2018 IEEE 11th International Conference on Software Testing, Verification and Validation

An Empirical Study on Software Failure Classification with Multi-Label and Problem-Transformation Techniques

Yang Feng*, James A. Jones*, Zhenyu Chen†, Chunrong Fang†
*Department of Informatics, University of California, Irvine, USA
†State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
*{yang.feng, jajones}@uci.edu,†{zychen, fangchunrong}@nju.edu.cn

***Abstract*—Classification techniques have been used in software-engineering research to perform tasks such as categorizing software executions. Traditionally, existing work has proposed single-label failure classification techniques, in which the training and subsequent executions are labeled with a singular fault attribution. Although such approaches have received substantial attention in research on automated software engineering, in reality, recent work shows that the assumption of such a single attribution is often unrealistic: in practice, the inherent characteristics of software behavior, such as multiple faults that contribute to failures and fault interactions, may negatively influence the effectiveness of these techniques. To relax this unrealistic assumption, in the machine learning field, researchers have proposed new approaches for multi-label classification. However, the effectiveness and efficiency of such approaches varies widely based upon application domains. In this paper, we empirically investigate the performance of these new approaches on the failure classification task under different application settings. We conducted experiments using eight classification techniques on five subject programs with more than 8,000 faulty versions to investigate how each such technique accounts for the intricacies of software behavior. Our experimental results show that multi-label techniques provide improved accuracy over single-label. We also evaluated the efficiency of the training and prediction phases of each technique, and offer guidance as to the applicability for each technique for different usage contexts.**

I. Introduction

Large software development organizations (*e.g.,* Microsoft [3] and Baidu [25]) receive thousands of crash and failure reports every day. Due to the overwhelming number of reports, manual diagnosis becomes an expensive and tedious task. One strategy that software engineers can employ to help categorize and triage these reports is to embed instrumentation to track execution information. This execution information can accompany reports, and machine-learning-classification techniques can recognize and classify them [17], [25]. Such classification can not only substantially reduce the efforts of processing these reports but also can help developers prioritize their efforts for corrective maintenance, even before the root causes of the failures have been diagnosed [8], [17].

Traditionally, software-engineering failure classification (*e.g.,* [1], [8], [11], [17]) has been performed using classic *single label* machine-learning classifiers, such as KNN, C45 and Naive Bayes, which are designed to classify crash and failure reports to only one label (*e.g.,* the fault that caused the failure). These single-label classification techniques build upon a simplifying assumption: *each crash or failure report can be classified to a single cause*. However, in reality, the cause of failures is often more complex and may be attributable to multiple root causes. For each software-release version, software engineers can receive a number of failure reports revealing various faults. Two distinct studies [4], [5] found that software faults almost always interacted to alter program behavior. These interactions may result in an under-estimation of the root causes for failure, and may subsequently cause additional maintenance efforts, *e.g.,* bug-report reopening. According to Shihab *et al.* [20], such under-estimation can lead to more than twice the bug-fixing time cost. As such, classifying failures to a single root cause may increase the cost of software maintenance.

In recent years, machine-learning research has produced new techniques for classification problems with more than one label (*e.g.,* [18], [21], [27], [28]), *i.e.,*, *multi-label classification*, which may be better suited to the practical failure-classification scenarioes. Based on these advancements, Feng [7] and Xia [25] propose multi-label failure report classification technique to improve this task. In these techniques, a multi-label classifier can automatically label a failure report as likely to show evidence that it failed due to multiple causes. Further, in addition to the recent advanced multi-label classification techniques, single-label techniques can be used with modifications of the problem space that allows multiple classifications. Such multi-label classification by using single-label techniques through transforming the problem space is called *problem transformation*.

Although both *problem transformation* and *multi-label classification* relax the *single label* assumption of traditional failure-classification techniques, the empirical performance of these techniques under different settings is still not clear for software-engineering researchers. Domain-specific empirical investigation is important because different data may affect classification effectiveness in unforeseen ways. Particularly, in the case of software-execution data (*e.g.,* execution profiles and coverage), the data may contain patterns and structure — it is not purely stochastic. Thus, empirically studying the effectiveness of the various classification techniques for software-execution data (and for a variety of circumstances,

0-7695-6388-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICST.2018.00039

e.g., fault quantity) is important to guide software engineers' choices for automated classification techniques.

As such, in this paper we empirically investigate the use of these multi-label techniques and related problem-transformation techniques that are capable of assigning multiple labels to software failures. We employ two of the leading and recent advances in *multi-label learning*: ML-KNN [27] and BP-MLL [26]. In addition, we also employ the most widely used problem-transformation methods [28]: Label Powerset (LP) and Binary Relavance (BR) in our study. We evaluate the performance of the existing and commonly used single-label classifiers and the extent to which they suffer from the single-label simplifying assumptions. Moreover, we evaluate the extent to which recent multi-label and problem-transformation approaches address any such limitations that arise from these assumptions.

We conducted an experiment to investigate the effectiveness and efficiency of the three types of classifiers: (1) single-label, (2) problem-transformation, and (3) multi-label. To conduct this study, we implemented and evaluated eight classification techniques: two single-label, four problem-transformed, and two multi-label techniques. We used five real-world software subjects and over 8000 faulty versions with multiple faults.

We found both surprising and unsurprising results — that is, some results confirmed general data-mining research of classifying generic data, and some results contradict generic data-mining research and as such were software-execution-data specific. These results have direct implications for the choice of classification techniques in practice, and in this paper, we highlight such conclusions. Perhaps predictably, we found that in general, traditional single-label techniques work best when fault quantities are low (*e.g.,* software is mature), and multi-label techniques work much better in these circumstances. Surprisingly, we found that problem-transformation techniques worked better than anticipated and did not suffer from the theoretical problems that data-mining research predicts. We analyze this situation for software-execution data and find characteristics of the data that influence this difference for our domain. Such results inform engineers' choices of techniques and open the door to the use of problem-transformation techniques that may be easier to implement and can extend traditional single-label techniques.

The main contributions of this work are as follows:

- We empirically evaluate the use of multi-label learning techniques and problem-transformation methods for a well-studied automated software-engineering task: failure classification. To the best of our knowledge, this is the first extensive study of investigating the effectiveness and efficiency of problem transformation for this task.
- We conducted a large study of 5 large C programs, with over 8000 faulty versions, with detailed comparison and analysis of the effectiveness and efficiency of single-label, transformed single-label, and multi-label classification.
- The results of our study offer suggestions for practical use of such classification techniques under different settings, *e.g.,* maturity of the software and density of bugs.

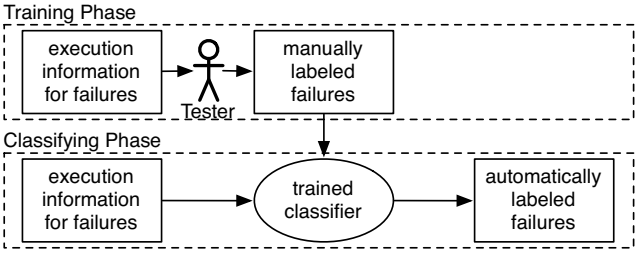


Fig. 1: Software failure classification.

