mathematics

MDPI

Article

A Model for Predicting Statement Mutation Scores

Lili Tan, Yunzhan Gong and Yawen Wang *

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China

* Correspondence: wangyawen@bupt.edu.cn

Received: 13 June 2019; Accepted: 15 August 2019; Published: 23 August 2019

Abstract: A test suite plays a key role in software testing. Mutation testing is a powerful approach to measure the fault-detection ability of a test suite. The mutation testing process requires a large number of mutants to be generated and executed. Hence, mutation testing is also computationally expensive. To solve this problem, predictive mutation testing builds a classification model to predict the test result of each mutant. However, the existing predictive mutation testing methods only can be used to estimate the overall mutation scores of object-oriented programs. To overcome the shortcomings of the existing methods, we propose a new method to directly predict the mutation score for each statement in process-oriented programs. Compared with the existing predictive mutation testing methods, our method uses more dynamic program execution features, which more adequately reflect dynamic dependency relationships among the statements and more accurately reflects information propagation during the execution of test cases. By comparing the prediction effects of logistic regression, artificial neural network, random forest, support vector machine, and symbolic regression, we finally decide to use a single hidden layer feedforward neural network as the predictive model to predict the statement mutation scores. In our two experiments, the mean absolute errors between the statement mutation scores predicted by the neural network and the real statement mutation scores both approximately reach 0.12.

Keywords: software testing; machine learning; mutation testing

1. Introduction

When a programmer writes a program, a mistake may occur in the code. For example, a programmer may incorrectly write x=x-1 as x=x+1, x=x*1, x=x%1, etc. This mistake is referred to as a software fault (i.e., a software bug). When this fault is executed, an incorrect execution result may appear on the corresponding statement. This incorrect execution result often is referred to as a software error and cannot be directly observed. When this software error propagates to an observable program output, a software failure occurs.

A strong-power test suite may detect more software faults than a weak-power one, thus measuring the fault detection capability of a test suite is an important question in software testing. Mutation testing is an approach to determine the effectiveness of a test suite [1–3].

The programs with software faults are called mutants. In mutation testing, mutants are generated through automatically changing the original program with mutation operators, where each mutation operator is a rule and can be applied to program statements to produce the program version with a software fault. A mutant is said to be identified by a test suite if at least one test case from the test suite has different execution results on the mutant and the original program. Mutation score, which is the ratio of all identified mutants to all mutants, has been widely used to assess the adequacy of a test suite.

Although mutation testing is obviously useful, it is extremely expensive [4,5]. For example, using 108 mutation operators, Proteum [6] generates 4937 mutants for tcas, which is the smallest program

among the Siemens programs and contains only 137 non-commenting and non-whitespace lines of code. Thus, testing a large number of mutants can be a big burden.

For solving this problem, researchers have proposed some optimization methods to reduce the cost of mutation testing, such as random mutation [7,8], mutant clustering [9] and selective mutation [10,11]. For quickly calculating the mutation score of the whole program, these methods attempt to use a mutant sample to represent all mutants. Random mutation randomly chooses some mutants from all mutants to construct mutation samples. A mutant clustering algorithm first classifies all mutants into different clusters so that the mutants in a cluster have similar identification difficulties, and then selects a small number of mutants from each cluster to construct the mutant sample. Selective mutation uses only a subset of mutation operators to generate a mutant sample.

Different from the above mutant reduction methods, the predictive mutation testing methods [12,13] have been proposed in recent years. The predictive mutation testing methods extract some features related to program structures and testing processes and apply machine learning to predict each mutant's test result (i.e., the identification result). Moreover, these predictive methods' execution time is short. However, the existing predictive mutation testing methods are all designed for object-oriented programs. The same as other methods, the existing predictive mutation testing methods are also mainly used for estimating the mutation score of the whole program. The main differences among the above mutant reduction methods can be shown in Table 1.

Table 1. Main differences among mutation reduction methods.

Method	Key Technology	Time Cost	Target
random mutation	simple random sampling	low	estimating program mutation score
mutant clustering	stratified sampling	low	estimating program mutation score
selective mutation	non-probability sampling	high	estimating program mutation score
predictive mutation	supervised learning	low	estimating program mutation score classifying mutants

To make up for the shortcomings of existing predictive mutation testing methods, based on the execution impact map [14] Goradia uses, we suggest a new predictive method. This new method is not only suitable for procedure-oriented programs but also can use a single hidden layer feedforward neural network and seven statement features to predict the mutation score of each program statement.

The prediction of the statement mutation scores includes two major phases: extracting the statement features and determining the mathematical form of predictive model. In the feature extraction phase, we obtain the following seven features to express the effect of a statement on the program outputs: number of executions, path impact factor, value impact factor, generalized path impact factor, generalized value impact factor, latent impact factor, and information hidden factor. In fact, among the above seven features, only a number of executions are adopted by existing predictive mutation testing methods. Compared with the existing predictive mutation testing methods, our method more accurately expresses information propagation among the statements. For a statement, except for the number of executions, its six other features are extracted from the following six aspects respectively:

When a test case executes on the statement containing a software fault , an error may be generated. This error either propagates along the original execution path or changes the original execution path.

(1) The fault in the statement may change the program output by generating the errors that propagate along the original execution paths. From this aspect, we extract the statement's value impact factor.

(2) The fault in the statement may change the program outputs by generating the errors altering the original execution paths. From this aspect, we extract the statement's path impact factor.

However, in a few cases, the change of execution path does not result in a change of program output. Therefore, we need to analyze further the features of the changed program branch in order to more accurately predict how likely the program output will be changed.

(3) The no longer executed branches lose their ability to pass their information along the original execution path to the program outputs. The loss of this capability may cause the program output to be changed. From this aspect, we extract the statement's generalized value impact factor.

(4) The no longer executed program branch is no longer able to influence the selection of subsequent program branches. Loss of this ability may also impact the program output. From this aspect, we extract the statement's generalized path impact factor.

(5) The fault in a statement may cause some program branches, which has not been executed, will be executed. Executing these new branches may cause the program output to change. From this aspect, we extract the statement's latent impact factor.

(6) Sometimes, the program under testing has multiple output statements, some of which happen to have the same output values. In this case, even if the software fault changes the execution path of the test case, the program outputs could still be the same. From this aspect, we extract a statement's information hidden factor.

Among these six factors, the first five factors facilitate program output changes, and the last one prevents program output from changing.

In the phase of determining mathematical form of the predictive model, we compared the following five machine learning models based on Brier scores: artificial neural network (ANN), logical regression (LR), random forest (RF), support vector machine (SVM) and symbolic regression (SR). From the experiment results, the artificial neural networks were identified as the most suitable predictive model.

With the methods in this article, we analyzed the two programs. In the two experiments, the mean absolute errors between the real statement mutation scores and predictive statement mutation scores are 0.1205 and 0.1198, respectively.

The remainder of this paper is organized as below: in Section 2, we introduce some basic terms used throughout the entire paper. In Section 3, we define seven statement features. In Section 4, we propose a method for quickly calculating statement features. In Section 5, we compare the prediction accuracy of five machine learning models. In Section 6, we introduce the structure of our automated prediction tool. In Section 7, we describe the work to be performed.

2. Basic Terms

Definition 1. *Original program and mutation score.*

In this paper, a program without any software fault is also called an original program. For example, Program 1 is an original program. It first outputs the factorial of the absolute value of the difference between m and n, and then classifies the factorial. Based on the relationships among m, n and the factorial, the execution results of the program are divided into three areas, the first and third of which belong to the first class, and the second of which belongs to the second class.

A program with software faults is called a mutant. In mutation testing, mutants are generated through automatically changing the original program with mutation operators. For example, in terms of Program 1, if the statement dist=m-n is changed into dist=m%n, then the mutant m_1 is generated as shown in Program 2. If a test suite (i.e., a collection of test cases) can identify the mutant m_1, it must satisfy the following conditions: there must be at least one test case in the test suite to execute the statement dist=m%n in m_1, the execution result of dist=m%n must be different from that of dist=m-n, and the difference must be propagated to the program output.

Program mutation score is the proportion of identified mutants in a program, which is used to assess how well the program is tested by the test suite. Statement mutation score is the the proportion

of identified mutants in a statement, which is used to assess how well the statement is tested by the test suite.

Definition 2. *Program statement and branch.*

In this article, we predict the ability of a test suite to test each line program code. A statement in the program under testing usually occupies one line. Because a control expression usually occupies a line in the program, in this paper, we also think of a controlling expression as a statement. As shown in Program 1, we denote gth statement as s_g. According to C programming language standard—C99 [15], a controlling expression can occur in "if", "switch", "while", "do while" and "for" statements and decides which of the program branches is executed.

In terms of if-else statement, if its controlling expression appears in the rth line, then we denoted its controlling expression as s_r, and use $B_{r,t}$ and $B_{r,f}$ to denote the true branch and false branch of s_r, respectively. In terms of a loop statement (such as while loop, do-while loop and for loop), we regard it as the combination of the controlling expression and the corresponding program branch. If a loop statement's controlling expression appears in the rth line, then its controlling expression is denoted as s_r, and the corresponding loop body is considered as the true branch of s_r, so that this loop body can also be denoted as $B_{r,t}$. According to this representation method, the program branch whose function is to exit the loop is denoted as $B_{r,f}$.

Program 1: An original program.

	#include <stdio.h>
	typedef int bool ;
	void fun(int m, int n) {
	int dist, fac ;
s_1	if(m>n)
s_2	dist=m-n ;
	else
s_3	dist=n-m;
s_4	fac=1;
s_5	while (dist>1) { // Loop for factorial
s_6	fac=fac ∗ dist ;
s_7	dist = dist -1 ;
	}
s_8	printf ("fac=%d \n", fac);
s_9	if (m<n) // classify the factorial
s_{10}	printf ("class 1 \n") ;
s_{11}	else if (fac<5)
s_{12}	printf ("class 2 \n") ;
	else
s_{13}	printf ("class 1 \n") ;
	}

Program 2: The mutant m_1 of Program 1.

	#include <stdio.h>
	typedef int bool ;
	void fun(int m, int n) {
	int dist, fac ;
s_1	if(m>n)
s_2	dist=m%n ;
	else
s_3	dist=n-m;
s_4	fac=1;
s_5	while (dist>1) { // Loop for factorial
s_6	fac=fac ∗ dist ;
s_7	dist = dist -1 ;
	}
s_8	printf (“fac=%d \n”, fac);
s_9	if (m<n) // classify the factorial
s_{10}	printf (“class 1 \n”) ;
s_{11}	else if (fac<5)
s_{12}	printf (“class 2 \n”) ;
	else
s_{13}	printf (“class 1 \n”) ;
	}

For example, in Program 1, s_9 is the controlling expression, the statement s_{10} constitutes its true branch $B_{9,t}$, and the statements s_{11}, s_{12} and s_{13} constitute its false branch $B_{9,f}$. The statements s_6 and s_7 constitute the loop body of the while loop, and, in this situation, the loop body is also considered as the true branch $B_{5,t}$ of the controlling expression s_5.

Definition 3. *Statement instance and branch instance.*

A statement may be executed multiple times by a test suite, so that multiple execution instances are generated. The statement's each execution instance is called its a statement instance. The hth execution instance of test case t_k on statement s_g is denoted as s^h_{g,t_k}. In this paper, the execution instance of a program output statement is called an *output statement instance*. In addition, the execution instance of a controlling expression is also considered as a special statement instance, and is called a *controlling expression instance*.

For example, when Program 1 is executed by test case $t_1(m = 4, n = 1)$, the assignment statement s_4, controlling expression s_5, controlling expression s_9, controlling expression s_{11} and output statement s_{13} are executed once, three times, once, once and once. This allows them to produce one, three, one, one and one execution instance, respectively, during the execution of the test case t_1. Among them, the controlling expression instances s^1_{5,t_1}, s^2_{5,t_1} and s^3_{5,t_1}, respectively, represent the first, second and third executions of the test case t_1 on the statement s_5.

A program branch may also be executed multiple times, so that many execution instances are generated. Each execution instance of the program branch is called *a branch instance*. Just as a

program branch consists of many statements, a branch instance consists of many statement instances. These statement instances are called the *statement instances in the branch instance*. A bit similar to the symbols of statement instances, we use B^{l}_{r,z,t_k} to represent the lth execution instance of the test case t_k on the program branch $B_{r,z}$, where z represents the true or the false branch, and its value is t or f. Whether $B_{r,z}$ is executed depends on the execution result of the controlling expression s_r.

For example, the branch instance B^1_{9,t,t_1} consists of s^1_{10,t_1}, and the branch instance B^1_{9,f,t_1} consists of s^1_{11,t_1}, s^1_{12,t_1} and s^1_{13,t_1}. In terms of the while statement in Program 1, s_5 is a controlling expression and generates three execution instances s^1_{5,t_1}, s^2_{5,t_1} and s^3_{5,t_1} during the execution of the test case t_1. Because the execution of B^1_{5,t,t_1} is the necessary condition for B^2_{5,t,t_1} to be executed, B^2_{5,t,t_1} is contained in B^1_{5,t,t_1}. As shown in Table 2, Figures 1 and 2, the first branch instance B^1_{5,t,t_1} of the while loop consists of the statement instances s^1_{6,t_1}, s^1_{7,t_1}, s^2_{5,t_1}, s^2_{6,t_1}, s^2_{7,t_1} and s^3_{5,t_1}, and the second branch instance B^2_{5,t,t_1} consists of the statement instances s^2_{6,t_1}, s^2_{7,t_1} and s^3_{5,t_1}.

Table 2. The execution history of the test cases.

Test Case	Program Output	Execution History	Branch Instances in Loop
m = 4, n = 1	fac = 6, class 1 (s_{13})	$H_1: s^1_{1,t_1}, s^1_{2,t_1}, s^1_{4,t_1}, s^1_{5,t_1}, s^1_{6,t_1}, s^1_{7,t_1}, s^2_{5,t_1}, s^2_{6,t_1}, s^2_{7,t_1}, s^3_{5,t_1}, s^1_{8,t_1}, s^1_{9,t_1}, s^1_{11,t_1}, s^1_{13,t_1}$	$B^1_{5,t,t_1} = \{s^1_{6,t_1}, s^1_{7,t_1}, s^2_{5,t_1}, s^2_{6,t_1}, s^2_{7,t_1}, s^3_{5,t_1}\}$, $B^2_{5,t,t_1} = \{s^2_{6,t_1}, s^2_{7,t_1}, s^3_{5,t_1}\}$
m = 2, n = 2	fac = 1,class 2 (s_{12})	$H_2: s^1_{1,t_2}, s^1_{3,t_2}, s^1_{4,t_2}, s^1_{5,t_2}, s^1_{8,t_2}, s^1_{9,t_2}, s^1_{11,t_2}, s^1_{12,t_2}$	
m = 1, n = 4	fac = 6, class 1 (s_{10})	$H_3: s^1_{1,t_3}, s^1_{3,t_3}, s^1_{4,t_3}, s^1_{5,t_3}, s^1_{6,t_3}, s^1_{7,t_3}, s^2_{5,t_3}, s^2_{6,t_3}, s^2_{7,t_3}, s^3_{5,t_3}, s^1_{8,t_3}, s^1_{9,t_3}, s^1_{10,t_3}$	$B^1_{5,t,t_1} = \{s^1_{6,t_1}, s^1_{7,t_3}, s^2_{5,t_3}, s^2_{6,t_3}, s^2_{7,t_3}, s^3_{5,t_3}\}$, $B^2_{5,t,t_3} = \{s^2_{6,t_3}, s^2_{7,t_3}, s^3_{5,t_3}\}$

Definition 4. *Original execution path of the test case.*

The execution history H_k of the test case t_k is formed when the test case t_k executes on an original program. The execution history H_k is an execution trace, each element of which is a statement instance. These statement instances are ordered by time until the last program output. In this paper, the execution history H_k of the test case t_k is also called the original execution path of t_k.

For example, consider the Program 1, where test case t_1 (m = 4, n = 1), test case t_2 (m = 2, n = 2), and test case t_3 (m = 1, n = 4) constitutes the test suite T. As shown in Table 2, when t_1 is executed, H_1 is generated, and the program outputs fac = 6 and class 1. When t_2 is executed, H_2 is generated, and the program outputs fac = 1 and class 2. When t_3 is executed, H_3 is generated and the program outputs fac = 6 and class 1.

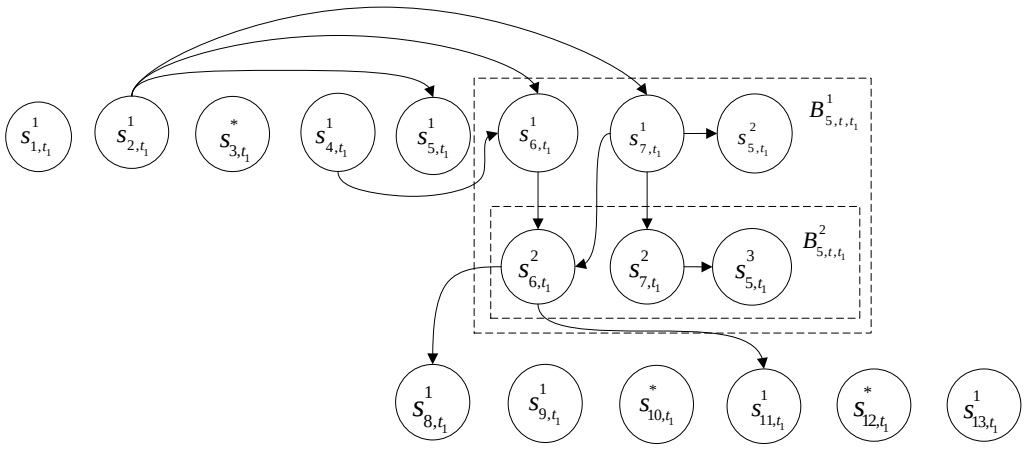

Figure 1. The execution impact graph G_1 formed when Program 1 is executed by test case 1.

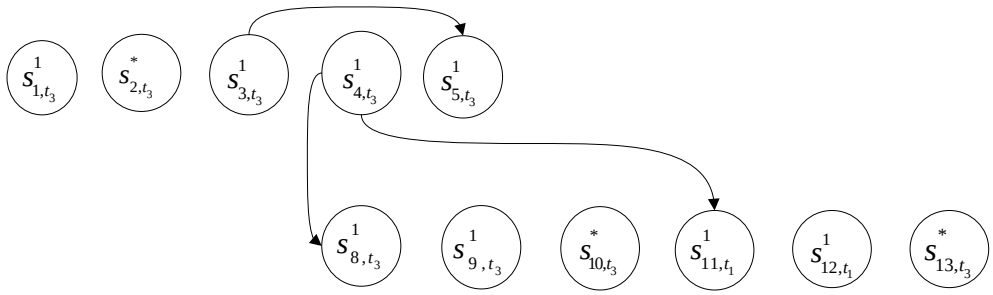

Figure 2. The execution impact graph G_2 formed when Program 1 is executed by test case 2.

