Novel Applications of Machine Learning in Software Testing

Lionel C. Briand
Simula Research Laboratory and University of Oslo
Oslo, Norway
briand@simula.no

Abstract

Machine learning techniques have long been used for various purposes in software engineering. This paper provides a brief overview of the state of the art and reports on a number of novel applications I was involved with in the area of software testing. Reflecting on this personal experience, I draw lessons learned and argue that more research should be performed in that direction as machine learning has the potential to significantly help in addressing some of the long-standing software testing problems.

1 The Role of Machine Learning in Software Testing

Machine learning is the science of identifying useful patterns in data [13]. The application of machine learning to large databases is also often denoted as data mining. The identified patterns can often highlight useful information about the process that generates the data and can then serve as a basis for decision-making. Machine learning techniques vary a great deal in terms of their underlying theory, assumptions, and model representations. Techniques also differ in terms of the research community they originate from, ranging from traditional statistics to heuristic learning algorithms and neural networks.

In the context of software engineering, many activities are performed by humans and are inherently error prone. Furthermore, many tasks require software engineers to identify trade-offs in terms of the amount of resources to assign to a task and where to focus their effort. Given this state of affairs the question is whether appropriate data can be collected and used by machine learning algorithms in order to help decision making in software engineering. There are many potential applications but let us restrict our discussions to software testing as it is the focus of this article.

Various types of data can be collected while testing software. Execution traces and coverage information for test cases can be captured at different levels of detail. Failure data that captures where and why a failure occurs is also of interest. The question then, is whether we can exploit such data on a project in order to tackle some of the long-standing testing problems such as the completeness of test suites, the automation of test oracles, the distribution of testing resources according to levels of risks (risk-driven testing), and localization of faults causing failures.

Figure 1 illustrates with an activity diagram the general role that machine learning can play in software engineering and in particular in the context of software testing. A software engineering "process" gets executed (e.g., running test cases), data is collected and fed to a learner which yields interesting "patterns", e.g., failure conditions. Such patterns are expected to be useful in supporting or automating decision-making, e.g., localization of faults in source code; though as we will see in many cases this remains to be investigated. Decision-making then yields a "plan" which can take various forms, such as a priority list for testing and inspections, an automated algorithm to detect failures (oracle), or possible improvements for a test specification.

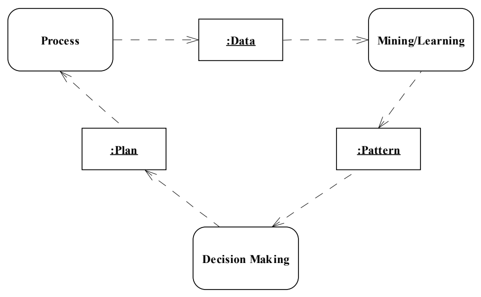

Figure 1 The Role of Machine Learning

As we will see in more detail in the following sections, despite the variety of applications, there are however recurrent issues in applying machine learning techniques to software testing. One is whether, for a given application, we can possibly obtain adequate data, both in terms of form and content, to learn interesting, actionable patterns. Patterns that are an inaccurate approximation of reality or cannot be easily used to drive beneficial actions are of no practical interest. Furthermore the benefits must be significantly higher than the cost of additional data collection and analysis. The complexity of patterns to be learned may be such that the data needs complex preprocessing before a machine learning algorithm becomes capable of learning useful patterns.

2 Examples of Testing Applications

Based on studies that I have personally been involved in, this section presents novel applications of machine learning in software testing. They were selected because of their potentially high impact and promising initial results.

2.1 Test specification and test suite refinement

In the context of open source development or software evolution, developers often face test suites which have been developed with no apparent rationale and which may need to be augmented or refined to ensure sufficient dependability, or even reduced to meet tight deadlines. We refer to this process as the re-engineering of test specifications and test suites. In this context, it would be important to provide both methodological and tool support to help people understand the limitations of test suites and their possible redundancies, so as to be able to refine them in a cost-effective manner. To address this problem in the case of black-box testing, a methodology based on machine learning was proposed in [3].

