sampling by generating synthetic instances, avoiding overfitting of random over–sampling. SMOTE generates synthetic instances by interpolating between minority examples that lie together, making the decision regions larger towards majority class and less specific. Synthetic examples are introduced along the line segment between each minority class example and one of its k minority class nearest neighbors. Its generation procedure for each minority class example consist in (1) choose one of its k minority class nearest neighbors, (2) take the difference between the two vectors, and (3) multiply the difference by a random number between 0 and 1, and add it to this example.

We believe that the use of SMOTE's similar techniques could improve the quality of the synthetic software instances during the diversification process. Furthermore, a heuristically guided search of the space of transformations could generate, of even improve, these software instances (e.g., sosies that passes a larger test case than the original version or that present a better representation or structure). However, as could be expected, the representation of software systems as instances for machine leaning results in a very challenging task.

B. Diversification in ensemble learning

In machining learning, the ensemble methodology consists in measuring a set of individual patterns using multiple learning algorithms and merge its results to obtain a better predictive performance (e.g., decreasing the error rate or improving accuracy). Ensemble learning can be seen as a learning strategy that addresses inadequacies in training data. Ultimately, an ensemble is less likely to misclassify than just a single component function. This approach is typical of supervised learning, in which fast algorithms such as decision trees are commonly used for improving the performance of the decision boundary. Similarly, ensemble techniques have been used in unsupervised learning scenarios, for example, in consensus clustering or in anomaly detection. Anywise, classifier ensembles have proven to significantly improve the accuracy of a single classifier [33].

There are two approaches to ensemble construction. One is to combine component functions that are homogeneous (derived using the same learning algorithm and being defined in the same representation formalism, for example, an ensemble of functions derived by decision tree methods). Another approach is to combine component functions that are heterogeneous (derived by different learning algorithms and being represented in different formalisms, for example, an ensemble of functions derived by decision trees, instance–based learning, bayesian learning, and neural networks). Two main issues exist in ensemble learning: ensemble construction and classification combination. There are bagging, cross-validation, boosting methods for constructing ensembles, weighted vote and unweighted vote for combining classifications. The Ada Boost algorithm is recognized as one of the best methods for constructing ensembles of decision trees.

Both homogeneous and heterogeneous ensembles could be perceived as an special case of software diversity. They share the same functionality (classification or prediction) but are based on the use of different learning algorithms (heterogeneous) or trained with different subsets of the data (homogeneous).

Empirically, is proven that ensembles tend to yield better results when there is a significant diversity among the models used [34]. However, it is still not clear how diversity affects classification performance, especially on minority classes, since diversity is one influential factor of ensemble. Because of the diversity affects the classifier ensembles' generalization ability, a reduction process of its instances must retain the classifier ensembles' diversity. If each classifier in an ensemble produces a very similar performance, such a classifier ensemble may not improve its generalization ability [35]. On the other hand, if an instance is classified into a wrong category by a classifier of an ensemble, other classifiers within the same ensemble may correct the wrong classification by combining the rest of the classifier's results.

Many diversity measures for ensemble learning have been proposed [35]. For example, [33] et al. present a new ensemble subset evaluation method that integrates classifier diversity measures into a novel classifier ensemble reduction framework. While few papers look into comparisons between different diversifying heuristics, it could be interesting to establish similarities between ensemble learning measures of diversity and the software diversification paradigm. The goal is to obtain more general criteria on the quality of the software systems diversity and exploring the diversification quality of the system.

C. Exploration of the diversification space

The different ways of transformation during software diversification is unlimited. Accordingly, to explore all the space of possible variants results in an unaffordable endeavor (this is the main motivation for the application of diversification on security). Meta-heuristic search algorithms, such as evolutionary algorithms, could be used to perform a more permeating diversification [18], [36]. A big question is how to identify the software engineering principles and evolution rules that drive the emergence and the constant renewal of diversity in software systems.

The mining of software repositories is a relatively novel research field that links software engineering to data mining.

Its goal is to analyze the rich data available in software repositories to uncover interesting and actionable information about software systems, projects and software engineering in general. Some commonly explored areas comprehend software evolution [37], models of software development processes [38], characterization of developers' behavior and their activities [39], prediction of future software qualities [40], use of machine learning techniques on software project data [41], software bug prediction [42], analysis of software change patterns [43], and analysis of code clones [44].

Repository mining offers a vast set of tools for analyzing natural software diversity in several software ecosystems, across multiple projects and platforms. Now we can explore the different facets of software diversity empirically, in a bigger and massive way. This includes not only the analysis of software as a product itself, but also the human interactions among developers to understand the way their perceived and conceived the software during the development and workflow of a project.