Definition 5. *Execution impact graph*

An execution impact graph G_k is formed when the test case t_k executes. The execution impact graph G_k consists of multiple impact arcs generally, and each impact arc expresses the information propagation between the statement instances. In terms of an impact arc, the arc tail s^j_{i,t_k} is called a direct impact predecessor, and the arc head s^h_{g,t_k} is called a direct impact successor. In the practical application, if a variable is assigned in the statement instance s^j_{i,t_k} and is directly used at the statement instance s^h_{g,t_k}, then s^j_{i,t_k} is a direct impact predecessor of s^h_{g,t_k}, and s^h_{g,t_k} is a direct impact successor of s^j_{i,t_k}. In the execution impact graph G_k, each node is expressed in the form of s^j_{i,t_k} or s^*_{i,t_k}, where s^j_{i,t_k} denotes a statement instance and the symbol $*$ indicates that the statement s_i is not executed by test case t_k.

For example, when program 1 is executed by test cases 1, 2, and 3, the corresponding execution impact graphs are generated respectively, as shown in Figures 1, 2, and 3. In Program 1, the variable dist is defined in the statement s_2 and is directly used in the statements s_5, s_6 and s_7. Hence, when the test case t_1 is executed, s^1_{2,t_1} becomes the direct impact predecessor of s^1_{5,t_1}, s^1_{6,t_1} and s^1_{7,t_1}, respectively. In this situation, s^1_{5,t_1}, s^1_{6,t_1} and s^1_{7,t_1} become the direct impact successors of s^1_{2,t_1}.

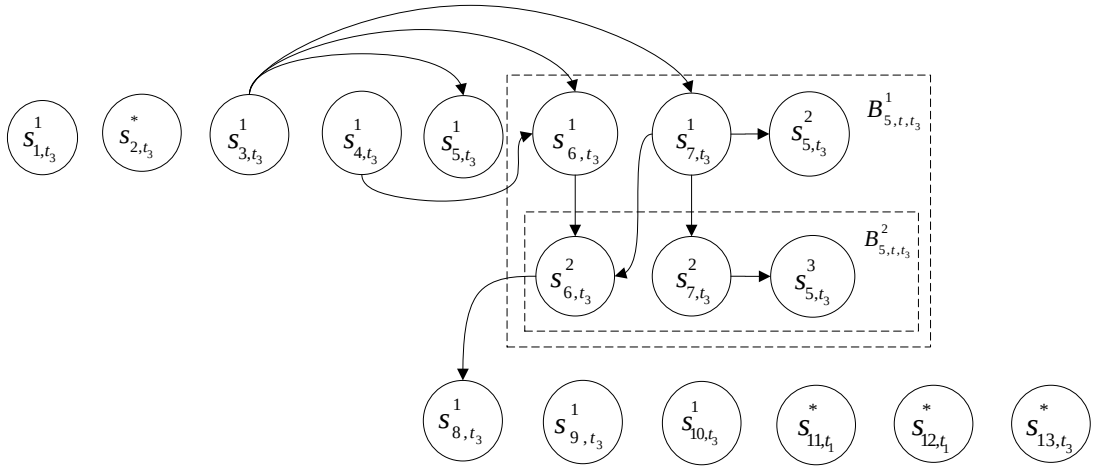

Figure 3. The execution impact graph G_3 formed when Program 1 is executed by test case 3.

Each direct impact successor of a statement instance may have its own direct impact successor. Thus, the impact successor is transitive. If a statement instance is the impact successor of the statement instance s^h_{g,t_k} but it is not the direct impact successor of s^h_{g,t_k}, then this statement instance is called the indirect impact successor of s^h_{g,t_k}. Thus, the impact successor can be divided into two types: the direct impact successor and the indirect successor.

For example, s^1_{5,t_1}, s^1_{6,t_1}, s^1_{7,t_1}, s^2_{5,t_1}, s^2_{6,t_1}, s^2_{7,t_1}, s^3_{5,t_1}, s^1_{8,t_1} and s^1_{11,t_1} are all the impact successors of s^1_{2,t_1}. However, s^1_{5,t_1}, s^1_{6,t_1} and s^1_{7,t_1} are the direct impact successors of s^1_{2,t_1}, and s^2_{5,t_1}, s^2_{6,t_1}, s^2_{7,t_1}, s^3_{5,t_1}, s^1_{8,t_1} and s^1_{11,t_1} are the indirect impact successors of s^1_{2,t_1}.

If there is a fault f_g in statement s_g, and s^h_{g,t_k} is an execution instance of statement s_g, then f_g may change the execution result of s^h_{g,t_k} during the execution of the test case t_k. If this change happens, we say that a error e^h_{g,t_k} *is generated from the statement instance* s^h_{g,t_k}. In this paper, an error is different from a fault . Errors are dynamic and are generated in the process of the test case execution. However, faults are static. Whether the program under testing is executed or not, they may exist in the program under testing.

3. Formal Definitions of Statement Features

In this section, we propose the seven features of a statement. The most of them are related to execution paths of test cases. When the statement containing a software fault is executed by a test case, an error may generate. After this error generates, it either propagates along the original execution path of the test case or changes the original execution path. The value impact factor describes the ability of the fault existing in a statement to affect the program output under the condition that the execution path is unchanged. The path impact factor, the generalized value impact factor, the generalized path impact factor and the latent impact factor describe the abilities of the fault existing in a statement to affect the program output under the condition that the execution path is changed by the generated error.

3.1. Value Impact Factor

The value impact factor of a statement expresses its ability to directly impact the program outputs along the execution paths of the test cases.

3.1.1. Value Impact Factor of Statement

The errors generated from the statement instance s^h_{g,t_k} may propagate along the original execution path H_k to some execution instances of the output statements. Each of these output statement instances is called the *value impact element of the statement instance* s^h_{g,t_k}. The collection consisting of all value impact elements of s^h_{g,t_k} is called *value impact set of the statement instance* s^h_{g,t_k}, and denoted as V^h_{g,t_k}.

A statement s_g has multiple execution instances generally and each execution instance has its own value impact set. The union of these value impact sets is called the *value impact set of* s_g, and is denoted as V_g. The element in V_g is called the value impact element of s_g. The number of value impact elements of s_g is called the value impact factor of s_g, and is denoted as $x_{vi}(s_g)$. Therefore, the following formula holds:

$$V_g = \bigcup_{k=1,2,\cdots,K} \bigcup_{h=1,2,\cdots,H_{gk}} V^h_{g,t_k}, \tag{1}$$

where K is the total number of test cases in the test suite, and H_{gk} is the total number of times the statement s_g is executed by the test case t_k.

Example 1. *From Table 2, we know that the statement* s_6 *has four execution instances* s^1_{6,t_1}, s^2_{6,t_1}, s^1_{6,t_3} *and* s^2_{6,t_3}. *If* s_6 *includes a fault, then each execution instance of* s_6 *may generate an error. The errors generated from* s^1_{6,t_1} *and* s^2_{6,t_1} *may propagate along the original execution path* H_1 *to the output statement instance* s^1_{8,t_1}. *Therefore,* $V^1_{6,t_1} = V^2_{6,t_1} = \{s^1_{8,t_1}\}$. *The errors generated from* s^1_{6,t_3} *and* s^2_{6,t_3} *may propagate along the original execution path* H_3 *to the output statement instance* s^1_{8,t_3}. *Therefore,* $V^1_{6,t_3} = V^2_{6,t_3} = \{s^1_{8,t_3}\}$. *According to Formula (1), we have* $V_6 = V^1_{6,t_1} \bigcup V^2_{6,t_1} \bigcup V^1_{6,t_3} \bigcup V^2_{6,t_3} = \{s^1_{8,t_1}, s^1_{8,t_3}\}$.

3.1.2. The Value Impact Relationship between Statement Instance and Its Direct Impact Successors

According to the relationship between the impact precursor and the impact successor, we get the following conclusion: If the statement instances $s^{q_1}_{p_1,t_k}, s^{q_2}_{p_2,t_k}, \cdots, s^{q_n}_{p_n,t_k}$ are all direct impact successors of the statement instance s^{h}_{g,t_k}, then we get

$$V^{h}_{g,t_k} = \bigcup_{c=1,2,\cdots,n} V^{q_c}_{p_c,t_k}. \tag{2}$$

Example 2. *From Figure 1, we can know that the direct impact successors of the statement instance* s^1_{7,t_1} *consist of* s^2_{5,t_1}, s^2_{6,t_1} *and* s^2_{7,t_1}. *Under the condition that we know* $V^2_{5,t_1} = \varnothing$, $V^2_{6,t_1} = \{s^1_{8,t_1}\}$ *and* $V^2_{7,t_1} = \varnothing$, *we have*

$$V^1_{7,t_1} = V^2_{5,t_1} \bigcup V^2_{6,t_1} \bigcup V^2_{7,t_1} = \{s^1_{8,t_1}\}.$$

This formula indicates that the errors generated from the statement instance s^1_{7,t_1} *can reach up to one output statement instance* s^1_{8,t_1} *when it propagates along the original execution path of the test case* t_1. *Using the same method, we also know* $V^1_{6,t_1} = \{s^1_{8,t_1}\}$, $V^2_{6,t_1} = \{s^1_{8,t_1}\}$, $V^2_{5,t_1} = \varnothing$ *and* $V^3_{5,t_1} = \varnothing$.

3.1.3. Value Impact Set of Branch Instance

The information expressed by the statement instances in the branch instance B^l_{r,z,t_k} can propagate along the original execution path H_k to some execution instances of the program output statements. These affected output statement instances constitute the *value impact set* V^l_{r,z,t_k} *of the branch instance* B^l_{r,z,t_k}. We can get the following formula:

$$V^l_{r,z,t_k} = \bigcup_{d=1,2,\cdots,n} V^{h_d}_{g_d,t_k}, \tag{3}$$

where $s^{h_1}_{g_1,t_k}, s^{h_2}_{g_2,t_k}, \cdots, s^{h_n}_{g_n,t_k}$ are all the statement instances in the branch instance B^l_{r,z,t_k}.

Example 3. *We can use formula (3) to calculate the value impact set of the branch instance* B^1_{5,t,t_1}. *From Example 2, we know both* $V^1_{6,t_1} = \{s^1_{8,t_1}\}$, $V^1_{7,t_1} = \{s^1_{8,t_1}\}$, $V^2_{5,t_1} = \varnothing$, $V^2_{6,t_1} = \{s^1_{8,t_1}\}$ $V^2_{7,t_1} = \varnothing$, $V^3_{5,t_1} = \varnothing$. *Because the branch instance* B^1_{5,t,t_1} *consists of the six statement instances* s^1_{6,t_1}, s^1_{7,t_1}, s^2_{5,t_1}, s^2_{6,t_1} s^2_{7,t_1}, *and* s^3_{5,t_1}, *we get* $V^1_{5,t,t_1} = V^1_{6,t_1} \bigcup V^1_{7,t_1} \bigcup V^2_{5,t_1} \bigcup V^2_{6,t_1} \bigcup V^2_{7,t_1} \bigcup V^3_{5,t_1} = \{s^1_{8,t_1}\}$.

3.1.4. Value Impact Set of the Special Statement Instance

If a statement instance is an output statement instance, it usually does not have any impact successors. We set its value impact set to itself because the change of its execution result is precisely the change of program output. If a statement instance is not an output statement instance and does not have any impact successors, then we set its value impact set to an empty set.

3.2. Path Impact Factor

The path impact factor of a statement expresses its ability to directly impact the execution paths of the test cases.

3.2.1. Path Impact Factor of Statement

The more controlling expression instances a statement impact, the more easily the fault in the statement changes the execution paths of the test cases. The more likely the execution path is changed, the more likely the program output will be changed. Therefore, we take the number of the control expression instances impacted by a statement during the test suite execution as a feature to describe

the effect of this statement on program output. For this purpose, we defined a statement's path impact factor.

The errors generated from the statement instance s^h_{g,t_k} may propagate along the original execution path H_k to some controlling expression instances. The collection of these controlling expression instances is called the *path impact set* P^h_{g,t_k} *of the statement instance* s^h_{g,t_k}. The element in P^h_{g,t_k} is called the *path impact element of* s^h_{g,t_k}. The path impact set of statement s_g is the union of path impact sets of execution instances of s_g, and denoted as P_g. In other words,

$$P_g = \bigcup_{k=1,2,\cdots,K} \bigcup_{h=1,2,\cdots,H_{gk}} P^h_{g,t_k}. \tag{4}$$

K is the total number of test cases in the test suite, and H_{gk} is the total number of times the statement s_g is executed by the test case t_k.

Example 4. *From Table 2, we know that the statement* s_6 *has four execution instances* s^1_{6,t_1}, s^2_{6,t_1}, s^1_{6,t_3} *and* s^2_{6,t_3}. *If* s_6 *includes a fault, then when* s_6 *is executed by the test suite, each execution instance may generate an error. The errors generated from first two statement instances* s^1_{6,t_1} *and* s^2_{6,t_1} *may propagate along the original execution path* H_1 *to the controlling expression instance* s^1_{11,t_1}. *Along the original execution path* H_3, *the errors generated from the last two statement instances* s^1_{6,t_3} *and* s^2_{6,t_3} *cannot be propagated to any controlling expression instance. Therefore,* $P^1_{6,t_1} = P^2_{6,t_1} = \{s^1_{11,t_1}\}$ *and* $P^1_{6,t_3} = P^2_{6,t_3} = \varnothing$. *Using Formula (4), we get*

$$P_6 = P^1_{6,t_1} \bigcup P^2_{6,t_1} \bigcup P^1_{6,t_3} \bigcup P^2_{6,t_3} = \{s^1_{11,t_1}\}.$$

3.2.2. The Path Impact Relationship of the Statement Instance and Its Direct Impact Successor

According to the relationship between the impact precursor and the impact successor, we get the following conclusion: If the statement instances $s^{q_1}_{p_1,t_k}, s^{q_2}_{p_2,t_k}, \cdots, s^{q_n}_{p_n,t_k}$ are all direct impact successors of the statement instance s^h_{g,t_k}, then

$$P^h_{g,t_k} = \bigcup_{c=1,2,\cdots,n} P^{q_c}_{p_c,t_k}. \tag{5}$$

Example 5. *From Figure 1, we can know the the direct impact successors of the statement instance* s^1_{7,t_1} *consist of* s^2_{5,t_1}, s^2_{6,t_1} *and* s^2_{7,t_1}. *Under the condition that we know* $P^2_{5,t_1} = \{s^2_{5,t_1}\}$, $P^2_{6,t_1} = \{s^1_{11,t_1}\}$ *and* $P^2_{7,t_1} = \{s^3_{5,t_1}\}$, *according to Formula (5), we have*

$$P^1_{7,t_1} = P^2_{5,t_1} \bigcup P^2_{6,t_1} \bigcup P^2_{7,t_1} = \{s^2_{5,t_1}, s^1_{11,t_1}, s^3_{5,t_1}\}.$$

Therefore, the errors generated from the statement instance s^1_{7,t_1} *can change up to three controlling expression instances* s^2_{5,t_1}, s^1_{11,t_1} *and* s^3_{5,t_1} *along the original execution path* H_1. *Using the same method, we can also get* $P^1_{6,t_1} = \{s^1_{11,t_1}\}$, $P^3_{5,t_1} = \{s^3_{5,t_1}\}$, *and so on.*

3.2.3. Path Impact Set of Branch Instance

The statement instances in the branch instance B^l_{r,z,t_k} may propagate their information along the original execution path H_k to some of the controlling expression instances outside of B^l_{r,z,t_k}. These controlling expression instances constitute the *path impact set* P^l_{r,z,t_k} *of the branch instance* B^l_{r,z,t_k}. The path impact set of B^l_{r,z,t_k} express the impact of B^l_{r,z,t_k} on the controlling expression instances outside of B^l_{r,z,t_k}. If $s^{h_1}_{g_1,t_k}, s^{h_2}_{g_2,t_k}, \cdots, s^{h_n}_{g_n,t_k}$ are all the statement instances in the branch instance B^l_{r,z,t_k}, then the following mathematical formula holds

$$P^l_{r,z,t_k} = \left(\bigcup_{d=1,2,\cdots,n} P^{h_d}_{g_d,t_k} \right) \Big\backslash B^l_{r,z,t_k}. \tag{6}$$

Example 6. *We illustrate the formula above by calculating the path impact set of the branch instance* B^1_{5,t,t_1}. *From Example 5, we know that* $P^1_{6,t_1} = \{s^1_{11,t_1}\}$, $P^1_{7,t_1} = \{s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}\}$, $P^2_{5,t_1} = \{s^2_{5,t_1}\}$, $P^2_{6,t_1} = \{s^1_{11,t_1}\}$, $P^2_{7,t_1} = \{s^3_{5,t_1}\}$, *and* $P^3_{5,t_1} = \{s^3_{5,t_1}\}$. *Because the branch instance* B^1_{5,t,t_1} *consists of the six statement instances* s^1_{6,t_1}, s^1_{7,t_1}, s^2_{5,t_1}, s^2_{6,t_1}, s^2_{7,t_1} *and* s^3_{5,t_1}, *we get*

$$P^1_{5,t,t_1} = (P^1_{6,t_1} \bigcup P^1_{7,t_1} \bigcup P^2_{5,t_1} \bigcup P^2_{6,t_1} \bigcup P^2_{7,t_1} \bigcup P^3_{5,t_1}) \setminus B^1_{5,t,t_1} = \{s^1_{11,t_1}\}.$$

3.2.4. Path Impact Set of the Special Statement Instance

If a statement instance is a controlling expression instance, it usually does not have any impact successors. We set its path impact set to itself because the change of its execution result is precisely the change of the program execution path. If a statement instance is not a controlling expression instance and does not have any impact successors, then we set its path impact set to an empty set.

3.3. Generalized Value Impact Factor

The generalized value impact factor of a statement expresses its ability to indirectly impact the program outputs.

