Towards More Accurate Multi-label Software Behavior Learning

Xin Xia*[♭], Yang Feng[†], David Lo[‡], Zhenyu Chen[†§], and Xinyu Wang*
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[†]State Key Laboratory for Novel Sofware Technology, Nanjing University, Nanjing, China
[‡]School of Information Systems, Singapore Management University, Singapore
xxkidd@zju.edu.cn, mf1132011@software.nju.edu.cn, davidlo@smu.edu.sg,
zychen@software.nju.edu.cn, and wangxinyu@zju.edu.cn

***Abstract*—In a modern software system, when a program fails, a crash report which contains an execution trace would be sent to the software vendor for diagnosis. A crash report which corresponds to a failure could be caused by multiple types of faults simultaneously. Many large companies such as Baidu organize a team to analyze these failures, and classify them into multiple labels (i.e., multiple types of faults). However, it would be time-consuming and difficult for developers to manually analyze these failures and come out with appropriate fault labels. In this paper, we automatically classify a failure into multiple types of faults, using a composite algorithm named *MLL-GA*, which combines various multi-label learning algorithms by leveraging genetic algorithm (GA). To evaluate the effectiveness of *MLL-GA*, we perform experiments on 6 open source programs and show that *MLL-GA* could achieve average F-measures of 0.6078 to 0.8665. We also compare our algorithm with Ml.KNN and show that on average across the 6 datasets, *MLL-GA* improves the average F-measure of Ml.KNN by 14.43%.**

***Index Terms*—Software Behavior Learning, Multi-label Learning, Genetic Algorithm**

I. Introduction

Software faults appear in all stages of software development lifecycle which necessitates perfective maintenance activities. A previous study shows that the cost of debugging in a software could consume 50% - 80% of the development and maintenance effort [1]. To improve the reliability of software systems, many vendors employ automated crash reporting systems [2]. When a program fails, these systems generate a crash report, corresponding to a *failure*, which contains an execution trace, which would be sent to the software vendor for diagnosis. An execution trace corresponds to a path that a program takes when executing from the start of the program till the end when it terminates, and a failure could be caused by multiple types of faults simultaneously. Many large companies such as Baidu[1] receive hundreds of thousands of crash reports (*failures*) every day. Thus, they organize project teams which analyze the *failures* they received, and categorize them into different labels (i.e., fault types). However, based on our interactions with Baidu developers, we find that there are two difficulties:

1) The crash reports only contain execution traces, and there are hundreds of thousands of crash reports that are received daily. Manually assigning labels (types of fault) to the failures would be a time-consuming and tedious work.
2) A failure could be caused by multiple types of faults simultaneously. Thus, often more than one label should be assigned to the failure, which makes the work even harder.

To address the two difficulties, Feng and Chen proposed multi-label software behavior learning, which automatically classifies a failure into one or more fault labels [3]. They use Ml.KNN [4], one of state-of-the-art multi-label learning algorithms, to solve the problem. Notice that multi-label software behavior learning is a difficult task: given there are a total of $|L|$ labels, since one failure could be assigned multiple labels, there would be $2^{|L|}$ combinations of labels that could be assigned to a failure.

A variety of methods have been developed to tackle multi-label learning problem. These methods can be divided into two main streams: problem transformation methods and algorithm adaptation methods [5], [6]. The problem transformation methods transform the multi-label classification task into multiple single-label classification tasks; they are usually based on either binary relevance (BR) (e.g., ensemble of classifier chains (ECC) [7]), or label powerset (LP) (e.g., random k-labelset (Rakel) [8]).[2] Algorithm adaptation methods extend specific learning algorithms in order to handle multi-label data directly (e.g., Ml.KNN [4]).

We notice that the performance of different multi-label learning algorithms differs for different software behavior datasets. Some algorithms perform much better than others on some datasets but lose to the others on other datasets.[3] In this paper, we propose a composite algorithm named *MLL-GA* which combines various multi-label learning algorithms by leveraging genetic algorithm (GA) [9]. In total, *MLL-GA* combines 12 multi-label learning algorithms including binary relevance (BR), ensemble of classifier chains (ECC), random k-labelset (RAKEL), with KNN, naive Bayes Multinomial,

[♭] The work was done while the author was visiting Singapore Management University.

[§] Corresponding author.

[1] Baidu is the largest Chinese-language Internet search provider.

[2] For the details of BR and LP algorithms, please refer to Section II.

[3] For more details, please refer to Section III.

C4.8 decision tree, and SVM [10] as their underlying classifiers, and Ml.KNN.

To examine the benefits of *MLL-GA*, we perform experiments on 3 real C programs and 3 Siemens test programs from the Software-artifact Infrastructure Repository (SIR), i.e., tcas, printtokens, printtokens2, replace, flex, and grep [11]. We show that *MLL-GA* could achieve average F-measure up to 0.8665. We also compare our algorithm with the state-of-the-art multi-label software behavior learning algorithm Ml.KNN used by Feng and Chen. The experiment results show on average across the 6 datasets, *MLL-GA* improves average F-measures of Ml.KNN by 14.43%. Moreover, since we combine 12 different multi-label learning algorithms, we also compare *MLL-GA* with each of them, and the experiment results show that on average across the 6 programs, *MLL-GA* could on average improve 10.42% over these 12 algorithms.

The main contributions of this paper are as follows:

1) We investigate the performance of many multi-label learning algorithms to solve software behavior learning problem. And we propose a composite algorithm named *MLL-GA* which combines various multi-label algorithms to achieve a better performance by leveraging genetic algorithm. Notice *MLL-GA* is an extendable algorithm, which could combine various multi-label learning algorithms.
2) We evaluate *MLL-GA* on 6 programs, and the experiment results show that *MLL-GA* could improve the state-of-the-art software behavior learning algorithm Ml.KNN by a substantial margin.

The remainder of the paper is organized as follows. We describe the background in Section II. We elaborate *MLL-GA* in Section III. We present our experiments in Section IV. We discuss related work in Section V. We conclude and mention future work in Section VI.

II. Background

In this section, we first present a motivating example to multi-label software behavior learning in Section II-A. Next, we describe the multi-label learning algorithms which would be used in this paper in Section II-B. Then, we elaborate the algorithm difference phenomenon in Section II-C.

