Article citation info:
SHAO Y, LIU B, WANG S, XIAO P. A novel test case prioritization method based on problems of numerical software code statement defect prediction. Eksploatacja i Niezawodnosc – Maintenance and Reliability 2020; 22 (3): 419–431, http://dx.doi.org/10.17531/ein.2020.3.4.

Yuanxun SHAO
Bin LIU
Shihai WANG
Peng XIAO

A NOVEL TEST CASE PRIORITIZATION METHOD BASED ON PROBLEMS OF NUMERICAL SOFTWARE CODE STATEMENT DEFECT PREDICTION

NOWATORSKA METODA PRIORYTETYZACJI PRZYPADKÓW TESTOWYCH OPARTA NA PROGNOZOWANIU BŁĘDÓW INSTRUKCJI KODU OPROGRAMOWANIA NUMERYCZNEGO

Test case prioritization (TCP) has been considerably utilized to arrange the implementation order of test cases, which contributes to improve the efficiency and resource allocation of software regression testing. Traditional coverage-based TCP techniques, such as statement-level, method/function-level and class-level, only leverages program code coverage to prioritize test cases without considering the probable distribution of defects. However, software defect data tends to be imbalanced following Pareto principle. Instinctively, the more vulnerable the code covered by the test case is, the higher the priority it is. Besides, statement-level coverage is a more fine-grained method than function-level coverage or class-level coverage, which can more accurately formulate test strategies. Therefore, we present a test case prioritization approach based on statement software defect prediction to tame the limitations of current coverage-based techniques in this paper. Statement metrics in the source code are extracted and data pre-processing is implemented to train the defect predictor. And then the defect detection rate of test cases is calculated by combining the prioritization strategy and prediction results. Finally, the prioritization performance is evaluated in terms of average percentage faults detected in four open source datasets. We comprehensively compare the performance of the proposed method under different prioritization strategies and predictors. The experimental results show it is a promising technique to improve the prevailing coverage-based TCP methods by incorporating statement-level defect-proneness. Moreover, it is also concluded that the performance of the additional strategy is better than that of max and total, and the choice of the defect predictor affects the efficiency of the strategy.

Keywords: *software defect prediction, test case prioritization, code statement metrics, machine learning, software testing.*

Metodę priorytetyzacji przypadków testowych (TCP) wykorzystuje się powszechnie do ustalania kolejności implementacji przypadków testowych, co przyczynia się do poprawy wydajności i alokacji zasobów w trakcie testowania regresyjnego oprogramowania. Tradycyjne techniki TCP oparte na pokryciu na poziomie instrukcji, metody/funkcji oraz klasy, wykorzystują pokrycie kodu programu tylko w celu ustalenia priorytetów przypadków testowych, bez uwzględnienia prawdopodobnego rozkładu błędów. Jednak dane o błędach oprogramowania są zwykle niezrównoważone zgodnie z zasadą Pareto. Instynktownie, im bardziej wrażliwy jest kod pokryty przypadkiem testowym, tym wyższy jest jego priorytet. Poza tym, pokrycie na poziomie instrukcji jest bardziej szczegółową metodą niż pokrycie na poziomie funkcji lub pokrycie na poziomie klasy, które mogą dokładniej formułować strategie testowe. Dlatego w artykule przedstawiamy podejście do priorytetyzacji przypadków testowych oparte na prognozowaniu błędów instrukcji oprogramowania, które pozwala zmniejszyć ograniczenia obecnych technik opartych na pokryciu. Wyodrębniono metryki instrukcji w kodzie źródłowym i zaimplementowano wstępne przetwarzanie danych w celu nauczania predyktora błędów. Następnie obliczono wskaźnik wykrywania błędów w przypadkach testowych poprzez połączenie strategii priorytetyzacji i wyników prognozowania. Wreszcie, oceniono wydajność ustalania priorytetów pod względem średnich procentowych błędów wykrytych w czterech zestawach danych typu open source. Kompleksowo porównano wydajność proponowanej metody w ramach różnych strategii ustalania priorytetów i predyktorów. Wyniki eksperymentów pokazują, że jest to obiecująca technika poprawy dominujących metod TCP opartych na pokryciu poprzez włączenie podatności na błędy na poziomie instrukcji. Ponadto stwierdzono również, że strategia dodatkowa cechuje się lepszą wydajnością niż strategie max i total, a wybór predyktora błędów wpływa na skuteczność strategii.

Słowa kluczowe: *przewidywanie błędów oprogramowania, priorytetyzacja przypadków testowych, metryki instrukcji kodu, uczenie maszynowe, testowanie oprogramowania.*

1. Introduction

Software testing, as an indispensable stage in software development life cycle, aims to discover defects in a software artefact as much as possible to assure its quality and reliability [10]. It is a very costly and resources consuming task that accounts for higher than 50% of development costs [15]. In previous work [51], test case selection (TCS), test case minimization (TCM) and TCP are three techniques maximize the value of test suites. TCP is often used to improve the

effectiveness of some performance objectives associated with regression testing [7, 10, 21, 26, 35, 36], but it does not increase software costs like TCS and TSM by ignoring some meaningful test cases [9]. It prioritizes the test suite used according to certain test goals and strategies to optimize the speed of the defect detection rate, which is conducive to improving testing efficiency and reducing time and resource costs. With the expansion of application scenarios, it has been successfully applied in different test areas, such as graphical user interface testing [7, 20], web application testing [13].

TCP can schedule the execution of test cases in a specific order in order to enhance the defect detection rate, that is, to discover most defects as early as possible by prioritizing the most relevant test cases. It can be roughly grouped into multiple major dimensions [21, 51], such as requirement-based TCP [1, 22, 41], search-based TCP [26], coverage-based TCP [35], history-based TCP [40]. For instance, some researchers have studied the impact of requirement-based or search-based TCP methods on improving defect detection rates. For example, Arafeen et al. [1] and Krishnamoorthi et al. [22] showed that using requirement information in test case sequencing can improve testing efficiency. There are two common coverage strategies: total and additional. Li et al. [26] employed search-based heuristics TCP approaches (e.g.2-optimal greedy, hill climbing, genetic algorithms) to deal with the NP-hard problem for regression testing. Compared results on Siemens suite and space programs showed that genetic algorithms perform well. In addition, Spieker et al. [40] used neural networks and reinforcement learning to automatically select and prioritize test cases in continuous integration testing and tamed history-based TCP adaption problem. Among all TCP approaches, the coverage-based prioritization approach is one of the most commonly used in TCP [8, 17, 21, 27, 35, 36], such as statement-level coverage, branch-level coverage, method/function-level coverage and class-level coverage. The structural coverage-based method usually prioritizes test cases based on coverage, such as the total number of classes or methods covered. For example, Rothermel et al. [11, 35, 36] proposed a series of TCP methods including additional and total strategies combined with code coverage information. Henard et al. [17] comprehensively compared well-established white-box (e.g. total statement, additional branch) TCP techniques with black-box (e.g. input model diversity) ones. It is found that defects detected by the two techniques have high overlap and small performance differences. However, the black-box TCP techniques are more suitable for testing without source code. On the contrary, coverage objects considered in white-box TCP are source code, which is also the object concerned in this paper.

Unlike software reliability models [33, 38], software defect prediction can forecast defect prone or the number of defects in a software system, which is conducive to allocate testing resources efficiently. Furthermore, previous work [29, 32, 34, 43, 46, 48, 50] has suggested that using the defect-proneness of a defect predictor to rank makes sense for test cases. In other words, the more the code tends to finds defects, the higher the defect detection rate. Current defect predictors are mainly oriented to modules (e.g. class level [16], method/function level [25, 28, 37, 39, 44]), which are a relatively coarse granularity for TCP. For instance, Tonella et al. [43] integrated user knowledge through the case-based ranking machine learning algorithm with multiple, diverse TCP indexes. The method can handle partial, inconsistent and high-level data in a low-cost knowledge acquisition manner. The preliminary results showed that it is close to the optimal strategy for moderate test suite size (no more than 60). Wang et al. [7] presented quality-aware test case prioritization which leverages an unsupervised statistic defect prediction (CLAMI [30]) and a static bug finder (FindBugs [19]) to detect defective code and then revise the existing coverage-based methods by considering the weighted detective code. Empirical results performed on 7 open-source Java projects showed that it could improve coverage-based methods for test cases. Xiao et al. [48] designed a clustering-based TCP approach that utilized the defect-proneness probability based on the results of a method-level defect prediction model. The approach used a support vector machine to build the defect-proneness prediction model and employed K-means to calculate the optimal number of clusters. However, it only considered four code metrics: Line of code, total operators, total operands and cyclometric complexity. Lachmann et al. [23] proposed a system-level black-box TCP approach based on a support vector machine, which utilized test case history and natural language-based test case descriptions to prioritize. The method is compared with random and manual prioritization on the two subject systems, which show that it is beneficial to improve the defect detection rate. Mirarab et al. [29] provided a TCP approach based on the class-level Bayesian networks defect prediction model, which integrated software defect-proneness, code modification information, and test coverage data. The obtained results on a Java application showed that the approach has better test performance when there are a reasonable number of defects. Paterson et al. [32] proposed a TCP strategy (G-clef) based on defect prediction (Schwa) to reorder a test suite. They utilized code attributes such as the number of authors and revisions to configure Schwa, and compared G-clef on three groups of strategies: single-version, test execution history and software history. The results revealed that applying defect prediction to rank test cases was appealing.