3.3.1. Generalized Value Impact Factor of Statement

The error generated from the statement instance s^h_{g,t_k} may propagate to some controlling expression instances along the original execution path of test case t_k, so that the execution results of these controlling expression instances may be changed. As long as the execution result of the control expression instance s^l_{r,t_k} is changed, the branch instance B^l_{r,z,t_k}, which appears in the original execution path H_k, will no longer be executed. This makes the statement instances in B^l_{r,z,t_k} no longer pass their information to some output statement instances. Thus, the execution results of these output statement instances may be changed. Therefore, the errors generated from the statement instance s^h_{g,t_k} may indirectly affect some output statement instances through the above error propagation process. These output statement instances that may be indirectly influenced by s^h_{g,t_k} form the *generalized value impact set of the statement instance* s^h_{g,t_k}. The generalized value impact set of the statement instance s^h_{g,t_k} is denoted as $\mathcal{V}^h_{g,t_k}$. The element in $\mathcal{V}^h_{g,t_k}$ is called the generalized value impact element of s^h_{g,t_k}. The number of generalized value impact element of s^h_{g,t_k} is called the generalized value impact factor of s^h_{g,t_k}, and is denoted as $x_{gvi}(s^h_{g,t_k})$.

A statement s_g has multiple execution instances generally and each execution instance has its own generalized value impact set. In order to describe this indirect effect of s_g on program output, the union of these generalized value impact sets is called the generalized value impact set of s_g. The generalized value impact set of s_g is denoted as $\mathcal{V}_g$, the element in $\mathcal{V}_g$ is called the generalized value impact element of s_g, and the number of the generalized value impact element of s_g is called the generalized value impact factor of s_g. In summary,

$$\mathcal{V}_g = \bigcup_{k=1,2,\cdots,K} \bigcup_{h=1,2,\cdots,H_{gk}} \mathcal{V}^h_{g,t_k}, \tag{7}$$

where K is the total number of test cases in the test suite, and H_{gk} is the total number of times the statement s_g is executed by the test case t_k.

Example 7. *In Program 1, the statement* s_7 *has four execution instances* s^1_{7,t_1}, s^2_{7,t_1} s^1_{7,t_3},*and* s^2_{7,t_3}. *Given that* $\mathcal{V}^1_{7,t_1} = \{s^1_{8,t_1}, s^1_{13,t_1}\}$, $\mathcal{V}^2_{7,t_1} = \varnothing$, $\mathcal{V}^1_{7,t_3} = \{s^1_{8,t_3}\}$, *and* $\mathcal{V}^2_{7,t_3} = \varnothing$, *we can use Formula (7) to calculate the generalized value impact set of the statement* s_7*:*

$$\mathcal{V}_7 = \mathcal{V}^1_{7,t_1} \bigcup \mathcal{V}^2_{7,t_1} \bigcup \mathcal{V}^1_{7,t_3} \bigcup \mathcal{V}^2_{7,t_3} = \{s^1_{8,t_1}, s^1_{13,t_1}, s^1_{8,t_3}\}.$$

3.3.2. Generalized Value Impact Set of the Special Statement Instance

A controlling expression instance s^l_{r,t_k} usually does not have any impact successors. Corresponding to s^l_{r,t_k}, there is usually a branch instance B^l_{r,z,t_k} that appears in the original execution path H_k. In this situation, the generalized value impact set of s^l_{r,t_k} is equal to the value impact set of the branch instance B^l_{r,z,t_k}. This conclusion can be interpreted as follows: If an error is generated from the controlling expression instance s^l_{r,t_k}, then the branch instance B^l_{r,z,t_k} will no longer be executed, so that the statement instances in B^l_{r,z,t_k} can no longer propagate their information along the original execution path H_k to some output statement instances. This error propagation process also exactly reflects the impact of branch instance B^l_{r,z,t_k} on program output. Hence, the above conclusion is proved. For example, the generalized value impact set of the controlling expression instance s^1_{5,t_1} is equal to the value impact set of the branch instance B^1_{5,t,t_1}.

If a statement instance is not a controlling expression instance and does not have any impact successors, then we set its generalized value impact set to an empty set.

3.3.3. The Generalized Value Impact Relationship between a Statement and Its Direct Impact Successors

According to the relationship between the impact precursor and the impact successor, we get the following conclusion: If the statement instances $s^{q_1}_{p_1,t_k}, s^{q_2}_{p_2,t_k}, \cdots, s^{q_n}_{p_n,t_k}$ are all direct impact successors of the statement instance s^h_{g,t_k}, then

$$\mathcal{V}^h_{g,t_k} = \bigcup_{c=1,2,\cdots,n} \mathcal{V}^{q_c}_{p_c,t_k}. \tag{8}$$

Example 8. *In Program 1, s^2_{5,t_1}, s^2_{6,t_1} and s^2_{7,t_1} are all direct impact successors of the statement instance s^1_{7,t_1}. Given that $\mathcal{V}^2_{5,t_1} = \{s^1_{8,t_1}\}$, $\mathcal{V}^2_{6,t_1} = \{s^1_{13,t_1}\}$, and $\mathcal{V}^2_{7,t_1} = \varnothing$, according to formula (8), we can get*

$$\mathcal{V}^1_{7,t_1} = \mathcal{V}^2_{5,t_1} \bigcup \mathcal{V}^2_{6,t_1} \bigcup \mathcal{V}^2_{7,t_1} = \{s^1_{8,t_1}, s^1_{13,t_1}\}.$$

In the same way, given that $\mathcal{V}^3_{5,t_1} = \varnothing$, we can get $\mathcal{V}^2_{7,t_1} = \mathcal{V}^3_{5,t_1} = \varnothing$. Given that $\mathcal{V}^2_{5,t_3} = \{s^1_{8,t_3}\}$, $\mathcal{V}^2_{6,t_3} = \varnothing$, and $\mathcal{V}^2_{7,t_3} = \varnothing$,we can get $\mathcal{V}^1_{7,t_3} = \mathcal{V}^2_{5,t_3} \bigcup \mathcal{V}^2_{6,t_3} \bigcup \mathcal{V}^2_{7,t_3} = \{s^1_{8,t_3}\}$

3.4. Generalized Path Impact Factor

The generalized path impact factor of a statement expresses its ability to indirectly change the program execution path.

3.4.1. Generalized Path Impact Factor of Statement

The error generated from the statement instance s^h_{g,t_k} may propagate to some controlling expression instances along the original execution path of test case t_k. As long as the execution result of the control expression instance s^l_{r,t_k} is changed, the branch instance B^l_{r,z,t_k} that appears in the original execution path H_k will no longer be executed. The statement instances in B^l_{r,z,t_k} will no longer pass their information to the controlling expression instances appearing after B^l_{r,z,t_k}. In this situation, the execution results of the controlling expression instances appearing after B^l_{r,z,t_k} may be changed because they are no longer influenced by the statement instances in B^l_{r,z,t_k}. Therefore, the errors generated from the statement instance s^h_{g,t_k} may indirectly affect some controlling expression instances appearing after B^l_{r,z,t_k} through the above error propagation process. These controlling expression instances that may be indirectly affected by s^h_{g,t_k} through the above error propagation process form the *generalized path impact set of the statement instance* s^h_{g,t_k}. This set is denoted as $\mathcal{P}^h_{g,t_k}$, the element of which is called the generalized path impact element of s^h_{g,t_k}. The number of generalized path impact elements of s^h_{g,t_k} is called the generalized path impact factor of s^h_{g,t_k}, and is denoted as $x_{gpi}(s^h_{g,t_k})$.

A statement s_g has one or more execution instances generally. Therefore, the generalized path impact set of s_g is defined as the union of generalized path impact sets of the execution instances of s_g, and is denoted as $\mathcal{P}_g$. In other words,

$$\mathcal{P}_g = \bigcup_{k=1,2,\cdots,K} \bigcup_{h=1,2,\cdots,H_{gk}} \mathcal{P}^h_{g,t_k}, \tag{9}$$

where K is the total number of test cases in the test suite, and H_{gk} is the total number of times the statement s_g is executed by the test case t_k. The element in $\mathcal{P}_g$ is called the generalized path impact element of s_g. The number of generalized path impact element of s_g is called the generalized path impact factor of s_g, and denoted as $x_{gpi}(s_g)$.

Example 9. *We explain the above definitions by calculating the generalized path impact set of the statement s_7. In Program 1, the statement s_7 has four execution instances s^1_{7,t_1}, s^2_{7,t_1}, s^1_{7,t_3} and s^2_{7,t_3}. Given that $\mathcal{P}^1_{7,t_1} = \{s^1_{11,t_1}\}$, $\mathcal{P}^2_{7,t_1} = \varnothing$, $\mathcal{P}^1_{7,t_3} = \varnothing$ and $\mathcal{P}^2_{7,t_3} = \varnothing$, we can use Formula (9) to calculate the generalized value impact set of the statement s_7.*

$$\mathcal{P}_7 = \mathcal{P}^1_{7,t_1} \bigcup \mathcal{P}^2_{7,t_1} \bigcup \mathcal{P}^1_{7,t_3} \bigcup \mathcal{P}^2_{7,t_3} = \{s^1_{11,t_1}\}.$$

3.4.2. Generalized Path Impact Set of the Special Statement Instance

If a statement instance s^l_{r,t_k} is a controlling expression instance, then it usually does not have any impact successors. Corresponding to s^l_{r,t_k}, there is usually a branch instance B^l_{r,z,t_k}, which exists in the original execution path H_k. In this situation, the generalized path impact set of s^l_{r,t_k} is precisely the path impact set of B^l_{r,z,t_k}. This conclusion can be interpreted as follows: Assume there is a software fault in statement s_r. If an error is generated from the controlling expression instance s^l_{r,t_k}, then the branch instance B^l_{r,z,t_k} will no longer be executed, the information expressed by the statement instances in B^l_{r,z,t_k} can no longer propagate along the original execution path H_k to some controlling expression instances outside of B^l_{r,z,t_k}. This error propagation process also exactly reflects the impact of branch instance B^l_{r,z,t_k} on the execution path of the test case t_k. Therefore, the generalized path impact set of s^l_{r,t_k} is equal to the path impact set of B^l_{r,z,t_k}. For example, the generalized path impact set of the controlling expression instance s^1_{5,t_1} is equal to the path impact set of the branch instance B^1_{5,t,t_1}. In other words, $\mathcal{P}^1_{5,t_1} = P^1_{5,t,t_1} = \{s^1_{11,t_1}\}$. Otherwise, if a statement instance is not a controlling expression instance and does not have any impact successors, then we set its generalized path impact set to an empty set.

3.4.3. The Generalized Path Impact Relationship between a Statement Instance and Its Direct Impact Successors

According to the relationship between the impact precursor and the impact successor, we get the following conclusion: If the statement instances $s^{q_1}_{p_1,t_k}$, $s^{q_2}_{p_2,t_k}$, $\cdots$, $s^{q_n}_{p_n,t_k}$ are all direct impact successors of the statement instance s^h_{g,t_k}, then

$$\mathcal{P}^h_{g,t_k} = \bigcup_{c=1,2,\cdots,n} \mathcal{P}^{q_c}_{p_c,t_k}. \tag{10}$$

Example 10. *In Program 1, s^2_{5,t_1}, s^2_{6,t_1} and s^2_{7,t_1} are all direct impact successors of the statement instance s^1_{7,t_1}. Given that $\mathcal{P}^2_{5,t_1} = \{s^1_{11,t_1}\}$, $\mathcal{P}^2_{6,t_1} = \varnothing$, and $\mathcal{P}^2_{7,t_1} = \varnothing$, according to formula (10), we can get*

$$\mathcal{P}^1_{7,t_1} = \mathcal{P}^2_{5,t_1} \bigcup \mathcal{P}^2_{6,t_1} \bigcup \mathcal{P}^2_{7,t_1} = \{s^1_{11,t_1}\}.$$

3.5. Latent Impact Factor

The fault in a statement may cause some program branches that have not yet been executed to be executed. The latent impact factor expresses the impact of these branches to be executed on the program output.

3.5.1. Latent Impact Factor of the Program Statement

Contrary to the branch instances that will no longer be executed, some branch instances may be going to be executed due to the error generated from the statement instance s^h_{g,t_k}. These branches to be executed may change the program outputs. For an example, in Program 1, if the assignment statement s_2 is mutated into dist=m%n, then the remainder dist becomes zero when test case t_1 runs. In this situation, the true branch $B_{11,t}$ of s_{11}, which consists of s_{12} and does not appear in the original execution path H_1, will be executed and change the program output.

These branch instances to be executed are divided into two classes. In the first class, each branch instance contains statement instances. In the second class, each branch instance does not. The first class branch instances constitute the *latent impact set of statement instance* s^h_{g,t_k}, and denotes as L^h_{g,t_k}. The element in L^h_{g,t_k} is called the *latent impact element of the statement instance* s^h_{g,t_k}. The number of latent impact elements of s^h_{g,t_k} is called *the latent impact factor of* s^h_{g,t_k} and denoted as $x_{li}(s^h_{g,t_k})$.

A statement s_g has multiple execution instances generally, and each of them has its own latent impact set. Therefore, the union of these latent impact sets is defined as the *latent impact set of the statement* s_g, and denoted as L_g. In other words,

$$L_g = \bigcup_{k=1,2,\cdots,K} \bigcup_{h=1,2,\cdots,H_{gk}} L^h_{g,t_k}, \tag{11}$$

where K is the total number of test cases in the test suite, and H_{gk} is the total number of times the statement s_g is executed by the test case t_k. The element in L_g is called the *latent impact element of* s_g. The number of latent impact element of s_g is called the *latent impact factor of the statement* s_g, and denoted as $x_{li}(s_g)$.

Example 11. *We are going to calculate the latent impact factor of the statement* s_7. *As shown in Table 2,* s_7 *has the four execution instances* s^1_{7,t_1}, s^2_{7,t_1}, s^1_{7,t_3} *and* s^2_{7,t_3}. *Assume four errors* e^1_{7,t_1}, e^2_{7,t_1}, e^1_{7,t_3} *and* e^2_{7,t_3} *are generated from the statement instances* s^1_{7,t_1}, s^2_{7,t_1}, s^1_{7,t_3} *and* s^2_{7,t_3}, *respectively. In this situation,* e^1_{7,t_1} *may propagate along the original execution path* H_1 *to the controlling expression instances* s^2_{5,t_1}, s^3_{5,t_1} *and* s^1_{11,t_1}. *When* e^1_{7,t_1} *propagates to* s^2_{5,t_1}, *the branch instances* B^2_{5,f,t_1} *that do not appear in the original execution path* H_1 *will be executed. However, the role of* $B_{5,f}$ *is to exit the loop, so that it does not contain any statements. Thus,* B^2_{5,f,t_1} *itself does not affect program output. This makes* B^2_{5,f,t_1} *not a latent impact element of* s^1_{7,t_1}. *When* e^1_{7,t_1} *propagates along the original execution path* H_1 *to* s^3_{5,t_1}, *the branch instance* B^3_{5,t,t_1} *that does not appear in the original execution path* H_1 *will be executed. The* $B_{5,t}$ *contains some statements so that the execution of* B^3_{5,t,t_1} *in itself may change the program outputs. Thus,* B^3_{5,t,t_1} *is a latent impact element of* s^1_{7,t_1}. *When* e^1_{7,t_1} *propagates along the original execution path* H_1 *to* s^1_{11,t_1}, *the branch instance* B^1_{11,t,t_1} *that does not appear in the original execution path* H_1 *will be executed. The program branch* $B_{11,t}$ *contains some statements so that the execution of* B^1_{11,t,t_1} *in itself may change the program outputs. Thus, the branch instance* B^1_{11,t,t_1} *is a latent impact element of* s^1_{7,t_1}. *From the above analysis, we can know that the latent impact set of* s^1_{7,t_1} *consists of* B^3_{5,t,t_1} *and* B^1_{11,t,t_1}. *In the similar way, we can know that the latent impact set of the statement instance* s^2_{7,t_1} *consists of* B^3_{5,t,t_1}. *The latent impact set of* s^1_{7,t_3} *consists of* B^3_{5,t,t_3}, *and that of* s^2_{7,t_3} *also consists of* B^3_{5,t,t_3}. *With Formula (11), we can get the latent impact set of the statement* s_7:

$$L_7 = L^1_{7,t_1} \bigcup L^2_{7,t_1} \bigcup L^1_{7,t_3} \bigcup L^2_{7,t_3} = \{B^3_{5,t,t_1}, B^1_{11,t,t_1}, B^3_{5,t,t_3}\}.$$

3.5.2. The Latent Impact Relationship between a Statement Instance and its Direct Impact Successors

According to the relationship between the impact precursor and the impact successor, we get the following conclusion: If the statement instances $s^{q_1}_{p_1,t_k}, s^{q_2}_{p_2,t_k}, \cdots, s^{q_n}_{p_n,t_k}$ are all direct impact successors of the statement instance s^{h}_{g,t_k}, then

$$L^{h}_{g,t_k} = \bigcup_{c=1,2,\cdots,n} L^{q_c}_{p_c,t_k}. \tag{12}$$

Example 12. *From Figure 1, we can know the the direct impact successors of the statement instance s^1_{7,t_1} consist of s^2_{5,t_1}, s^2_{6,t_1} and s^2_{7,t_1}. Given that $L^2_{5,t_1} = \varnothing$, $L^2_{6,t_1} = \{B^1_{11,t,t_1}\}$ and $L^2_{7,t_1} = \{B^3_{5,t,t_1}\}$, according to Formula (12), we have*

$$L^1_{7,t_1} = L^2_{5,t_1} \bigcup L^2_{6,t_1} \bigcup L^2_{7,t_1} = \{B^1_{11,t,t_1}, B^3_{5,t,t_1}\}.$$

In addition, under the condition that we know $L^2_{5,t_3} = \varnothing$, $L^2_{6,t_3} = \varnothing$ and $L^2_{7,t_3} = \{B^3_{5,t,t_3}\}$, using the same method, we can still get

$$L^1_{7,t_3} = L^2_{5,t_3} \bigcup L^2_{6,t_3} \bigcup L^2_{7,t_3} = \{B^3_{5,t,t_3}\}.$$

Furthermore, we can get

$$L_7 = L^1_{7,t_1} \bigcup L^2_{7,t_1} \bigcup L^1_{7,t_3} \bigcup L^2_{7,t_3} = \{B^1_{11,t,t_1}, B^3_{5,t,t_1}, B^3_{5,t,t_3}\}.$$

3.5.3. Latent Impact Set of the Special Statement Instance

If a statement instance s^l_{r,t_k} is a controlling expression instance and the branch instance B^l_{r,z,t_k} does not appear in the original execution path H_k; then, in the condition that B^l_{r,z,t_k} is not empty, we set B^l_{r,z,t_k} as the only element in the latent impact set of s^l_{r,t_k}. If a statement instance is not a controlling expression instance and does not have any impact successors, then we set the latent impact set of s^l_{r,t_k} to an empty set.