II. Background: Software Failure Classification

Typically, failure-classification work utilizes supervised learning, *i.e.,* training data instances are manually labeled to build a model and label the failure reports. As described by Podgurski *et al.* [8], [17] in early work in this area, developers "often receive many more failure reports than they have time to investigate thoroughly" and moreover, "to facilitate corrective maintenance, it is desirable to identify these groups *before* the causes of the failures are diagnosed." Dang *et al.* [3] describe a process whereby clusters (or "buckets" in their parlance) help to identify which failure reports are causing the most number of crashes and help to identify duplications. Xia *et al.* [25] report their experience working with Baidu (the largest Chinese internet search provider) developers: (1) hundreds of thousands of crash and failure reports are received every day, (2) teams within the company are assigned to the task of manually labeling a sampled subset of the failure reports, and (3) many of the failures have multiple causes.

Figure 1 depicts this process of training the classifier and classification model with training data, and then using that classifier to automatically label unseen execution information of the failure report. In all such cases, the failure reports do not contain the complete execution history, which may involve billions of instructions or events, but instead are abstracted to a more compact form that is amenable to analysis. In practice, failure reports contain some combination of different forms of program-execution spectra [12] and/or textual descriptions. Frequently used program-execution spectra for representing executions include statement and method coverage, runtime context (including hardware and software configurations), and stack traces (at time of crash). This recorded execution information is utilized to characterize the data instances, for both the training and evaluation phases of the learning techniques.

Motivations for Study on Software-Failure Data. The goal of this study is to investigate the performance of various failure classification techniques under different settings, and further, to provide actionable advice for improving the practical application of these techniques. Even though failure classification techniques have received substantial attention in academia in the past couple decades, the previous works mainly limit themselves to improving the performances under the *single-label* assumption. In the past, Feng *et al.* [7] proposed *multi-label* failure-classification techniques to relax the unrealistic assumption that failures can be attributed to strictly one fault. Xia *et al.* [25] extended the *multi-label* failure classification

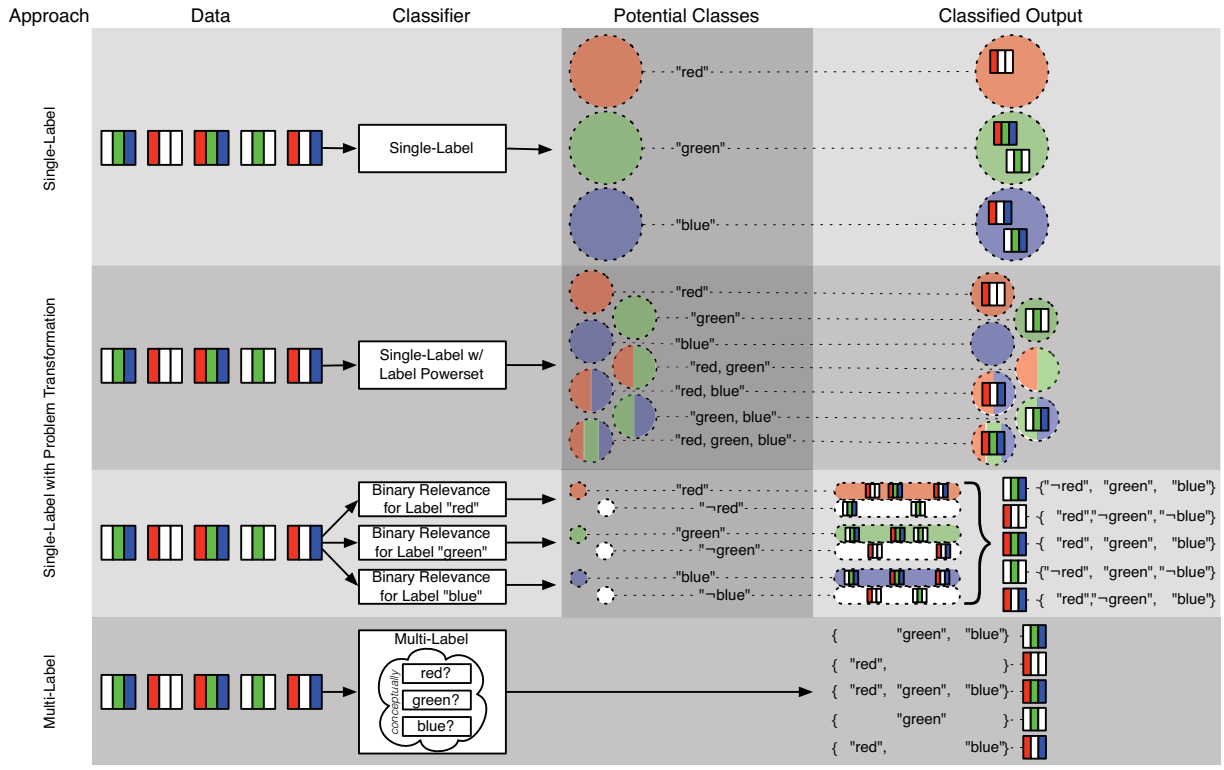


Fig. 2: Conceptual depiction of the difference of the studied classification techniques.

by employing genetic algorithms to improve effectiveness. In this work, we go beyond such early work to evaluate a variety of failure classification techniques (single-label, multi-label, and single-label problem-transformed techniques) on real-world software subjects (*i.e.,* not limited to the "Siemens suite") and investigate the effects of different settings, such as fault quantity and software maturity. The investigation of such features on real software are crucial to inform application of such techniques in practice.

In addition to studying the various classification techniques, it is also important to study how domain-specific software-execution data influences the performance of the techniques. In the machine-learning and data-mining research communities, performance of a variety of techniques have been evaluated on generic data. However, in those communities it is recognized that domain-specific data characteristics affect the efficacy of the techniques. For example, data that is not stochastic and has structure changes the applicability and effectiveness of classification techniques. Software-execution data is known to have structure in two main ways:

1) Runtime events follow the structure of logic of the program code, and as such, inherently have structure and are not purely stochastic. Examples of such events include: use of particular software features; inputs or variables taking on certain values; execution of particular statements, branches, or paths; and the occurrence of certain data flows, information flows, object states, state transitions, sequences of events, and timing patterns.
2) Failure attribution to the faults that caused them has been shown to be sensitive to fault interference [4], [5]. That is, multiple faults interact to cause failures, and moreover sometimes interact to obscure or increase failures.

For these reasons, study of such classification techniques on domain-specific software failure data is essential to inform guidance on the use of such techniques in practice.