However, as mentioned above, most of the current work focused on TCP methods based on coarse-grained software defect prediction models, such as class-level [29], method/function-level [7, 48], system-level [23]. Moreover, some techniques [32, 48] leverage only a small amount of metric information or user experience [43] to build defect-proneness models. If the granularity can be subdivided into the code statement-level, more accurate test resource allocation or case prioritization can be formulated, such as white-box TCP based on statement coverage [21, 35, 36]. However, these methods rank test cases in terms of the total number of statements covered without considering the probable distribution of code defects, that is, assuming that defects are evenly distributed in the source code. Nevertheless, the distribution of defects is skewed in most cases, with approximately 80% of defects occurring in 20% of the code [5]. In this paper, the defect-proneness probability of all valid statements in the code is predicted to guide coverage-based TCP. Therefore, integrating statement-based software defect prediction into a coverage-based test case prioritization has great theoretical and practical engineering value.

Furthermore, to the best of our present knowledge, there is no software defect prediction model that can predict defect prone of code statements. The key issue is how to select and extract statement metrics from source code. Besides, the existing mainstream research [29, 32, 34, 43, 46, 48, 50] only incorporates a single predictor into the TCP methods and does not compare the effects of various predictors on defect detection rate of test cases.

To overcome these shortcomings, we put forward a novel test case prioritization method based on software code statement defect prediction (TCP-SCSDP). The main idea of our method TCP-SCSDP is first to inject defects and extract code statement features to form a labelled defect dataset and perform data pre-processing on it. And then a supervised machine learning algorithm (e.g. random forest) is selected to build a software defect prediction model based on code statements. The model can assess the defect probability of statements and achieve the fine-grained prediction at the statement-level. Taking the white box test cases based on code coverage as the subject (test set), the prioritization strategy is utilized to calculate the defect detection rate of each test case. Finally, the weighted average percentage of defect detection is utilized to measure the ranking results. It can be observed that the code statement-level defect prediction method can not only improve the prediction granularity problem, but also fill the gap of the test case prioritization method based on the code statement-level defect-proneness prediction.

Our contributions to the current research are as follows:

(1) We propose a software defect prediction model based on code statement-level, which addresses the problems of acquiring statement metrics and prediction granularity.
(2) We incorporate code statement level software defect prediction into the test case prioritization process, which improves the defect detection rate.
(3) We systematically compare the effects of four common prioritization strategies and three classic predictors on test performance and find that the additional strategy is to the max and total ones and the choice of predictors has an impact on the prioritization strategy.
(4) We present an empirical evaluation using four open-source projects from the Software Infrastructure Repository (SIR). The experimental results indicate that our proposed TCP-SCSDP is effective and feasible.

The remainder of this paper is organized as follows: The remainder of this paper is organized as follows: Section 2 presents the test case prioritization based on code statement defect prediction. Section 3 is devoted to the experimental setup. Experimental results and discussion are described in Section 4. Some threats to validity are described in Section 5. Finally, conclusions and future work are generalized in Section 6.

2. Test case prioritization based on code statement defect prediction

2.1. The proposed overall framework

The proposed method TCP-SCSDP is divided into two parts, code statement level defect prediction, and the prioritization strategy. The framework is presented in Fig. 1. The software defect prediction method based on code statement-level is a more fine-grained prediction than the traditional method based on function level or class level, which helps to solve the problem of unable to predict accurately. Firstly, metrics and defect information of code statements are extracted from the source code and software test report to form a labeled training dataset. And selecting a supervised learning algorithm (e.g. RF, GBRT, LRM) to train a software defect prediction model. Similarly, the features of the new software code statements are used as the validation test dataset. Secondly, the proposed prediction model is used to forecast the defect proneness of each line of code statement. Then we apply the test case priority such as total strategy and additional strategy [35] to calculate the defect probability of each test case. Finally, the test cases are sorted in terms of the probability of defects.

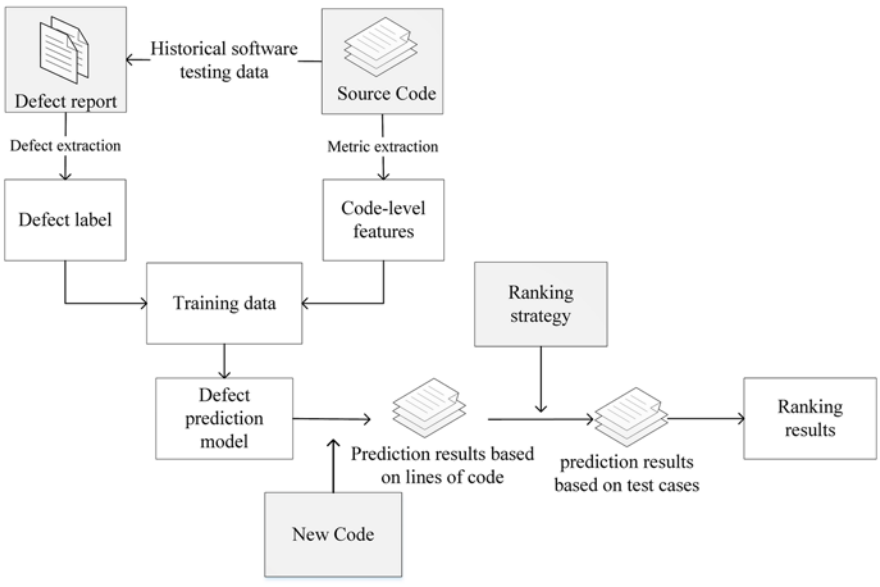


Fig. 1. The proposed test case prioritization framework based on statement-level defect prediction

2.2. Software defect prediction based on code statement level

2.2.1. Description of program code statement features

A key issue in statement-based software defect prediction is the way to extract code features. Traditional code feature (attribute or metric) such as McCabe, Halsted, and other metrics describes a module static metric information [28]. And machine learning algorithms are utilized to build the relationship between software module metrics and defect. The main difference from the traditional code features is that code vocabulary is used as independent variables in the software defect prediction based on code statement.

Regardless of the programming language, source code is composed of three vocabularies: language keywords, operators and operands.

- Language keywords, called reserved words, indicate an identifier that has a special meaning defined in advance by programming language. They can be used to describe a data type, the logical structure of a control program, and so on. In general, they cannot be used arbitrarily as variable names, method names, class names, package names, and parameters.

Table 1. 32 C language keywords defined by ANSI-C.

Type	Description
Type keywords	Basic type: void, char, int, short, long, double, float, signed, unsigned Complex type: enum, struct, union
Control keywords	Cycle control: for, do, while Condition control: if, else, switch, case, default Jump control: break, continue, goto, return
Storage keywords	auto, extern, register, static
Modifier keywords	const, sizeof, typedef, volatile

- Operators are symbols that represent specific operations and can be used to construct program language expressions. Common operators are mainly divided into five categories: arithmetic operators, relational operators, logical operators, assignment operators, as well as operators that complete bit operations - bitwise operators.
- An operand is an entity on which an operator acts and specifies the number operation variable in the instruction.

For example, the American National Standards Institute (ANSI) defines 32 language keywords for C programs. According to the characteristics of the programming language, four keywords are designed: type, control, data storage, and other keywords, among which type keywords indicate the type of data, control keywords are responsible for the logic of the processing program, store keywords declare variable range. Some of these types can be refined in combination with the functional struc-

Table 2. List of C language operators

Arithmetic operators	Relational operators	Logical operators	Bitwise operators	Assignment operators
+:addition or unary plus -:subtraction or unary minus *:multiplication /:division %:remainder ++:prefix increment –:prefix decrement	==:equal to !=:not equal to >:greater than >=:greater than or equal to <:less than <=:less than or equal to	&&: logical AND ||:logical OR !:logical NOT	&:bitwise AND |:bitwise OR :bitwise complement ~:bitwise exclusive OR <<:left shift >>:right shift	=:basic assignment +=:addition assignment -=:subtraction assignment *=:multiplication assignment /=:division assignment %=:remainder assignment &=:bitwise AND assignment |=:bitwise OR assignment ~=:bitwise XOR assignment <<=:left shift assignment >>=: right shift assignment

ture. As shown in Table 1, control keywords can be divided into loop control, condition control, and jump control.

Operators are binary operators that assign the right operand to the left. Operators in C language are mainly divided into five categories, as shown in Table 2.

Obviously, there may be differences between keywords and operators for different programming languages, which are related to its own characteristics and design style. Java keywords are shown in Table 3. By comparing the keywords of C and Java language, we can see that C is a structured oriented language and Java is a typical object-oriented programming language. Therefore, there are many keywords describing classes in Java, such as abstract and interface, which are impossible to appear in C. In addition, although Java provides keywords such as try and catch for exception handling mechanisms, there is little difference between C and Java in a logic structure such as control and type keywords.