The analysis of the natural diversity using the techniques offered by software repository mining could serve as a baseline for generating synthetic diversity. Furthermore, data in software repositories represents a natural field for the application of machine learning methods, big data analysis and deep learning to software engineering. Some interesting applications include prediction of software defects using classification and regression, clustering of similar developing patterns and code reuse, analysis of natural language artifacts and interactions among developers, empirical studies on extracting knowledge from large projects via association rules mining and visualization techniques to summarize source code data, etc.

III. Automatic Test Amplification

Software testing is closely related to software quality. Several testing methodologies have been implemented to verifying the correctness of a software system and ensuring that a program meets certain specifications. While software testing is a significant step during the development process, it is also very expensive as it should take place throughout the whole software development cycle. Various studies indicate that the time and effort spent during software testing usually is greater than the overall system implementation cost [45].

Automatic test generation is a traditional subject of software testing. Its aim is to provide faster and cheaper testing by generating more efficient and accurate testing cases, without requiring special skills or knowledge of the system's behavior. Automatic testing also loses the testing activities from cognitive bias and could produce less errors during testing.

The increasing use and expansion of strong test suits, such as JUnit for Java, has promoted a vast amount of manually well–written test cases. In this context, test amplification has gained and special attention [10]. This is a special variant of automatic test generation in which pre–existing test cases are used to assist the automated generation of additional test cases. The objective of test amplification techniques is improving the value of existing test suits for achieving an specific engineering goal (e.g, increasing test coverage [46], improve observability [47], assess properties of the test suits [48] or detect faults [49]).

Despite the recent progress made in this field, many challenges still remain open. For instance, there are some difficulties in how to make a better use of the information contained in the existing tests in order to synthesizing new ones. Furthermore, it is not clear how the changes on the existing test statements affect the quality of the test suits, which has several implications in terms of scalability of the testing system. On the other hand, Danglot et al. [10] also note the absence of comparison works between traditional test generation (generating test cases from scratch) and test amplification (generating tests from existing tests).

In this context, machine learning methods are interesting tools in the domain of testing amplification. There have been proposed several approaches that apply mutations to the existing tests for effectively generating new tests cases. Search–based methods represent a more efficient way for exploring the testing input requirements, in order to tackling with the almost inevitable updates of the software system. In this section, we address some ideas about the application of machine learning methods for improving test amplification tasks.

A. *Search–based test amplification*

Search–based test data generation is a form of dynamic testing in which additional test data is synthesized following some search heuristics. The idea of using existing test data in order to generate additional test examples renders very well to search–based software testing. Meta–heuristic search algorithms have proven to achieve great success for performing the analysis and expansion of the search space.

Genetic algorithms [50] is the most widely used strategy to generate synthetic test cases that satisfied desired testing requirements [51]. For this particular purpose, the algorithm does not search for a single optimal solution, instead, it automatically searches the space for suitable testing cases while a fitness function that evaluates the requirements is continuously updated. The issue of premature convergence to local optima has been a common problem in genetic algorithms so far. To overcome this problem, many improvements to the fitness function have been proposed.

Baudry et al. [52] presented the bacteriologic algorithm for test case optimization. The algorithm applies several mutations on an initial test suite and incrementally evolves an improved test suite that is considered to be superior to the original one in terms of a mutation score that they defined previously. In this manner, most meta–heuristic algorithms that have been used for test data generation require one or more initial solutions from which to start the search.

In the work of Yoo and Harman [53], they propose a search–based test data regeneration algorithm based on the hill climbing strategy, which adopts random restart in order to avoid local optima. This test data regeneration technique assumes that the existing test data belong to global optima, and, therefore, always starts from a global optimum that

corresponds to the existing test data. Interestingly, they found that the mutation faults detected by the generated test data are different from those that are detected by the original test suite.

There is a vast amount of additional meta–heuristic and optimization techniques that could be used for refining the search space in accordance to some coverage criteria [30] (e.g., ant colony optimization, firefly algorithm, particle swarm optimization, simulate annealing, artificial bee colony algorithm). As in ensemble learning, a hybrid approach using these techniques may offer a best overall result of the search space.

In many real testing scenarios a single-objective optimization approach is unrealistic [54]. Developers usually want to find test sets that meet several objectives simultaneously in order to maximize the value obtained from the inherently expensive process of running the test cases and examining the output they produce. Several multi-objective evolutionary algorithms have been applied to the test data generation problem [55]–[57]. However, as far as we known, there has been no work on multi-objective test amplification reported in the literature.

On the other hand, as testing can only detect the existence of faults, and not the lack of them, executing additional test cases can only increase the confidence in the program under test. Furthermore, it may be possible to utilize test data regeneration not only for creating more tests cases, but also to improve existing test suites. Summarizing, meta–heuristic techniques are good since they reward individuals with high score but they do not favor diversity and the search may converge to many local optima.