III. Machine-Learning Classification

Classification techniques attempt to take data instances and infer labels in order to place data instances into classes. In this section, we give an overview of the three types of classification: single-label, problem transformation, and multi-label. Figure 2 conceptually depicts four main types of classification: one single-label, two kinds of problem-transformation, and one multi-label classification techniques — each technique is shown in a row. In this figure, data instances are depicted as a triple — these depictions can be interpreted as metaphors for a software execution, where the colors (red, green, and blue) represent whether a fault influenced the software failure. For example, the first data instance of [▯▯▯] represents an execution that failed due to the green fault and the blue fault (but not the red fault). The "Potential Classes" column represents the possible outputs, given the chosen classification technique — these depictions can be interpreted as metaphors for the attribution of the classifier, *i.e.,* the faults that can be diagnosed for the execution failures. On the right of the figure in the "Classified Output" column, we depict a hypothetical best-case

output of the classifier, given the input data instances. Such an output can be interpreted as describing the recommended fault attributions for each failure execution. We now describe each classification technique depicted in each row.

A. Single-Label Classification

Perhaps the most common type of classification used by software-engineering researchers is *single-label*, where a single classification is placed on each data instance — that is, there is no possibility for a data instance to be classified to more than one label. In Figure 2, the single-label classification has three possible classes: "red", "green", or "blue". Notice that because there are only three classes (and no possibility of a data instance belonging to multiple classes), this form of single-label classification can cause classification errors when the data can exhibit features that do not map perfectly into the class set.

B. Single-Label Classification with Problem-Transformation

To solve the classification problems that involved more than one label, machine-learning researchers have proposed some methods to transform multi-label problems into single-label problems. Two common problem-transformation methods are called Label Powerset (LP) and Binary Relevance (BR) [28].

Label Powerset (LP) (also referred to in the literature as the *label combinations*) is a simple but effective method to transform multi-label problems into single-label problems. It accomplishes this task by enumerating all label combinations as atomic labels, *i.e.,* each label combination is treated as a class in one single label problem [22]. The second row of Figure 2 depicts such a transformation of the classification problem to accommodate classification to multiple labels. The Label Powerset approach works by enumerating all possible combinations — in the figure, the powerset of "red," "green," and "blue" is produced so that each class is labeled with an atomic label, such as "green, blue". Applying the conceptual figure to the software failure classification problem, we can say that the set of possible classes includes failure caused by each individual fault (*e.g.,* red, green, or blue) or each possible combination thereof. Note that the set of possible combinations can grow quickly with large numbers of labels.

The other common problem-transformation method to allow single-label classification to work for multi-label problems is called *Binary Relevance* (BR). Binary-Relevance classification treats each label of the label set independently. By training an independent classifier for each of the labels, a Binary-Relevance method will produce a combined classification result, *i.e.,* the final output of BR contains the classification result from the independent classifier of each labels. The third row of Figure 2 depicts the transformation of the single-label classification into three independent classifiers. In the figure, a classifier is used to classify each datum as "red" or "¬red", "green" or "¬green", and "blue" or "¬blue". In the end, these results are aggregated to form a single, multi-label classification for each input datum. Note that such independent classifiers for each label cannot leverage co-occurrence information (*e.g.,* red and green often co-occur, and green and blue rarely co-occur).

C. Multi-Label Classification

In recent years, multi-label classification has been achieved through classification techniques that directly target the multi-label problem. Zhang and Zhou [27] created an *algorithmic* transformation (rather than a problem- or usage-transformation) of a single-label classification technique to more directly target multi-label classification. They define a technique, called ML-KNN, which is derived from the traditional K-nearest neighbors (1-KNN) technique. The technique uses the maximum *a posteriori* (MAP) principle to assist in determining label sets for unseen instances. Subsequently, Spyromitros *et al.* [21] created another multi-label classification technique BR-KNN, which is a binary-relevance transformation of 1-KNN, to more directly target multi-label tasks. These techniques can be thought as categorizing the data instances into one or more than one *label buckets*. These techniques have been applied to numerous applications, such as assigning movie-topic tags, determining the set of topical subjects that are discussed in text documents (*e.g.,* religion, politics, finance, or education), and associating genes with functional classes.

The fourth row of Figure 2 depicts one way in which multi-label classification can be conceptualized. Please note that the classification of each label is presented as independent only to aid its depiction and comprehension. These multi-label techniques do not decompose the problem into multiple independent binary classification problems. Instead, they do indeed consider correlations among labels. Nevertheless, this conceptualization is useful to understand the result: multiple labels, each of which can be ascribed independently, without the need to enumerate all possible combinations.

Motivation for Study of Classification Techniques. The machine-learning research community has analyzed the theoretical performance of such multi-label and problem-transformation techniques, and has also conducted empirical studies thereof. Moreover, the literature also stipulates that the nature of the data to be classified can play a strong role in determining efficacy for a given task. In this subsection, we discuss the theoretical strengths and limitations of each such technique, and we motivate the need for such scientific study for the software-engineering domain, specifically for the well-studied field of software failure classification.

In the machine-learning research community, the problem transformation methods of single-label classifiers are recognized to have potential limitations in practice. Zhang and Zhou [28] discuss the potential limitations of the Label Powerset transformation approach. They refer to two limitations: (1) *incompleteness* — LP can only predict label sets that appear in the training set, and given the large combinatorial space for all labels, such incompleteness can be likely; and (2) *inefficiency* — when the powerset of all labels is large the

training phase can be expensive. Moreover, the Binary Relevance transformation cannot leverage common co-occurrences of labels, because it treats all labels independently, and may suffer from class-imbalance the set of classes is large and label density is low [28].

Each technique provides advantages and disadvantages in different contexts, including training costs and prediction costs. As the machine learning literature recommends, the application task and nature of the data to be classified can play a strong role in effectiveness. To inform actionable guidance of which and how software failure classification techniques should be used in practice, we must study how such machine-learning classifiers perform in the specific software-engineering task.

IV. Experiment

To empirically investigate the degree to which the simplifying assumptions of prior software-engineering research affects software failure classification, we conducted an experiment. For this experiment, we had two goals: (1) to evaluate the degree to which single-label classification techniques are effective and efficient for performing software-behavior classification in the presence of multiple faults; and (2) to evaluate how pure single-label, problem-transformation methods, and multi-label classification techniques compare, in terms of effectiveness (*i.e.,* accuracy of classification) and efficiency (*i.e.,* computational time required for the analysis). To achieve those goals, we conducted a controlled experiment that includes a number of software subjects, each containing various numbers and combinations of faults that caused failure.

We posed the following two research questions:

RQ1. To what degree of accuracy does single-label, problem transformation, and multi-label classification techniques properly classify software execution behavior, at various fault quantities?

RQ2. What is the runtime expense of each such classification techniques?