A. A Motivating Example

To help understand multi-label software behavior learning, we present a motivating example in Figure 1, and this example is a problem that currently Baidu experiences. In practice, there are two ways to describe a failure. First, users can write a textual description of a failure, and provide the input and output of the failure, the execution trace and runtime context. Examples of this type of failure descriptions are bug reports submitted in bug tracking systems. For example, Baidu employs many temporary testers in its crowdsourcing platform,[4] and they would detect a lot of failures and report them to the development team. For this type of failure descriptions, it is easy to identify the fault types, since there are plenty of textual descriptions, input and output information. For example, consider the following failure report:

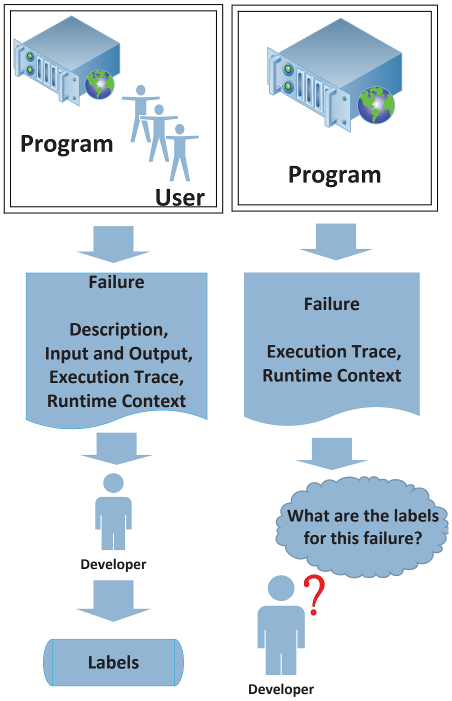

Fig. 1. A Motivating Example.

> *We have derived a Wizard from the Project Wizard, and there would be an exception when insert a new page. Moreover, edit/delete buttons in the previous pages are enabled although nothing is selected.*

After a developer reads this failure description, he/she could easily decide which types of fault the failure should be assigned to, i.e., it is assigned to both the “insert page fault”, and “edit/delete button fault” categories.

However, besides the bug reports, there is another type of failure descriptions called crash reports [2]. Crash reports contain only execution traces, and runtime contexts. In Baidu, there are hundreds of thousands of such crash reports received every day. Since there are only execution traces and runtime contexts, it would be difficult for a developer to decide the proper labels.

Thus, if we could train a model from failures with known labels and use the model for predicting the labels of failure descriptions which only contain execution traces, then the workload for developers could be reduced. As far as we know, currently Baidu organizes a team to analyze the software behaviors included in crash reports, and label these failures into different categories to help in the debugging process. In this work, we want to automate this manual work that Baidu developers are currently doing.

B. Multi-label Learning

Multi-label learning can be defined as follows. Let χ denotes the input space and let L denotes the set of labels. Given a

[4] http://test.baidu.com/

multi-label training dataset $D = \{(X_i, Y_i)\}_{i=1}^{n}$ where $X_i \in \chi$ and $Y_i = \{0, 1\}^{|L|}$ ($Y_i = 1$ indicates that the instance is assigned to the i^{th} label and $Y_i = 0$ indicates otherwise), the goal of multi-label learning is to learn a hypothesis $h : \chi \rightarrow 2^{|L|}$ which is used to predict the label set for a new instance [5], [6]. Problem transformation methods and algorithm adaptation methods are two main families of approaches to solve multi-label learning problems. In the following paragraphs, we describe binary relevance (BR) algorithm, label powerset (LP) algorithm, and Ml.KNN, which would be used in our *MLL-GA*, respectively.

1) Binary Relevance (BR): Given a set of labels L, binary relevance (BR) algorithm creates $|L|$ binary datasets from the input dataset. Each of the $|L|$ binary datasets corresponding to one label from L; Each instance in a binary dataset corresponding to a label l is relabeled as +ve, if it has the label l, or -ve, otherwise. The multi-label learning problem is then decomposed to $|L|$ single-label binary classification problems, and for each binary classification problem, a separate classifier would be built. BR algorithm ignores the label correlations, which causes it not to perform well in many datasets.

There are many extensions of BR which address the label correlations. Class chain (CC) and ensemble of classifier chains (ECC) algorithms consider label correlation by inputting the predicted labels into feature space [7]. Specially, CC also builds $|L|$ binary classifiers in sequence, and each classifier predicts one label in L. For the $(n + 1)$ binary classifier, for each instance, it would input its original features, and also the predicted labels (i.e., labels predicted in the previous n classifiers) into the $(n+1)$ classifier. For example, consider 5 labels, i.e., $\{l_1, l_2, l_3, l_4, l_5\}$. CC first randomly selects a sequence to build the 5 binary classifiers, e.g., $\{l_5, l_3, l_2, l_1, l_4\}$. Next, to build the l_5 classifier, it would use the original features in the dataset; to build the l_3 classifier, it would use the original features, and also the label predicted by the l_5 classifier; to build the l_2 classifier, it would use the original features, and also the labels predicted by l_5 and l_3 classifiers; and so on. Since CC randomly selects a sequence of labels, to increase the overall accuracy and avoid over-fitting, ECC trains an ensemble of CC classifiers.

2) Label Powerset (LP): Label powerset (LP) algorithm treats each unique label set as a new single label, and then it uses the multi-class classification methods to complete the learning task with the new single labels [5]. Given a set of labels L, LP potentially generates $2^{|L|}$ labels for multi-class classification. For the label set $L = \{1, 2, 3, ..., |L|\}$, LP transform it to $L^{'} = \{1, 2, 3, ..., 2^{|L|}\}$, then the multi-label classification problem is reduced to a multi-class classification problem. LP method considers label correlation, but since there are $2^{|L|}$ labels, it suffers from label explosion problem. With $2^{|L|}$ labels, the training set becomes extremely sparse which will cause the under-fitting problem.

Similar to BR algorithm, there are many extensions to the original LP. Most of these algorithms try to address the label explosion problem. Random k-labelsets (RAKEL) randomly chooses k labels from the L labels, and uses an LP classifier to compute the result [8]. With a set of LP classifiers, it outputs the final result using a voting-based mechanism. For example, suppose there are 3 labels, i.e., $\{l_1, l_2, l_3\}$. We select $k = 2$ labels from the 3 labels to construct LP classifiers, and we build 3 such LP classifiers correspond to e.g., $\{l_1, l_2\}$, $\{l_2, l_3\}$, and $\{l_1, l_3\}$. For a new instance, let its predicted labels by the 3 LP classifiers be $\{l_2\}$, $\{l_2, l_3\}$, and $\{l_1, l_3\}$. Since the vote for l_1 is 1, which means only 1/3 of the LP classifiers vote for it, which is less than the 0.5 (i.e., at least half of the classifiers predict it), l_1 would not be assigned to the instance; similarly, the vote for l_2 and l_3 are 2, which means 2/3 of the LP classifiers vote for them, so l_2 and l_3 would be assigned to the instance.

Notice BR and LP algorithms could use different underlying classification algorithms, such as KNN, naive Bayes multinomial, C4.8 decision tree, and SVM [10]. We refer to these kinds of algorithms (e.g., BR and LP) as *meta algorithms* [12]. In this paper, we denote BR and LP with an underlying classification algorithm c as BR^c and LP^c. For example, random k-labelsets with SVM as its underlying classifier is denoted as $RAKEL^{SVM}$.