To avoid this drawback, Boussaa et al. [58] proposed a new search–based approach for test data generation with the goal of achieving more diversity in the testing space. The idea consists in an adaptation of the Novelty Search Algorithm [59]. They define a new measure of novelty based on distances in order to maximize a fitness function that evaluates generated test cases. Thereby, individuals in this evolving population are selected based on how different they are compared to other solutions evaluated so far.

The patterns that conform the existing testing libraries represent a structured way of knowledge. Consequently, it might be feasible to incorporate machine learning techniques into the whole flow of software development and maintenance processes. For instance, unit tests could allow us to learn from a huge set of predefined testing examples. Furthermore, it could be interesting to use hybridized methods, guided by multi-objective optimization criteria of diversity, to generate improved and amplified testing suits. The application of machine learning for test design and pattern detection is a promising area still under research [60].

IV. Conclusions

This report presented some ideas for integrating machine learning principles to software engineering. In particular, we discuss some applications to the fields of both software diversification and automatic test amplification. We review various important areas such as the generation of synthetic versions of software and the use of ensemble learning approaches, the exploration of source code and repository mining, and the automatic amplification and refinement of test cases. To sum up, we identify the following interesting lines:

- Represent source code structures, or even entire programs, as instances for performing machine learning tasks.
- Study the diversity measures proposed for ensemble learning and its application to assess software diversity.
- Analyze natural software diversity using repository mining techniques.
- Investigate the advantages of machine learning methods, in conjunction with novel search–based approaches, for automatic test data amplification.

This report serve as a starting point to the author in order to strengthen his understanding on these issues with the purpose of identifying novel and promising future research directions.

References

[1] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[2] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, M. Kaufmann, Ed. Morgan Kaufmann Publishers, 2006.

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

[4] D. Zhang, *Advances in machine learning applications in software engineering*. IGI Global, 2006.

[5] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

[6] K. Dejaeger, W. Verbeke, D. Martens, and B. Baesens, "Data mining techniques for software effort estimation: a comparative study," *IEEE transactions on software engineering*, vol. 38, no. 2, pp. 375–397, 2012.

[7] L. An and F. Khomh, "An empirical study of crash-inducing commits in mozilla firefox," in *Proceedings of the 11th international conference on predictive models and data analytics in software engineering*. ACM, 2015, p. 5.

[8] L. L. Minku and X. Yao, "How to make best use of cross-company data in software effort estimation?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 446–456.

[9] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 16, 2015.

[10] B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry, "The emerging field of test amplification: A survey," *arXiv preprint arXiv:1705.10692*, 2017.

[11] Y. Yang, S. Zhu, and G. Cao, "Improving sensor network immunity under worm attacks : A software diversity approach," *Ad Hoc Networks*, vol. 0, pp. 1–15, 2016.

[12] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. of the Int. Symp. on Fault-Tolerant Computing (FTCS78)*, 1978, pp. 3–9.

[13] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, no. 2, pp. 220–232, 1975.

[14] Z. Lin, R. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout." in *DIMVA*, vol. 9. Springer, 2009, pp. 107–126.

[15] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. of the conf. on Computer and communications security (CCS)*, no. 272–280, 2003.

[16] D. Mendez, B. Baudry, and M. Monperrus, "Empirical evidence of large-scale diversity in api usage of object-oriented software," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 2013, pp. 43–52.

[17] J. E. Just and M. Cornwell, "Review and analysis of synthetic diversity for breaking mono–cultures," in *Proceedings of the 2004 ACM workshop on Rapid malcode (WORM 04)*, ACM, Ed., 2004, pp. 23–32.

[18] B. Baudry, M. Monperrus, C. Mony, F. Chauvel, F. Fleurey, and S. Clarke, "Diversify: Ecology-inspired software evolution for diversity emergence," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 395–398.

[19] S. Allier, O. Barais, B. Baudry, J. Bourcier, F. Fleurey, M. Monperrus, H. Song, and M. Tricoire, "Multi-tier diversification in web-based software applications," *IEEE Software, Institute of Electrical and Electronics Engineers*, vol. 32, no. 1, pp. 83–90, 2015.

[20] D. Partridge and W. Krzanowskib, "Software diversity : practical statistics for its measurement," 1997.

[21] D. Posnett, R. DSouza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 452–461.

[22] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, "Automatic software diversity in the light of test suites," *arXiv preprint arXiv:1509.00144*, 2015.

[23] R. Feldt., "Generating diverse software versions with genetic programming: an experimental study," in *IEE Proceedings-Software*, vol. 145, no. 6, 1998, pp. 228–236.

[24] M. Rinard, "Obtaining and reasoning about good enough software," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 930–935.

[25] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.

[26] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[27] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *arXiv preprint arXiv:1204.4224*, 2012.