Software Subject Programs. We provide the detailed information of the subject programs in Table I. We used the compression tool GZIP, the regular-expression evaluation tool GREP, two versions of the lexical analyzer FLEX, the text parser and transformer SED, and the software dependency and build tool MAKE. All of these programs, their source code, faults, and inputs are from the Software-artifact Infrastructure Repository (SIR) [19]. All of these programs are mature UNIX tools and written in C, and we compiled and instrumented them for method coverage using GCC (version 4.7.2).

Experimental Variables. We experimented using two independent variables: (1) the classification technique and (2) the number of faults that contributed to failure.

Independent Variable 1: Classification Technique. For the first independent variable, we introduce the classical classification method, K-Nearest Neighbors (KNN), and the popular technique Multilayer Perceptron (MP), which is a feed-forward artificial neural-network model. Based on these two techniques, we also introduce the two most common problem-transformation methods, Label Powerset and Binary Relevance [28]. For multi-label classification, we chose the Multi-Label K-Nearest Neighbors (ML-KNN) technique [27] and the BP-MLL [26], which is a back-propagation neural-network algorithm. We classify based on method coverage using Euclidean distance measures, for all techniques.

TABLE I: Experimental Software Subjects

Program	Version	$\|T\|$	$\|T_{\|F\|>1}\|$	$\|F\|$	$\|FV\|$	$\|E\|$
Gzip	1.1.2	214	42	7	85	18190
Grep	2.4	809	675	9	208	168272
FlexV2	2.4.7	567	549	16	3017	1710639
FlexV3	2.5.1	567	544	14	2379	1348893
Make	3.76.1	1043	503	19	3020	3149860
Sed	3.01	360	90	6	48	17280
Totals		3560	2403	71	8757	6413134

$|T|$: number of test cases
$|T_{|F|>1}|$: number of test cases capable of detecting more than one fault
$|F|$: number of discrete faults
$|FV|$: number of multi-fault versions (between 1 fault and $|F|$ faults) that we generated by combining the faults in F at varying quantities of faults (according the rules described in Independent Variable 2)
$|E|$: number of executions, which is the product of the number of test cases $|T|$ and the number of faulty versions $|FV|$

Technique 1: 1-KNN. Single-Label K-Nearest Neighbors technique.

Technique 2: 1-MP. Single-Label back-propagation Multilayer Perceptron.

Technique 3: LP-KNN. Label Powerset K-Nearest Neighbors technique, implemented in RAkLE [24].

Technique 4: LP-MP. Label Powerset back-propagation Multilayer Perceptron technique, implemented in RAkEL.

Technique 5: BR-KNN. Binary Relevance K-Nearest Neighbors technique.

Technique 6: BR-MP. Binary Relevance back-propagation Multilayer Perceptron technique.

Technique 7: ML-KNN. Multi-Label K-Nearest Neighbors technique [27].

Technique 8: ML-MP. Multi-Label back-propagation Multilayer Perceptron technique [26].

Independent Variable 2: Fault Quantity. For the second independent variable, we varied the number of faults in the multi-fault versions of the software to determine the effect of fault quantity on failure-classification effectiveness. Such investigation may inform guidance of which techniques may be more applicable at different stages of software maturity.

We set five fault quantities: 0%, 25%, 50%, 75%, and 100%. At the 0% level, the number of possible faulty-versions equals the number of faults $|F|$. At the $n = 25\%$, 50%, or 75% levels, each is comprised of randomly chosen fault combinations that contain up to n of all faults in F. At the 100% level, a single multi-fault version that contains all faults for the software program — at this fault quantity, there can only be one combination version. For fault quantities greater than 0%, if the number of possible combinations exceeds 1000, we generated only up to 1000 randomly chosen combinations.

Dependent Variable 1: F-Measure. To assess the accuracy of the classifiers, we use the F-Measure to evaluate the classification algorithms on software behavior. The classical F-Measure is related to the *precision* P and *recall* R. Given a label set $Y = \{Y_j | j = 1, 2..., m\}$, and an unseen instance e'_i of the testing multi-label software behavior classification data set E', with the following two equations, the P and R can

be computed based on the output label set y_i and the ground truth label set g_i.

$$P = \frac{1}{|E'|}\sum_{i=1}^{|E'|}\frac{|y_i \bigcap g_i|}{|y_i|}, R = \frac{1}{|E'|}\sum_{i=1}^{|E'|}\frac{|y_i \bigcap g_i|}{|g_i|} \quad (1)$$

In our experiment, F-Measure is a harmonic mean of precision and recall. Besides the failure-report classification, F-Measure is also used in many other software-engineering studies (*e.g.,* [15], [16]).

$$\text{F-Measure} = \frac{2 \times P \times R}{P + R} \quad (2)$$

Dependent Variable 2: Training and Classification Time. To evaluate the efficiency of the classification techniques, we measured the time required to train each classifier with all training data and evaluate all evaluation data using 10-fold cross-validation [10].

Experimental Setup. In this experiment, we implement our techniques atop the widely-used open-source machine-learning libraries, MULAN (version 1.5.0) [23] and WEKA (version 3.7.6) [9] to conduct the experiment.

Each classification technique that we studied is dependent upon some parameters. For the KNN-based techniques, the key parameter is k, which denotes the number of nearest neighbors. For the MP-based techniques, the $epoch$ parameter denotes the number of epochs to train through, and the *learning-rate* parameter denotes the learning rate for the neural network. In order to determine the value of these parameters and reduce any bias, we conducted a preliminary training model on some faulty versions. We set the parameters $k = 5$, $epoch = 10$ and *learning-rate* $= 0.1$ based on the default and recommended parameter settings of the algorithm inventors of MULAN and WEKA for a wide array of various tasks and used by other researchers. We then validated these parameters by sampling five percent of faulty versions and determined that these recommended settings were applicable.

For Label Powerset, we implemented the Random k-Labelsets (RAkEL) [24] algorithm, which is widely used in practice. RAkEL randomly chooses k disjoint subsets from the label set, and in the prediction phase, uses an ensemble method voting scheme. RAkEL requires that the number of labels is larger than the number of subsets, and as such cannot be applied to single-label (*i.e.,* single fault) versions.

In order to investigate the performance of the techniques, we evaluated on faulty versions of varying numbers of faults. For each of the subject programs, we generated multi-fault versions, using combinations of all faults in F: 0% (actually denotes a single fault), 25%, 50%, 75%, and 100% (which denotes a version with all faults in F) percent of $|F|$. If the number of combinations of these quantities of faults exceeds 1000, we randomly choose 1000 faulty versions to facilitate the experiment. Otherwise, we exhaustively generate all possible combinations of faults to produce the faulty versions.