Figure 2 provides an overview of the steps involved in the MELBA (MachinE Learning based refinement of BlAck-box test specification) methodology. This technique requires as input both a test suite and a test specification in the form of Category-Partition (CP) categories and choices [10]. The CP method seeks to generate test cases that cover the various execution conditions of a function. To apply the CP method, one first identifies the input and environment parameters of each function, the characteristics (categories) of each parameter and the choices of each category. Categories are properties of parameters that can have an influence on the behavior of the software under test (e.g., size of an array in a sorting algorithm). Choices (e.g., whether an array is empty) are the potential values of a category. Test frames and test data are generated according to the categories and choices defined.

In practice, the test specification may not exist to start with, especially if no black-box strategy was used to identify the test cases. In this case, which is likely to be the most common situation in practice, the test specification has to be either reverse-engineered or created from high-level system specifications. Furthermore, to enable learning, the output domain of the program under test has to be divided into equivalence classes.

As the input domain is modeled using CP categories and choices, the test suite is then transformed into an *abstract test suite* (Activity 1, Figure 2) in order to enable effective learning. An abstract test case shows an output equivalence class and pairs (category, choice) that characterize its inputs and environment parameters (e.g., execution configuration), instead of raw inputs. In other words, we use the test specification to move test cases to a higher level of abstraction that will help us learn more useful and interesting patterns. Once an abstract test suite is available, the C4.5 machine learning algorithm is used to learn rules that relate pairs (category, choice), modeling input properties, to output equivalence classes (Activity 2). These rules are in turn analyzed (Activity 3), using a number of heuristics, to determine potential problems that may indicate redundancy among test cases and the need for additional test cases (Activity 4). Those rules may also indicate that the CP specification needs to be improved, e.g., an important category is missing or certain choices are ill-defined (Activity 5).

The improvement process is iterative as improvements to either the test suite or test specification can lead to the identification of new problems to be addressed. The learning algorithm will therefore be repeatedly executed (edges from Activities 4 and 5 to Activity 1, followed by Activity 2), which is not an issue as obtaining C4.5 decision trees for a few thousands of (abstract) test cases and a few dozen categories is quick. The process stops when no more problems can be found in the rules learnt by the machine learning algorithm (Activity 3).

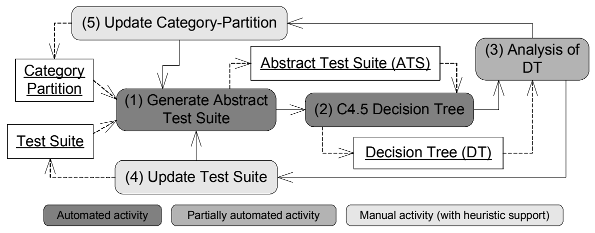
(5) Update Category-Partition
Category Partition
Abstract Test Suite (ATS)
(3) Analysis of DT
(1) Generate Abstract Test Suite
(2) C4.5 Decision Tree
Test Suite
Decision Tree (DT)
(4) Update Test Suite
Automated activity
Partially automated activity
Manual activity (with heuristic support)

Figure 2 The MELBA Process

In the MELBA context, we can identify a number of *potential* problems when analyzing C4.5 decision tree:

Case 1—Instances (test cases) can be misclassified: the wrong output equivalence class is associated to a test case. In other words, a test case belongs to a tree leaf where the majority of instances belong to another output equivalence class.

Case 2—Certain categories or choices are not used in the tree (i.e., they are not selected as attributes to split a (sub)set of instances in the tree).

Case 3—Certain combinations of choices, across categories, are not present on any path, from the root node to any leaf of the tree.

Case 4—A leaf of a tree contains a large number of instances (test cases).

All of the above cases can be automatically detected by a dedicated tool. However, determining the exact cause of the problem can only be facilitated by heuristics but not be entirely automated.