Table 3. List of Java language keywords

Keywords	Description
Access Modifiers	Private, protected, public
Class, Method, and	Abstract, class, extends, final, implements, interface, native, new
Variable Modifiers	Static, strictfp, synchronized, transient, volatile
Process control	Break, continue, return, do-while, if-else, for, instanceof, switch-case-default
Exception handling	Try catch, throw, throws
Package	Import, package
basic data types	Boolean, byte, char, double, float, int, long, short, null, true, false
variable reference	Super, this, void
reserved word	Goto, const

2.2.2. Feature extraction based on code statement

In the previous section, it was emphasized that the program code is composed of three vocabularies, that is, most of the defects in the code are basically related to them. For example, the wrong variable type, the wrong operator, and the wrong operand are defined respectively. Regardless of the defects that may be caused by the code running environment, the defects are highly related to these vocabularies. In this paper, keywords, operators, operands, and lines are mainly utilized as the metrics of each line of code for statement-based software defect prediction, as illustrated in Table 4.

Fig. 2 shows the process of extracting features from valid statements, which do not contain comment lines, empty lines, and brackets. The current code is line 8, including 1 basic type, 2 arithmetic

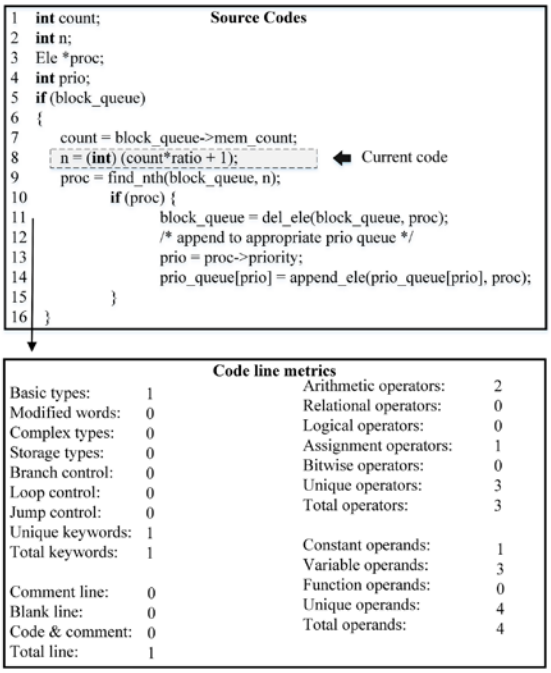


Fig. 2 The process of code feature extraction.

operators, 3 variable operands, etc. The features and a defective class label together constitute the original dataset of software defect prediction.

2.2.3. Feature selection

It is a known fact that attribute selection has been extensively used in SDP [2, 16, 18, 24, 28, 45]. However, the following issues may arise during model training without using attribute selection (or feature selection): 1) The noise, redundancy and irrelevant features may be included. As all features collected in a dataset can be used for different tasks, not all of which contribute to SDP evaluation. 2) The accuracy, interpretability and generalization ability of the predictor may be affected.

Attribute selection can be primarily split into filter-based and wrapper-based approaches. The wrapper-based method obtains the best attribute subset by interacting with the learner feedback, while the filter-based method evaluates the feature subset on the training set without relying on the learner. Compared with the filter-based method, the wrapper-based method is inclined to more computationally complexity. Hence, a filter-based attribute selection method is preferred.

In this study, the classic correlation-based feature selection (CFS) filter method [14] is employed in software defect prediction [2, 3, 6].

Table 4. Simple description of code metrics

Features	Description
Keywords	Basic types, modified words, complex types, storage types, branch control, loop control, jump control, unique keywords, total keywords
Operators	Arithmetic operators, relational operators, logical operators, assignment operators, unique operators, total operators
Operands	Constant operands, variable operands, function operands, unique operands, total operands
Lines	Total line, comment line, blank line, code & comment
Defect	Bug

A subset of metrics is treated as input rather than all attributes of the dataset according to the heuristic function. This function (Equation 1) assigns a high score to a subset of metrics that are highly related to the class and have a low correlation with each other. Redundant metrics are therefore discriminated against because they are highly associated with one or more other metrics.

$$Merit_s = \frac{\overline{r_{cf}}}{\sqrt{k + k \bullet (k-1) \bullet \overline{r_{ff}}}} \quad (1)$$

Where Merits is the metric subset containing k metrics, $\overline{r_{cf}}$ is the average metric-class correlation, and $\overline{r_{ff}}$ is the average metric-metric intercorrelation.

2.2.4. Building the prediction model

The software defect prediction model based on valid code statements is a method that uses a predictor to find the relationship between features and the defect class label. Generally, software defect prediction models are trained and tested on the same project dataset [25, 28, 37, 39]. However, it is difficult to build the software defect prediction model when there is not sufficient historical data [44]. In order to avoid this problem, features and the defective label of the context scene are obtained by selecting similar tested software from other projects as training samples. In addition, metrics of the context scene is extracted from the tested software code as new data. By building and testing the statement-based prediction model in the training dataset and the new target testing dataset respectively, the probability of defect proneness of each line of code is calculated, as shown in Fig. 3.

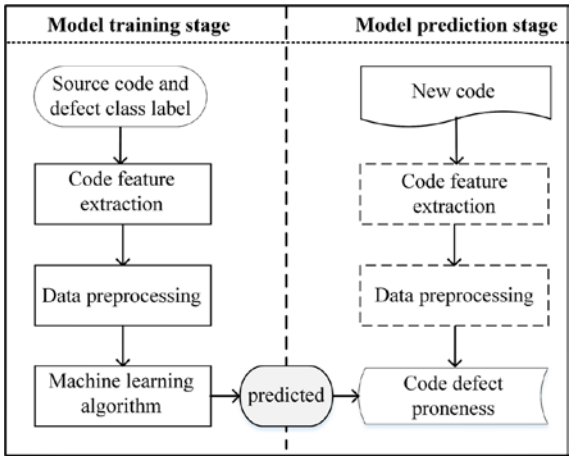


Fig. 3. The proposed software defect prediction framework based on code statementlevel

There is no difference between the defect prediction based on code context scenario and the traditional approach based on method/function or class defect prediction in the construction phase. The main difference lies in feature extraction that has been introduced in the previous part. There are plenty of machine learning methods, including supervised learning, unsupervised learning, and semi-supervised learning. Common supervised learning algorithms have been investigated for software reliability prediction and software defect prediction, such as Naive Bayes (NB) [2, 16, 28], linear Regression Model (LRM) [34], non-linear classifiers (e.g. support vector machines (SVM), neural network (NN)) [3, 49], ensemble learning (e.g. random forest (RF), bagging, Gradient Boost Regression Tree (GBRT)) [25, 47], and tree/rule-based classifiers (e.g. OneR, RIPPER, decision tree (C4.5), decision table (DT), partial decision trees (PART)) [28, 42]. Existing software defect predictors are mainly used for classification and ranking. Classification aims to predict software sample entities into defect proneness or non-defect proneness and the ranking task is to prioritize the samples according to the predicted code defect proneness probability. Compared with software defect classification, ranking results are more flexible and easier to use. Defect prediction can guide software testing to optimize resource allocation, which is essentially a test priority determination problem. This article uses code metrics for training to forecast the probability of defect proneness in code statements and then ranks them. Fig. 4 shows an example of the prediction results of the proposed method with the actual program code.

Lines	Probability	Source codes
7208	0.0175	if (fastmap && startpos < total_size && !bufp->can_be_null)
7209	0.0000	{
7210	0.0210	if (range > 0) /* Searching forwards. */
7211	0.0000	{
7212	0.0210	register const char *d;
7213	0.0210	register int lim = 0;
7214	0.0131	int irange = range;
7215	0.0000	
7216	0.0061	if (startpos < size1 && startpos + range >= size1)
7217	0.0025	lim = range - (size1 - startpos);
7218	0.0000	
7219	0.0025	d = (startpos >= size1 ? string2 - size1 : string1) + startpos;
7220	0.0000	
7221	0.0000	/* Written out as an if-else to avoid testing `translate'
7222	0.0000	inside the loop. */
7223	0.0104	if (translate)
7224	0.0324	while (range > lim
7225	0.0025	&& !fastmap[(unsigned char)
7226	0.0025	translate[(unsigned char) *d++]])
7227	0.0025	range--;
7228	0.0025	else
7229	0.0025	while (range > lim && !fastmap[(unsigned char) *d++])
7230	0.0104	range--;
7231	0.0000	
7232	0.0139	startpos += irange - range;
7233	0.0000	}
7234	0.0139	else /* Searching backwards. */
7235	0.0000	{
7236	0.0139	register char c = (size1 == 0 || startpos >= size1
7237	0.0139	? string2[startpos - size1]

Fig. 4. An example of defect prediction result based on code statement-level

In Fig. 4, the leftmost column is the code line, the middle column is the defect prediction probability, and the right is the source code. We can see from this example that the software code statement-level defect prediction is feasible and has great research potential.

2.2. Test case Prioritization

The TCP problem is formalized as follows [35]:

Assumption T: A Test suite; PT: The set of permutations of T; f: a function from PT to the real numbers.

Problem: find $T' \in PT$ for $(\forall T'')(T'' \in PT)$ meets $f(T') \geq f(T'')$, where $(T'' \neq T')$.