3) Ml.KNN: Ml.KNN is one of the algorithm adaptation methods [4]. For a new instance X_{new}, Ml.KNN first gets its k-nearest neighbors $knn(X_{new})$ from the training dataset. For a label l in the label set L, it would compute the number of instances in $knn(X_{new})$ with the label l. We denote the number of data instances label l as $C_{X_{new}}(l)$.

Next, based on the above count, Ml.KNN computes the estimated probability of the new instance X_{new} to belong to the label l (denoted as $H_1^l(X_{new})$) and the estimated probability of the new instance to NOT belong to label l (denoted as $H_0^l(X_{new})$). These two estimates do not necessarily sum up to 1. The above two estimated probabilities are computed for every label in the label set L. If H_1^l is larger than H_0^l, the label l would be assigned to X_{new}.

C. Why A Composite Model?

To investigate whether different multi-label learning algorithms would perform differently on different datasets, we evaluate 12 multi-label learning algorithms on printtokens2 and flex programs.[5] The 12 algorithms are selected as follows: we choose BR, ECC, and RAKEL as meta algorithms, and use KNN, naive Bayes multinomial, C4.5, and SVM as their corresponding underlying classification algorithms. Then, we have 11 multi-label learning algorithms.[6] The twelfth algorithm is Ml.KNN. Table I presents the average F-measure scores for the 12 algorithms on printtokens2 and flex programs. The best performing algorithm for printtokens2 is RAKEL with C4.5, while the best one for flex is RAKEL with SVM. Thus, for different datasets, the best performing algorithms could be different.

We refer to this phenomenon as the *algorithm difference* phenomenon. Due to this phenomenon, if we poorly choose

[5] For the details of printtokens2 and flex, please refer to Section IV.

[6] Naive Bayes multinomial cannot work with ECC since it only handles numeric features.

TABLE I
AVERAGE F-MEASURE SCORES FOR 12 MULTI-LABEL LEARNING ALGORITHMS ON PRINTTOKENS2 AND FLEX PROJECT.

Projects	printtokens2	flex
Ml.KNN	0.7123	0.6540
BR^{KNN}	0.6991	0.6759
BR^{NBM}	0.6994	0.5394
$BR^{C4.5}$	0.8042	0.6685
BR^{SVM}	0.8018	0.7167
ECC^{KNN}	0.6965	0.6514
$ECC^{C4.5}$	0.8089	0.6344
ECC^{SVM}	0.7930	0.7030
$RAKEL^{KNN}$	0.6974	0.6828
$RAKEL^{NBM}$	0.6987	0.6082
$RAKEL^{C4.5}$	0.8091	0.7206
$RAKEL^{SVM}$	0.7951	0.7349

an algorithm (e.g., BR^{NBM} for flex), then the prediction performance would be poor. To address this, in this work, we propose a technique that combines various multi-label learning algorithms to achieve a better performance.

III. PROPOSED ALGORITHM

In this section, we first describe the overall framework of *MLL-GA* in Section III-A. Then, we present the details of the *MLL-GA* in Section III-B.

A. Overall Framework

Figure 2 presents the overall framework of *MLL-GA*. *MLL-GA* has two phases: training phase and prediction phase. In the training phase, our goal is to build a composite model learned from historical training data (i.e., failures). In the prediction phase, we apply this model to predict the proper set of labels (i.e., types of faults) for a new unlabeled data (i.e., a new failure).

Our framework takes as inputs instances from historical failures with known labels. A failure corresponds to a program execution trace, and the executions or non-executions of the various program elements are the set of binary features characterizing the failure. Next, for the i^{th} multi-label algorithm, we build a classifier M_i on the historical training data; in total, we build n multi-label classifiers (Step 1). Then, our framework searches for the near optimal composition of these n multi-label classifiers (Step 2), and after enough number of iterations, it outputs a near-optimal composite model (*MLL-GA* classifier) (Step 3). The *MLL-GA* classifier is a machine learning classifier which assigns multiple labels to a new failure based on its execution trace.

After the model is constructed, in the prediction phase it is then used to predict a set of labels for a new failure. We input the execution trace of a new failure into the *MLL-GA* classifier (Step 4). It would then output the prediction result which is a set of labels (Step 5).

B. MLL-GA: A Composite Algorithm

Since different multi-label learning algorithms would exhibit different performance on a software behavior dataset, the basic idea behind *MLL-GA* is that we assign high weights for the algorithms which perform well on the dataset, and assign low weights for the algorithms which perform poorly on the same dataset.

Formally, given n multi-label algorithms, we first build n multi-label classifiers (M_1 to M_n) on the training failures. Then, we search for a near-optimal composition of these n multi-label classifiers, and output the *MLL-GA* classifier. Given a instance j, M_i would predict its labels. For a label l, we denote $Label_i(j,l)$ as the *label vote* of l by M_i on instance j ($Label_i(j,l) = 1$ means label l is assigned to instance j; $Label_i(j,l) = 0$ means otherwise). *MLL-GA* would compute a weighted sum of all *label votes* assigned by the n multi-label classifiers, and for each label l, predict whether l is assigned to the instance j based on a *threshold score* of l. Definition 1 provides a more formal definition of the *MLL-GA* classifier.

Definition 1: (**MLL-GA Classifier**) Consider n multi-label classifiers (M_1 to M_n) built on the historical training data D. A MLL-GA classifier composes these n classifiers and assigns a label l to an instance j as follows:

$$Label(j,l) = \begin{cases} 1, & if\ Composite(j,l) \geq threshold_l \\ 0, & Otherwise \end{cases}$$

where,

$$Composite(j,l) = \sum_{i=1}^{n} \alpha_i \times Label_i(j,l) \qquad (1)$$

In the above equation, $Label_i(j,l)$ is the label vote of label l outputted by the i^{th} classifier M_i for instance j, α_1 to α_n are the weights of the n classifiers, $threshold_l$ is the boundary used to decide whether label l is assigned to an instance. Notice for each label l, there is a boundary $threshold_l$; in total, there would be $|L|$ such boundaries ($threshold_1$ to $threshold_{|L|}$). Label l would be assigned to instance j if its composite score $Composite(j,l)$ is larger or equal than $threshold_l$ (i.e., $Label(j,l) = 1$); otherwise l is not assigned to j. Note that α_1 to α_n, and $threshold_1$ to $threshold_{|L|}$ are the parameters of a MLL-GA classifier. Thus, we denote a GA classifier as ($\sum_{i=1}^{n} \alpha_i M_i$, $threshold_1$ to $threshold_{|L|}$) where each M_i is a multi-label classifier, α_i is the weight of M_i, and $threshold_1$ to $threshold_{|L|}$ are the boundary for each label in the label set L.