For example, as far as the controlling expression instance s^2_{5,t_1} is concerned, although the branch instance B^2_{5,f,t_1} does not appear in the original execution path H_1, B^2_{5,f,t_1} does not include any statement instance. Hence, B^2_{5,f,t_1} is not a latent impact element of s^2_{5,t_1}, and we set the latent impact set of s^3_{5,t_1} to an empty set. As far as the controlling expression instance s^3_{5,t_1} is concerned, because the branch instance B^3_{5,t,t_1} not only does not appear in the original execution path H_1 but also is not empty, we set s^3_{5,t_1} as the only element in the latent impact set of B^3_{5,t,t_1}.

3.6. Information Hidden Factor

The last feature of a statement is its information hiding feature. Sometimes, the program has multiple output statements, and some of them happen to generate same outputs. In this case, even if the software fault in a statement changes the execution path of the test case, the output of the program may still not be changed.

This phenomenon make the faults in statements difficult to identify. For a statement s_g, we use the information hiding factor to express this feature. The information hiding factor of s_g can be calculated in the following way. We use the test cases that execute s_g to construct sub test suite T_g. When we execute T_g, the program under testing generates some outputs. The information entropy of the output distribute is called the information hidden factor of statement s_g, and denoted as $x_{ih}(s_g)$. In other words,

$$x_{ih}(s_g) = -\sum_i p_i \log_2 p_i, \tag{13}$$

where p_i is the probability that the test cases executing the statement s_g generate the ith program output class.

Example 13. *We calculate the information hidden factors of the statements s_9 and s_{11}, respectively. In Program 1, the test suite consists of the test cases t_1, t_2 and t_3. These three test cases all execute statement s_9. Their executions generate three program outputs (fac = 6, class 1), (fac = 1 class 2) and (fac = 6 class 1), respectively. Hence, the probability that the program output (fac = 6, class 1) is 0.67, and the probability that the program output (fac = 1 class 2) is 0.33. According to Formula (13), the information hidden factor of statement s_9 is 0.9182 bit. The test cases t_1 and t_2 execute the statement s_{11}. Their executions generate two program outputs (fac = 6, class 1) and (fac = 1 class 2), respectively. Hence, the probabilities that the program output (fac = 6, class 1) and (fac = 6, class 1) are both 0.5. According to Formula (13), the information hidden factor of statement s_{11} is 1.0 bit.*

4. Calculation of Statement Features

First, we propose an iterative method to compute statement features, and then compare the time cost of this method with that of direct mutant testing.

4.1. Calculation Process

We divide the calculation of all the statement features into two parts. The first part calculation includes the first five statement features: the value impact factor, the path impact factor, the generalized value impact factor, the generalized path impact factor, and the latent impact factor. The second part calculation includes includes the last two statement features: the number of times a statement is executed, and information hidden factor.

The first part of the calculation takes much more time than the second one. For reducing the computational complexity, we propose an iterative method. Generally, if a statement instance has at least one impact successor, then we can calculate its first five features according to the formulas (2), (5), (8), (10), and (12). Otherwise, we use the methods mentioned in Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3 to calculate its first five features.

The computation of the statement features is divided into two corresponding stages. The first stage, including steps 1–6, calculate the first part of statement features. The second stage including steps 7 and 8, calculate the second part of statement features. The overall computation steps are as follows:

Step 1 Set test case serial number $k = 1$.

Step 2 Construct the execution impact graph G_k of the test case t_k.

Step 3 First, from the original execution path of the test case t_k, find all statement instances that have not been analyzed. From these unanalyzed statement instances, find the last executed statement instance. We might as well denote this statement instance as s^h_{g,t_k}.

(1) If s^h_{g,t_k} has one or more impact successors, then we construct the impact sets of its first five features according to the formulas (2), (5), (8), (10) and (12).

(2) If s^h_{g,t_k} does not have any impact successors, then we construct the impact sets of its first five features according to the methods mentioned in Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3.

Step 4 If there are some statement instances which appear in the original execution path of test case t_k but have not yet been analyzed, go to step 3, else go to step 5.

Step 5 If test case t_k is not the last test case in test suite, then k = k + 1, and go to step 2, else go to step 6.

Step 6 First, construct each program statement's value impact set, path impact set, generalized value impact set, generalized path impact set and latent impact set by formulas (1), (4), (7), (9) and (11). Next, for each program statement , calculate its value impact factor, path impact factor, generalized value impact factor, generalized path impact factor factor and the latent impact factor.

Step 7 For each statement in program under testing, compute the total number of times it is executed by the test cases in the test suite.

Step 8 For each statement in the program under testing, compute its information hidden factor by formula (13).

Example 14. *We illustrate the above process by extracting the features of each statement in Program 1. In terms of the first stage of extracting the statement features, whether the statement instances are generated during the execution of test case 1, test case 2 or test case 3, the methods for calculating features of the statement instance are the same. Therefore, with regard to steps 1 to 5, we only explain in detail how to calculate the features of the statement instances generated during test case t_1 execution. The detailed calculation process is as follows.*

We first set $k = 1$, execute test case t_1, and construct the execution impact graph G_1 of test case t_1 as shown in Figure 1.

The first analyzed statement instance is the last executed statement instance in original execution path H_1. Thus, we first analyze the output statement instance s^1_{13,t_1}. According to Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3, we get $V^1_{13,t_1} = \{s^1_{13,t_1}\}$, $P^1_{13,t_1} = \varnothing$, $\mathcal{V}^1_{13,t_1} = \varnothing$, $\mathcal{P}^1_{13,t_1} = \varnothing$ and $Ł^1_{13,t_1} = \varnothing$.

The second analyzed statement instance s^1_{11,t_1} is the penultimate element in original execution path H_1. Because it is a controlling expression, we get $V^1_{11,t_1} = \varnothing$, $P^1_{11,t_1} = \{s^1_{11,t_1}\}$, $\mathcal{V}^1_{11,t_1} = V^1_{11,t,t_1} = V^1_{13,t_1} = \{s^1_{13,t_1}\}$, $\mathcal{P}^1_{11,t_1} = \varnothing$ and $L^1_{11,t_1} = \{B^1_{11,t,t_1}\}$ according to Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3.

The third analyzed statement instance s^1_{9,t_1} is the antepenultimate element in H_1. Because s^1_{9,t_1} is a controlling expression instance, we get $V^1_{9,t_1} = \varnothing$, $P^1_{9,t_1} = \{s^1_{9,t_1}\}$, $\mathcal{V}^1_{9,t_1} = V^1_{9,t,t_1} = V^1_{11,t_1} \bigcup V^1_{13,t_1} = \{s^1_{13,t_1}\}$, $\mathcal{P}^1_{9,t_1} = P^1_{9,t,t_1} = (P^1_{11,t_1} \bigcup P_{13,t_1}) \setminus B^1_{9,t,t_1} = (s^1_{11,t_1} \bigcup \varnothing) \setminus B^1_{9,t,t_1} = \varnothing$ and $Ł^1_{9,t_1} = \{B^1_{9,t,t_1}\}$ according to Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3.

The fourth analyzed statement instant s^1_{8,t_1} is the fourth element from the end of H_1. Because s^1_{8,t_1} is an output statement instance, $V^1_{8,t_1} = \{s^1_{8,t_1}\}$, $P^1_{8,t_1} = \varnothing$, $\mathcal{V}^1_{8,t_1} = \varnothing$, $\mathcal{P}^1_{8,t_1} = \varnothing$ and $Ł^1_{8,t_1} = \varnothing$ according to Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3.

The fifth analyzed statement instant s^3_{5,t_1} is the fifth element from the end of H_1. Because s^3_{5,t_1} is a controlling expression instance of zero length, $V^3_{5,t_1} = \varnothing$, $P^3_{5,t_1} = \{s^3_{5,t_1}\}$, $\mathcal{V}^3_{5,t_1} = V^3_{5,f,t_1} = \varnothing$, $\mathcal{P}^3_{5,t_1} = P^1_{5,t,t_1} = \varnothing$ and $Ł^3_{5,t_1} = \{B^3_{5,t,t_1}\}$ according to Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3.

The sixth analyzed statement instance s^2_{7,t_1} is the sixth statement instance from the end of H_1. Because the direct impact successors of s^2_{7,t_1} consist of s^3_{5,t_1}, we get $V^2_{7,t_1} = V^3_{5,t_1} = \varnothing$, $P^2_{7,t_1} = P^3_{5,t_1} = \{s^3_{5,t_1}\}$, $\mathcal{V}^2_{7,t_1} = \mathcal{V}^3_{5,t_1} = \varnothing$, $\mathcal{P}^2_{7,t_1} = \mathcal{P}^3_{5,t_1} = \varnothing$, $Ł^2_{7,t_1} = Ł^3_{5,t_1} = \{B^3_{5,t,t_1}\}$ according to formulas (2), (5), (8), (10) and (12).

The seventh analyzed statement instance s^2_{6,t_1} is the seventh element from the end of H_1. Becasue the direct impact successors of s^2_{6,t_1} consist of s^1_{8,t_1} and s^1_{11,t_1}, we get $V^2_{6,t_1} = V^1_{8,t_1} \bigcup V^1_{11,t_1} = \{s^1_{8,t_1}\}$, $P^2_{6,t_1} = P^1_{8,t_1} \bigcup P^1_{11,t_1} = \{s^1_{11,t_1}\}$, $\mathcal{V}^2_{6,t_1} = \mathcal{V}^1_{8,t_1} \bigcup \mathcal{V}^1_{11,t_1} = \{s^1_{13,t_1}\}$, $\mathcal{P}^2_{6,t_1} = \mathcal{P}^1_{8,t_1} \bigcup \mathcal{P}^1_{11,t_1} = \varnothing$ and $L^2_{6,t_1} = L^1_{8,t_1} \bigcup L^1_{11,t_1} = \{B^1_{11,t,t_1}\}$ according to formulas (2), (5), (8), (10) and (12).

The eighth analyzed statement instance s^2_{5,t_1} is the eighth element from the end of H_1. Because s^2_{5,t_1} is a controlling expression instance, we get $V^2_{5,t_1} = \varnothing$, $P^2_{5,t_1} = \{s^2_{5,t_1}\}$, $\mathcal{V}^2_{5,t_1} = V^2_{5,t,t_1} = V^2_{6,t_1} \bigcup V^2_{7,t_1} \bigcup V^3_{5,t_1} = \{s^1_{8,t_1}\}$, $\mathcal{P}^2_{5,t_1} = P^2_{5,t,t_1} = (P^2_{6,t_1} \bigcup P^2_{7,t_1} \bigcup P^3_{5,t_1}) \setminus P^2_{5,t,t_1} = \{s^1_{11,t_1}\}$ and $Ł^2_{5,t_1} = \varnothing$ according to Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3.

The ninth analyzed statement instance s^1_{7,t_1} is the ninth element from the end of H_1. Because the direct impact successors of s^1_{7,t_1} consist of s^2_{5,t_1}, s^2_{6,t_1} and s^2_{7,t_1}, we get $V^1_{7,t_1} = V^2_{5,t_1} \bigcup V^2_{6,t_1} \bigcup V^2_{7,t_1} = \{s^1_{8,t_1}\}$, $P^1_{7,t_1} = P^2_{5,t_1} \bigcup P^2_{6,t_1} \bigcup P^2_{7,t_1} = \{s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}\}$, $\mathcal{V}^1_{7,t_1} = \mathcal{V}^2_{5,t_1} \bigcup \mathcal{V}^2_{6,t_1} \bigcup \mathcal{V}^2_{7,t_1} = \{s^1_{8,t_1}, s^1_{13,t_1}\}$, $\mathcal{P}^1_{7,t_1} = \mathcal{P}^2_{5,t_1} \bigcup \mathcal{P}^2_{6,t_1} \bigcup \mathcal{P}^2_{7,t_1} = \{s^1_{11,t_1}\}$ and $L^1_{7,t_1} = L^2_{5,t_1} \bigcup L^2_{6,t_1} \bigcup L^2_{7,t_1} = \{B^3_{5,t,t_1}, B^1_{11,t,t_1}\}$ according to formulas (2), (5), (8), (10) and (12).

The tenth analyzed statement instance s^1_{6,t_1} is the tenth element from the end of H_1. Because the direct impact successors of s^1_{6,t_1} consist of s^2_{6,t_1}, we get $V^1_{6,t_1} = V^2_{6,t_1} = \{s^1_{8,t_1}\}$, $P^1_{6,t_1} = P^2_{6,t_1} = \{s^1_{11,t_1}\}$,

$\mathcal{V}^1_{6,t_1} = \mathcal{V}^2_{6,t_1} = \{s^1_{13,t_1}\}$, $\mathcal{P}^1_{6,t_1} = \mathcal{P}^2_{6,t_1} = \varnothing$ and $L^1_{6,t_1} = \{B_{11,t,t_1}\}$ according to formulas (2), (5), (8), (10) and (12).

The eleventh analyzed statement instance s^1_{5,t_1} is the eleventh statement instance from the end from H_1. Because s^1_{5,t_1} is a controlling expression instance, we get $V^1_{5,t_1} = \varnothing$, $P^1_{5,t_1} = \{s^1_{5,t_1}\}$, $\mathcal{V}^1_{5,t_1} = V^1_{5,t,t_1} = \{s^1_{8,t_1}\}$, $\mathcal{P}^1_{5,t_1} = P^1_{5,t,t_1} = \{s^1_{11,t_1}\}$ and $L^1_{5,t_1} = \varnothing$ according to Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3.

The twelfth analyzed statement instance s^1_{4,t_1} is the twelfth element from the end of H_1. Because the direct impact successors of s^1_{4,t_1} consist of s^1_{6,t_1}, we get $V^1_{4,t_1} = V^1_{6,t_1} = \{s^1_{8,t_1}\}$, $P^1_{4,t_1} = P^1_{6,t_1} = \{s^1_{11,t_1}\}$, $\mathcal{V}^1_{4,t_1} = \mathcal{V}^1_{6,t,t_1} = \{s^1_{13,t_1}\}$, $\mathcal{P}^1_{4,t_1} = \mathcal{P}^1_{6,t,t_1} = \varnothing$, $L^1_{4,t_1} = L^1_{6,t_1} = \{B^1_{11,t,t_1}\}$ according to formulas (2), (5), (8), (10) and (12).

The thirteenth analyzed statement instance s^1_{2,t_1} is the thirteenth element from the end of H_1. The direct impact successors of s^1_{2,t_1} consist of s^1_{5,t_1}, s^1_{6,t_1} and s^1_{7,t_1}. According to formulas (2), (5), (8), (10) and (12), we get $V^1_{2,t_1} = V^1_{5,t_1} \bigcup V^1_{6,t_1} \bigcup V^1_{7,t_1} = \{s^1_{8,t_1}\}$, $P^1_{2,t_1} = P^1_{5,t_1} \bigcup P^1_{6,t_1} \bigcup P^1_{7,t_1} = \{s^1_{5,t_1}, s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}\}$, $\mathcal{V}^1_{2,t_1} = \mathcal{V}^1_{5,t_1} \bigcup \mathcal{V}^1_{6,t_1} \bigcup \mathcal{V}^1_{7,t_1} = \{s^1_{8,t_1}, s^1_{13,t_1}\}$, $\mathcal{P}^1_{2,t_1} = \mathcal{P}^1_{5,t_1} \bigcup \mathcal{P}^1_{6,t_1} \bigcup \mathcal{P}^1_{7,t_1} = \{s^1_{11,t_1}\}$ and $L^1_{2,t_1} = L^1_{5,t_1} \bigcup L^1_{6,t_1} \bigcup L^1_{7,t_1} = \{B^3_{5,t,t_1}, B^1_{11,t,t_1}\}$.

The fourteenth analyzed statement instance s^1_{1,t_1} is the fourteenth element from the end of H_1. Because s^1_{1,t_1} is a controlling expression instance, we get $V^1_{1,t_1} = \varnothing$, $P^1_{1,t_1} = \{s^1_{1,t_1}\}$, $\mathcal{V}^1_{1,t_1} = V^1_{1,t,t_1} = V^1_{2,t_1} = \{s^1_{8,t_1}\}$, $\mathcal{P}^1_{1,t_1} = P^1_{1,t,t_1} = P^1_{2,t_1} \setminus B^1_{1,t,t_1} = \{s^1_{5,t_1}, s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}\}$ and $L^1_{1,t_1} = B^1_{1,f,t_1}$ according to Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3.

Similar to the above procedure, we can continue to calculate the above first five impact sets for all statement instances generated during the executions of test case 2 and test case 3, respectively. Thus far, steps 1–5 are completed, and their final results are shown in Table 3.

After calculating the first five impact sets of each statement instance, based on Table 3 and Formulas (1), (4), (7), (9) and (11), we can get the first five impact sets of each statement, as shown in Table 4. The corresponding impact factors are shown in Table 5.

After calculating the first five impact factors of each statement, we calculate the execution number of each statement, as shown in Table 6.

Finally, similar to Example 13, we use the Formula (13) to calculate the information hidden factor of each statement. The calculation process are shown in Table 7.

4.2. Computational Complexity Analysis

Through the analysis of computational complexity, we can draw the following conclusion: Compared with the time required for direct mutation testing, the time used to calculate all statement features can be neglected. The computation time of statement features consists of two parts. The first part of time overhead is used to calculate the value impact factor, path impact factor, generalized value impact factor, generalized path impact factor and potential impact factor. The calculation of these features is relatively complex. The second part of time overhead is used to calculate the other two statement features. The calculation of these two features is relatively simple. Therefore, we can approximatively consider the first part of time overhead as the total time overhead for computing all statement features. In this situation, to prove the conclusion, we only need to prove that the first part of time overhead is much lower than that used to directly execute mutation testing. We can get this conclusion from the following four steps:

In the first steps, we first suppose the time overhead that the computer spends to executes one statement once is T_0. According the Section 4.1, we can get the following conclusion: the time overhead used to compute a factor of a statement instance is also roughly equal to T_0. If a statement instance has at least one impact successor, then we use the formulas (2), (5), (8), (10) and (12) to calculate its first five impact factors, respectively. Otherwise, this statement does not have any impact successors, and we use the method in the Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3 to calculate them, respectively.