The problems discussed above all have one or more potential causes, as summarized in Figure 3 and are discussed in detail in [3].

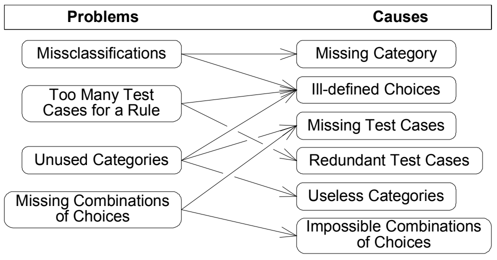

Figure 3 Mapping Problems to Causes

Based on such mappings precise heuristics were developed to help testers identify plausible causes for problems identified in the decision tree. The case studies performed so far have shown that trained testers can iteratively achieve expert-level Category-Partition test specifications and test suites when guiding by the MELBA process. Our taxonomies of decision tree problems and root causes have so far appeared to be complete. The refinement of test suites led to a significant increase in fault detection. Though more empirical studies are obviously required, preferably in realistic settings, these initial results are encouraging.

2.2 Debugging / Fault localization

Using machine learning, the paper in [4] proposes a method (RUBAR: RUle-BAsed statement Ranking) to identify suspicious statements, which are likely to contain a fault related to a failure observed during testing. This can be used to help fault localization during debugging. The RUBAR technique is based on principles similar to Tarantula [4] but addresses its main flaw: its difficulty to deal with the presence of multiple faults as it implicitly assumes that failing test cases execute the same fault(s). The improvement we present in this paper results from the use of C4.5 decision trees to learn various failure conditions based on information regarding the test cases' inputs and outputs. Failing test cases executing under similar conditions are then assumed to fail due to the same fault(s). Statements are then considered suspicious if they are covered by a large proportion of failing test cases that execute under similar conditions. Case studies also show that failure conditions, as modeled by a C4.5 decision tree, accurately predict failures and can therefore be used to help debugging as the failure conditions modeled by the tree are overall accurately describing actual failure conditions.

Let us now focus on the machine learning aspects of this approach. In order to obtain meaningful and accurate machine learning rules, like for the MELBA methodology in the previous section, we cannot simply use the test case input and output values. Our experience is that it would typically lead to meaningless and inaccurate rules because the machine learning algorithm cannot learn what input or output properties are potentially of interest but only which ones matter once they are defined. In other words, without some additional guidance, the learning algorithm is unlikely to find the precise conditions under which test cases fail. This guidance, like for MELBA, comes in the form of categories and choices, as required by CP [10]. Once again here, initial test cases are transformed into abstract test cases, as previously described. In a black-box testing context, such CP specifications are readily available.

Let's take a simple but hopefully illustrative example: for the well-known Triangle program, the input values characterize the length of triangle sides. We can use CP to define categories on the relationships among the lengths of the triangle sides, e.g., whether they are equal or otherwise. In this way, the input of the Triangle program can be described as `[compare(side1, side2), compare(side2, side3), compare( side3, side1)]` instead of simply `[side1, side2, side3]`, where `compare (s, s')` determines whether `s` is larger, equal, or smaller than `s'`. Then the test cases can be expressed as tuples in terms of these properties and we refer to these tuples as abstract test cases, e.g., input data `(1, 1, 2)` becomes `(side1=side2, side2<side3, side3>side1)`. If we obtain failures when the triangle sides are of equal length, based on a set of abstract test cases, the learning algorithm will be able to determine that when all sides are of equal length the program fails to recognize this is an equilateral triangle. A fictitious decision tree modeling failure conditions for this example is depicted in Figure 4. An alternative to abstract test cases would be to consider higher-order learning algorithms (e.g., Foil) but this is not a practical option at this stage of maturity of the technology.