In the formal description, PT represents all possible TCP suites. The input of function f is the specified priority order generated by the ranking target, and the output result is linked to the ranking target. Generally speaking, typical ranking targets include code cover-

age, defect detection rate, defect important, test cost and so on. In this paper, the code coverage-based TCP method concentrates on three aspects: prioritization strategy, prioritization criteria, and prioritization search.

Prioritization criteria describes the coverage criterion of the internal structure of the program code, such as method coverage, statement coverage, branch coverage and modified condition/decision coverage (MC/DC), which reflects the test adequacy. This paper associates code statement defect-proneness prediction with statement coverage to achieve test case sequencing. Prioritization search refers to the search algorithm used in sorting test cases such as greedy algorithm, genetic algorithm [26]. Most studies have adopted the greedy method, and empirical studies [11, 36] also prove that the greedy method is a simple and efficient ranking method. Prioritization strategy refers to the method used to prioritize test cases in a test suite. The typically used prioritization methods are as follows [11, 35]:

- Random prioritization: It assigns the priority of test cases in the test case set without any sorting criteria. It is used as a baseline for performance comparison in this paper.
- Optimal prioritization: It is based on the fact that each test case in the test suite can expose software errors, and determines the optimal sorting of test cases to maximize the defect detection rate. However, it is not a practical method because it requires pre-determining the defect information, which is often not available before testing. It was chosen to serve as an upper bound on the effectiveness of ranking strategies to distinguish the gaps between various strategies and the optimal solution.
- Total prioritization: It is a completely static strategy that directly counts the coverage of each test case in the test suite. The defect detection rate of test cases is calculated according to the code coverage and the test cases are sorted.
- Additional prioritization: It is a feedback mechanism strategy that takes into account overall coverage. Software entities (e.g. functions, statements, branches) covered by test cases are no longer considered. Therefore, after executing a test case, the covered software entities are eliminated and the remaining test cases are reordered. With the continuous execution of test cases, software entities will gradually be covered. When they are all covered, these entities need to be reset to the uncovered state, and the above process is repeated for the remaining test cases.
- Max prioritization: It takes the maximum defect prediction probability of the covered program code as the defect detection rate and the test cases are prioritized.

Prioritization strategy is usually combined with sorting criteria and sorting search. It is closely related to the average percentage faults detected (APFD) value, and different prioritization strategies often lead to different prioritization results. In this paper, we adopt a greedy search method based on code statement coverage to prioritize test cases on different strategies.

2.3. The defect detection rate of test cases

The proposed TCP-SCSDP determines the priority of test cases by their defect detection rate–a measure of how quickly defects are detected during software testing [11, 35]. Fig. 5 shows an example of defect detection rates for the total, additional and max prioritization strategies. It can be seen that the defect detection rate varies with the prioritization strategy.

Where the probability is obtained from software defect prediction based on code statement. The defect detection rate is computed according to the selected prioritization strategy and the code coverage of the test case. For example, the defect detection rate of test case 1 is 0.686 in total, 0.387 in max and 0.686 in additional, and the defect detection rate of test case 2 is 0.672 in total, 0.209 in max and 0.343 in additional prioritization. It is worth noting that the additional strategy is a feedback strategy, which will eliminate the covered code after the last test. After the defect detection rate of test cases is obtained and sorted, the APFD value is finally calculated.

Source codes	Probability	Test case 1	Test case 2
int main(int argc, char *argv[]	-		
{	-		
int x,y,z,m;	0.012	*	*
x=atoi(argv[1]);	0.015	*	*
y=atoi(argv[2]);	0.015	*	*
z=atoi(argv[3]);	0.015	*	*
if(y<z)	0.116	*	*
if(x<y)	0.116	*	*
m=y;	0.387	*	
else if(x<z)	0.174		*
m=y;	0.387		
else m=z;	0.209		*
else if(x>y)	0.174		
m=y;	0.387		
else if(x>z)	0.174		
m=x;	0.387		
else m=z;	0.209		
}	-		
Total		0.686	0672
Max		0.387	0.209
Additional		0.686	0.343

Fig. 5. Defect detection rate of test cases under different ranking strategies

3. Experimental setup

3.1. Research questions

In order to study the performance of the proposed method TCP-SCSDP systematically and objectively, we present three questions as follows:

RQ1: What is the performance comparison of different prioritization strategies?

To answer this question, we compared the APFD values under the five prioritization strategies of random, optimal, max, additional and total. Among them, the random strategy and the optimal strategy are used as a comparison baseline to evaluate the test performance of total, max and additional strategies.

RQ2: How do different software defect predictors affect test case prioritization?

The test prioritization task in this paper is the result of software defect prediction based on the code statement-level. Its test performance depends on the defect predictor. To answer this question, we choose LRM, GBRT and RT classifiers for comparative analysis.

3.2. Benchmark classifiers

For experimental comparison research, three distinct types of machine learning methods are used: Linear Regression Model (LRM), Random Forest (RF), and Gradient Boosting Regression Tree (GBRT). The three classifiers are widely used in software defect prediction and show superior prediction performance [25, 34, 47]. In addition, because this paper is a test case prioritization based on code statement-level defect prediction, all three algorithms can well support the defect prediction.

LRM: A curve that is called the best fitting curve is utilized to fit the data points, and the fitting process is called regression. When the curve is linear, the process is called the linear regression. The main idea of linear regress model is to use the pre-determined weights to combine the attributes to represent the categories.

$$x = w_0 + w_1 a_1 + w_2 a_2 + \cdots + w_k a_k \quad (2)$$

where x is a class label, $a_1, a_2, \cdots, a_k$ is an attribute value, $w_1, w_2, \cdots, w_k$ is a weight value. In general, LRM can be resolved by the least square method.

RF: It is an improved decision tree algorithm, which is a typical bagging ensemble learning, mainly to handle the over-fitting problem of decision tree. It has the advantages of high accuracy, not easy to overfit and can handle high-dimensional data [47]. It uses multiple the mechanism of decision trees for voting to improve the prediction results. Assume that random forest is composed of m trees, where each tree is generated by a certain amount of training samples n. In order to ensure the generalization ability of random forest, n samples are generated by bootstrapping and the final prediction results are obtained by bagging.

GBRT: It is a gradient boosting algorithm, which was first put forward by Friedman [12]. It fits a regression tree by using the fastest descent approximation method, that is, using the value of the negative gradient of the loss function in the current model as the approximate value (pseudo-residual) of the residual of the lifting tree algorithm in the regression problem. In simple terms, each tree in a progressive gradient regression tree is learned from the residuals of all previous trees.

3.3. Benchmark datasets

In this paper, four open-source C program datasets are chosen as experimental objects, which are *Gzip*, *Grep*, *Flex* and *Sed*. Among them, *Gzip* is a widely-used file compression program of GNU free software, *Grep* is a text search tool running under Linux which can search text using specific pattern matching including regular expressions, *Flex* is a program for SQL lexical analysis in Linux environment and sed is a tool for running Linux instructions. These programs are intensively studied in the field of software engineering [4, 17]. Their source code and related materials can be accessed from SIR. According to the defect information found in the historical version of these codes, defect injection is carried out selectively, in which the defect of the deleted class and non-modified code class are not injected. Table 5 describes the basic information of the datasets.

Table 5. An overview of subjects used in this study

Dataset	Size	Injected defects	Defect rate per thousand lines of code (KLOC)	Number of test cases	Source	Description
Gzip	5680	37	6.51	279	SIR	File compression utility
Grep	10068	47	4.67	669	SIR	Text search tool
Flex	10459	32	3.06	447	SIR	SQL parsing tool
Sed	14427	27	1.87	261	SIR	Linux command run tool

3.4. Performance evaluation measures

The goal of test case prioritization is to find as many software defects as early as possible, so as to they can be fixed early in testing. It can improve the effectiveness of software testing and shorten the software development life cycle. Generally, APFD is invoked as the performance evaluation indicator of the priority ranking method. Suppose there are n test cases in a test case T, m defects found in a defective set F, and the test case rank T', its APFD is as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_i + \cdots TF_m}{m * n} + \frac{1}{2n} \quad (3)$$

In Equation 3, TF_i is the position of the first test case with defect i found in sorting strategy T'. The APFD value ranges from 0 to 1. The higher its value is, the better the test case prioritization is. For instance, there is a test case set as shown in Table 6. If it is sorted according to $T_1 - T_{10}$, the APFD value of the test sequence is $APFD = 0.46$. If it is sorted according to $T_{10} - T_1$, the APFD value of the test sequence is $APFD = 0.7875$.

Table 6. Correspondence between test cases and defects.