[28] B. Baudry, S. Allier, and M. Monperrus, "Tailored source code transformations to synthesize computationally diverse program variants," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 149–159.

[29] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," *IEEE International Conference on Software Maintenance*, pp. 388–391, 2013.

[30] O. Sahin and B. Akay, "Comparisons of metaheuristic algorithms and fitness functions on software test data generation," *Applied Soft Computing*, vol. 49, pp. 1202–1214, 2016.

[31] M. L. Collard and J. I. Maletic, "srcml 1.0: Explore, analyze, and manipulate source code." in *ICSME*, 2016, p. 649.

[32] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, pp. 341–378, 2002.

[33] G. Yao, H. Zeng, F. Chao, C. Su, C.-M. Lin, and C. Zhou, "Integration of classifier diversity measures for feature selection-based classifier ensemble reduction," *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, vol. 20, no. 8, pp. 2995–3005, 2016.

[34] L. I. Kuncheva and C. J. Whitaker, "Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy," *Machine learning*, vol. 51, no. 2, pp. 181–207, 2003.

[35] B. Sun, J. Wang, H. Chen, and Y.-t. Wang, "Diversity measures in ensemble learning," *Control and Decis*, vol. 29, no. 3, pp. 385–395, 2014.

[36] K. Yeboah-Antwi and B. Baudry, "Embedding adaptivity in software systems using the ecselr framework," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 839–844.

[37] C. Zhu, Y. Li, J. Rubin, and M. Chechik, "A dataset for dynamic discovery of semantic changes in version controlled software histories," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 523–526.

[38] G. Robles, J. M. González-Barahona, C. Cervigón, A. Capiluppi, and D. Izquierdo-Cortázar, "Estimating development effort in free/open source software projects by mining software repositories: a case study of openstack," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 222–231.

[39] M. Mäntylä, B. Adams, G. Destefanis, D. Graziotin, and M. Ortu, "Mining valence, arousal, and dominance: possibilities for detecting burnout and productivity?" in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 247–258.

[40] P. Thongtanunam, R. G. Kula, A. Erika, and C. Cruz, "Improving code review effectiveness through reviewer recommendations," pp. 1–4, 2014.

[41] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. ACM, 2011, p. 11.

[42] R. Coelho, L. Almeida, G. Gousios, A. Van Deursen, and C. Treude, "Exception handling bug hazards in android-results from a mining study and an exploratory survey." *Empirical Software Engineering*, vol. 22, no. 3, pp. 1264–1304, 2017.

[43] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo, "A deeper look into bug fixes: Patterns, replacements, deletions, and additions," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 512–515.

[44] D. Steidl and N. Göde, "Feature-based detection of bugs in clones," in *Proceedings of the 7th International Workshop on Software Clones*. IEEE Press, 2013, pp. 76–82.

[45] P. Ammann and J. Offutt, *Introduction to software testing*, 2008.

[46] J. Zhang, Y. Lou, L. Zhang, D. Hao, L. Zhang, and H. Mei, "Isomorphic regression testing: executing uncovered branches without test augmentation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 883–894.

[47] M. Patrick and Y. Jia, "Kd-art: Should we intensify or diversify tests to kill mutants?" *Information and Software Technology*, vol. 81, pp. 36–51, 2017.

[48] B. H. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Empirical Software Engineering*, vol. 14, no. 3, pp. 341–379, 2009.

[49] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, "From genetic to bacteriological algorithms for mutation-based testing," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 73–96, 2005.

[50] D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning," *Reading: Addison-Wesley*, 1989.

[51] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[52] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, "Automatic test case optimization: A bacteriologic algorithm," *ieee Software*, vol. 22, no. 2, pp. 76–82, 2005.

[53] S. Yoo and M. Harman, "Test data regeneration: generating new test data from existing test data," *Software Testing, Verification and Reliability*, vol. 22, no. 3, pp. 171–201, 2012.

[54] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1098–1105.

[55] R. A. Matnei Filho and S. R. Vergilio, "A mutation and multi-objective test data generation approach for feature testing of software product lines," in *Software Engineering (SBES), 2015 29th Brazilian Symposium on*. IEEE, 2015, pp. 21–30.

[56] J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Software: Practice and Experience*, vol. 42, no. 11, pp. 1331–1362, 2012.

[57] R. A. Matnei Filho and S. R. Vergilio, "A multi-objective test data generation approach for mutation testing of feature models," *Journal of Software Engineering Research and Development*, vol. 4, no. 1, p. 4, 2016.

[58] M. Boussaa, O. Barais, G. Sunyé, and B. Baudry, "A novelty search approach for automatic test data generation," in *Proceedings of the Eighth International Workshop on Search-Based Software Testing*. IEEE Press, 2015, pp. 40–43.

[59] J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty," in *ALIFE*, 2008, pp. 329–336.

[60] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.