The search space of all possible compositions corresponds to the various assignments of values to the weights α_1 to α_n, and the thresholds $threshold_1$ to $threshold_{|L|}$. Each weight is a real number from zero to one and each boundary is a real number from zero to n. For example, suppose there are 3 multi-label learning algorithms M_1, M_2, M_3, and 5 labels l_1 to l_5. The weights for the 3 algorithms are $\{\alpha_1 = 0.7, \alpha_2 = 0.4, \alpha_3 = 0.8\}$, and the boundaries for the 5 labels are $\{1, 0.8, 1.5, 1.4, 0.5\}$. Table II presents the prediction results of an instance of *MLL-GA*. For label l_1, M_1 and M_3 predict that it is assigned to the instance, so its composite score is $0.7 \times 1 + 0.4 \times 0 + 0.8 \times 1 = 1.5$, which

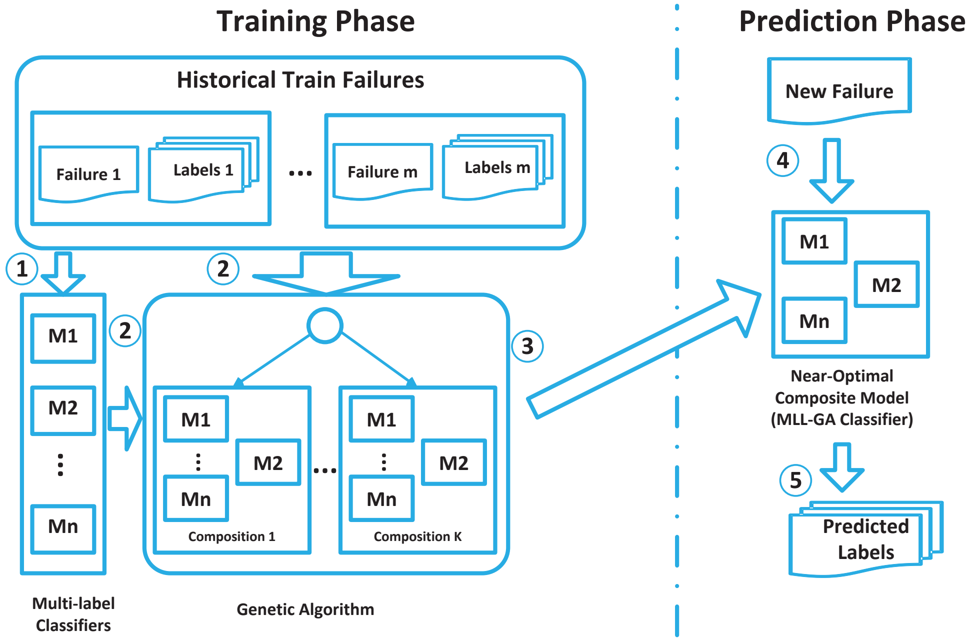

Fig. 2. Overall Framework of *MLL-GA*.

TABLE II
AN EXAMPLE WITH 5 LABELS AND 3 MULTI-LABEL LEARNING ALGORITHMS.

Algorithms	l_1	l_2	l_3	l_4	l_5
M_1	1	0	1	1	0
M_2	0	1	1	1	0
M_3	1	0	1	0	1
Composite score	1.5	0.4	1.9	1.1	0.8
MLL-GA	**1**	**0**	**1**	**0**	**1**

is above $threshold_1 = 1$. Thus, *MLL-GA* would assign l_1 to the instance.

1) Fitness Function: Fitness function measures the quality of a solution in a search space. We define a fitness score f so that the higher the score is, the better performance the *MLL-GA* classifier achieves, and the goal is to find the candidate compositions (i.e., weights and thresholds) which could maximize the score of the fitness function f. We set the fitness function as the average F-measure score (See Section IV), i.e., after we choose the weights and the thresholds, we use the composite model (i.e., *MLL-GA* classifier) to predict the label of instances in the historical training failures and compute the resulting F-measure score.

2) Detailed Procedure: We employ one of the state-of-the-art search algorithms named genetic algorithms to learn the weights (α_1 to α_n) and the thresholds ($threshold_1$ to $threshold_{|L|}$). In genetic algorithm, the solutions in a search space are modeled as *chromosomes* [13], [9]. In *MLL-GA*, a solution is a set of values for the weights and the thresholds. Genetic algorithm starts with a random selection of chromosomes, referred to as the initial *population*. Then, it evolves the population by generating subsequent *generations*, where each generation is a population of chromosomes. GA evolves the population by 3 operations: selection, crossover, and mutation. Selection refers to the process of selecting *parent* chromosomes according to their fitness scores. Crossover refers to the process of exchanging the genes of selected parents to produce offsprings with a given probability. Mutation refers to the process that the genes of new chromosomes would be modified according to a given probability. More details about GA can be found in [13], [9].

We use a simple GA [13], [9] implemented in jgap [14] in this paper. Chromosomes are represented as an array of $(n + |L|)$ doubles – n doubles represent the weights α_1 to α_n whose values are between 0 and 1, and $|L|$ doubles represent the thresholds $threshold_1$ to $threshold_{|L|}$ whose values are between 0 and n. Roulette wheel selection procedure [13], [9] is used in the selection process, which assigns a high probability to a chromosome with a higher fitness score to be selected. Single point crossover operation is used in the crossover process, which processes pairs of parent chromosomes and for each pair, it randomly picks a gene (i.e., a double value) from a parent chromosome with a certain probability, and swaps that gene and the subsequent ones with corresponding genes from the other parent chromosome. Random mutation operation is used in the mutation process, where for each gene (i.e., a double), it randomly swaps the gene with another double value with a certain probability.

Algorithm 1 presents the training phase of *MLL-GA*. For the i^{th} multi-label learning algorithms, we first build a classifier M_i based on the instances in D (Line 10), and we randomly sample a small dataset D_s from D according to the $SampleSize$ (Line 11). In this paper, we set $SampleSize$ as 30% of the total instances in D, i.e., only 30% of instances would be used to search for the near-optimal composition parameters. The reason we reduce the number of instances is to reduce the training time needed by the genetic algorithm. Then, we create an initial population (i.e., P) containing a total of $PopSize$ chromosomes (i.e., solutions) which are generated randomly, and we record the best solution (i.e., the solution with the maximum average F-measure score on D_s) among the solutions in P (Lines 12 and 13). Remember that each solution in P is a set of weights α_1 to α_n and thresholds $threshold_1$ to $threshold_{|L|}$. Next, we evolve the population in $MaxGen$ iterations; for each iteration, we perform the selection, crossover, and mutation operations on the current population, and record the best solution found so far (Lines 15 to 21). The algorithm returns the $\{\alpha_1$ to $\alpha_n\}$, and $\{threshold_1$ to $threshold_{|L|}\}$ values which maximize the average F-measure on D_s (i.e., the best solution among solutions in the initial population and the populations generated in the $MaxGen$ generations).