In the experiment, we use ten-fold cross-validation to evaluate the effectiveness of our models. We are forced to give special considerations for the single-label techniques (which we denote as the "1-" based techniques). Particularly, in the training phase, we have to choose one label to be the "correct one", even when the failure is attributable to multiple faults. Thus, we randomly choose one from the ground-truth label set to train the single-label models. However, given this required compromise to accommodate the less-expressive single-label models, it is relatively difficult to evaluate the accuracy of its classification in a way that is fair to all techniques. Thus, in our experiment, for single-label techniques for multiple-fault versions, if the output label is any one element of the ground-truth label set, we treat the output as correct. That is to say, that in a way our results for single-label output receives an advantage — for example, if an evaluation execution failed due to Fault-1 and Fault-2, we consider either classification of "Fault-1" *or* "Fault-2" as a correct classification. The alternative would have been to treat either classification as incorrect, because the single-label technique *cannot* produce the fully-correct multiple-fault classification, by definition. These experiments were performed with a 1600MHz CPU of 8 cores, on 64-bit Ubuntu 14.04, and 12GB of RAM.

V. EXPERIMENTAL RESULTS

Figure 3 and Table II present the accuracy results of our experiment. The boxplots in Figure 3 depict the F-Measure values for each subject program, each technique, and each fault quantity, with the exception of the 100% fault quantity due to the fact that 100% of the faults produces only a single all-fault version, and as such has no variance to show with the box plot. The detailed numerical results for the mean F-Measure values for each subject program, technique, and fault combination can be found in Table II, including the 100% all-fault version. Note that for the 1-fault versions (*i.e.,* at the 0% level), the Label Powerset as implemented using the RAkEL technique is inapplicable, and as such, those cells are left blank. For convenience to the reader, we highlight the most accurate results (the highest score and all scores within 0.01 of the highest) in the table.

For the accuracy evaluation, we first compare the overall accuracy of all techniques. The Label Powerset, Binary Relevance and Multilabel techniques, regardless of whether using the KNN or MP approach, substantially outperform the single-label techniques in terms of accuracy. For the problem-transformation methods, LP-KNN, BR-KNN, LP-MP, and BR-MP provide notably similar results. The multi-label ML-KNN technique also provide similarly accurate results, whereas the ML-MP technique provided less accurate results, particularly when the version had few faults.

When considering how the number of faults affects the accuracy of the results, we note that the single-label techniques provide substantially reduced accuracy as the number of faults increases, which is unsurprising given the power and aim of such single-label classification techniques. The problem-transformation techniques and the ML-KNN technique provides only slightly reduced accuracy as the number of faults increases. Some decrease in accuracy should be expected, given that there are more labels to predict (and

TABLE II: Mean F-Measure of Each Technique and Fault Quantity

		Single-Label Alg		Label Powerset		Binary Relevance		Multilabel Alg	
Prog	Percentage	1-KNN	1-MP	LP-KNN	LP-MP	BR-KNN	BR-MP	ML-KNN	ML-MP
Sed	0%	0.8884	0.8434			**0.9597**	**0.9519**	**0.9556**	0.7208
	25%	0.7914	0.7095	**0.9283**	0.9156	**0.9275**	0.9088	0.9175	0.7463
	50%	0.6664	0.5120	**0.8823**	**0.8735**	**0.8809**	0.8526	0.8708	0.7661
	75%	0.5974	0.4449	**0.8638**	**0.8570**	**0.8638**	0.8391	**0.8568**	0.6944
	100%	0.5651	0.3583	**0.8522**	0.8380	**0.8494**	0.8297	**0.8431**	0.7608
Gzip	0%	0.8862	0.8762			**0.9780**	**0.9769**	**0.9807**	0.9237
	25%	0.7924	0.7745	**0.9601**	**0.9562**	**0.9601**	0.9496	**0.9612**	0.9441
	50%	0.7185	0.6647	**0.9462**	0.9335	**0.9462**	0.9258	**0.9438**	**0.9387**
	75%	0.6720	0.5558	**0.9361**	0.9200	**0.9361**	0.9082	**0.9300**	0.9233
	100%	0.6033	0.4880	**0.9296**	0.9086	**0.9296**	0.8962	**0.9201**	**0.9289**
Grep	0%	0.8815	0.8892			**0.9703**	**0.9795**	**0.9769**	0.6299
	25%	0.7015	0.6670	0.9431	**0.9600**	0.9443	**0.9591**	**0.9528**	0.8340
	50%	0.5555	0.5115	0.9353	**0.9516**	0.9367	**0.9509**	**0.9431**	0.8981
	75%	0.4703	0.4234	0.9309	**0.9441**	0.9323	**0.9446**	**0.9373**	0.9210
	100%	0.3848	0.3255	0.9231	**0.9356**	0.9267	**0.9370**	**0.9321**	**0.9320**
FlexV2	0%	0.8310	0.7991			0.8680	0.8873	**0.8898**	0.7567
	25%	0.5336	0.4319	0.8218	**0.8531**	0.8202	0.8451	0.8408	0.8442
	50%	0.3388	0.2475	0.8472	**0.8726**	0.8453	**0.8728**	**0.8725**	0.8627
	75%	0.2601	0.1955	0.8508	**0.8760**	0.8501	**0.8770**	**0.8769**	**0.8685**
	100%	0.2073	0.1452	0.8541	**0.8766**	0.8520	**0.8786**	**0.8785**	**0.8698**
FlexV3	0%	0.9216	0.9181			**0.9869**	**0.9869**	**0.9869**	0.8650
	25%	0.8152	0.7755	**0.9614**	**0.9618**	**0.9610**	**0.9596**	**0.9571**	0.8174
	50%	0.7244	0.6259	**0.9365**	**0.9389**	**0.9362**	**0.9338**	**0.9311**	0.8695
	75%	0.6310	0.4909	**0.9130**	**0.9148**	**0.9126**	**0.9098**	**0.9064**	0.8799
	100%	0.5194	0.3758	0.8888	**0.9106**	0.8899	0.8878	0.8802	0.8109
Make	0%	0.6158	0.5942			**0.9549**	**0.9551**	**0.9542**	0.5145
	25%	0.4069	0.3510	**0.8856**	**0.8838**	**0.8840**	0.8745	0.8747	0.6992
	50%	0.3157	0.2790	**0.8124**	**0.8062**	**0.8098**	0.7962	0.7923	0.6594
	75%	0.2435	0.2275	**0.7662**	**0.7612**	**0.7628**	0.7556	0.7473	0.6559
	100%	0.2111	0.1787	**0.7296**	**0.7308**	**0.7293**	**0.7326**	0.7207	0.6688

thus get wrong), however, it is notable how much better these techniques perform (in terms of accuracy) over the single-label techniques. Finally, the exception is ML-MP, which has improved accuracy (and more stable accuracy) as the number of faults increases.