Test Case ID	Defect ID							
	F1	F2	F3	F4	F5	F6	F7	F8
T1								
T2								
T3		※					※	
T4	※		※				※	
T5		※					※	
T6								
T7		※	※	※	※			
T8	※		※					
T9						※		
T10		※		※				※

4. Experimental results and discussion

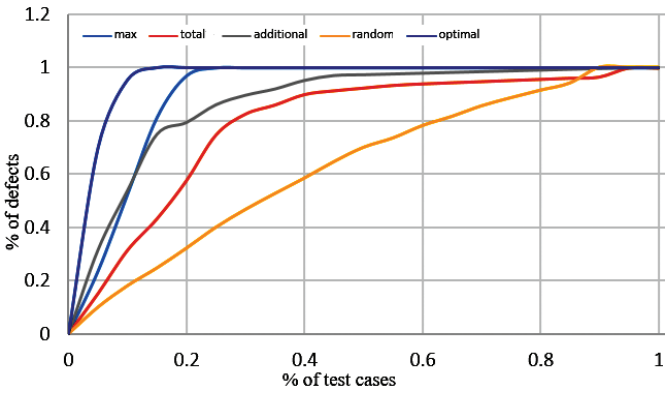
In order to systematically study the problems raised, four open-source program datasets of SIR website are taken as experimental objects. Taking the set of Siemens programs (*Gzip*, *Grep*, *Flex* and *Sed*) as the data source, data samples are established through the automatically extracted code features and defect label as training dataset of the prediction model. And the testing dataset is removed from the training dataset to avoid over-fitting seen in Table 7. A prediction model trained from the historical data across other projects is utilized to predict defects in the project to solve the problem of insufficient historical defect data. For instance, assuming that the training dataset is {*Grep*, *Flex*, *Sed*} then the *Gzip* program is the testing dataset, and then the *Gzip* program is the testing dataset.

Aiming at the first problem RQ1, five test case prioritization strategies, max, total, additional, random and optimal, are compared. Because the random strategy has randomness in selecting test cases, the average result of 20 times will be used as a comparison value to objectively evaluate its performance. In addition, for the second problem RQ2, three different predictors, LRM, RF, and GBRT,

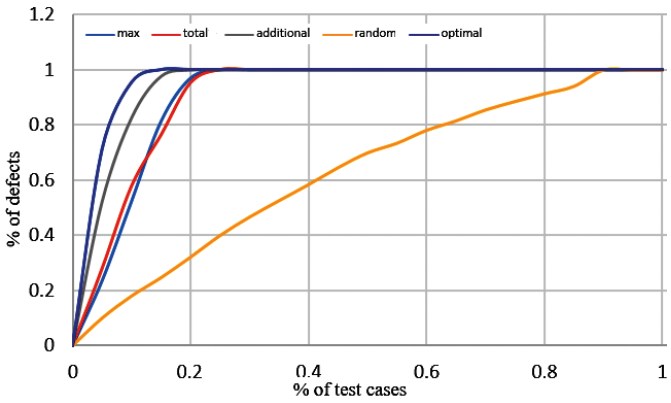
Table 7. Case studies

Training	Testing (case study)
Grep, Flex, Sed	Gzip
Gzip, Flex, Sed	Grep
Gzip, Grep, Sed	Flex
Gzip, Grep, Flex	Sed

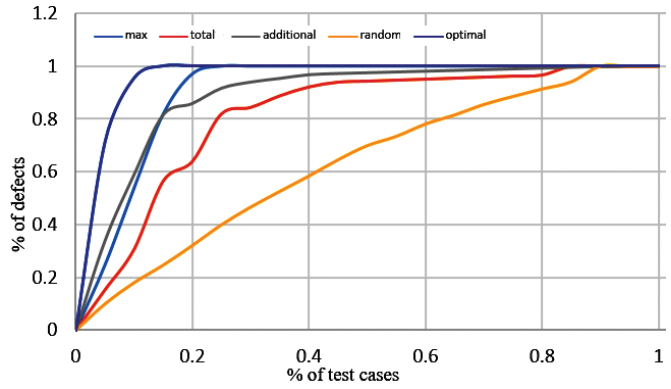
are used to analyse the impact of on the ranking of test cases. Fig. 6-9 show the Alberg diagrams [31, 34] of the experimental results, where the x-axis indicates the proportion of test cases used in the total test cases and the y-axis indicates the percentage of defects found in the total defects.



(a) LRM



(b) RF



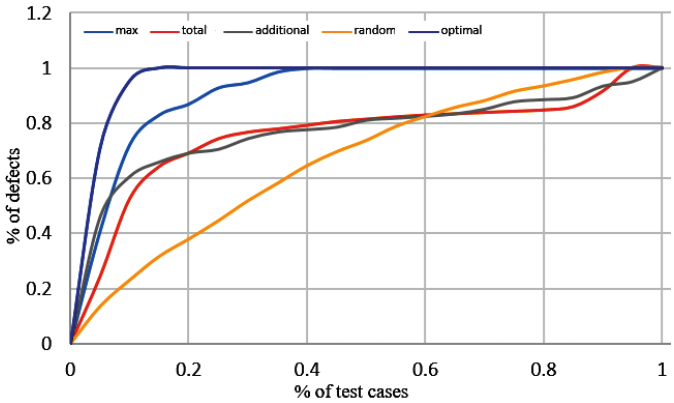
(c) GBRT

Fig. 6. Results on Flex dataset from different combinations of predictors and test prioritization strategies

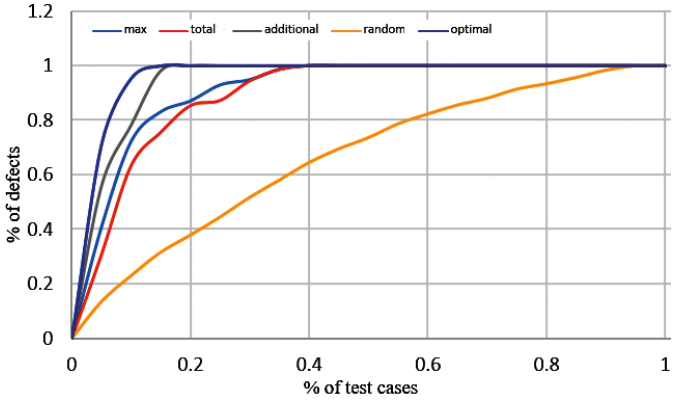
4.1. RQ1: What is the performance comparison of different prioritization strategies?

From the observation of Fig. 6-9 and Table 8, it can be seen that the performance of different prioritization strategies is different.

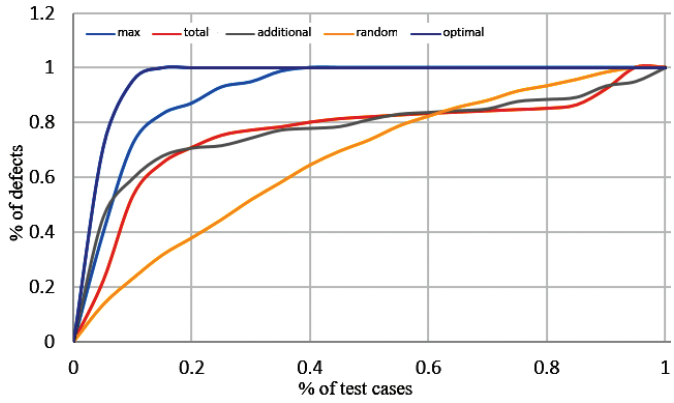
- The optimal strategy has the best performance, which is the upper theoretical limit of test case prioritization. The APFD values of the optimal strategy in *Gzip* and *Sed* are 0.933 and 0.948, respectively.



(a) LRM



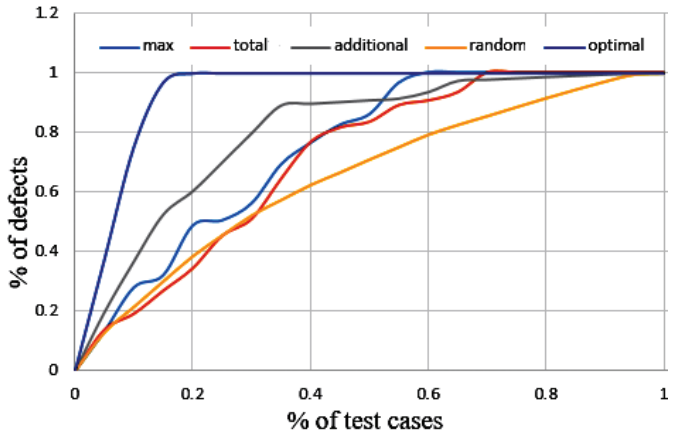
(b) RF



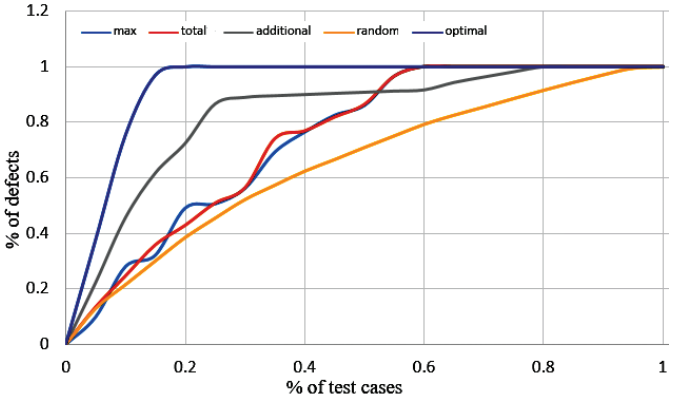
(c) GBRT

Fig. 7. Results on Grep dataset from different combinations of predictors and test prioritization strategies