Whichever method is used, the calculation is simple. Therefore, we can consider that the time overhead used to compute an impact factor of the statement instance is roughly equal to T_0.

Table 3. The first five impact sets of all statement instances in Program 1.

Statement Instance	Direct Impact Successors	Value Impact Set	Path Impact Set	Generalized Value Impact Set	Generalized Path Impact Set	Latent Impact Set
s^1_{13,t_1}	$\varnothing$	s^1_{13,t_1}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^1_{11,t_1}	$\varnothing$	$\varnothing$	s^1_{11,t_1}	s^1_{13,t_1}	$\varnothing$	B^1_{11,t,t_1}
s^1_{9,t_1}	$\varnothing$	$\varnothing$	s^1_{9,t_1}	s^1_{13,t_1}	$\varnothing$	B^1_{9,t,t_1}
s^1_{8,t_1}	$\varnothing$	s^1_{8,t_1}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^3_{5,t_1}	$\varnothing$	$\varnothing$	s^3_{5,t_1}	$\varnothing$	$\varnothing$	B^3_{5,t,t_1}
s^2_{7,t_1}	s^3_{5,t_1}	$\varnothing$	s^3_{5,t_1}	$\varnothing$	$\varnothing$	B^3_{5,t,t_1}
s^2_{6,t_1}	$s^1_{8,t_1}, s^1_{11,t_1}$	s^1_{8,t_1}	s^1_{11,t_1}	s^1_{13,t_1}	$\varnothing$	B^1_{11,t,t_1}
s^2_{5,t_1}	$\varnothing$	$\varnothing$	s^2_{5,t_1}	s^1_{8,t_1}	s^1_{11,t_1}	$\varnothing$
s^1_{7,t_1}	$s^2_{5,t_1}, s^2_{6,t_1}, s^2_{7,t_1}$	s^1_{8,t_1}	$s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}$	$s^1_{8,t_1}, s^1_{13,t_1}$	s^1_{11,t_1}	$B^3_{5,t,t_1}, B^1_{11,t,t_1}$
s^1_{6,t_1}	s^2_{6,t_1}	s^1_{8,t_1}	s^1_{11,t_1}	s^1_{13,t_1}	$\varnothing$	B^1_{11,t,t_1}
s^1_{5,t_1}	$\varnothing$	$\varnothing$	s^1_{5,t_1}	s^1_{8,t_1}	s^1_{11,t_1}	$\varnothing$
s^1_{4,t_1}	s^1_{6,t_1}	s^1_{8,t_1}	s^1_{11,t_1}	s^1_{13,t_1}	$\varnothing$	B^1_{11,t,t_1}
s^1_{2,t_1}	$s^1_{5,t_1}, s^1_{6,t_1}, s^1_{7,t_1}$	s^1_{8,t_1}	$s^1_{5,t_1}, s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}$	$s^1_{8,t_1}, s^1_{13,t_1}$	s^1_{11,t_1}	$B^3_{5,t,t_1}, B^1_{11,t,t_1}$
s^1_{1,t_1}	$\varnothing$	$\varnothing$	s^1_{1,t_1}	s^1_{8,t_1}	$s^1_{5,t_1}, s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}$	B^1_{1,f,t_1}
s^1_{12,t_2}	$\varnothing$,	s^1_{12,t_2}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^1_{11,t_2}	$\varnothing$,	$\varnothing$	s^1_{11,t_2}	s^1_{12,t_2}	$\varnothing$	B^1_{11,f,t_2}
s^1_{9,t_2}	$\varnothing$	$\varnothing$	s^1_{9,t_2}	s^1_{12,t_2}	$\varnothing$	B^1_{9,t,t_2}
s^1_{8,t_2}	$\varnothing$	s^1_{8,t_2}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^1_{5,t_2}	$\varnothing$	$\varnothing$	s^1_{5,t_2}	$\varnothing$	$\varnothing$	B^1_{5,t,t_2}
s^1_{4,t_2}	$s^1_{8,t_2}, s^1_{11,t_2}$	s^1_{8,t_2}	s^1_{11,t_2}	s^1_{12,t_2}	$\varnothing$	B^1_{11,f,t_2}
s^1_{3,t_2}	s^1_{5,t_2}	$\varnothing$	s^1_{5,t_2}	$\varnothing$	$\varnothing$	B^1_{5,t,t_2}
s^1_{1,t_2}	$\varnothing$	$\varnothing$	s^1_{1,t_2}	$\varnothing$	s^1_{5,t_2}	B^1_{5,t,t_2}
s^1_{10,t_3}	$\varnothing$	s^1_{10,t_3}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^1_{9,t_3}	$\varnothing$	$\varnothing$	s^1_{9,t_3}	s^1_{10,t_3}	$\varnothing$	B^1_{9,f,t_3}
s^1_{8,t_3}	$\varnothing$	s^1_{8,t_3}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^3_{5,t_3}	$\varnothing$	$\varnothing$	s^3_{5,t_3}	$\varnothing$	$\varnothing$	B^3_{5,t,t_3}
s^2_{7,t_3}	s^3_{5,t_3}	$\varnothing$	s^3_{5,t_3}	$\varnothing$	$\varnothing$	B^3_{5,t,t_3}
s^2_{6,t_3}	s^1_{8,t_3}	s^1_{8,t_3}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^2_{5,t_3}	$\varnothing$	$\varnothing$	s^2_{5,t_3}	s^1_{8,t_3}	$\varnothing$	$\varnothing$
s^1_{7,t_3}	$s^2_{5,t_3}, s^2_{6,t_3}, s^2_{7,t_3}$	s^1_{8,t_3}	s^2_{5,t_3}, s^2_{5,t_3}	s^1_{8,t_3}	$\varnothing$	B^3_{5,t,t_3}
s^1_{6,t_3}	s^2_{6,t_3}	s^1_{8,t_3}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^1_{5,t_3}	$\varnothing$	$\varnothing$	s^1_{5,t_3}	s^1_{8,t_3}	$\varnothing$	$\varnothing$
s^1_{4,t_3}	s^1_{6,t_3}	s^1_{8,t_3}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s^1_{3,t_3}	$s^1_{5,t_3}, s^1_{6,t_3}, s^1_{7,t_3}$	s^1_{8,t_3}	$s^1_{5,t_3}, s^2_{5,t_3}, s^3_{5,t_3}$	s^1_{8,t_3}	$\varnothing$	B^3_{5,t,t_3}
s^1_{1,t_3}	$\varnothing$	$\varnothing$	s^1_{1,t_3}	s^1_{8,t_3}	$s^1_{5,t_3}, s^2_{5,t_3}, s^3_{5,t_3}$	B^1_{1,t,t_3}

Table 4. The first five impact sets of all statements in Program 1.

Statement instance	Value Impact Set	Path Impact Set	Generalized Value Impact Set	Generalized Path Impact Set	Latent Impact Set
s_1	$\varnothing$	$s^1_{1,t_1}, s^1_{1,t_2}, s^1_{1,t_3}$	s^1_{8,t_1}, s^1_{8,t_3}	$s^1_{5,t_1}, s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}$ $,s^1_{5,t_2}, s^1_{5,t_3}, s^2_{5,t_3}, s^3_{5,t_3}$	B^1_{1,f,t_1} $,B^1_{5,t,t_2}, B^1_{1,t,t_3}$
s_2	s^1_{8,t_1}	$s^1_{5,t_1}, s^2_{5,t_1}, s^3_{5,t_1}, s^1_{11,t_1}$	$s^1_{8,t_1}, s^1_{13,t_1}$	s^1_{11,t_1}	$B^3_{5,t,t_1}, B^1_{11,t,t_1}$
s_3	s^1_{8,t_3}	$s^1_{5,t_2}, s^1_{5,t_3}, s^2_{5,t_3}, s^3_{5,t_3}$	s^1_{8,t_3}	$\varnothing$	$B^1_{5,f,t_2}, B^3_{5,t,t_3}$
s_4	s^1_{8,t_1}, s^1_{8,t_2} $,s^1_{8,t_3}$	$s^1_{11,t_1}, s^1_{11,t_2}$	$s^1_{13,t_1}, s^1_{12,t_2}$	$\varnothing$	$B^1_{11,t,t_1}, B^1_{11,f,t_2}$
s_5	$\varnothing$	$s^3_{5,t_1}, s^2_{5,t_1}, s^1_{5,t_1}, s^1_{5,t_2}$, $s^3_{5,t_3}, s^2_{5,t_3}, s^1_{5,t_3}$	s^1_{8,t_1}, s^1_{8,t_3}	s^1_{11,t_1}	$B^3_{5,t,t_1}, B^1_{5,t,t_2}, B^3_{5,t,t_3}$
s_6	s^1_{8,t_1}, s^1_{8,t_3}	s^1_{11,t_1},	s^1_{13,t_1}	$\varnothing$	B^1_{11,t,t_1}
s_7	s^1_{8,t_1}, s^1_{8,t_3}	$s^3_{5,t_1}, s^2_{5,t_1}, s^1_{11,t_1}, s^3_{5,t_3}$, s^2_{5,t_3}	$s^1_{8,t_1}, s^1_{13,t_1}, s^1_{8,t_3}$	s^1_{11,t_1}	$B^3_{5,t,t_1}, B^1_{11,t,t_1}, B^3_{5,t,t_3}$
s_8	$s^1_{8,t_1}, s^1_{8,t_2}, s^1_{8,t_3}$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s_9	$\varnothing$	$s^1_{9,t_1}, s^1_{9,t_2}, s^1_{9,t_3}$	$s^1_{13,t_1}, s^1_{12,t_1}, s^1_{10,t_3}$	$\varnothing$	$B^1_{9,t,t_1}, B^1_{9,t,t_2}, B^1_{9,f,t_3}$
s_{10}	s^1_{10,t_3}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s_{11}	$\varnothing$	$s^1_{11,t_1}, s^1_{11,t_2}$	$s^1_{13,t_1}, s^1_{12,t_2}$	$\varnothing$	$B^1_{11,t,t_1}, B^1_{11,f,t_2}$
s_{12}	s^1_{12,t_2}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
s_{13}	s^1_{13,t_1}	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Table 5. The first five impact factors of all statements in Program 1.

Statement	Value Impact Factor	Path Impact Factor	Generalized Value Impact Factor	Generalized Path Impact Factor	Latent Impact Factor
s_1	0	3	2	8	3
s_2	1	4	2	1	2
s_3	1	4	1	0	2
s_4	3	2	2	0	2
s_5	0	7	2	1	3
s_6	2	1	1	0	1
s_7	2	5	3	1	3
s_8	3	0	0	0	0
s_9	0	3	3	0	3
s_{10}	1	0	0	0	2
s_{11}	0	2	2	0	2
s_{12}	1	0	0	0	0
s_{13}	1	0	0	0	0

Table 6. The execution number of each statement in Program 1.

s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}
3	1	2	3	7	4	4	3	3	1	2	1	1

Table 7. Computation for information hidden factor of each statement in Program 1.

Statement	Ratio (fac = 6 class 1)	Ratio (fac = 1 class 2)	Information Hidden Factor
s_1	2/3	1/3	$-\left(\frac{2}{3}\right)\log_2\left(\frac{2}{3}\right)-\left(\frac{1}{3}\right)\log_2\left(\frac{1}{3}\right)=0.9182$
s_2	1/1		$-\left(\frac{1}{1}\right)\log_2\left(\frac{1}{1}\right)=0$
s_3	1/2	1/2	$-\left(\frac{1}{2}\right)\log_2\left(\frac{1}{2}\right)-\left(\frac{1}{2}\right)\log_2\left(\frac{1}{2}\right)=1.0$
s_4	2/3	1/3	$-\left(\frac{2}{3}\right)\log_2\left(\frac{2}{3}\right)-\left(\frac{1}{3}\right)\log_2\left(\frac{1}{3}\right)=0.9182$
s_5	2/3	1/3	$-\left(\frac{2}{3}\right)\log_2\left(\frac{2}{3}\right)-\left(\frac{1}{3}\right)\log_2\left(\frac{1}{3}\right)=0.9182$
s_6	2/2		$-\left(\frac{2}{2}\right)\log_2\left(\frac{2}{2}\right)=0$
s_7	2/2		$-\left(\frac{2}{2}\right)\log_2\left(\frac{2}{2}\right)=0$
s_8	2/3	1/3	$-\left(\frac{2}{3}\right)\log_2\left(\frac{2}{3}\right)-\left(\frac{1}{3}\right)\log_2\left(\frac{1}{3}\right)=0.9182$
s_9	2/3	1/3	$-\left(\frac{2}{3}\right)\log_2\left(\frac{2}{3}\right)-\left(\frac{1}{3}\right)\log_2\left(\frac{1}{3}\right)=0.9182$
s_{10}	1/1		$-\left(\frac{1}{1}\right)\log_2\left(\frac{1}{1}\right)=0$
s_{11}	1/2	1/2	$-\left(\frac{1}{2}\right)\log_2\left(\frac{1}{2}\right)-\left(\frac{1}{2}\right)\log_2\left(\frac{1}{2}\right)=1.0$
s_{12}		1/1	$-\left(\frac{1}{1}\right)\log_2\left(\frac{1}{1}\right)=0$
s_{13}	1/1		$-\left(\frac{1}{1}\right)\log_2\left(\frac{1}{1}\right)=0$

In the second step, we can conclude that the time overhead used to calculate all factors of all statement instances is roughly equal to five times $\sum_{g=1}^{G}\sum_{k=1}^{K} H_{gk}T_0$, where G is the total number of statements in program under testing, K is the total number of test cases in test suite, and H_{gk} is the number of times statement s_g is executed by the test case t_k. Because statement s_g generates $\sum_{k=1}^{K} H_{gk}$ execution instances, in terms of the program under testing, the total number of executed statement instances by the tests suite is $\sum_{g=1}^{G}\sum_{k=1}^{K} H_{gk}$. Combining with the conclusion in the first step, we get the conclusion: In terms of the program under testing, the time overhead for computing all features of all statement instances is roughly equal to five times $\sum_{g=1}^{G}\sum_{k=1}^{K} H_{gk}$.

In the third step, we can conclude that the time overhead for direct mutation testing is $\sum_{g=1}^{G}\sum_{k=1}^{K} n_g|P_k|T_0$, where we suppose that the statement s_g generates n_g mutants, and the test case t_k executes $|P_k|$ statement instances. In the direct mutation testing, the program under testing generates $\sum_{g=1}^{G} n_g$ mutants, each mutant is tested by the test suite, and the time overhead for the test suite to test each mutant is $\sum_{k=1}^{K}|P_k|T_0$. Therefore, the time overhead for direct mutation testing is $\sum_{g=1}^{G} n_g \times \sum_{k=1}^{K}|P_k|T_0 = \sum_{g=1}^{G}\sum_{k=1}^{K} n_g|P_k|T_0$.

In the fourth step, we compare the time overhead used to calculate all features of all statement instances and the overhead used in the direct mutation testing. The ratio of the two time overheads is $5\sum_{g=1}^{G}\sum_{k=1}^{K} H_{gk}/\sum_{g=1}^{G}\sum_{k=1}^{K} n_g|P_k|$. Because $n_g \gg 5$, $|P_k| \gg H_{gk}$, we can get the final conclusion: Compared with the time required for direct mutation testing, the time overhead used to calculate all statement features can be neglected.

5. Machine Learning Algorithms Comparison and Modelling

Taking the Brier scores as a criterion, we compared the prediction effects of the following five models on statement mutation scores: artificial neural networks (ANN), logical regression (LR), random forests (RF), support vector machines (SVM) and symbolic regression (SR). The experiment result shows the artificial neural network algorithm has the highest prediction precision.

We did not try very complex models because the model should not be too complicated. First, our sample size should not be too large. Our data records need to be extracted in real time, so that the excessively large sample size will cause the user to wait a long time. In the case of a small sample size, over-complexing models can cause over-fitting. Secondly, according to the introduction of the Section 3, we can know that the relationship between the dependent variable and each independent variable is monotonic, so we estimate that the available model should not be very complicated.

5.1. Experimental Subjects

In this paper, there are two programs under testing: schedule.c and tcas.c. We explain our experiment with schedule.c as the main part and tcas.c as the auxiliary part. The program schedule.c realizes a CPU process management, and the program tcas.c realizes an aircraft early warning system. A more specific introduction is as follows.

The program schedule.c [16] realizes a priority scheduling algorithm. A computer has only one CPU, but sometimes multiple programs simultaneously request to be executed. For solving this problem, the priority scheduling algorithm assigns each program a priority. When a program needs to use CPU, it is first stored in a queue so that the program with a higher priority gets a CPU, whereas the program with a lower priority can wait. The schedule.c consists of 73 lines of C code including one branch statement, two single-loop statements and two double-loop statements. The test cases are included in its usage instructions. We take these test cases as a test suite of schedule.c.

The program tcas.c [17] is used to avoid collision of aircraft, which consists of 135 lines of C code with 40 program branch statements and 10 compound predicates. The tcas.c is able to monitor the traffic situation around a plane and offer information on the altitude of other aircraft. It can also generate collision warnings that another aircraft is in close vicinity by calculating the vertical and horizontal distances between the two aircrafts. The Software artifact Infrastructure Repository (SIR) also supplies some types of test case suites for tcas.c. From the SIR, we randomly selected a branch coverage test suite *suite*122 as the test suite used in our experiment.

5.2. The Construction Method of Data Set

To compare the prediction accuracy of the five machine learning models, we did two experiments with schedule.c and tcas.c, respectively. No matter the experiment, the data set is created in the same way. In each experiment, the data set contains 200 data records. Each data record r_p is established with one corresponding mutant sample m_p and contains seven independent variables and one dependent variable. If m_p is generated by modifying the statement s_q, then the seven independent variables of r_p are the seven features of the statement s_q, and the dependent variable of r_p is the identification result of the mutant m_p.

We take an example to explain the construction process of a data record. We might as well assume that a mutant sample m_p is generated by modifying statement s_2 and identified by the test suite. Now, we use m_p to construct one data record r_p. Because m_p is generated by modifying s_2, the values of seven independence variables in r_p are the seven features of statement s_2, i.e., $(1, 4, 2, 1, 2, 1, 0)$ as shown in Tables 5–7. Because m_p is identified by the test suite, the value of dependence variable in r_p is 1. Therefore, the data record r_p is $(1, 4, 2, 1, 2, 1, 0, 1)$.

5.3. Performance Metrics

A model may be considered good when it is evaluated using a metric, but, at the same time, the model may be considered bad when assessed against other metrics. For this reason, we will compare a few different common evaluation metrics and decide which of them is more suitable to our statement mutation score prediction.