For the efficiency evaluation, we present the results in Table III. For each subject program, we show two rows: one labeled "T" for the training phase, and one labeled "P" for the prediction phase. Note that in practice, the training phase may be performed less often than the prediction phase, as the training of the classifier can occur only when retraining is needed and prediction can occur each time that new data needs to be classified. The single-label techniques greatly outperform the problem-transformation and multi-label techniques in terms of efficiency in both training and prediction phases. For the problem transformation methods, we found that the efficiency is mainly influenced by the basic classification algorithms. We found that while the 1-KNN, LP-KNN and BR-KNN techniques have an efficient training phase (average 0.04s, 733.43s, and 66.42s), they have a relatively inefficient prediction phase (average 11.51s, 12484.74s and 6261.22s). In contrast, the 1-MP, LP-MP and BR-MP techniques spend more time in training (average 422.04s, 433287.01s and 211123.79s) and less time in prediction (average 3.73s, 3204.78s and 1534.94s).

Within the same problem-transformation method, the KNN-based algorithms spend less time in training phase and more time in prediction phase than 1-MP-based algorithms. Across the problem-transformation methods, for the same basic classification algorithm, we can find the Binary Relevance method is more efficient than Label Powerset.

As for the multi-label techniques, contrary to the 1-KNN technique and the corresponding problem-transformation methods, ML-KNN cost more time in training (average 8037.91s) and less time in prediction (average 889.09s). Compared with the LP-MP and BR-MP techniques, ML-MP has relatively higher efficiency in both training (average 4066.54s) and prediction (average 168.29s). We also get the average time-cost of each of the algorithms on the six subjective programs. Except the single-label learning technique, the BR-KNN technique has the highest efficiency in training phase (66.42s) and the ML-MP technique has the most efficient prediction phase (168.29s).

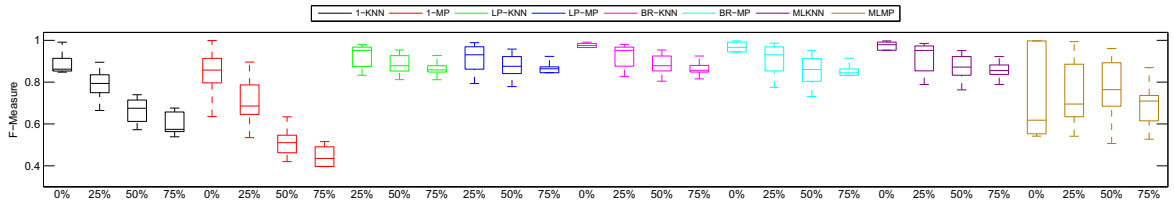
VI. Discussion

When reviewing all results from Section V, we observe some surprising, and some not-so-surprising findings.

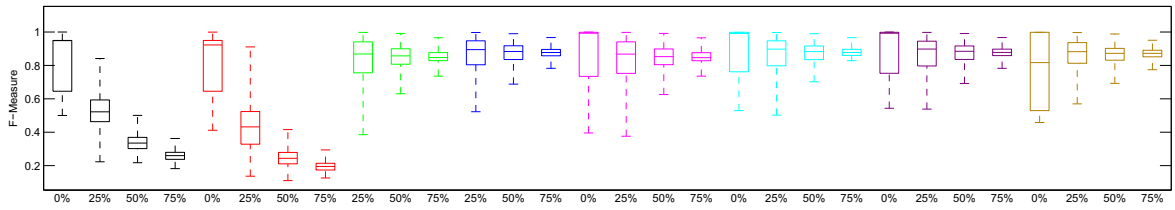
Finding 1. For single-label classification tasks, single-label techniques present an acceptable F-Measure result (0.6158–0.9216) and an exceptionally low time cost in both training and testing phases.

It is not surprising to find that single-label techniques perform acceptably for versions with only one fault — these are the circumstances for which single-label techniques were designed to best handle. This empirical result indicates the single-label classification techniques may be a good choice for usage scenarios with high efficiency requirements and a likelihood for single-label attributions.

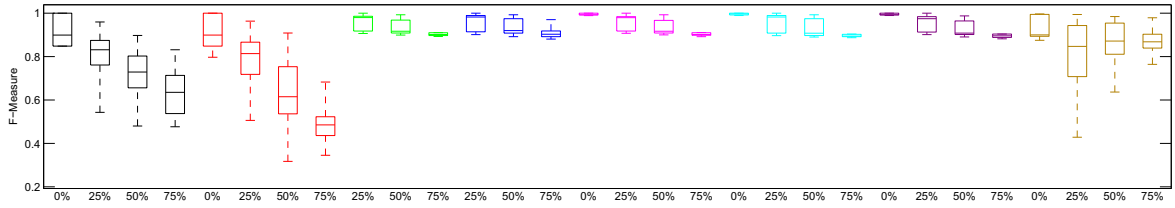
Implication. For the programs that are considered to be stable, given the high efficiency and large number of users, single-label failure classification techniques may provide adequate effectiveness at a low-runtime-overhead cost.



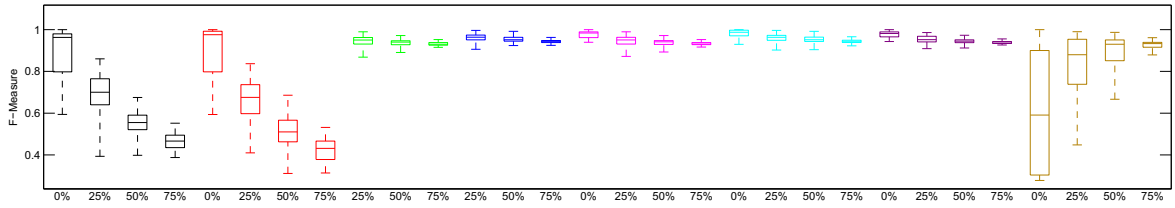
(a) Sed



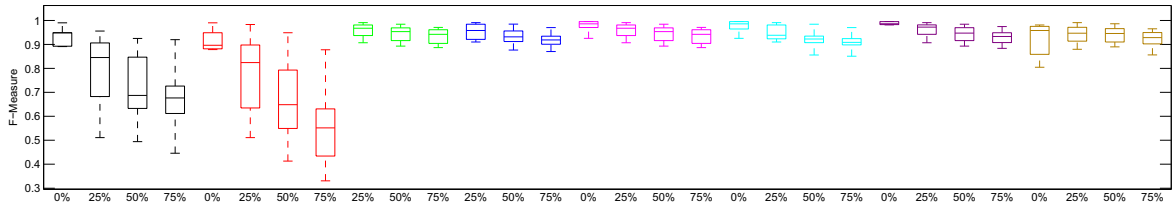
(b) Flex V2



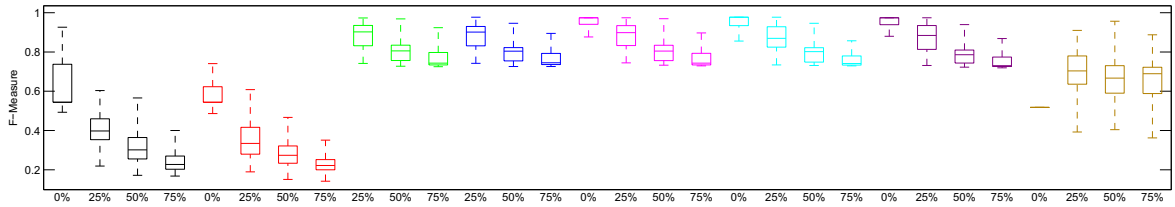
(c) Flex V3



(d) Grep



(e) Gzip



(f) Make

Fig. 3: Boxplots depicting the median (center line) and variance of the F-Measure for each subject program, each technique, and each fault quantity. The black boxes represent 1-KNN, red represents 1-MP, green represents LP-KNN, blue represents LP-MP, magenta represents BR-KNN, cyan presents BR-MP, purple presents ML-KNN, and brown presents ML-MP. (The 100% fault quantity is omitted because it represents a single faulty version and as such can be interpreted fully from the results in Table II.)