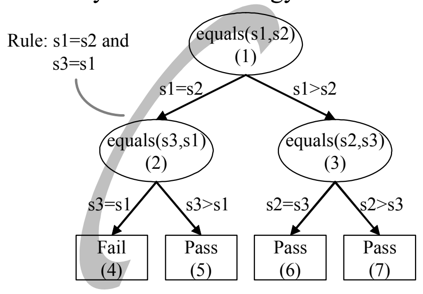

Figure 4 Modeling Failure Conditions

The case study presented in [4] shows that the decision tree we obtained had high (> 90%) precision and recall, with certain branches/rules predicting a near 100% probability of failure. We were therefore able to identify, in a very accurate manner, various conditions leading to failures. The statement ranking we obtained performed much better than Tarantula in the presence of multiple faults, as illustrated by the case study depicted in Figure

5. Based on the obtained statement ranking, the X-axis is the percentage of statements that need to be investigated to detect the percentage of faulty statements on the Y-axis. Observations are the program statements ordered in decreasing likelihood of containing a fault. Because certain faults made some statements unreachable, the debugging process proceeded in two steps in this case study, as denoted by the vertical line in Figure 5. We can see that both in the first and second steps, most faulty statements are placed early in the ranking. In particular, we can see a very sharp increase in the number of faulty statements covered in the higher 20% range of the ranked statements, which is roughly the range of practical interest in our situation, as discussed by Jones *et al.* [4].

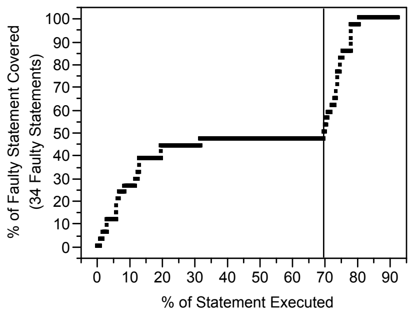

Figure 5 RUBAR Statement Ranking

Despite encouraging results, there are several important questions that research will have to answer. Perhaps the most important one is whether providing statement ranking and failure conditions in terms of inputs and environment parameters is of practical interest. It is not at all obvious how such information can be exploited.

2.3 Risk-driven testing

Regardless of the testing strategy adopted, in practice there are rarely enough human resources and time to thoroughly test every part of a system to a satisfactory extent. Testing teams must focus and prioritize their testing efforts. Usually this is done by analyzing the "risk" associated with a functionality or system components, depending on the testing level. Risk is usually defined as a combination of probability of failures and the damages they can potentially cause. Though there is a variety a ways of tackling the problem, many studies [14] have focused on the construction of models predicting the location of faults across files or classes. Many of these models make use of machine learning algorithms such as decision trees and neural networks, to name a few.

The reasons why machine learning algorithms have been popular with building such prediction models are that they often do not require the type of stringent assumptions associated with statistical models, e.g., logistic regression, and, for some of them such as decision trees and induction rules, are easier to interpret and understand for practitioners.

Though the construction of fault prediction models is not a *novel* application of machine learning, recent developments are worth discussing. Though there was a historical focus on using software structural measures to predict faults, there has been an increasing realization that other factors needed to be accounted for to obtain reasonably accurate prediction models. As a result, in many recent studies (e.g., [15]), though the data used in input varies, it usually includes structural complexity measures of components (e.g., class coupling), component change information from recent releases, history data (e.g., faults) associated with components, and developers' information regarding, for example, their experience with the system being changed.