5.3.1. Area under Curve

The two coordinates of the receiver operating characteristic (ROC) curve represent sensitivity and specificity, respectively. Through these two indicators, the ROC curve displays the two types of errors for all possible thresholds. The area AUC under the ROC curve is the quantitative indicator commonly used to evaluate a binary classification algorithm [18].

5.3.2. Logarithmic Loss

Logarithmic Loss works by penalising the false classifications [18]. It works well for both binary classification and multi-class classification generally. For a binary classification, the logarithmic function

$$-\frac{1}{n}\sum_{i=1}^{n} I(y_i = 1)\log\left[\hat{p}(Y = 1|x_i)\right] + I(y_i = 0)\log\left[1 - \hat{p}(Y = 1|x_i)\right]$$

is often used as a classifier's loss function. Logarithmic Loss closer to 0 indicates higher accuracy for the classifier.

5.3.3. Brier Score

The basic idea of Brier score is to compute the mean squared error (MSE) between the predicted probability scores and the true class indicator [19], where the positive class is coded as $y_i = 1$, and negative class $y_i = 0$. The most common formulation of the Brier score is shown as follows:

$$BS = \frac{1}{n}\sum_{i=1}^{n}[y_i - \hat{p}(Y = y_i|x_i)]^2.$$

The Brier score is a loss function, which means the lower its value, the better the machine learning model.

5.3.4. Metric Comparison

In the cross-validation process, we choose the Brill score as the model evaluation criterion. Our purpose is only to tell our users how likely the software bug in a statement will be detected by a test suite. Therefore, AUC is not suitable for us because it is also not directly related to the predicted probability. Because the logarithmic loss function may lead to an infinite penalty, it is also not used by us. The Brier score is the good score function because it is related to the predicted probability and is bounded. For the above reasons, we take the Brier score as an evaluation criterion in the cross-validation.

5.4. Model Comparing and Tuning

Under the condition of the same partitioning of the data set, we take the Brier score as a standard to evaluate the model. In our experiment, we tune hyperparameters and compare the prediction accuracies of five machine learning models. We use the same partitioning of the data set and the repeated 5-fold cross-validation to evaluate the prediction accuracy of the models because of the two following reasons.

(1) We tune some hyperparameters to find the optimal model settings with the help of the repeated 5-fold cross-validation method. During the 5-fold cross-validation, the samples are randomly partitioned into five equally sized folds. Models are then fitted by repeatedly leaving out one of the folds. In our each experiments, our data set contains 200 data records, so that the training and validation sets contain 160 and 40 data records, respectively. However the result from cross-validation is more or less uncertain generally. Therefore, in our experiment, five repeats of 5-fold cross-validation are used to effectively reduce this uncertainty and increase the precision of the estimates. Because each

5-fold cross-validation supplies a Brier score, five repeats of 5-fold cross-validation supply 5 Brier scores. Under each candidate combination of hyperparameters, we use the average of the five Brier scores to represent the prediction effects of the corresponding model.

(2) Because the performance metric is sensitive to the data splits, we thus compare the machine learning models based on the same partitioning of the data. Otherwise, the difference in performance will come from two different sources: the differences among the data splits and the differences among the models themselves. If one model is better than the other, we don't know if all performance differences are caused by model differences.

The compared models include the logistic regression, random forest, neural network, support vector machine and symbolic regression. We use their average Brier scores to assess their prediction effects.

5.4.1. Logistic Regression

(1) Introduction to Logistic Regression

Conventional logistic regression [20,21] can predict the occurrence probability of a specific outcome. The conditional probability of a positive outcome could be expressed with the formula below:

$$p(x_i) = p(Y = 1|x_i) = \frac{1}{1 + e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_d x_d}},$$

where β_i is the coefficient for the ith feature, and d is the total number of features. $\beta_1, \beta_2, \cdots, \beta_d$ can be solved by the elastic net approach [22–24] as follows:

$$max_{\beta_0, \boldsymbol{\beta}} \frac{1}{n} \sum_{i=1}^{n} \left\{ I(y_i = 1) log p(x_i) + I(y_i = 0) log\left(1 - p(x_i)\right) \right\} - \lambda \left[(1 - \alpha) \frac{1}{2} \|\boldsymbol{\beta}\|_2^2 + \alpha \|\boldsymbol{\beta}\|_1 \right], \quad (14)$$

where

$$\|\boldsymbol{\beta}\|_2^2 = \beta_1^2 + \beta_2^2 + \cdots + \beta_d^2 \quad and \ \|\boldsymbol{\beta}\|_1 = |\beta_1| + |\beta_2| + \cdots + |\beta_d|.$$

(2) Logistic regression tuning

Glmnet [25] is an R language software package that can fit linear, logistic and multinomial, Poisson, and Cox regression models by maximizing the penalized likelihood. In order to predict the mutant score of each program statement in schedule.c, we use the ridge penalty algorithm in a glmnet software package to fit the logistic regression mode. Hence, during tuning hyper parameters, the penalized parameter α in the formula (14) is set to 0, and the penalized parameter λ is set to 10^i where i takes each integer from -7 to 7 in turn. In the cross-validation process, we use the Brier score as the model evaluation criterion. Under each penalized parameter, the five repeats of 5-fold cross-validation generate five Brier scores. We calculate the average of the five Brier scores under each candidate penalized parameter λ, so that we can use the average Brier score to represent the prediction effect of the model under the each candidate penalized parameter.

Figure 4 and Table 8 show the average Brier Score under each candidate value of the penalized parameter λ . In Figure 4, the profile shows a decrease in the average Brier score until the penalized value λ is 10^{-2}. Therefore, the numerically optimally value of the penalized parameter is 10^{-2}.

Table 8. Average Brier scores for the logistic regression model.

λ	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}
Mean	0.1095	0.1062	0.1075	0.1016	0.0955
λ	10^{-2}	10^{-1}	1	10^{1}	10^{2}
Mean	0.0950	0.1090	0.1321	0.1374	0.1380
λ	10^{3}	10^{4}	10^{5}	10^{6}	10^{7}
Mean	0.1381	0.1381	0.1381	0.1381	0.1381

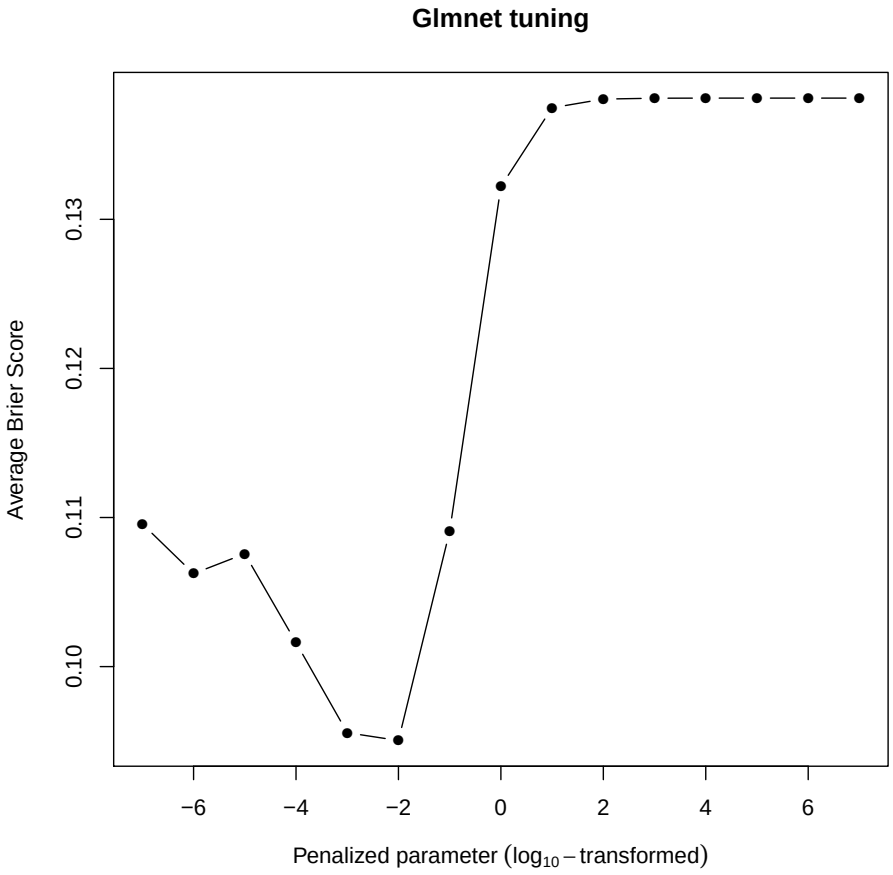

Figure 4. The performance profile of the logistic regression for predicting the statement mutation scores.

5.4.2. Random Forests

(1) Introduction to Random Forest

The random forest model [26,27] can work for regression tasks and classification tasks generally. It is a tree-based model consisting of multiple decision trees. Each decision tree is created on an independent and random sample taken from the training data set.

The decision tree algorithm [18,28] is a top-down "greedy" approach that partitions the dataset into smaller subsets. This algorithm has a tree-like structure that predicts the value of a target variable based on several input variables. At each decision node, the features are split into two subsets and this process is repeated until the number of data in the splits falls below some threshold. According to the target variable's type, decision trees can be divided into regression trees and classification trees. The purpose of classification tree is to classify, and its target variable takes discrete values. The purpose of regression trees is to build a regression model, its target variable takes continuous values.

For regression, the regression tree algorithm begins with the entire data set S and searches every distinct value of every independent variable to find the appropriate independent variable and split its value that partitions the data into two subsets (S_1 and S_2) such that the overall sums of squares error

$$SSE = \sum_{i \in S_1} (y_i - \overline{y}_1)^2 + \sum_{i \in S_2} (y_i - \overline{y}_2)^2 \tag{15}$$

are minimized, where $\overline{y}_1$ and $\overline{y}_2$ are the averages of the outcomes within subsets S_1 and S_2, respectively. Then, within each of subsets S_1 and S_2, this method searches again for the independent variable and splits its value that best reduces SSE. Because of the recursive splitting nature of regression trees, this method is also known as recursive partitioning.

For classification, the aim of classification trees is to partition the data into smaller, more homogeneous groups. Homogeneity in this context means that the nodes of the split are more pure. This purity is usually quantified by the entropy or Gini index. For the two-class problem, the Gini index for a given node is defined as

$$p_1(1 - p_1) + p_2(1 - p_2), \tag{16}$$

where p_1 and p_2 are the probabilities of Class 1 and Class 2, respectively.

In order to make a prediction for a given observation, the regression tree first analyzes which class this observation belongs to, and then takes the mean of the training data in the class as the prediction of this observation. When random forest algorithms are used, the result of regression question can be obtained by averaging predictions across all regression trees, and the result of the classification question can be obtained by a majority vote across all classification trees, respectively. The generalization error of a random forest depends on the errors of individual trees and the correlation between the trees.

(2) Random forest tuning

The randomForest package [29] implements the random forest algorithm in the R environment. We use this software to predict statement mutation scores generated when the test suite executes on the program schedule.c. Because the statement mutation score can be considered as the probability of positive class in binary classification, we denote the positive class and negative class as 1 and 0, respectively, and let Random Forests run under regression mode to predict the probability of a positive class [30]. In order to obtain a good prediction model, the different hyper parameter combinations are tried. The most important hyper parameter is mtry, which is the number of independent variables randomly selected at each split. In our experiment, we tried multiple candidate values of mtry (from 1 to 7). The other important tuning parameter is ntree, which is the number of bootstrap samples in the random forest algorithm. In theory, the performance of a random forest model should be a monotonic function of the number of trees (ntree). However, when ntree is greater than a certain number, the performance of a random forest model can only improve slowly. In our experiment, ntree is set to 1000. Under each candidate value of the parameter mtry, we calculate the average of the five Brier scores generated from the five repeats of 5-fold cross-validation. Furthermore, we use these averages to express the prediction effects of the random forest model under different candidate values of hyperparameter mtry. Figure 5 and Table 9 show the average Brier score under each candidate value of the hyperparameter mtry. As shown in Figure 5, the average Brier scores show a U shape, whose minimum value occurs in $mtry = 3$. Therefore, 3 is the optimal value of mtry.

Table 9. Average Brier scores for the random forest model.

mtry	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Mean	0.1166	0.0923	0.0888	0.0897	0.0914	0.0928	0.0925

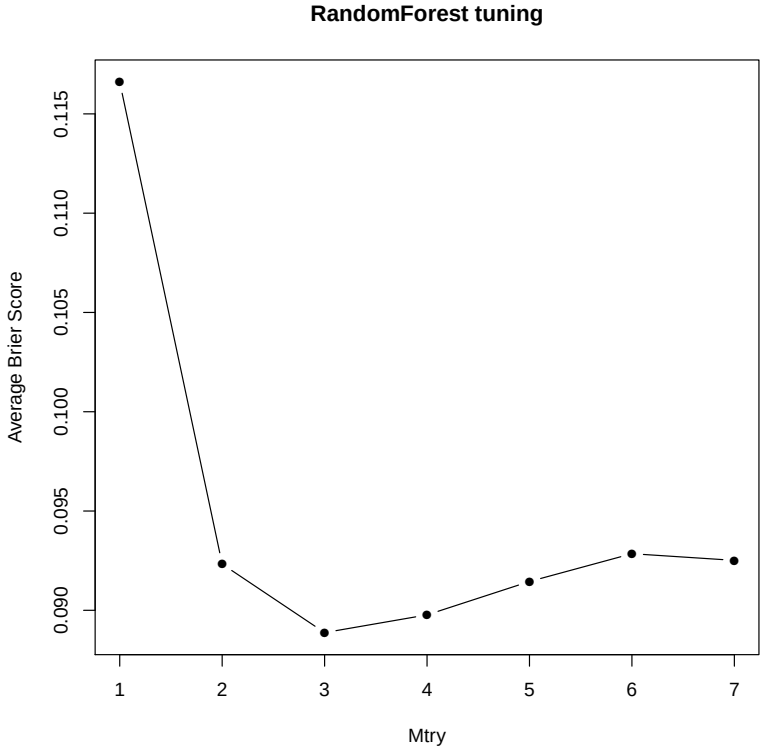

Figure 5. The performance profile of the random forest for predicting the statement mutation scores.

5.4.3. Artificial Neural Networks

(1) Introduction to neural network

Neural networks [18,31] can be used not only for regression but also for classification. The outcome of a neural network is modeled by an intermediary set of unobserved variables called hidden units. The simplest neural network architecture is the single hidden layer feed-forward network.

The working process of the single hidden layer feed-forward neural network is as follows. During the entire work of the neural network, all input neurons representing the original independent variables $x_1, x_2, \cdots, x_s$ are first activated through the sensors perceiving the environment. Next, inside each hidden unit h_k, all original independent variables are linearly combined to generate

$$u_k(\boldsymbol{x}) = \beta_{0k} + \sum_{j=1}^{s} \beta_{jk} x_j, \tag{17}$$

where $k = 1, 2, \cdots, r$ and r is the number of the hidden units. Then, by a nonlinear function g_k, $u_k(\boldsymbol{x})$ is typically transformed into the output of hidden unit h_k as follows:

$$g_k(\boldsymbol{x}) = \frac{1}{1 + e^{-u_k(\boldsymbol{x})}}. \tag{18}$$

(i) When treating the neural network as a regression model, all $g_k(\boldsymbol{x})$ are linearly combined to form the output of neural network:

$$f(\boldsymbol{x}) = \gamma_0 + \sum_{k=1}^{r} \gamma_k g_k(\boldsymbol{x}). \tag{19}$$

All of the parameters $\boldsymbol{\beta}$ and γ can be solved by minimizing the the penalized sum of the squared residuals:

$$\sum_{i=1}^{n}(y_i - f(\boldsymbol{x}_i))^2 + \lambda \sum_{k=1}^{r}\sum_{j=0}^{s}\beta_{jk}^2 + \lambda \sum_{k=0}^{r}\gamma_k^2, \tag{20}$$

where $f(\boldsymbol{x}_i)$ and y_i are the predicted result and the actual result related to the ith observed data, respectively.

(ii) Neural networks can also be used for classification. Unlike neural networks for regression, an additional nonlinear transformation is used on the linear combination of the outputs of hidden units.

When the neural network is used for binary classification, it uses

$$f^*(x) = \frac{1}{1+e^{-f(\boldsymbol{x})}} = \frac{1}{1+e^{-(\gamma_0+\sum_{k=1}^{r}\gamma_k g_k(\boldsymbol{x}))}} \tag{21}$$

to predict the class probability. The estimation of the parameters γ and $\boldsymbol{\beta}$ can be solved by minimizing the penalized cross-entropy

$$-\sum_{i=1}^{n} y_i \log f(\boldsymbol{x}_i) + (1-y_i)\log(1-f(\boldsymbol{x}_i)) + \lambda \sum_{k=1}^{r}\sum_{j=0}^{s}\beta_{jk}^2 + \lambda \sum_{k=0}^{r}\gamma_k^2, \tag{22}$$

where y_i is the $0/1$ indicator for the positive class. The neural network algorithm can also be used for multi-class classification. In this situation, the softmax transform outputs the probability that the sample $\boldsymbol{x}$ belongs to the lth class. Except the single hidden layer feed-forward network, there are many other types of models. For example, the famous deep learning approaches consist of multiple hidden layers.

(2) Neural network tuning

As we said before, the our model must not be too complicated, so we select R package nnet [32] to predict the statement mutation scores of the test suite on schedule.c. The software package nnet implements a feed-forward neural network with a single hidden layer. The λ and r in formula (22) represent the weight decay and the number of units in the hidden layer, respectively. They are denoted as *decay* and *size* in nnet package, respectively. Therefore, *decay* is the regularization parameter to avoid over-fitting.

In our experiment, *size* is set in turn to each integer value between one and then. At the same time, the *decay* was set to 10^i where i takes each integer value from -4 to 5 in turn.

Figure 6 and Table 10 show the average Brier scores under the each candidate combinations of *size* and *decay*. From them, we can know that the optimal combination of the weight decay and hidden unit number is $decay = 10^{-2}$ and $size = 8$ because, at this time, the minimum average Brier score appears.

Table 10. Average Brier scores for neural network models.