TABLE III: Time Cost of Each Technique (seconds)

		Single-Label Alg		Label Powerset		Binary Relevance		Multilabel Alg	
Prog	Phase	1-KNN	1-MP	LP-KNN	LP-MP	BR-KNN	BR-MP	ML-KNN	ML-MP
Sed	T	0.01	16.05	1.12	209.05	0.15	121.96	4.73	3.94
	P	0.40	0.17	4.04	1.38	3.08	0.94	0.88	0.25
Gzip	T	0.01	9.56	2.59	357.47	0.36	205.27	11.49	6.53
	P	0.29	0.09	9.16	2.43	5.51	1.71	2.03	0.72
Grep	T	0.01	75.78	19.85	9378.79	2.07	3492.52	143.49	74.52
	P	1.79	0.71	284.66	61.60	156.93	24.30	15.92	3.57
FlexV2	T	0.00	75.17	155.73	104136.38	16.06	48345.61	783.73	708.46
	P	1.15	0.61	1503.10	649.59	759.76	291.13	96.52	32.67
FlexV3	T	0.00	77.08	10.97	8650.59	1.63	6164.17	125.51	85.76
	P	1.04	0.62	99.64	49.18	88.19	33.91	14.91	3.56
Make	T	0.01	168.40	543.16	310554.74	46.15	152794.27	6968.96	3187.34
	P	6.84	1.53	10584.13	2440.60	5247.74	1182.95	758.84	127.52
Sum.	T	0.04	422.04	733.43	433287.01	66.42	211123.79	8037.91	4066.54
	P	11.51	3.73	12484.74	3204.78	6261.22	1534.94	889.09	168.29

Finding 2. Multi-label and problem-transformation techniques achieved relatively high accuracy and outperform the single-label techniques in both single-label and multi-label tasks.

It is also not surprising that the multi-label and problem-transformation techniques performed better when more faults were present. We were intrigued to see that the problem-transformation techniques performed surprisingly well, in terms of accuracy. Considering the theoretical limitations of such techniques (as described in Section III) the specific nature and structure of program spectra for failure classification (also described in Section III) makes such techniques feasible options to consider. This result implies that these techniques are better choices for the multi-label classification tasks, *i.e.,* the real problem domain of many software engineering tasks.

Implication. For the programs that are relatively immature (*e.g.,* in alpha- or beta-testing versions), given the high effectiveness of multi-label classification techniques and relative small number of users, multi-label failure classification techniques may provide greater accuracy, particularly when such accuracy is most needed.

Finding 3. Although ML-KNN and problem-transformation techniques are similarly effective for many multi-fault versions, the ML-KNN technique is more efficient in the prediction phase.

ML-KNN presents a relatively high time cost (8037.91s in total) in the training phase, but it achieved a low prediction time cost (only 889.09s).

Implication. Compared with problem-transformation techniques, ML-KNN is an advisable choice in situations where multiple attributions are likely and training is a less frequent requirement for multiple predictions, which is likely to be the case for many software-engineering tasks such as failure classification.

Finding 4. Execution data exhibits a degree of structure that makes problem-transformation techniques a viable option.

Besides the above summarized three findings, we conducted an initial deeper investigation of such characteristics of the data to potentially explain the accuracy successes of the problem-transformation methods, although these will need to be further investigated and validated in future work. Upon looking at the failure-causality matrices, we found that many execution failures were caused by common patterns of faults. For example, for many failures, they may have all been caused by faults F1, F5, F8, F9; and for many other failures, they may have been caused by faults F3, F11, F12. That is to say, that although there are theoretically 2^L combinations of possible fault causalities, in practice, and in our experiments, we found that far fewer of these combinations were exercised than the theoretical combinations thereof.

Implication. When adaptation of existing single-label techniques is too difficult, problem-transformation (such as label-powerset) of the single-label technique may be an acceptable choice in practice for software-failure classification.

It is worth noting that in practice, for real-world software, running "in the wild," where many faults can interact in unforeseen ways, the problem-transformation techniques may suffer from the detriments that the theoretical literature predicts, and these effects need to be further investigated in future work. That said, we have evidence to show that some form of multi-label classification, whether through problem transformation (*e.g.,* LP-KNN, BR-KNN, LP-MP, or BR-MP) or through algorithmic adaptation (*e.g.,* ML-KNN or ML-MP) can provide accuracy benefits for software that potentially contains multiple faults.

Overall, to Research Question 1, we find that for subject programs with multiple faults, where multitudes of those faults do interact to cause failures, that both problem-transformation (*i.e.,* Label Powerset and Binary Relevance) and multi-label classification techniques provide greater accuracy, in general, over single-label. But also, when a program has few (or single) faults, single-label techniques can be accurate.

Overall, to Research Question 2, we find that the single-label and problem-transformed KNN variant techniques are relatively efficient to train, but also more costly for prediction. We also find that the multi-label KNN, ML-KNN, and all MP-based techniques (single-label, problem-transformed, and multi-label) are relatively more expensive to train but also relatively efficient for prediction.

VII. Threats to Validity

The results of our experiments and their findings cannot be safely generalized to all software and all faults. Note that all subject programs are mature Unix tools, written in C. As such,

practitioners would be advised to verify the applicability of the classification techniques for their software and faults.

We presented our effectiveness results in terms of the F-measure, which integrates the constituent metrics of precision and recall. Although space constraints preclude our presentation of the constituent precision and recall scores, we note that many of the results in Table II are close to 1.0, and as such, both constituent scores must be near that value as well.

VIII. Related Work

Supervised Software Behavior Learning. Supervised software behavior learning has been widely used to assist software engineering tasks (*e.g.,* [2], [11], [13], [14]). Bowring *et al.* label branch profiles into two labels: "fail" or "passed" by applying an active learning technique and introducing a Markov model to train a classifier [2]. Haran *et al.* [11] proposed three classification methods that can be applied to different application scenarios, *i.e.,* random forests, basic association trees, and adaptive sampling association trees. Aiming at the same goal, Lo *et al.* [14] apply the pattern mining technique to reveal a set of discriminative features that can be representative of the events of programs. Based on the features, they build a model to classify the patterns into the same binary label set. Kim *et al.* [13] present a change classification technique for predicting latent software bugs. In order to classify the software changes into two labels: "clean" or "buggy", this technique treats the revision history stored in the revision control system as the features to train the classifier.