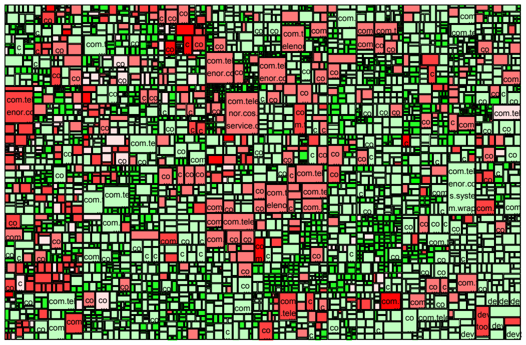

Figure 6 System Tree Map

Once a fault prediction model has been developed then it is possible to rank system components (files, packages, classes) according to their likelihood to contain a fault. The result can also be more easily visualized with tree maps, as illustrated by Figure 6. In this example, the tree map shows a hierarchical system decomposition down to the class level. At the lowest level rectangles represent classes and their size is proportional to the size of the class itself. Such low-level predictions can also be combined to provide higher-level predictions for packages or subsystems. Classes are color-coded according to their likelihood of containing a fault. Though this appears to be intuitively useful, one difficulty associated with fault prediction models has been the lack of strategies to exploit them and the lack of evidence concerning the cost-benefit of applying them to focus and drive verification. However, a recent study has shown that their use could lead to significant return on investment [15]. Much more research needs to be done though to devise optimal strategies and from an economical standpoint, to build and exploit fault prediction models and demonstrate their cost-effectiveness.

2.4 Test oracles

The automation of test oracles is probably one of the most difficult problems in software testing. Though there is no generally applicable solution to automate test oracles, there are a number of situations where machine learning can help. This is the case in the context of iterative development and testing, when algorithms are constantly refined and re-tested, and there is no precise oracle that can be defined. There are, for instance, many such cases in the domains of image and speech processing. The example we use here is the segmentation of the heart ventricle in 3-D images [7] (see example in Figure 7).

Image segmentation is the act of extracting particular structures of interest (e.g., ventricle) from an image (e.g., heart). A lot of time and effort is spent in order to evaluate image segmentation algorithms. If the image segmentation algorithm does not provide accurate enough results from a medical diagnostic point of view, this is considered a failure and the technical expert needs to modify the algorithm and re-run the whole test suite to verify it. This process is repeated as the image segmentation algorithm evolves to its final acceptable version where the test suite passes. This evaluation process is mostly done manually and is therefore very time consuming, requiring the presence of reliable experts. The main reason for this is that no segmentation will ever be perfect and the evaluation of any algorithm will always need to account for some degree of error in the segmentations it generates. In [7], the authors propose to automate the re-testing of images with new segmentation algorithms by using machine learning techniques. During the initial learning phase, the expert is required to evaluate segmentations manually. This is used by a machine learning algorithm to learn a model that can classify a pair of segmentations as being diagnostically consistent or inconsistent based on a large set of similarity measures. In a second phase, once a valid machine learning model is learnt, a segmentation produced by any new version of the image segmentation algorithm under test will be automatically deemed correct or incorrect depending on whether or not it is diagnostically consistent with the segmentations previously deemed correct or incorrect. In this second phase, because there is no need for any intervention from human experts, substantial saving can be obtained during the re-testing of segmentations.

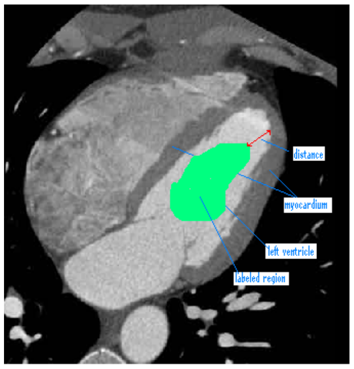

Figure 7 Heart Segmentation Example

Similarity measures comparing old and new segmentations, along with expert-provided assessments about their validity (from a diagnostic viewpoint), can then be used to build a model, using machine learning algorithms, to predict the diagnostic consistency of segmentations. In our case, we used once again, decision trees though similar results were obtained with other standard algorithms (e.g., PART, Ripper, logistic regression [13]). The decision tree based on the entire training set only includes four leaves, as illustrated in Figure 8. It is worth noting that despite its simplicity (Only three similarity measures—three non-terminal nodes—were selected in the tree.), the tree yielded very good results. A cross-validation analysis yielded a ROC (Receiving Operator Characteristic) area [13] of 90% and a classification accuracy of 87%.