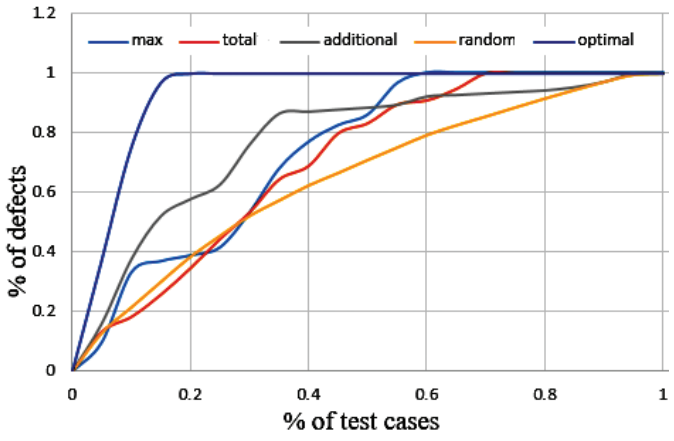
- The random strategy has the worst performance, which is caused by its indiscriminate random selection of test cases. Its APFD values in *Gzip*, *Grep*, *Flex* and *Sed* are 0.628, 0.65, 0.62 and 0.551, respectively, which are close to the theoretical values of 0.5. However, the actual result of the random strategy is higher than the theoretical value. This is because the corresponding relationship between the designed test cases and defects is not one-to-one, that is, a defect can be found by multiple test cases, which is closer to the actual test situation. This leads to an increased probability of finding defects in the test case set. Therefore, the probability of finding defects in the test case is higher than the theoretical value.
- The curve of the max strategy on the *Gzip* and *Sed* datasets is closer to that of the total (see Fig. 8 and Fig. 9), and it is closer to the additional strategy on the *Grep* and *Flex* datasets (see



(a) LRM

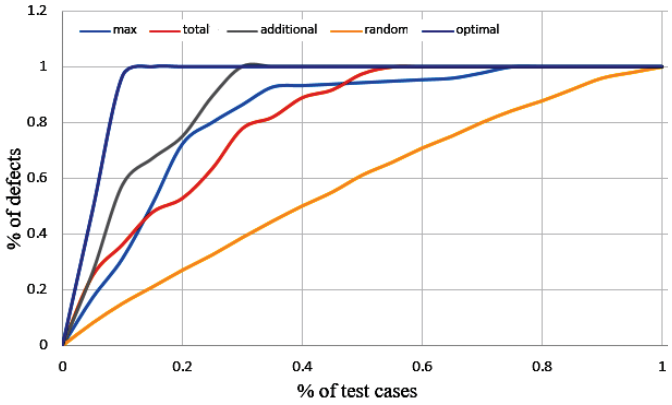


(b) RF

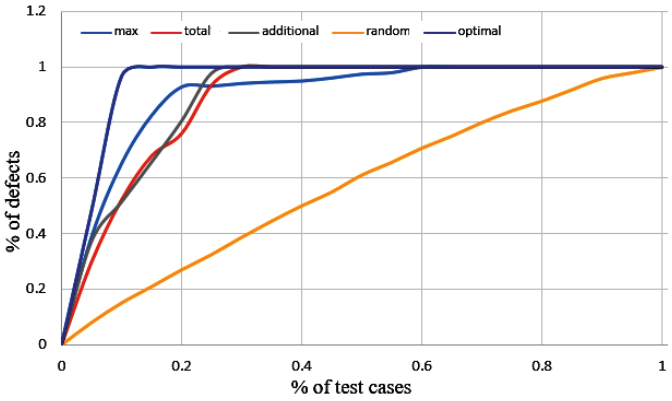


(c) GBRT

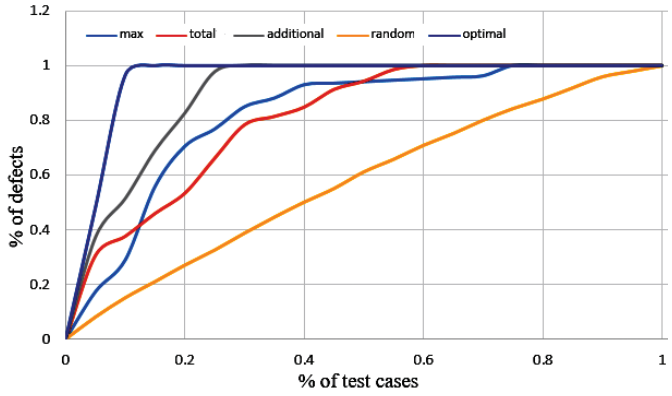
Fig. 8. Results on Gzip dataset from different combinations of predictors and test prioritization strategies



(a) LRM



(b) RF



(c) GBRT

Fig. 9. Results on Sed dataset from different combinations of predictors and test prioritization strategies

Fig. 7 and Fig. 6). The APFD values of the max strategy are generally higher than those of the total strategy (see Table 8). However, the starting curve of the total is higher than the max, which indicates that the total is actually better than the max in the early stage of testing. This is mainly because the total is on the overall coverage strategy. It is easier to prioritize test cases with wider coverage in the early stage so that the probability of finding defects is higher. However, it is prone to falling into the situation of repeatedly covering the tested code later in the testing. The max strategy takes into account the greatest probability of detecting defects in the code. Although it is not as good as the total strategy in the early stage of testing, it is easier to prioritize the execution of defective test cases in the later stage.

- The additional is a strategy with excellent performance, which is only slightly lower than the optimal strategy. Although the additional is not as good as the max on *Grep* and *Flex* datasets in some cases, the overall curve in the figures shows that the additional is worse than the max and total. This is because both the max and total are static strategies, while the additional is a dynamic one. After each test execution, the defect detection rate of the test case will be readjusted according to the coverage of the test code. This feedback method is helpful for the additional strategy to optimize its test case selection behaviour in real-time.
- In addition, compared with the random strategy, the test performance gain curves of the three TCP strategies max, total and

optimal are convexly shown in Fig. 10. As the proportion of test cases increases, the test performance difference tends to increase firstly and then decreased. Moreover, when 20%-30% of the test cases are selected, three test performance gain curves reach the maximum and the efficiency ratio is the largest.

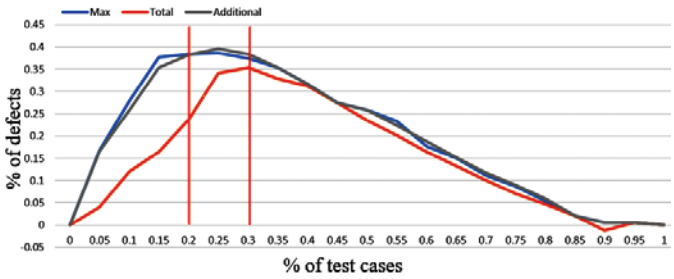


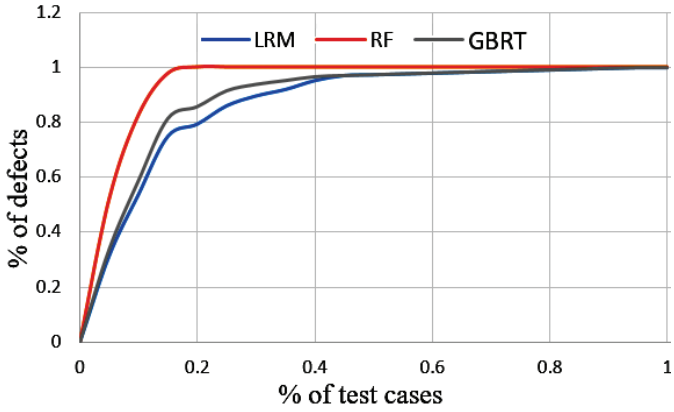
Fig. 10. The test performance gains of the three strategies max, total and optimal compared to the random strategy

Based on the above results, the following conclusions can be drawn: The proposed model is suitable for test case prioritization. In the test case prioritization methods, the APFD performance of the additional strategy is preferable to max strategy and total strategy.

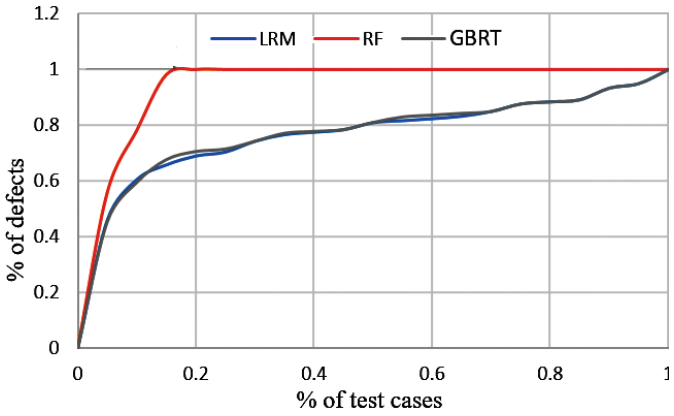
5.2. RQ2: How do different software defect predictors affect test case prioritization?