Size \ *Decay*	10^{-4}	10^{-3}	10^{-2}	10^{-1}	1	10^{1}	10^{2}	10^{3}	10^{4}	10^{5}
1	0.1180	0.0984	0.0928	0.0968	0.1111	0.1397	0.1956	0.2419	0.2491	0.2499
2	0.0915	0.0869	0.0883	0.0876	0.1104	0.1379	0.1889	0.2404	0.2489	0.2498
3	0.0953	0.0890	0.0865	0.0867	0.1099	0.1371	0.1835	0.2390	0.2488	0.2498
4	0.0897	0.0889	0.0863	0.0881	0.1094	0.1367	0.1791	0.2375	0.2486	0.2498
5	0.0874	0.0890	0.0865	0.0880	0.1093	0.1366	0.1753	0.2361	0.2484	0.2498
6	0.0881	0.0869	0.0865	0.0881	0.1093	0.1365	0.172	0.2348	0.2483	0.2498
7	0.0896	0.0887	0.0862	0.0878	0.1093	0.1364	0.1694	0.2334	0.2481	0.2498
8	0.0884	0.0878	0.0856	0.0882	0.1093	0.1364	0.1670	0.2321	0.2479	0.2497
9	0.0884	0.0871	0.0869	0.0881	0.1093	0.1364	0.1649	0.2308	0.2478	0.2497
10	0.0870	0.0880	0.0867	0.0878	0.1093	0.1365	0.1631	0.2295	0.2476	0.2497

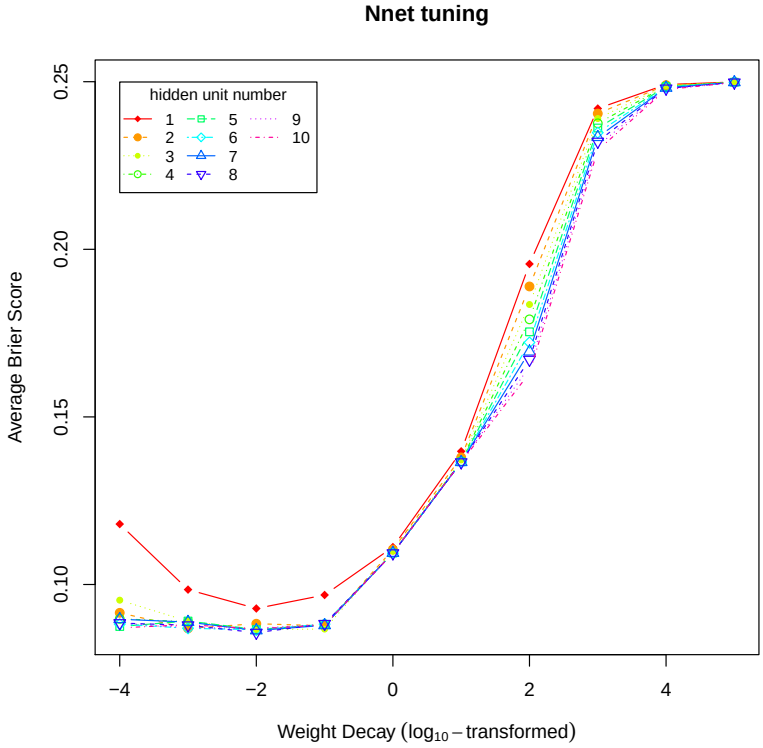

Figure 6. The performance profile of the neural network for predicting the statement mutation scores.

5.4.4. Support Vector Machines

(1) Introduction to support vector machine

Given a set of n training instances $\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}$, the goal of support vector machine is to find a hyperplane that separates the positive and the negative training instances with the maximum margin and minimum misclassification error. The training of support vector machine is equivalent to solving the following optimization problem:

$$\begin{aligned} &\min_{w,b,\zeta_i} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\zeta_i \\ &\textit{subject to} \quad y_i(\boldsymbol{w}^T\boldsymbol{x_i} + b) \geq 1 - \zeta_i \\ &\qquad\qquad \zeta_i \geq 0, \quad i = 1, 2, \cdots, n, \end{aligned}$$

where $\boldsymbol{w}$ is the normal vector of the maximum-margin hyperplane $\boldsymbol{w}^T\boldsymbol{x} + b = 0$, C is the regularization parameter, ζ_i indicates a non-negative slack variable to tolerant some training data falling in the wrong side of the hyperplane, and b is a bias. The parameter C specifies the cost of a violation to the margin. When C is small, the margins will be wide and many support vectors will be on the margin or will violate the margin. When C is large, the margins will be narrow and there will be few support vectors on the margin or violating the margin.

The maximum-margin hyperplane can be obtained by solving the above problem. Given new data $\boldsymbol{x}$, $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$ represents the signed distance between x and the hyperplane. We can classify the new data $\boldsymbol{x}$ based on the sign of $f(\boldsymbol{x})$.

If the original problem is stated in a finite-dimensional space, it often happens that the sets to discriminate are not linearly separable. For solving this problem, a support vector machine maps the original finite-dimensional space into a higher-dimensional space, making the separation easier.

Let $\phi(\boldsymbol{x})$ denote the vector after $\boldsymbol{x}$ mapping. In this higher-dimensional space, the optimization problem can be rewritten into

$$\min_{w,b,\zeta_i} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\zeta_i \tag{23}$$

$$subject\ to \quad y_i(\boldsymbol{w}^T\phi(\boldsymbol{x_i}) + b) \geq 1 - \zeta_i \tag{24}$$

$$\zeta_i \geq 0, \quad i = 1, 2, \cdots, n \tag{25}$$

or expressed in the dual form

$$\min_{\boldsymbol{\alpha}} \quad -\sum_{i=1}^{n}\alpha_i + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j k(\boldsymbol{x_i}, \boldsymbol{x_j}) \tag{26}$$

$$subject\ to \quad \sum_{i=1}^{n}\alpha_i y_i = 0 \tag{27}$$

$$0 \leq \alpha_i \leq C, i = 1, 2, \cdots, n, \tag{28}$$

where $k(\boldsymbol{x_i}, \boldsymbol{x_j}) = \phi(\boldsymbol{x_i})^T\phi(\boldsymbol{x_j})$ defines the kernel function greatly reducing the computational cost.

By solving the above optimization problem, the optimal α_i^* and b^* can be obtained. Therefore, the maximum-margin hyperplane in the higher-dimensional space is

$$\boldsymbol{w^*}^T\phi(\boldsymbol{x}) + b = \sum_{i=1}^{n}\alpha_i^* y_i\phi(\boldsymbol{x_i})^T\phi(\boldsymbol{x}) + b^* = \sum_{i=1}^{n}\alpha_i^* y_i k(\boldsymbol{x_i}, x) + b^*. \tag{29}$$

The kernel trick allows the support vector machine model to produce extremely flexible decision boundaries. The most common kernel functions are listed in Table 11:

Table 11. Kernel functions.

Name	Expression	Parameter
linear kernel	$k(x_i, x_j) = \boldsymbol{x_i}^T\boldsymbol{x_j}$	
polynomial kernel	$k(x_i, x_j) = (\gamma\boldsymbol{x_i}^T\boldsymbol{x_j} + \theta)^d$	γ, θ, d
radial kernel	$exp(-\sigma\|x_i - x_j\|^2)$	σ

The original SVM can be used for classification and regression without probability information. To solve this problem, Platt [33] proposed to use a logistic function to convert the decision value from a binary support vector machine to a probability. Formally, the probability of data $\boldsymbol{x_i}$ being a positive instance is defined as follows:

$$P(y_i = 1|\boldsymbol{x_i}) = \frac{1}{1 + exp(Af(\boldsymbol{x_i}) + B)},$$

where $f(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{x}) + b$ is the maximum-margin hyperplane. The parameters A and B are derived by minimizing the negative log-likelihood of the training data:

$$-\sum_{i=1}^{n}[t_i log(P(y_i = 1|\boldsymbol{x_i})) + (1 - t_i)log(1 - P(y_i = 1|\boldsymbol{x_i}))]$$

where

$$t_i = \begin{cases} \frac{n_+ + 1}{n_+ + 2} & if\ y_i = 1, \\ \frac{1}{n_- + 2} & if\ y_i = -1. \end{cases}$$

n_+ denotes the number of positive training instances (i.e., $y_i = 1$), and n_- denotes the number of negative training instances (i.e., $y_i = -1$). Newton's method with backtracking is a commonly used

approach to solve the above optimization problem [34] and is implemented in LibSVM. Besides the binary classification, the support vector machine can also compute the class probabilities for the multi-class problem using one-against-one (i.e., pairwise) approach [35] .

(2) Support vector machine tuning

Support vector machine algorithms are provided in the software package kernlab [36] written in the R language. We built the support vector machine based on the radial basis kernel function provided by this package. A radial basis kernel function maps the independent variables to an infinite-dimensional space. The regularization parameter C in formula (23) is called cost parameter in kernlab. A smaller C results in a smoother decision surface and a larger C results in a flexible model that strives to classify all training data correctly. The radial basis kernel function in kernlab package is shown in Table 11, where the parameter σ represents the inverse kernel width. A larger σ means a narrower radial basis kernel function.

When we use kernlab to predict the statement mutation scores of schedule.c, we hope to get the Brier score as small as possible by tuning C and σ. For this purpose, we first set the parameter σ to the median of $\|\boldsymbol{x} - \boldsymbol{x}'\|^2$ [18,37,38]. Next, let the parameter C take respectively as 2^{-5}, 2^{-3}, 2^{-1}, 2^1, 2^3, 2^5, 2^7, 2^9, 2^{11}, 2^{13} and 2^{15}. Then, under each candidate value of C, we use the five repeats of 5-fold cross-validation to calculate the average Brier scores.

Figure 7 and Table 12 show the average Brier score generated by five repeats of 5-fold cross-validation at each candidate value of C. As shown in Figure 7, although there was a relatively large fluctuation, the average Brier score shows a general trend of first decreasing and then rising. From this figure, we can know that $C = 2^{-1}$ is the optimally value of the regularization parameter. At this time, the average Brier score reaches a minimum 0.0933.

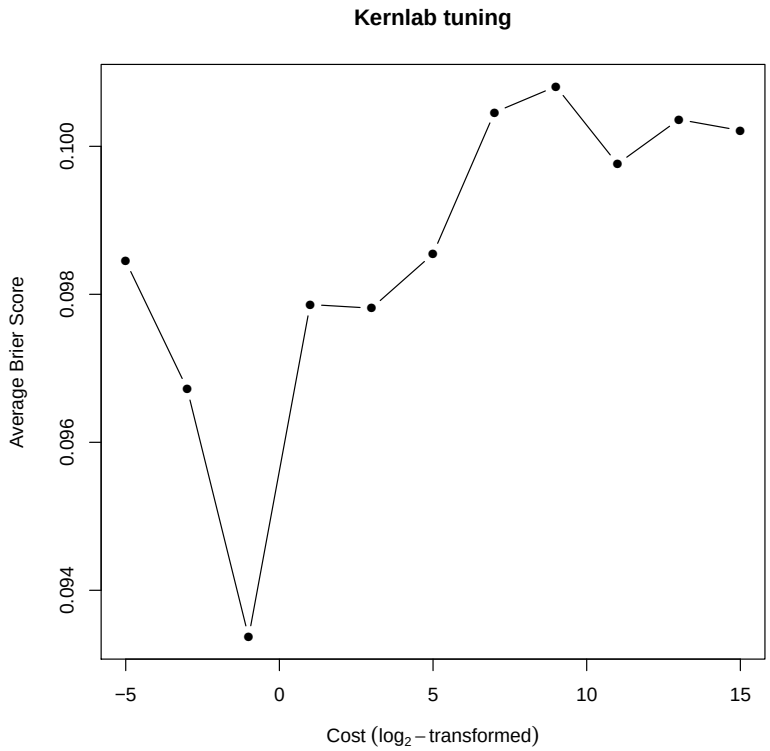

Figure 7. The performance profile of the support vector for predicting the statement mutation scores.

Table 12. Resampled Brier score for the support vector machine model.

C	2^{-5}	2^{-3}	2^{-1}	2^{1}	2^{3}	2^{5}	2^{7}	2^{9}	2^{11}	2^{13}	2^{15}
Mean	0.0984	0.0967	0.0933	0.0978	0.0978	0.0985	0.1004	0.1008	0.0997	0.1003	0.1002

5.4.5. Symbolic Regression

(1) Introduction to symbolic regression

Symbolic regression can also be called function modeling. Based on the given data, it can automatically find the functional relationship, such as $2x^3 + 5$, $\cos(x) + 1/e^y$, etc., between independent variables and dependent variables.

Throughout the modeling process, a function model $f(\boldsymbol{x})$ is always coded as a symbolic expression tree. The input of symbolic regression is a data set, and the genetic programming method is often used to determine $f(\boldsymbol{x})$. The genetic programming constantly changes an old function model into a new better fitted one by selecting the function with the better fitness value. A possible and frequently used fitness function is the average squared difference between the values predicated by $f(\boldsymbol{x})$ and the actually observed values $\boldsymbol{y}$ as follows:

$$MSE(f(\boldsymbol{x}), \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}(f(\boldsymbol{x}_i) - y_i)^2.$$

Mutation operations and crossover operations are the two important ways to change function model $f(\boldsymbol{x})$. A mutation operation directly changes a symbolic expression sub tree, and a crossover operation cuts a symbolic expression sub tree and replace it with a sub tree in another symbolic expression tree.

(2) Symbolic regression tunning

The symbolic regression tool rgp [39] is an implementation of genetic programming methods in the R environment. We use rgp to predict the statement mutation scores of schedule.c. In our symbolic regression experiment, the most basic mathematical operators are set to the operators +, −, *, /, sin. An important tuning parameter in rgp is *populationSize*, which means the number of individuals included in a population, and is set to 100 in our experiment.

Another important tuning parameter is the number of evolution generations. Too few evolutionary generations produce an under-fitting, whereas too many evolutionary generations produce an over-fitting. We did a grid search to determine the optimal number of the generations, which minimizes the average Brier score. Because we need to complete the model fitting in a relatively short time, the number of evolution generations cannot be set too large. In our experiment, the candidate number of evolution generations is set to 3, 6, 9, 12, 15, 18, respectively. The five repeats of five-fold cross-validation are used to calculate the evolution effects (i.e., the average Brier scores) under each candidate number of evolution generations.

As shown in Table 13 and Figure 8, the average Brier scores oscillated down. In the 12th generation evolution, the smallest average Brier score 0.1504 appeared.

Table 13. Average Brier scores for the symbolic regression.

Generation	**3**	**6**	**9**	**12**	**15**	**18**
Mean	0.1640	0.1546	0.1548	0.1504	0.1527	0.1515

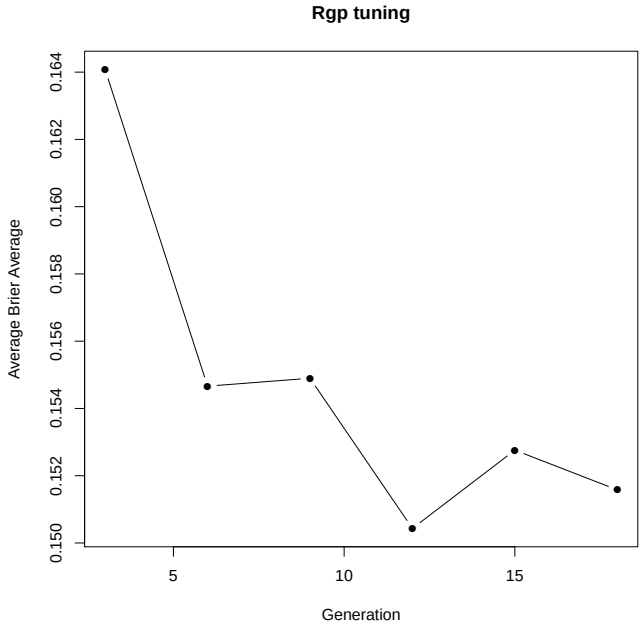

Figure 8. Rgp tuning.

5.4.6. Comparing Models

Once the hyper parameters in the above five models have been determined for the above five models, we face the question: how do we choose among multiple models? The logistic regression model is used to set the baseline performance because its mathematical expression is simple and operation speed is fast. If other predictive models do not surpass it, the logistic regression model is used in future actual forecasting.

The boxplot in Figure 9 shows, under the condition that the Brier score is the standard, the neural network does the best job about predicting the statement mutation scores. The second best is the random forest model, which is a little better than the support vector machine model. The logistic model is second to last and greatly exceeded the symbolic regression.

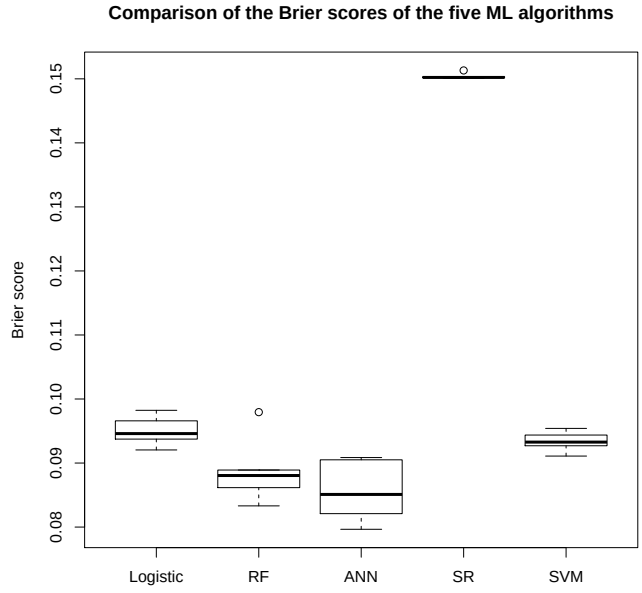

Figure 9. Comparison of the Brier scores of the five machine learning models for schedule.c.

5.4.7. Testing Predictions in Practice

According to Figure 9 and Tables 8–10, 12 and 13, we know that, in the process of repeated cross-validation, the average Brier scores of the logistic regression, random forest, neural network, support vector machine and symbolic regression are 0.0950, 0.0888, 0.0856, 0.0933 and 0.1504, respectively. Therefore, the neural network is the best model because its average Brier score is lower than other models. To further demonstrate the predictive effect of the neural network model on the schedule.c, we did the two following things. Firstly, we apply the neural network model, whose hyper-parameters have been tuned according to the method in Section 5.4.3, to predict the statement mutation scores of schedule.c. Under the condition that the schedule.c is used as the experiment subject, we calculate the mean absolute error between all statement mutation scores obtained by the neural network prediction and all real statement mutation scores. The experiment result shows the mean absolute error reaches 0.1205. Secondly, we randomly select 34 statements in schedule.c, and their two kinds of statement mutation scores are shown in Figure 10. In this figure, the horizontal coordinate represents the real statement mutation score, and the vertical coordinate represents the statement mutation score predicted by the neural network model. Each circle represents a statement, and the distance between each short dashed line and diagonal line is 0.1. From this figure, we can see that more than 60% of the statements are located between the two short dashed lines.