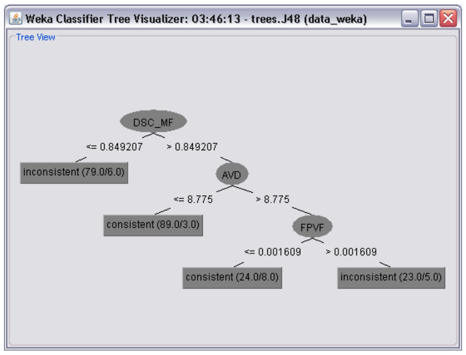

Figure 8 Decision Tree Example

In the leaf nodes, the left number in the parentheses represents the total number of instances belonging to the leaf and the right number represents the average number of instances that have incorrectly been predicted to have the classification of the leaf node. Based on the probabilities associated with the tree leaves, we assess that more than 80% of the comparisons can be trusted with a high level of confidence (i.e., > 90% chances of predicting correctly (in)consistency). So, though more sophisticated similarity measures could be considered to improve the results, with the current decision tree, in 80%

of the cases no manual verification of segmentation results would be necessary, thus saving substantial expert time. Further investigations are of course required in order to determine the reasons for the remaining uncertainty in the models and how to reduce it.

3 Brief Overview of Other Related Works

This section presents a brief overview of other works using machine learning in the area of software testing. These works differ with respect to the inputs they expect, their learning objectives, and the learning mechanisms they employ.

Similar to our RUBAR method, the work in [2] aims at learning program behavior but uses execution traces: the technique records the execution of branches in the program (inter-procedural) control flow graph. The classifier being learnt is a map between execution traces (i.e., branch execution profiles) and behavior classes (e.g., fail/pass), and can be used to guide the construction (or extension) of a test suite. A classifier is first built using execution traces and corresponding behavior classes. Any newly defined and executed test case is then either used to refine the classifier (when the test case provides a new mapping between a profile and behavior class) or discarded (if the mapping is already in the classifier). In the former case, the tester has to provide a behavior class, typically by building an oracle. Although the classifier helps the tester identify (somewhat) redundant test cases, thus avoiding the cost of building an oracle for them, the technique requires that the test case be defined by other means and executed before determining whether it is a useful addition or not. The strategy can only be practical if the tester can ensure that the new test data have high chances of triggering new behavior (otherwise, building test cases and more so executing them may be expensive), e.g., if an automatic test data generator is used, as suggested by the authors, there is a risk that many executed test cases will be redundant and recognized by the classifier, therefore providing little help. Though the approach is very effective at determining whether a new test case is a useful addition, it provides little help for the definition of new, interesting test cases (the authors assume an automated test input and test case generation procedure is available).

Another execution trace based approach to test suite evaluation and extension is proposed in [9]. As long as the reverse-engineered specification, in the form of so-called 'likely invariants' [6], does not change when adding a test case to a test suite, the test case is deemed redundant. This approach does not require the construction of oracles, as opposed to [2]. The quality of the result however depends on the program invariant patterns that are used. The program invariants being discovered from execution information must be instances of a set of pre-defined invariant patterns. An invariant that does not fall into this category cannot be recognized. Furthermore, the approach does not provide guidance regarding the definition of new test cases.

This approach is expanded upon in [11] where the authors use a specification (invariants), reverse-engineered from passing test cases, to determine whether new automatically-generated (mainly randomly) test inputs are illegal, legal, or fault revealing based on the run-time monitoring of invariants. Illegal test cases violate the reverse-engineered pre and post conditions of a method, whereas fault-revealing test cases only violate post conditions. However, such a test case may not necessarily represent a failure since the pre and post conditions may not be complete (they have been learnt from a set of test cases that may not have exercised every possible behavior of the method). The technique is further expanded in [5] where reverse-engineered likely invariants [6] are analyzed by a constraint solver to automatically produce test inputs and test cases. Constraint solving is however limited; for example, integer variables are supported but not floating point variables. In both [5] and [11], only likely invariants matching pre-defined patterns are considered (recall the discussion above).