This problem explores the impact of different defect prediction classifiers on test results. It can be found that from Table 8 that different predictors have an impact on the additional, max and total strategies, while the optimal and random strategies will not be disturbed because they are not related to the predictor. For the convenience of analysis, this section analyses the performance of the three predictors on *Gzip*, *Grep*, *Flex* and *Sed* by taking the additional strategy as the object as illustrated in Figure 11.

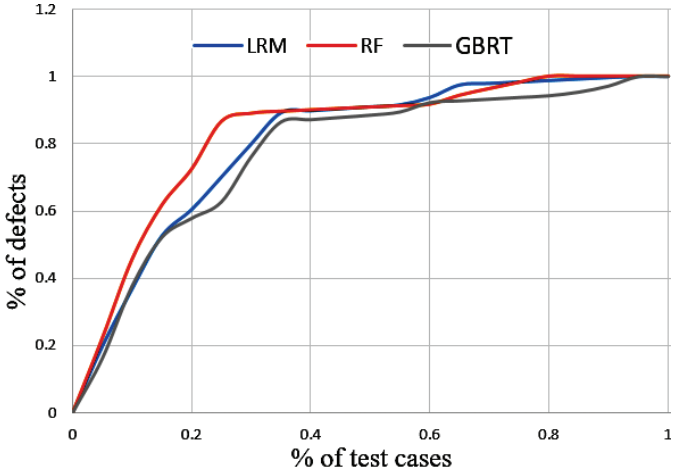
It can be seen from Fig. 11 that using the RF-based predictor for the additional test case prioritization strategy is better than using



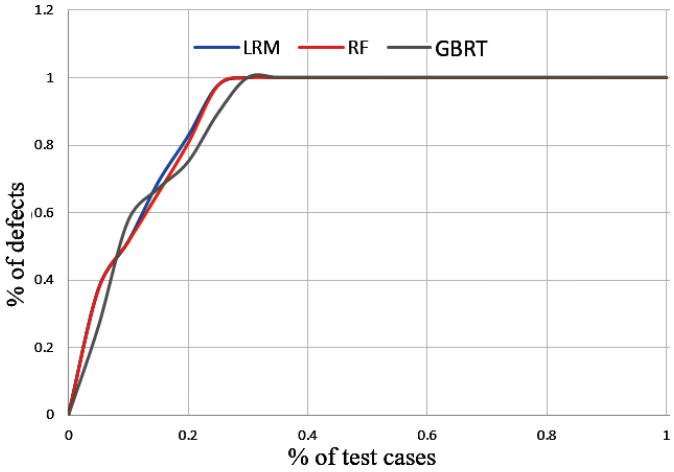
(a) Flex dataset



(b) Grep dataset



(c) Gzip dataset



(d) Sed dataset

Fig. 11. Impact of LRM, RF and GBRT predictors on the additional prioritization strategy

On the whole, the performance of the test case prioritization strategy is optimal, additional, max, total and random in turn from high to low. In order to quantitatively analyse the performance difference, the APFD average values under different classifiers and test case prioritization strategies are calculated as shown in Table 8. The average value is used as total, max, additional and optimal strategies are 0.7918 (-16.05%), 0.8451 (-10.72%), 0.8473 (-10.50%), and 0.9523 respectively. The percentiles in parentheses are the performance loss ratio relative to the optimal strategy. The smaller the loss ratio, the better the ranking.

the LRM-based or GBRT-based on *Gzip*, *Grep*, *Flex* and *Sed* datasets. Especially on the *Grep* dataset, the test performance is significantly higher than the other two predictors. This shows that the choice of the predictor has an influence on the test case prioritization based on code statement level defect prediction, and its degree of influence is also inconsistent with the different experimental objects.

Based on the above results, the following conclusions can be drawn: The prediction performance of the defect predictor will affect the efficiency of the test case prioritization method, that is, different

Table 8. APFD values under different prioritization and predictors

Dataset	prioritization	Machine learning algorithm		
		LRM	RF	GBRT
Gzip	Max	0.740	0.739	0.731
	Total	0.701	0.740	0.698
	Additional	0.794	0.821	0.766
	Random	0.628	0.628	0.628
	Optimal	0.933	0.933	0.933
Grep	Max	0.910	0.910	0.908
	Total	0.746	0.894	0.751
	Additional	0.746	0.941	0.766
	Random	0.650	0.650	0.650
	Optimal	0.964	0.964	0.964
Flex	Max	0.900	0.900	0.908
	Total	0.776	0.907	0.802
	Additional	0.855	0.943	0.874
	Random	0.620	0.620	0.620
	Optimal	0.964	0.964	0.964
Sed	Max	0.814	0.873	0.808
	Total	0.801	0.886	0.799
	Additional	0.880	0.889	0.892
	Random	0.551	0.551	0.551
	Optimal	0.948	0.948	0.948

defect predictors will bring different prediction probabilities to the test case prioritization.

5. Threats to validity

As an empirical study, its potential limitations must be taken into account when interpreting its results. This section describes several potential threats to the validity of our models.

5.1. Internal Validity

Internal validity mainly refers to the correctness and reproducibility of our empirical results. We implement all the baseline predictors (RF, LRM and GBRT) with default settings invoking WEKA to reduce the potential possibility to make mistakes. The optimal parameters of the predictors may be different for different defect datasets, which may lead to different results. However, it does not hinder the feasibility and effectiveness of the test case prioritization based on code statement-level software defect prediction. And all the algorithms involved adopting the same data pre-processing (e.g. CFS-based attribute selection [14]) to minimize redundancy. Besides, we have double-checked all of our experiments, but there may be a few errors.

5.2. External Validity

External validity relates to the generalization ability of our empirical results. The proposed approach was compared and analysed in four selected subjects, which may have data quality issues. And the attributes collected are all code statement metrics from SIR Repository and the samples are abstracted at code statement level from C programming language. Although our proposed model could be employed in other programming languages (i.e., Java, C++), we cannot guarantee the same empirical results. Besides, we only employed the three prevailing defect classifiers (RF, LRM, and GBRT). As we all know, there are a large number of predictive classifiers [6, 25, 47]. We could not validate all other algorithms due to time and space constraints. However, it does not dispute that choosing different predictors affects test case prioritization results. To reduce the external threats, more programming languages, high-quality defect datasets, and the predictors should be utilized in the future.

5.3. Construct Validity

Construct validity refers to the suitability of the test performance evaluation measure. There are some several measures [7], such as average severity of faults detected (ASFD), coverage effectiveness (CE), total percentage of faults detected (TPFD), average percentage faults detected (APFD). However, in fact, there is no studies have applied all of the measures to evaluate test case prioritization. We chose carefully the most commonly used measure APFD to prioritize test cases. Besides, in order to reduce construct validity, we also use the Alberg diagram to visually describe the curve of the proportion of found defects to the total number of defects as the proportion of test cases to the total number of test cases increases.

6. Conclusion

This paper proposes a novel test case prioritization method based on code statement-level defect prediction named TCP-SCSDP, which takes into account the possible distribution of defects and prediction granularity. The proposed feature set for measuring code statements first is used as the input for the statement software defect prediction model, and data pre-processing is performed on the software defect dataset. Secondly, the predictor is applied to predict the defect proneness probability of valid code statements. Then the defect detection rate of all test cases is calculated by using the test case prioritization strategy, and they are sorted from high to low. Finally, APFD is used to evaluate the prioritization.

Experimental results on 4 open source datasets show that the proposed approach is feasible and effective, and the test performance will be affected by the predictor and the test case prioritization strategy.

Our future work will focus on the following aspects: (1) Collect more open source software projects, programming languages (e.g. C++, Java) and predictors (e.g. neural network, k nearest neighbour), as mentioned earlier, to validate the generality of our method. (2) Optimize the test case prioritization strategy to improve software testing efficiency.

Acknowledgement

The research work is supported by a grant from the Science & Technology on Reliability & Environmental Engineering Laboratory of China (Grant No.614200404031117). And this work is also partially supported by No. 61400020404. We would be grateful to the editor and anonymous reviewers for their insightful and valuable comments and suggestions to improve the paper.

References

1. Arafeen M J, Do H. Test case prioritization using requirements-based clustering. Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation 2013: 312-321, https://doi.org/10.1109/ICST.2013.12.
2. Arar O F, Ayan K. A feature dependent naive bayes approach and its application to the software defect prediction problem. Applied Soft Computing 2017; 59: 197-209, https://doi.org/10.1016/j.asoc.2017.05.043.