Algorithm 1 The Training phase of *MLL-GA*.

```
1: MLL-GA($D$, $n$, $PopSize$, $MaxGen$, $SampleSize$)
2: Input:
3: $D$: Historical training failures
4: $n$: Number of $n$ multi-label learning algorithms
5: $PopSize$: Number of chromosomes in a population
6: $MaxGen$: Maximum number of generations
7: $SampleSize$: Sample Size
8: Output: Composite MLL-GA Classifier ($\sum_{i=1}^{n} \alpha_i M_i$, $threshold_1$ to $threshold_{|L|}$).
9: Method:
10: For the $i^{th}$ multi-label learning algorithm, Build a classifier $M_i$ on $D$;
11: Let $D - s$ as a set of instances from $D$ randomly sampled from $D$.
12: Sample a small size instances $D_s$ from $D$ according to $SampleSize$;
13: Let $P$ = Initial population with $PopSize$ members;
14: Evaluate $P$ and record the best solution (i.e., the solution with the maximum average F-measure scores on $D_s$) found so far;
15: Let $curGen$ = 0
16: while $curGen < MaxGen$ do
17:     Let $P^{'} = select(P)$;
18:     $P^{'} = crossover(P^{'})$;
19:     $P^{'} = mutation(P^{'})$;
20:     Evaluate $P^{'}$ and record the best solution so far;
21:     $curGen$ = $curGen$ + 1;
22: end while
23: Output ($\sum_{i=1}^{n} \alpha_i M_i$, $threshold_1$ to $threshold_{|L|}$) which achieves the highest average F-measure score.
```

IV. Experiments

In this section, we evaluate the effectiveness of *MLL-GA*. The experimental environment is a Windows 7, 64-bit, Intel(R) Xeon(R) 2.53GHz server with 24GB RAM. We first present our experiment setup, evaluation metrics, and 3 research questions in Section IV-A, IV-B, and IV-C, respectively. We then present our experiment results that answer the 3 research questions (Sections IV-D, IV-E, and IV-F).

A. Experiment Setup

We evaluate *MLL-GA* on 3 Siemens test programs and 3 real C programs from the Software-artifact Infrastructure Repository (SIR), i.e., tcas, printtokens, printtokens2, replace, flex, and grep [11]. For each of such programs, we inject multiple faults [15] into it, and each fault represents a type of fault (i.e., a label). In SIR, for each program, there are multiple faulty versions. We collect faults from these versions and categorize them into types. We randomly pick one fault per type and inject them to each of the programs. Then, we run the test cases for these programs, and collect the execution traces. For programs tcas, printtokens, printtokens2, and replace, since they are small programs, we collect execution traces at the statement level. For programs flex and grep, since they are larger programs, we collect execution traces at the method (function) level. Next, for the test cases which fail (i.e., failures), we analyze their root causes and assign them into multiple fault types (i.e., labels). Table III presents the statistics of the 6 programs. The columns correspond to the program name (Program), the lines of source code (LOC), the number of functions (# Func.), the number of test cases (# Test Cases), the number of failures (# Fail.), and the number of types of faults (# Faults). Notice the number of failures, and the number of types of faults correspond to the number of instances, and the number of labels in the multi-label learning setting. Thus, the last two columns of Table III are in bold.

TABLE III
Statistics of Collected Datasets.

Project	LOC	# Func.	# Cases	# Fail.	# Faults
tcas	173	9	1,608	**497**	**9**
printtokens	726	18	4,130	**252**	**5**
printtokens2	570	19	4,115	**1,072**	**10**
replace	564	21	5,542	**870**	**7**
flex	10,459	162	567	**466**	**7**
grep	10,068	146	809	**657**	**7**

We implemented *MLL-GA* on top of Mulan [16], which is a Java library for multi-label learning. We choose Ml.KNN and 3 multi-label meta algorithms, i.e., binary relevance (BR), ensemble of classifier chains (ECC), and random k-labelsets (RAKEL) algorithms, since they are state-of-the-art multi-label learning algorithms, c.f., [5], [6]. We choose KNN, naive Bayes Multinomial, C4.8 decision tree, and SVM [10] as the underlying classifiers for these 3 multi-label meta algorithms. These underlying classifiers are widely used in software engineering studies (e.g., bug triaging [17], defect prediction [18], reopened bug prediction [19], etc.), and data mining studies [10]. We use the implementation of these 4 underlying classifiers in Weka [20]. In total, we combine 12 multi-label learning algorithms. The detailed configuration of the algorithms are as follows: for Ml.KNN and KNN, we set number of neighbors as 10. For naive Bayes Multinomial and C4.8 decision tree, we use the default settings in Weka. For SVM, we use the polynomial kernel. For ECC, we set the ensemble time as 10. For RAKEL, we use the random 3-labelsets, i.e., we select 3 labels to form the label powerset (LP) classifier. With the 12 multi-label learning algorithms, we use jgap, a Java implementation of genetic algorithm to combine them. We set the population size ($PopSize$ in Algorithm 1) as 1,000, and the number of iterations ($MaxGen$ in Algorithm 1) as 1,000.

Ten-fold cross validation [12] is used to evaluate the performance of *MLL-GA*, i.e., we randomly divide the dataset into 10 folds, and of these 10 folds, 9 folds are used to train a classifier, while the remaining one is used to evaluate the performance. The whole process iterates 10 times. The overall performance score across the 10 iterations is reported.

We compare *MLL-GA* with the state-of-the-art work on multi-label software behavior learning by Feng and Chen [3]. Feng and Chen makes use of Ml.KNN, and thus we compare *MLL-GA* against Ml.KNN. The number of neighbors of Ml.KNN is set to 10 – this setting is also used in Feng and Chen's paper.

B. Evaluation Metrics

Give a label l in the label set L, for an instance in the multi-label software behavior learning dataset, there are four outcomes: An instance is assigned to label l when it truly belongs to l (true positive, TP_l); it assigned to label l when it actually does not belong to l (false positive, FP_l); it is not assigned to label l when it actually belong to l (false negative, FN_l); or it is not assigned to label l when it actually does not belong to l (true negative, TN_l). Based on these possible outcomes, precision, recall and F-measure for label l are defined as:

- **Precision for l:** the proportion of instances that are correctly labeled as l among those labeled as l.
$$P_l = TP_l/(TP_l + FP_l) \tag{2}$$
- **Recall for l:** the proportion of instances labeled as l that are correctly labeled.
$$R_l = TP_l/(TP_l + FN_l) \tag{3}$$
- **F-measure for l:** a summary measure that combines both precision and recall for label l - it evaluates whether an increase in precision (recall) outweighs a reduction in recall (precision).

$$F_l = (2 \times P_l \times R_l)/(P_l + R_l) \tag{4}$$

Given the precision, recall, and F-measure for l, the average precision, recall, and F-measure across the $|L|$ labels are given as:

$$\begin{aligned} Ave.P &= \frac{1}{|L|}\sum_{l\in L} P_l \\ Ave.R &= \frac{1}{|L|}\sum_{l\in L} R_l \\ Ave.F &= \frac{1}{|L|}\sum_{l\in L} F_l \end{aligned} \tag{5}$$

Notice that the average precision, recall, and F-measure measure the prediction performance across all the $|L|$ labels, which are also used in previous software behavior learning [3], and many multi-label learning studies [8], [7].