Podgurski *et al.* [12] use passing and failing execution profiles to help debugging. Profiles entail recorded information that is not limited to predicates but can contain any data that can be collected about program executions (e.g., a program variable value, a call stack). The program execution features in the profile that can best distinguish passing from failing executions are first identified using logistic regression. Because the level of abstraction of the collected data needs to be increased to help predict failure, failing profiles are then grouped using either a cluster analysis technique or a multivariate visualization technique. Those groups are then analyzed by the tester to understand failing execution conditions. The type of properties that can be uncovered using this approach is limited and remains to be investigated.

In [8], the use of learning algorithms to understand program executions is applied to the problem of profiling deployed software, i.e., collecting execution information in the field. The objective is to be able to accurately classify program executions as fail or pass executions based on low-cost, limited instrumentation of deployed programs, rather than to improve a test suite.

In [1] the authors present an adaptive sampling mechanism to identify feasible paths in a control flow graph with high traversing probability. Instead of uniformly sampling the set of paths in the control flow graph of a program (which usually contains a very large number of unfeasible paths), the authors devise an adaptive (learning) approach to ease the identification of feasible paths. The adaptive aspect of the approach is to learn, from the structure of known feasible paths, which branches to traverse in order to obtain a new feasible path.

There is also a great deal of work on building defect prediction models using machine learning algorithms. This article is however not the right venue for such an extensive survey and the reader is pointed to articles [14] , [15], and [16] for further discussions on this topic. What must be noted though is that very little of this work is concerned with *applying* defect prediction models for supporting decision-making. At this point, we cannot make any claim regarding the potential benefits of such prediction models in practice. Another noteworthy observation is that what specific machine learning technique is used does not seem, in most cases, to make any significant difference. Simple standard techniques (e.g., C4.5 decision trees), as long as they are well used, seem to do just fine.

4 Discussion

In general, all the studies presented above seek to "summarize" some form of experience (e.g., assessing test results, test failures, fault occurrences) into a form that is exploitable for decision-making, whether automated or in interaction with a human.

One general problem is to ensure that relevant data is available in a form that facilitates learning using standard algorithms and techniques. We have seen, for example when learning about relationships between inputs and outputs for refining test specifications, that a human expert needs to provide inputs in the form of categories and choices. This is required to transform raw test data into what we called abstract test cases. This kind of pre-processing is often necessary to make learning effective through standard techniques.

In many situations simple machine learning techniques, such as decision trees, perform as well in terms of prediction accuracy as more complex ones, e.g., neural networks, support vector machines. Moreover, techniques like decision trees or induction rules produce models that are easier to interpret by practitioners. In the case of fault prediction models, it is very important for users of these prediction models to understand why they obtain a particular prediction. It makes the adoption of such models significantly easier and allows them to determine what factor could be missing that would invalidate the prediction. For other applications, such as test specification refinement or fault localization, we have seen the interpretability of models was a key component of the methods. Therefore, depending on the application at hand, accuracy is far from being the only criterion by which to select a machine learning technique.

For most if not all applications of machine learning in software testing, it is in general important to determine the role of the human in providing adequate input information and exploiting the models. Human experts are good at certain tasks but poor at others. For example, it is common for testers with a thorough domain expertise to perform black-box testing by identifying execution conditions of interest. This is for example what Category-Partition is designed for. However, it is hard for any human, based on the execution results of a test suite to determine failure conditions. This is why our technique to support fault localization described above was designed to derive failure conditions assuming testers would provide Category-Partition specifications. This expert input is probably what will make this approach practical and scalable.

Last but not least, there is very little evidence, for all existing applications of machine learning in software testing, that such techniques bring any benefits. Much more attention must be dedicated to studying the cost-effectiveness of machine learning models to support decision-making.

Acknowledgments

I wish to thank my co-authors from the papers from which material has been reused in the current article: Erik Arisholm, Yvan Labiche, Kambiz Frounchi, Xutao Liu, and Zaheer Bawar. I am also grateful to Rajwinder Panesar-Walawege for reviewing a draft of this paper.