3. Arar O F, Ayan K. Software defect prediction using cost-sensitive neural network. Applied Soft Computing 2015; 33(C): 263-277, https://doi.org/10.1016/j.asoc.2015.04.045.
4. Bertolino A, Miranda B, Pietrantuono R, et al. Adaptive coverage and operational profile-based testing for reliability improvement. Proceedings of the 39th International Conference on Software Engineering 2017: 541-551.
5. Boehm B, Basili V R. Software defect reduction top 10 list. Computer 2001; 34(1): 135-137, https://doi.org/10.1109/2.962984.
6. Catal C, Diri B. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. Information Sciences 2009; 179(8): 1040-1058, https://doi.org/10.1016/j.ins.2008.12.001.
7. Catal C, Mishra D. Test case prioritization: A systematic mapping study. Software Quality Journal 2013; 21(3): 445-478, https://doi.org/10.1007/s11219-012-9181-z.
8. Di Nardo D, Alshahwan N, Briand L, et al. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. Software Testing, Verification and Reliability 2015; 25(4): 371-396, https://doi.org/10.1002/stvr.1572.
9. Do H, Mirarab S, Tahvildari L, et al. The effects of time constraints on test case prioritization: A series of controlled experiments. IEEE Transactions on Software Engineering 2010; 36(5): 593-617, https://doi.org/10.1109/TSE.2010.58.
10. Durelli V H S, Durelli R S, Borges S S, et al. Machine learning applied to software testing: A systematic mapping study. IEEE Transactions on Reliability 2019; 68(3): 1189-1212, https://doi.org/10.1109/TR.2019.2892517.
11. Elbaum S, Malishevsky A G, Rothermel G. Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering 2002; 28(2): 159-182, https://doi.org/10.1109/32.988497.
12. Friedman J H. Stochastic gradient boosting. Computational statistics & data analysis 2002; 38(4): 367-378.
13. Garg D, Datta A. Test case prioritization due to database changes in web applications. Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation 2012: 726-730.
14. Hall M, Frank E, Holmes G, et al. The weka data mining software: An update. ACM SIGKDD explorations newsletter 2009; 11(1): 10-18.
15. Harrold M J. Testing: A roadmap. Proceedings of the Future of Software Engineering 2000: 61-72.
16. He P, Li B, Liu X, et al. An empirical study on software defect prediction with a simplified metric set. Information and Software Technology 2015; 59(C): 170-190, https://doi.org/10.1016/j.infsof.2014.11.006.
17. Henard C, Papadakis M, Harman M, et al. Comparing white-box and black-box test prioritization. Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) 2016: 523-534, https://doi.org/10.1145/2884781.2884791.
18. Hosseini S, Turhan B, Mäntylä M. A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. Information and Software Technology 2017; 95: 296-312, https://doi.org/10.1016/j.infsof.2017.06.004.
19. Hovemeyer D, Pugh W. Finding bugs is easy. Acm sigplan notices 2004; 39(12): 92-106, https://doi.org/10.1145/1052883.1052895.
20. Huang C Y, Chang J R, Chang Y H. Design and analysis of gui test-case prioritization using weight-based methods. Journal of Systems and Software 2010; 83(4): 646-659, https://doi.org/10.1016/j.jss.2009.11.703.
21. Khatibsyarbini M, Isa M A, Jawawi D N A, et al. Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology 2018; 93: 74-93, https://doi.org/ 10.1016/j.infsof.2017.08.014.
22. Krishnamoorthi R, Sahaaya Arul Mary S A. Requirement based system test case prioritization of new and regression test cases. International Journal of Software Engineering and Knowledge Engineering 2009; 19(3): 453-475, https://doi.org/10.1142/S0218194009004222.
23. Lachmann R, Schulze S, Nieke M, et al. System-level test case prioritization using machine learning. Proceedings of the 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA) 2016: 361-368, https://doi.org/10.1109/ICMLA.2016.163.
24. Laradji I H, Alshayeb M, Ghouti L. Software defect prediction using ensemble learning on selected features. Information and Software Technology 2015; 58: 388-402, https://doi.org/10.1016/j.infsof.2014.07.005.
25. Lessmann S, Baesens B, Mues C, et al. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering 2008; 34(4): 485-496, https://doi.org/10.1109/TSE.2008.35.
26. Li Z, Harman M, Hierons R M. Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering 2007; 33(4): 225-237, https://doi.org/10.1109/TSE.2007.38.
27. Mei H, Hao D, Zhang L, et al. A static approach to prioritizing junit test cases. IEEE Transactions on Software Engineering 2012; 38(6): 1258-1275, https://doi.org/10.1109/TSE.2011.106.
28. Menzies T, Greenwald J, Frank A. Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering 2006; 33(1): 2-13, https://doi.org/10.1109/TSE.2007.256941.
29. Mirarab S, Tahvildari L. A prioritization approach for software test cases based on bayesian networks. Proceedings of the International Conference on Fundamental Approaches to Software Engineering 2007: 279-290.
30. Nam J, Kim S. Clami: Defect prediction on unlabeled datasets. Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2015: 452-463, https://doi.org/10.1109/ASE.2015.56.
31. Ohlsson N, Alberg H. Predicting fault-prone software modules in telephone switches. IEEE Transactions on Software Engineering 1996; 22(12): 886-894, https://doi.org/10.1109/32.553637.
32. Paterson D, Campos J, Abreu R, et al. An empirical study on the use of defect prediction for test case prioritization. Proceedings of the 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST) 2019: 346-357, https://doi.org/10.1109/ICST.2019.00041.
33. Peng R, Zhai Q. Modeling of software fault detection and correction processes with fault dependency. Eksploatacja i Niezawodnosc – Maintenance and Reliability 2017; 19(3): 467-475, https://doi.org/10.17531/ein.2017.3.18.
34. Peng X, Liu B, Wang S. Feedback-based integrated prediction: Defect prediction based on feedback from software testing process. Journal of Systems and Software 2018; 143: 159–171, https://doi.org/10.1016/j.jss.2018.05.029.
35. Rothermel G, Untch R H, Chu C, et al. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering 2001; 27(10): 929-948, https://doi.org/10.1109/32.962562.
36. Rothermel G, Untch R H, Chu C, et al. Test case prioritization: An empirical study. Proceedings of the IEEE International Conference on Software Maintenance-1999 (ICSM'99)'Software Maintenance for Business Change'(Cat No 99CB36360) 1999: 179-188.
37. Shao Y, Liu B, Wang S, et al. A novel software defect prediction based on atomic class-association rule mining. Expert Systems with Applications 2018; 114: 237-254, https://doi.org/10.1016/j.eswa.2018.07.042.

38. Shatnawi O. Measuring commercial software operational reliability: An interdisciplinary modelling approach. Eksploatacja i Niezawodnosc – Maintenance and Reliability, 2014; 16(4): 585-594.
39. Song Q, Jia Z, Shepperd M, et al. A general software defect-proneness prediction framework. IEEE Transactions on Software Engineering 2011; 37(3): 356-370, https://doi.org/10.1109/TSE.2010.90.
40. Spieker H, Gotlieb A, Marijan D, et al. Reinforcement learning for automatic test case prioritization and selection in continuous integration. Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis 2017: 12-22, https://doi.org/10.1145/3092703.3092709.
41. Srikanth H, Hettiarachchi C, Do H. Requirements based test prioritization using risk factors: An industrial study. Information and Software Technology 2016; 69: 71-83, https://doi.org/10.1016/j.infsof.2015.09.002.
42. Sun Z, Song Q, Zhu X. Using coding-based ensemble learning to improve software defect prediction. IEEE Transactions on Systems Man and Cybernetics Part C 2012; 42(6): 1806-1817, https://doi.org/10.1109/TSMCC.2012.2226152.
43. Tonella P, Avesani P, Susi A. Using the case-based ranking methodology for test case prioritization. Proceedings of the 2006 22nd IEEE International Conference on Software Maintenance 2006: 123-133.
44. Turhan B, Menzies T, Bener A c, ,e B., et al. On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 2009; 14(5): 540-578, https://doi.org/10.1007/s10664-008-9103-7.
45. Wang H, Khoshgoftaar T M, Napolitano A. Software measurement data reduction using ensemble techniques. Neurocomputing 2012; 92(3): 124-132, https://doi.org/10.1016/j.neucom.2011.08.040.
46. Wang S, Nam J, Tan L. Qtep: Quality-aware test case prioritization. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering 2017: 523-534, https://doi.org/10.1145/3106237.3106258.
47. Wang S, Yao X. Using class imbalance learning for software defect prediction. IEEE Transactions on Reliability 2013; 62(2): 434-443, https://doi.org/10.1109/TR.2013.2259203.
48. Xiao L, Miao H, Zhuang W, et al. An empirical study on clustering approach combining fault prediction for test case prioritization. Proceedings of the 2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS) 2017: 815-820.
49. Xu J, Meng Z, Xu L. Integrated system of health management-oriented reliability prediction for a spacecraft software system with an adaptive genetic algorithm support vector machine. Eksploatacja i Niezawodnosc – Maintenance and Reliability 2014; 16(4): 571-578.
50. Yang X, Tang K, Yao X. A learning-to-rank approach to software defect prediction. IEEE Transactions on Reliability 2015; 64(1): 234-246, https://doi.org/10.1109/TR.2014.2370891.
51. Yoo S, Harman M. Regression testing minimization, selection and prioritization: A survey. Software Testing, Verification and Reliability 2012; 22(2): 67-120, https://doi.org/10.1002/stv.430.

Yuanxun SHAO
Bin LIU
Shihai WANG
School of Reliability and Systems Engineering
Beihang University
No.37 Xueyuan RD. Haidian, 100191, Beijing, China

Peng XIAO
Ji'an Municipal Industry and Information Technology Bureau
11/F, Block B, Administration Center Building
Jizhou District, Ji'an, Jiangxi, 343000, China

E-mails: yuanxunshao@buaa.edu.cn, liubin@buaa.edu.cn, wangshihai@buaa.edu.cn, buaaxp@foxmail.com