C. Research Questions

In this paper, we are interested in answering these research questions:

RQ1 *How effective is MLL-GA? How much improvement can it achieve over Ml.KNN proposed by Feng and Chen [3]?*

We need to compare *MLL-GA* with other state-of-the-art multi-label software behavior learning algorithms. Answer to this research question would shed light to the extent *MLL-GA* advances the state-of-the art algorithms. To answer this research question, we compare *MLL-GA* with Ml.KNN proposed by Feng and Yang. We compute the average precision, recall, and F-measure scores to evaluate the performance of these 2 approaches on the 6 programs from SIR.

RQ2 *What is the performance of the 12 multi-label learning algorithms proposed to solve software behavior learning problem?*

Our *MLL-GA* is a composite model which combines 12 multi-label learning algorithms. Each of these 12 algorithms could also be used separately to solve the software behavior learning problem. However, due to the *algorithm difference phenomenon*, the selection of the best multi-label learning algorithm would be a difficult problem. Answer to this research question could validate whether *algorithm difference phenomenon* happens in other programs, and also whether the composite model is better than these single models. To answer this research question, we run these 12 multi-label learning algorithms on the 6 programs, and record their average F-measure scores, and compare with *MLL-GA*.

RQ3 *How much time does it take for MLL-GA to run?*

The efficiency of *MLL-GA* would affect its practical usage. *MLL-GA* combines various multi-label learning algorithms by leveraging genetic algorithms. Training time refers to the time to build the *MLL-GA* classifier, which contains two parts: the time to build a classifier of each of the algorithms, and the time to choose near-optimal composition of these algorithms. Prediction time refers to the time to output the final prediction set of labels: first, a new instance would be input into each of multi-label classifiers which would output the intermediary prediction results; then, these results are composed to predict the final set of labels. To answer this research question, we compare the training and prediction time of *MLL-GA* with those of Ml.KNN.

D. RQ1: MLL-GA vs. Ml.KNN

Table IV compares the performance of *MLL-GA* and Ml.KNN in terms of average F-measure, precision, and recall. The average F-measure, precision, and recall of *MLL-GA* vary from 0.6078 to 0.8665, 0.5861 to 0.8804, and 0.7170 to 0.8623, respectively.

From Table IV, the improvement of our method over Ml.KNN is substantial. Across the 6 programs, *MLL-GA* outperforms Ml.KNN by 14.43%, 5.35%, and 21.66% for average F-measure, precision, and recall, respectively. In the printtokens program, *MLL-GA* achieves the highest improvement of 29.30%, 8.33%, and 40.56% over Ml.KNN for average F-measure, precision, and recall, respectively.

Average precision and recall are both important metrics for multi-label software behavior learning since they measure quality in two aspects. If the average precision is low, then the developer would not use the tool, due to a high number of false labels. On the other hand, if the average recall is low, which means that most correct labels are not assigned to the failures,

TABLE IV
EXPERIMENT RESULTS OF MLL-GA COMPARED WITH ML.KNN.

Project	Average F-measure			Average Precision			Average Recall		
	MLL-GA	Ml.KNN	Impro.	MLL-GA	Ml.KNN	Impro.	MLL-GA	Ml.KNN	Impro.
tcas	0.6078	0.5855	3.81%	0.5861	0.5720	2.47%	0.7170	0.6410	11.86%
printtokens	0.7816	0.6045	29.30%	0.7935	0.7325	8.33%	0.7947	0.5654	40.56%
printtokens2	0.8124	0.7123	14.50%	0.8145	0.7858	3.65%	0.8193	0.6764	21.13%
replace	0.8665	0.7535	15.00%	0.8804	0.8423	4.52%	0.8623	0.7219	19.45%
flex	0.7856	0.6540	20.12%	0.7646	0.6790	9.93%	0.8608	0.6690	28.67%
grep	0.8015	0.7683	4.32%	0.7978	0.7728	3.23%	0.8290	0.7655	8.30%
Average.	**0.7759**	**0.6797**	**14.43%**	**0.7698**	**0.7307**	**5.35%**	**0.8139**	**0.6732**	**21.66%**

TABLE V
AVERAGE F-MEASURE SCORES OF MLL-GA COMPARED WITH THE OTHER 12 MULTI-LABEL LEARNING ALGORITHMS. THE LAST COLUMN SHOWS THE AVERAGE IMPROVEMENT OF MLL-GA OVER THE OTHERS.

Projects	tcas	printtokens	printtokens2	replace	flex	grep	Average.	Ave.Impro.
MLL-GA	0.6078	0.7816	0.8124	0.8665	0.7856	0.8015	**0.7759**	**0%**
Ml.KNN	0.5855	0.6045	0.7123	0.7535	0.6540	0.7683	**0.6797**	**14.43%**
BR^{KNN}	0.5913	0.5903	0.6991	0.7454	0.6759	0.7837	**0.6810**	**14.36%**
BR^{NBM}	0.5437	0.6110	0.6994	0.6459	0.5394	0.7672	**0.6344**	**23.36%**
$BR^{C4.5}$	0.6038	0.7217	0.8042	0.8363	0.6685	0.7889	**0.7372**	**5.45%**
BR^{SVM}	0.5822	0.6964	0.8018	0.8446	0.7167	0.7947	**0.7394**	**5.17%**
ECC^{KNN}	0.5918	0.6049	0.6965	0.7384	0.6514	0.7573	**0.6734**	**15.39%**
$ECC^{C4.5}$	0.6097	0.7364	0.8089	0.8351	0.6344	0.7870	**0.7353**	**5.95%**
ECC^{SVM}	0.6119	0.7374	0.7930	0.8259	0.7030	0.7997	**0.7452**	**4.11%**
$RAKEL^{KNN}$	0.5913	0.6085	0.6974	0.7443	0.6828	0.7651	**0.6816**	**13.99%**
$RAKEL^{NBM}$	0.5598	0.6161	0.6987	0.6950	0.6082	0.7951	**0.6622**	**17.73%**
$RAKEL^{C4.5}$	0.6097	0.7288	0.8091	0.8459	0.7206	0.7877	**0.7503**	**3.42%**
$RAKEL^{SVM}$	0.6097	0.7799	0.7951	0.8474	0.7349	0.7955	**0.7604**	**2.00%**

TABLE VI
AVERAGE TRAINING AND PREDICTION TIME (SECONDS) FOR MLL-GA COMPARED WITH ML.KNN.