References

[1] Baskiotis N., Sebag M., Gaudel M.-C. and Gouraud S., "A Machine Learning Approach for Statistical Software Testing," *Proc. International Joint Conference on Artificial Intelligence*, 2007.

[2] Bowring J. F., Rehg J. M. and Harrold M. J., "Active Learning for Automatic Classification of Software Behavior," *Proc. ACM International Symposium on Software Testing and Analysis*, 2004.

[3] Briand L. C., Labiche Y. and Bawar Z., "Using Machine Learning to Refine Black-box Test Specifications and Test Suites," forthcoming in the *Proc. IEEE International Conference on Quality Software (QSIC)*, 2008.

[4] Briand L. C., Labiche Y. and Liu X., "Using Machine Learning to Support Debugging with Tarantula," *Proc. IEEE International Symposium on Software Reliability Engineering*, 2007.

[5] Csallner C. and Smaragdakis Y., "DSD-Crasher: A Hybrid Analysis Tool for Bug Finding," *Proc. ACM International Symposium on Software Testing and Analysis*, 2006.

[6] Ernst M. D., Cockrell J., Griswold W. G. and Notkin D., "Dynamically discovering likely program invariants to support program evolution," *IEEE Transaction on Software Engineering*, vol. 27 (2), pp. 1-25, 2001.

[7] Frounchi K., Briand L., Labiche Y., “Learning a Test Oracle towards Automating Image Segmentation Evaluation”, Carleton University TR SCE-08-02.

[8] Haran M., Karr A., Last M., Orso A., Porter A., Sanil A. and Fouche S., “Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks,” *IEEE Transactions on Software Engineering*, vol. 33 (5), pp. 1-18, 2007.

[9] Harder M., Mellen J. and Ernst M. D., “Improving Test Suites via Operational Abstraction,” *Proc. 25th International Conference on Software Engineering*, 2003.

[10]Ostrand T. J. and Balcer M. J., “The Category-Partition Method for Specifying and Generating Functional Test,” *Comm. of the ACM*, vol. 31 (6), pp. 676-686, 1988.

[11] Pacheco C. and Ernst M. D., “Eclat: Automatic generation and Classification of Test Inputs,” *Proc. European Conference on Object-Oriented Programming*, LNCS 3586, 2005.

[12] Podgurski A., Leon D., Francis P., Masri W. and Minch M., “Automated Support for Classifying Software Failure Reports,” *Proc. 25th International Conference on Software Engineering*, 2003.

[13] Witten I. H. and Frank E., *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufman, 2005.

[14] Briand L. and Wuest J., "Empirical Studies of Quality Models in Object-Oriented Systems," *Advances in Computers*, vol. 59, pp. 97-166, 2002.

[15] Arisholm E., Briand L., and Fuglerud M., "Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software," *proc. 18th IEEE International Symposium on Software Reliability,* 2007.

[16] Menzies T., Greenwald J., and Frank. A., “Data Mining Code Attributes to Learn Defect Predictors,” *IEEE Transactions on Software Engineering*, vol. 33 (1), 2007.

Short Biography:

Lionel Briand is a professor of software engineering at the Simula Research Laboratory and University of Oslo, Norway. He is currently on leave from the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he is full professor and has been granted the Canada Research Chair in Software Quality Engineering. Before that Lionel was the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany. He also worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland.

He has been on the program, steering, or organization committees of many international, IEEE and ACM conferences. He is editor-in-chief of Empirical Software Engineering (Springer) and is a member of the editorial boards of Systems and Software Modeling (Springer) and Software Testing, Verification, and Reliability (Wiley). He was on the board of IEEE Transactions on Software Engineering from 2000 to 2004.
His research interests include: model-driven development, testing and quality assurance, and empirical software engineering. In his opinion, software engineering research should entail both rigor and practicality, as any other engineering discipline. This implies that research be much more problem-driven, and less technology-driven, a significant change from what has been the trend in the past decades.