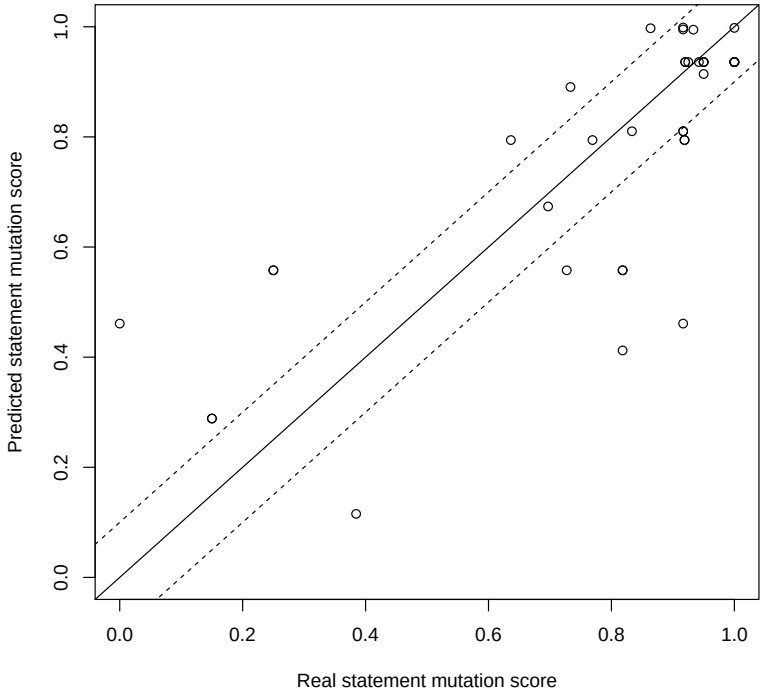

Figure 10. Comparing the real statement mutation scores and the predicted statement mutation scores in schedule.c by the artificial neural network.

5.5. Further Confirmation of the Optimization Model

To further confirm that the prediction effect of the neural network is the best, we compared the five machine learning models again under the condition that the program tcas.c is used as the experimental subject. In the process of repeated cross-validation, the average Brier score of the neural network model reaches 0.1164. The average Brier scores of the logistic regression, the support vector machine, the random forest and the symbolic regression are 0.1233, 0.1249, 0.1289 and 0.1373, respectively. Therefore, the neural network is once again considered the best model because its average Brier score is lower than other models. To further demonstrate the predictive effect of the neural network model on the tcas.c, we apply the neural network model, whose hyper-parameters have been tuned according to the method in Section 5.4.3, to predict the statement mutation scores of tcas.c. Under the condition that the

tcas.c is used as the experiment subject, the mean absolute error between real statement mutation scores and the statement mutation scores predicted by the tuned neural network reaches 0.1198. In order to illustrate the prediction results of the neural network more vividly, we randomly selected 31 statements in the program tcas.c. Their real statement mutation scores and the corresponding predicted mutation scores are shown in Figure 11.

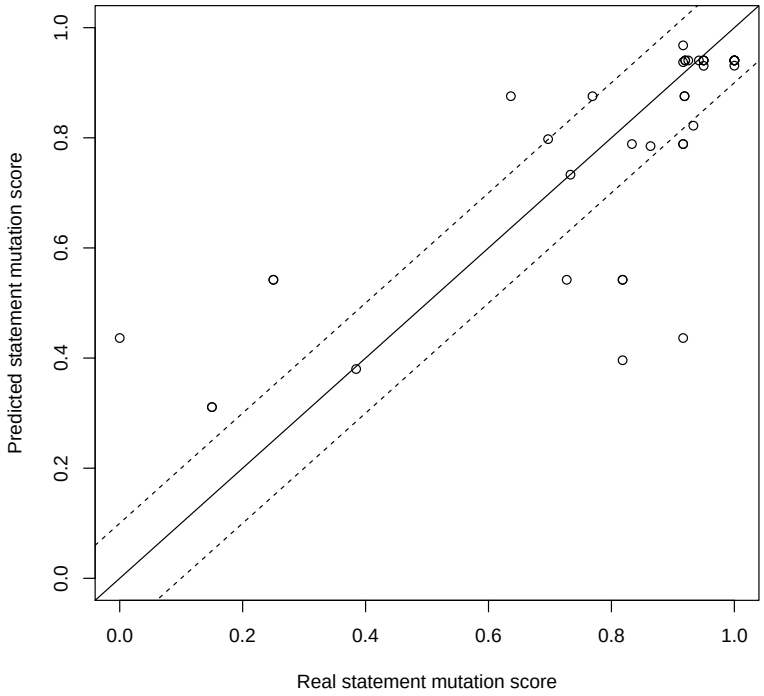

Figure 11. Comparing the real statement mutation scores and the predicted statement mutation scores in tcas.c by the artificial neural network.

Through the above analysis, we can see that, whether the experiment subject is schedule.c or tcas.c, the average Brier scores of the neural network are both the minimums. Thus, we recommend the single hidden layer feedforward neural network as the best model. In the two experiments, the mean absolute error between the statement mutation scores predicted by the neural network model and the real statement mutation scores both approximately reach 0.12.

6. Structure of the Automated Prediction Tool

The work process of our automatic analysis tool consists of the five parts, as shown in Figure 12: extracting the features of the statements in the program under testing, generating mutants, executing test suite on the each mutants, establishing the neural network model, and predicting the statement mutation scores.

In the first part, we extract the features of statements in the program under testing. First, we execute each test case and construct its execution impact graph with the open source software giri [40]. Giri was originally a dynamic program slice tool and is currently modified by us. Next, we traverse the statement instances in reverse order of the execution history of the test cases. Whenever we visit a statement instance, we compute its features. After calculating the features of each statement instance, we calculate the features of each program statement according to the corresponding the statement instances.

In the second part, we generate mutants. We first build a mutation operator set. In our experiments, the mutation operator set consists of the 22 mutation operators, which exist in the open source mutant generate tool ProteumIM2.0 [12]. These operators include u-Cccr, u-OEAA, u-OEBA,

u-OESA, u-CRCR, u-Ccsr, u-OAAN, u-OABN, u-OALN, u-OARN, u-OASN, u-OCNG, u-OLAN, u-OLBN, u-OLLN, u-OLNG, u-OLRG, u-OLSN, u-ORBN, u-ORLN, u-ORRN and u-ORSN. Next, we use these mutation operators to randomly construct 200 mutants, each of which is the program with a software bug.

In the third part, we execute the test suite on each mutant and record the corresponding identification result.

In the fourth part, we take the features of the mutant as independent variables and the identification result of the mutant as dependent variables to construct the prediction model with the neural network.

In the fifth part, we predict the mutation scores of each program statement with the constructed model.

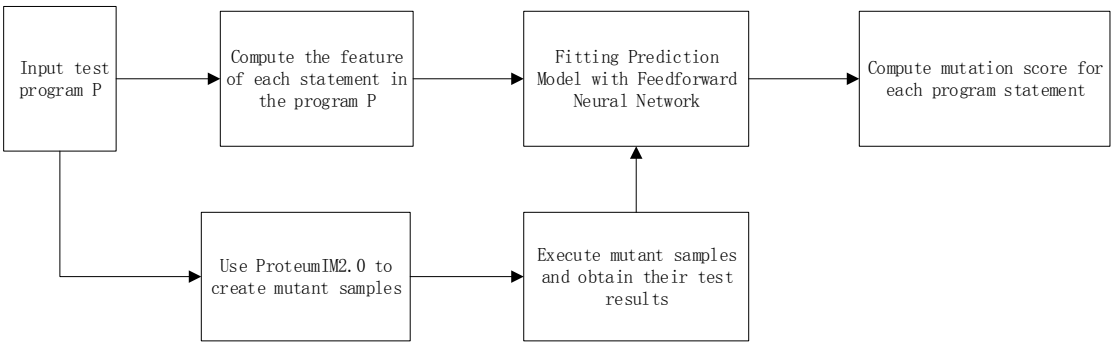

Figure 12. The structure of Automated Analysis Tool.

7. Conclusions

In this paper, we predicted statement mutation scores while using a single hidden layer feedforward neural network and seven statement features. As analyzed in Section 5, each experimental result shows that the neural network is the best prediction model from the standpoint of the mean absolute error. The experimental results on two c programs demonstrate that our method can directly predict statement mutant scores approximately. The experiment results also show that the seven statement features that represent the dynamic program execution and testing process can basically reflect the impact of statements on program output.

However, two shortcomings need to be improved. Firstly, a part of statement features weakly related to program outputs still need to be discovered. If the real mutation score of a statement is low, then this statement usually only has some statement features weakly related to program outputs. In this case, the prediction effect of my model is not good because we only found a part of weakly relevant statement features, and the other part of the weakly correlated sentence features still need to be discovered.

Secondly, in this paper, we assume the controlling expression has no side effect. However, in a few cases, a controlling expression has a side effect. In this case, the execution instance of the controlling expression may have some impact successors. For example, if there is a controlling expression if(x > y++) in the original program, then the execution result of y++ will be changed when it is executed a test case, so that it impacts subsequent statement instances containing y variables. In this situation, the methods mentioned in Sections 3.1.4, 3.2.4, 3.3.2, 3.4.2 and 3.5.3 are no longer applicable, and the corresponding algorithm needs to be redesigned.

In the future, we also plan to predict the statement mutation scores with the prediction model established by other programs. In this case, the users can train a prediction model with the data records from other programs beforehand. Using this pre-trained model, users can directly predict the statement mutation scores of the current program.

Author Contributions: Conceptualization, L.T.; methodology, L.T.; Supervision, Y.W. and Y.G.; writing—original draft preparation, L.T.

Funding: This work was supported by the National Natural Science Foundation of China (No. U1736110), the National Natural Science Foundation of China (No. 61702044), and the Fundamental Research Funds for the Central Universities (No. 2017RC27).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

V_g	value impact set of the statement s_g
$x_{vi}(s_g)$	value impact factor of the statement s_g
V^h_{g,t_k}	value impact set of the statement instance s^h_{g,t_k}
V^l_{r,z,t_k}	value impact set of the branch instance B^l_{r,z,t_k}
P_g	path impact set of the statement s_g
$x_{pi}(s_g)$	path impact factor of the statement s_g
P^h_{g,t_k}	path impact set of the statement instance s^h_{g,t_k}
P^l_{r,z,t_k}	path impact set of the branch instance B^l_{r,z,t_k}
$\mathcal{V}_g$	generalized value impact set of the statement s_g
$x_{gvi}(s_g)$	generalized value impact factor of the statement s_g
$\mathcal{V}^h_{g,t_k}$	generalized value impact set of the statement instance s^h_{g,t_k}
$\mathcal{P}_g$	generalized path impact set of the statement s_g
$x_{gpi}(s_g)$	generalized path impact factor of the statement s_g
$\mathcal{P}^h_{g,t_k}$	path impact set of the generalized statement instance s^h_{g,t_k}
L_g	latent impact set of statement s_g
$x_{li}(s_g)$	latent impact factor of statement s_g
L^h_{g,t_k}	latent impact set of statement instance s^h_{g,t_k}
$x_{ih}(s_g)$	information hidden factor of statement s_g

References

1. Andrews, J.H.; Briand, L.C.; Labiche, Y. Is mutation an appropriate tool for testing experiments? In Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, 15–21 May 2005; pp. 402–411.
2. DeMillo, R.A.; Lipton, R.J.; Sayward, F.G. Hints on test data selection: Help for the practicing programmer. *Computer* **1978**, *11*, 34–41. [CrossRef]
3. Mirshokraie, S.; Mesbah, A.; Pattabiraman, K. Efficient JavaScript mutation testing. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 18–22 March 2013; pp. 74–83.
4. Jia, Y.; Harman, M. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **2011**, *37*, 649–678. [CrossRef]
5. Frankl, P.G.; Weiss, S.N.; Hu, C. All-uses vs mutation testing: An experimental comparison of effectiveness. *J. Syst. Softw.* **1997**, *38*, 235–253. [CrossRef]
6. Maldonado, J.C.; Delamaro, M.E.; Fabbri, S.C.; da Silva Simão, A.; Sugeta, T.; Vincenzi, A.M.R.; Masiero, P.C. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation Testing for the New Century*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 113–116.
7. Acree, A.T., Jr. *On Mutation*; Technical Report; Georgia Inst of Tech Atlanta School of Information and Computer Science: Atlanta, GA, USA, 1980.
8. Zhang, L.; Hou, S.S.; Hu, J.J.; Xie, T.; Mei, H. Is operator-based mutant selection superior to random mutant selection? In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa, 1–8 May 2010; Volume 1, pp. 435–444.
9. Hussain, S. Mutation Clustering. Master's Thesis, Kings College London, London, UK, 2008.

10. Gligoric, M.; Zhang, L.; Pereira, C.; Pokam, G. Selective mutation testing for concurrent code. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, Lugano, Switzerland, 15–20 July 2013; pp. 224–234.
11. Offutt, A.J.; Rothermel, G.; Zapf, C. An experimental evaluation of selective mutation. In Proceedings of the 1993 15th International Conference on Software Engineering, Baltimore, MD, USA, 17–21 May 1993; pp. 100–107.
12. Zhang, J.; Zhang, L.; Harman, M.; Hao, D.; Jia, Y.; Zhang, L. Predictive mutation testing. *IEEE Trans. Softw. Eng.* **2018** . [CrossRef]
13. Jalbert, K.; Bradbury, J.S. Predicting mutation score using source code and test suite metrics. In Proceedings of the First, International Workshop on Realizing AI Synergies in Software Engineering, Zurich, Switzerland, 5 June 2012; pp. 42–46.
14. Goradia, T. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis, Cambridge, MA, USA, 28–30 June 1993; Volume 18, pp. 171–181.
15. C programming Language Standard—C99. Available online: https://en.wikipedia.org/wiki/C99 (accessed on 19 August 2019).
16. The Program schedule.c. Available online: https://www.thecrazyprogrammer.com/2014/11/c-cpp-program-forpriority-scheduling-algorithm.html (accessed on 19 August 2019).
17. The Program tcas.c. Available online: https://sir.csc.ncsu.edu/php/showfiles.php (accessed on 19 August 2019).
18. Kuhn, M.; Johnson, K. *Applied Predictive Modeling*; Springer: Berlin/Heidelberg, Germany, 2013; Volume 26.
19. Brier, G.W. Verification of forecasts expressed in terms of probability. *Mon. Weather Rev.* **1950**, *78*, 1–3. [CrossRef]
20. Hosmer, D.W., Jr.; Lemeshow, S.; Sturdivant, R.X. *Applied Logistic Regression*; John Wiley & Sons: Hoboken, NJ, USA, 2013; Volume 398.
21. Harrell, F.E. *Regression Modeling Strategies*; Springer: Berlin/Heidelberg, Germany, 2001.
22. Hastie, T.; Tibshirani, R.; Wainwright, M. *Statistical Learning with Sparsity: The Lasso and Generalizations*; Chapman and Hall/CRC:Boca Raton, FL, USA, 2015.
23. Friedman, J.; Hastie, T.; Tibshirani, R. Regularization paths for generalized linear models via coordinate descent. *J. Stat. Softw.* **2010**, *33*, 1. [CrossRef] [PubMed]
24. Tibshirani, R. Regression Shrinkage and Selection Via the Lasso. *J. R. Stat. Soc. Ser. B* **1994**, *58*, 267–288. [CrossRef]
25. Glmnet. Available online: https://CRAN.R-project.org/package=glmnet (accessed on 19 August 2019).
26. Breiman, L. *Some Infinity Theory for Predictor Ensembles*; Technical Report 579; Statistics Dept. UCB: Berkeley, CA, USA, 2000.
27. Breiman, L. *Consistency for a Simple Model of Random Forests*; Technical Report (670); University of California at Berkeley: Berkeley, CA, USA, 2004.
28. Quinlan, J.R. Simplifying decision trees. *Int. J. Hum.-Comput. Stud.* **1999**, *51*, 497–510. [CrossRef]
29. The randomForest. Available online: https://cran.r-project.org/web/packages/randomForest/index.html (accessed on 19 August 2019).
30. Li, C. Probability Estimation in Random Forests. Master's Thesis, Department of Mathematics and Statistics, Utah State University, Logan, UT, USA, 2013.
31. Demuth, H.B.; Beale, M.H.; De Jess, O.; Hagan, M.T. *Neural Network Design*, 2nd ed.; Martin Hagan: Stillwater, OK, USA, 2014.
32. R Package nnet. Available online: https://CRAN.R-project.org/package=nnet (accessed on 19 August 2019).
33. Platt, J.C. Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods. In *Advances in Large Margin Classifiers*; MIT Press: Cambridge, MA, USA, 1999; pp. 61–74.
34. Lin, H.T.; Lin, C.J.; Weng, R.C. A note on Platt's probabilistic outputs for support vector machines. *Mach. Learn.* **2007**, *68*, 267–276. [CrossRef]
35. Hsu, C.W.; Lin, C.J. A comparison of methods for multiclass support vector machines. *IEEE Trans. Neural Netw.* **2002**, *13*, 415–425. [PubMed]
36. Software Package Kernlab. Available online: https://CRAN.R-project.org/package=kernlab (accessed on 19 August 2019).

37. Caputo, B.; Sim, K.; Furesjo, F.; Smola, A. Appearance-based object recognition using SVMs: Which kernel should I use? In Proceedings of the NIPS Workshop on Statistical Methods for Computational Experiments in Visual Processing and Computer Vision, Whistler, BC, Canada, 12–14 December 2002; Volume 2002.
38. Karatzoglou, A.; Smola, A.; Hornik, K.; Zeileis, A. kernlab—An S4 package for kernel methods in R. *J. Stat. Softw.* **2004**, *11*, 1–20. [CrossRef]
39. The Symbolic Regression Tool Rgp. Available online: http://www.rdocumentation.org/packages/rg (accessed on 19 August 2019).
40. Sahoo, S.K.; Criswell, J.; Geigle, C.; Adve, V. Using likely invariants for automated software fault localization. In Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, 16—20 March 2013; ACM SIGARCH Computer Architecture News; Volume 41, pp. 139–152.