However, while a large number of single-label classification based software engineering techniques have been proposed, there are only some limited studies on multi-label software behavior learning [7], [25]. Feng and Chen first propose the techniques of multi-label software behavior learning [7], and Xin and Feng proposes a technique that is based on genetic algorithms to improve the performance of multi-label software behavior learning techniques [25]. This paper extends such work in four important ways: (1) it focuses on relatively efficient classification techniques instead of the expensive composite genetic-algorithm approach that combines individual techniques; (2) it studies more real-world software subjects of practical size; (3) it evaluates how the quantity of faults (*e.g.,* software maturity) affects performance of techniques, and (4) it finds that domain-specific software-execution data often has structure, and as such opens the possibility of using problem-transformation techniques.

Multi-label Learning. Multi-label learning techniques have gained wide attention in the machine learning community. Zhang and Zhou [28] provide a comprehensive literature survey of multi-label learning algorithms. For the multi-label learning algorithms, Zhang *et al.* [27] propose MLKNN, which is a k-nearest neighbors-based lazy learning technique. By adopting the *maximum margin* strategy with the optimized linear classifier to minimize the label ranking loss of multi-label data, Elisseeff *et al.* [6] present the Rank-SVM algorithm. Furthermore, Zhang and Zhou [26] created a technique called BPMLL, which is a surrogate ranking loss in the exponential form for neural networks.

Another option for achieving multi-label classification is through the use of single-label techniques by way of problem transformation. The most two common and straightforward problem-transformation methods are Label Powerset (LP) and Binary Relevance (BR). As we described in Section II, LP has two major limitations: (1) it cannot generalize the prediction result beyond the training label combination set, and (2) when the size of original label set is large, it could suffer from a prohibitively high time cost. By invoking LP only on size-k subset of the label space to guarantee an efficient computation, Tsoumakas *et al.* [24] propose random k-labelset (RAkEL) algorithm, which also uses an ensemble of label powerset (LP) classifiers to achieve prediction completeness.

IX. Conclusion

In this paper, we investigate the effectiveness and efficiency of multi-label classification and single-label classification on failure report classification, as well as the use of single-label classification for multiple labels through the use of problem transformations. With five real-world subject programs and 8757 faulty versions, we compared the multi-label classification methods ML-KNN and ML-MP with the transformed and original single-label classification techniques. We found that the single-label classification techniques provided relatively poor accuracy when compared to the multi-label and problem-transformed techniques for multi-label condition, which is common in practical software failure classification. We also found that the software-engineering data influences the effectiveness of the machine-learning techniques, *e.g.,* problem-transformation techniques provide adequate effectiveness. As such, this data suggests that future software-classification research should pay particular attention to situations in which more than a single label may apply. In such situations, the traditional simplifying assumption that single-label classification provides adequate accuracy should be questioned, and some form of multi-label classification should be assessed or used, whether through means of problem transformation (*e.g.,* Label Powerset or Binary Relevance) or a dedicated multi-label classification technique.

In the future, we will further investigate the ways in which fault co-occurrence affects classification, and devise specialized variants of the classifiers to potentially target the ways in which faults and their effects tend to co-occur. We also seek to further explore the ways in which such findings affect the use of machine-learning techniques in the domain of software-engineering research. Finally, we will further study the ways in which faults in deployed software may or may not behave differently for such classification problems.

Acknowledgments

This work is supported by the National Science Foundation under awards CAREER CCF-1350837, Huawei-Nanjing University Technical Collaboration Project on Automatic Precise Software Testing (YBN2016120004) and Jiangsu Prospective Project of Industry-University-Research (BY2015069-03).

REFERENCES

[1] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Software behavior: Automatic classification and its applications. 2003.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 195–205. ACM, 2004.

[3] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.

[4] V. Debroy and W. E. Wong. Insights on fault interference for programs with multiple bugs. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 165–174. IEEE, 2009.

[5] N. DiGiuseppe and J. A. Jones. Fault interaction and its repercussions. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 3–12. IEEE, 2011.

[6] A. Elisseeff and J. Weston. A kernel method for multi-labelled classification. In *Advances in neural information processing systems*, pages 681–687, 2001.

[7] Y. Feng and Z. Chen. Multi-label software behavior learning. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1305–1308. IEEE Press, 2012.

[8] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 451–462. IEEE, 2004.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[10] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Elsevier, 2011.

[11] M. Haran, A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil, and S. Fouche. Techniques for classifying executions of deployed software to support software engineering tasks. *Software Engineering, IEEE Transactions on*, 33(5):287–304, 2007.

[12] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *ACM SIGPLAN Notices*, volume 33, pages 83–90. ACM, 1998.

[13] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[14] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 557–566. ACM, 2009.

[15] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391. IEEE Press, 2013.

[16] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Multi-layered approach for recovering links between bug reports and fixes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 63. ACM, 2012.

[17] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 465–475. IEEE, 2003.

[18] J. Read, B. Pfahringer, G. Holmes, and E. Frank. Classifier chains for multi-label classification. *Machine learning*, 85(3):333–359, 2011.

[19] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software-artifact infrastructure repository, 2006.

[20] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 249–258. IEEE, 2010.

[21] E. Spyromitros, G. Tsoumakas, and I. Vlahavas. An empirical study of lazy multilabel classification algorithms. In *Artificial Intelligence: Theories, Models and Applications*, pages 401–406. Springer, 2008.

[22] G. Tsoumakas, I. Katakis, and I. Vlahavas. Mining multi-label data. In *Data mining and knowledge discovery handbook*, pages 667–685. Springer, 2010.

[23] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas. Mulan: A java library for multi-label learning. *The Journal of Machine Learning Research*, 12:2411–2414, 2011.

[24] G. Tsoumakas and I. Vlahavas. Random k-labelsets: An ensemble method for multilabel classification. In *Machine learning: ECML 2007*, pages 406–417. Springer, 2007.

[25] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang. Towards more accurate multi-label software behavior learning. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 134–143, Feb 2014.

[26] M.-L. Zhang and Z.-H. Zhou. Multilabel neural networks with applications to functional genomics and text categorization. *Knowledge and Data Engineering, IEEE Transactions on*, 18(10):1338–1351, 2006.

[27] M.-L. Zhang and Z.-H. Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern recognition*, 40(7):2038–2048, 2007.

[28] M.-L. Zhang and Z.-H. Zhou. A review on multi-label learning algorithms. *Knowledge and Data Engineering, IEEE Transactions on*, 26(8):1819–1837, 2014.