Project	Training Time		Prediction Time	
	MLL-GA	Ml.KNN	MLL-GA	Ml.KNN
tcas	182.36	0.26	2.10	0.03
printtokens	57.69	0.25	1.44	0.03
printtokens2	629.75	3.40	38.92	0.39
replace	296.10	1.53	11.60	0.18
flex	182.22	0.74	6.08	0.09
grep	333.03	1.15	12.00	0.13
Average.	**280.19**	**1.22**	**12.02**	**0.14**

developers would not use the tool also. There is a trade off between precision and recall [12]. One can increase precision by sacrificing recall (and vice versa). In our framework, we can sacrifice precision (recall) to increase recall (precision), by manually lowering (increasing) the value of the $threshold_l$ parameters for each label l in Equation (1). F-measure, which is the harmonic mean of precision and recall, is often used to judge whether an increase in precision outweighs a loss in recall (and vice versa) [12]. Thus, in many past papers, e.g., [3], [21], [22], [23], it is often used as a summary measure.

E. RQ2: Performance of Different Multi-label Learning Algorithms

In this section, we would like to show that the composite algorithm could achieve a better performance than a separate algorithm, and also investigate whether *algorithm difference phenomenon* exists in the programs. Table IV presents the average F-measure scores of *MLL-GA* compared with the other 12 multi-label learning algorithms. Notice in this paper, *MLL-GA* is composed of these 12 algorithms. The improvement of *MLL-GA* over the other algorithms varies from 2% ($RAKEL^{SVM}$) to 23.36% (BR^{NBM}). On average across the 12 algorithms, *MLL-GA* improves the average F-measure score of 10.42%.

Moreover, the *algorithm difference phenomenon* exists in the programs. For example, in the tcas and grep programs, it would be best to select the ECC^{SVM}, and in the printtoken2 program, it would be best to select the $ECC^{C4.5}$. And for other programs, it would be best to select the $RAKEL^{SVM}$. However, in practice, we could not get these average F-measure scores before we run the algorithms. *MLL-GA* simplifies the algorithm selection process, i.e., the users just need to input their algorithms into *MLL-GA*, and *MLL-GA* would output a near-optimal composite model, and this model could on average achieve a better performance than the best single algorithms.

From Table IV, we notice that among the 3 meta algorithms, random k-labelset (RAKEL) algorithm performs the best, followed by ensemble of class chain (ECC), and binary relevance (BR). Across the 6 programs, the average F-measure scores of RAKEL, ECC, and BR varies from 0.6622 to 0.7604, 0.6734 to 0.7452, and 0.6344 to 0.7394, respectively. The reason that BR does not perform well is it ignores the label correlation, while RAKEL and ECC consider the label correlation by using an ensemble of classifiers. Moreover, among the 4 underlying classifiers, SVM performs the best, followed by C4.5, KNN, and naive Baeyes multinomial. For example, across the 6 programs, $RAKEL^{SVM}$ could achieve the average F-measure score of 0.7604, while $RAKEL^{NBM}$ only achieves 0.6622.

F. RQ3: Time Efficiency

Table VI presents the average training and prediction time needed for MLL-GA and Ml.KNN. We notice that the training and the prediction time of *MLL-GA* are more expensive than those of Ml.KNN. However, they are still reasonable. On average, we need about 5 minutes (280 seconds) to build a *MLL-GA* classifier, and 12 seconds to predict the labels for the instances in the test set, respectively. Note that the training phase can be done offline (e.g., overnight), and the learned model could be used to predict labels of many instances.

G. Threats to Validity

Threats to internal validity relate to errors in our experiments. We have double checked our experiments and the datasets collected from the 6 programs, still there could be errors that we did not notice.

Threats to external validity relate to the generalizability of our results. We have analyzed 16,771 execution traces of test cases from 6 programs in SIR, and extract 3,814 failures, and assigned them into multiple types of faults. In the future, we plan to reduce this threat further by analyzing more failures from more software projects.

Threats to construct validity refer to the suitability of our evaluation measures. We use average F-measure scores as the main evaluation metric which is also used by past studies to evaluate the effectiveness of a prediction technique in previous multi-label software behavior learning study [3], and other software engineering studies [21], [22], [23]. Thus, we believe there is little threat to construct validity.

V. Related Work

A. Software Behavior Learning

1) Multi-label Software Behavior Learning: To our best knowledge, there are limited studies on multi-label software behavior learning [3]. Feng and Chen first propose the concept of multi-label software behavior learning [3]. They study the *failures* in software projects, and find that a failure could be caused by multiple types of faults simultaneously which corresponds to multiple labels. Thus, they propose multi-label software behavior learning, and leverage Ml.KNN to solve the problem. Our work extends their work: we propose a new multi-label learning algorithm to solve the problem. We consider the *algorithm difference phenomenon*, and propose a composite algorithm *MLL-GA* which composes different algorithms by leveraging genetic algorithm.

2) Supervised Software Behavior Learning: There have been a number of studies on supervised software behavior learning [24], [25], [26]. Bowring et al. propose an active learning algorithm to classify program executions into one of the two labels: “fail” or “passed” [24]. In their model, the classifier is updated incrementally with a series of labeled data. Haran et al. propose three techniques, i.e., random forests, basic association trees, and adaptive sampling association trees, to automatically classify program executions into the same 2 labels as proposed by Bowring et al. [25]. Lo et al. propose a pattern mining algorithm to classify program execution traces into the same two labels [26]. Our study is related to, and yet is different from the above studies: the above studies work on single-label learning setting, where one instance could only belong to one label; our study works on multi-label learning settings, where one instance could be assigned to multiple labels simultaneously.

3) Unsupervised Software Behavior Learning: There have been a number of studies on unsupervised software behavior learning [27], [28], [2]. Dickinson et al. cluster program execution traces to help developers find failures [27]. They use an agglomerative hierarchical clustering algorithm and consider various kinds of distance metrics and different numbers of clusters. Podgurski et al. propose a hybrid algorithm to cluster failures such that failures caused by the same/similar faults are grouped together [28]. They first perform a feature selection step to select important features that discriminates failures from successful executions. These features are then used to cluster failures into groups of similar failures. Liu et al. present different metrics that measures the similarity (aka. proximity) of two failures [2]. These metrics can then be used to cluster failures into groups of similar failures. Our work is orthogonal to the above studies, rather than clustering failures into groups, we assign a set of labels, corresponding to fault types, to each failure. In the above studies, when failures are clustered to groups, each failure can only belong to one group (i.e., one label). In our work, each failure can be assigned multiple labels.

B. Multi-label Learning

There have been a number of studies on multi-label learning [5]. Tsoumakas et al. [5], and Zhang and Zhou [6] provide a survey of multi-label learning studies in data mining and machine learning literature. Zhang et al. propose Ml.KNN algorithm which predicts the labels for a new instance using its k-nearest neighbors [4]. Tsoumakas et al. propose random k-labelset (RAKEL) algorithm, which uses an ensemble of label powerset (LP) classifiers [8]. Read et al. propose ensemble of class chains (ECC), which builds an ensemble of sets of binary relevance (BR) classifiers [7]. In this paper, we build a composite model which combines the above algorithms to achieve a better performance.

There have been a number of studies on multi-label learning in software engineering [29], [30]. Xia et al. propose *TagCombine* to recommend tags in software information sites [29]. Xia et al. propose *DevRec* to recommend bug resolvers [30]. Each of these two studies makes use of a multi-label learning algorithm. Our work is orthogonal to the above studies since we study a different problem - we focus on predicting the fault types of a failure rather than recommending a set of tags or resolvers. Also, different from the above studies, in this study we use a genetic algorithm to combine 12 different multi-label learning algorithms.

C. Search-based Software Engineering

There have been a number of studies on search-based software engineering [31], [32], [33]. Harman et al. provide a review and classification of search-based software engineering algorithms [31]. Panichella et al. propose a search-based genetic algorithm which tunes Latent Dirichlet Allocation (LDA) parameters, and apply it in different software engineering tasks [32]. Goues et al. propose *GenProg* to automatically repair defects by using genetic algorithm [33]. In this work, we also use a search-based technique to learn a near-optimal composition of multi-label classifiers. Different from the above mentioned studies, we address a different problem namely multi-label software behavior learning.

VI. Conclusion and Future Work

In this paper, we address multi-label software behavior learning problem, which classifies a failure into one or more faults types (i.e., labels). Due to the *algorithm difference phenomenon*, we propose a composite algorithm named *MLL-GA* which combines different multi-label learning algorithms by leveraging genetic algorithm. We set the fitness function in GA as average F-measure scores to adapt to different algorithms, which makes *MLL-GA* achieve a better performance. In total, we combine 12 multi-label learning algorithms. We perform experiments on 6 programs in SIR, and *MLL-GA* could achieve average F-measures of 0.6078 to 0.8665. We also compare our algorithm with Ml.KNN used by Feng and Chen. The experiment results show on average across the 6 datasets, *MLL-GA* improves the average F-measures of Ml.KNN by 14.43%. Moreover, we compare *MLL-GA* with each of the 12 different multi-label learning algorithms, and the experiment results show that on average across the 6 programs, *MLL-GA* could on average improve 10.42% over these 12 algorithms.

In the future, we plan to evaluate *MLL-GA* with datasets from more software projects, and develop a better algorithm which could improve the prediction performance (i.e., average F-measure) further. We also plan to investigate the effect of varying amount of training data on the performance of our approach.

Acknowledgment

This research is sponsored in part by National Basic Research Program of China (973 Program 2014CB340702), NSFC Program (No.61103032, 61170067, and 61373013), and National Key Technology R&D Program of the Ministry of Science and Technology of China (No2013BAH01B03). The authors would thank the developers in Baidu for describing some problems that they experience.

References

[1] J. S. Collofello and S. N. Woodfield, “Evaluating the effectiveness of reliability-assurance techniques,” *JSS*, pp. 191–195, 1989.

[2] C. Liu, X. Zhang, and J. Han, “A systematic study of failure proximity,” *TSE*, pp. 826–843, 2008.

[3] Y. Feng and Z. Chen, “Multi-label software behavior learning,” in *ICSE*, 2012, pp. 1305–1308.

[4] M.-L. Zhang and Z.-H. Zhou, “Ml-knn: A lazy learning approach to multi-label learning,” *Pattern Recognition*, pp. 2038–2048, 2007.

[5] G. Tsoumakas and I. Katakis, “Multi-label classification: An overview,” *IJDWM*, pp. 1–13, 2007.

[6] M. Zhang and Z. Zhou, “A review on multi-label learning algorithms,” 2013.

[7] J. Read, B. Pfahringer, G. Holmes, and E. Frank, “Classifier chains for multi-label classification,” *Machine Learning*, pp. 333–359, 2011.

[8] G. Tsoumakas, I. Katakis, and I. Vlahavas, “Random k-labelsets for multilabel classification,” *TKDE*, pp. 1079–1089, 2011.

[9] D. E. Goldberg and J. H. Holland, “Genetic algorithms and machine learning,” *Machine learning*, pp. 95–99, 1988.

[10] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip *et al.*, “Top 10 algorithms in data mining,” *Knowledge and Information Systems*, pp. 1–37, 2008.

[11] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact,” *Empirical Software Engineering*, pp. 405–435, 2005.

[12] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*, 2006.

[13] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*, 2010, vol. 2.

[14] K. Meffert, N. Rotstan, C. Knowles, and U. Sangiorgi, “Jgap-java genetic algorithms and genetic programming package,” *URL: http://jgap. sf. net*, 2011.

[15] V. Debroy and W. E. Wong, “Insights on fault interference for programs with multiple bugs,” in *ISSRE*, 2009, pp. 165–174.

[16] G. Tsoumakas, E. S. Xioufis, J. Vilcek, and I. P. Vlahavas, “Mulan: A java library for multi-label learning.” *Journal of Machine Learning Research*, pp. 2411–2414, 2011.

[17] J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in *ICSE*, 2006, pp. 361–370.

[18] F. Peters, T. Menzies, and A. Marcus, “Better cross company defect prediction,” in *MSR*, 2013, pp. 409–418.

[19] X. Xia, D. Lo, X. Wang, X. Yang, S. Li, and J. Sun, “A comparative study of supervised learning algorithms for re-opened bug prediction,” in *CSMR*, 2013, pp. 331–334.

[20] 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*, pp. 10–18, 2009.

[21] 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 *FSE*, 2012, p. 63.

[22] Y. Tian, J. Lawall, and D. Lo, “Identifying linux bug fixing patches,” in *ICSE*, 2012, pp. 386–396.

[23] J. Nam, S. J. Pan, and S. Kim, “Transfer defect learning,” in *ICSE*, 2013, pp. 382–391.

[24] J. F. Bowring, J. M. Rehg, and M. J. Harrold, “Active learning for automatic classification of software behavior,” in *ISSTA*, 2004, pp. 195–205.

[25] 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,” *TSE*, pp. 287–304, 2007.

[26] 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 *KDD*, 2009, pp. 557–566.

[27] W. Dickinson, D. Leon, and A. Podgurski, “Finding failures by cluster analysis of execution profiles,” in *ICSE*, 2001, pp. 339–348.

[28] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, “Automated support for classifying software failure reports,” in *ICSE*, 2003, pp. 465–475.

[29] X. Xia, D. Lo, X. Wang, and B. Zhou, “Tag recommendation in software information sites,” in *MSR*, 2013, pp. 287–296.

[30] ——, “Accurate developer recommendation for bug resolution,” in *WCRE*, 2013, pp. 72–81.

[31] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Computing Surveys (CSUR)*, p. 11, 2012.

[32] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, “How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms,” in *ICSE*, 2013, pp. 522–531.

[33] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *TSE*, pp. 54–72, 2012.