Hindawi
Scientific Programming
Volume 2021, Article ID 8325417, 20 pages
https://doi.org/10.1155/2021/8325417

Research Article

An Empirical Assessment and Validation of Redundancy Metrics Using Defect Density as Reliability Indicator

Dalila Amara,[1] **Ezzeddine Fatnassi**,[1,2] **and Latifa Ben Arfa Rabai**[1]

[1]*Université de Tunis, Institut Supérieur De Gestion De Tunis, SMART Lab, Tunis, Tunisia*
[2]*Insitut des Hautes Etudes de Tunis, Tunis, Tunisia*

Correspondence should be addressed to Dalila Amara; dalilaa.amara@gmail.com

Received 7 December 2019; Revised 22 December 2020; Accepted 30 January 2021; Published 19 February 2021

Academic Editor: Přemek Brada

Software metrics which are language-dependent are proposed as quantitative measures to assess internal quality factors for both method and class levels like cohesion and complexity. The external quality factors like reliability and maintainability are in general predicted using different metrics of internal attributes. Literature review shows a lack of software metrics which are proposed for reliability measurement and prediction. In this context, a suite of four semantic language-independent metrics was proposed by Mili et al. (2014) to assess program redundancy using Shannon entropy measure. The main objective of these metrics is to monitor program reliability. Despite their important purpose, they are manually computed and only theoretically validated. Therefore, this paper aims to assess the redundancy metrics and empirically validate them as significant reliability indicators. As software reliability is an external attribute that cannot be directly evaluated, we employ other measurable quality factors that represent direct reflections of this attribute. Among these factors, defect density is widely used to measure and predict software reliability based on software metrics. Therefore, a linear regression technique is used to show the usefulness of these metrics as significant indicators of software defect density. A quantitative model is then proposed to predict software defect density based on redundancy metrics in order to monitor software reliability.

1. Introduction

Software quality is one of the main concerns of all organizations using software systems. According to [1], it means the capability of a software process to produce software product with good quality. In both software process and product, quality is described through a set of attributes or characteristics that may be internal or external. Concerning product quality, reliability is identified as one of the most important software quality attributes. Different techniques including fault prevention, removal, fault tolerance, and fault forecasting are defined to produce reliable software systems [2, 3]. Furthermore, literature shows that software reliability like other external attributes is difficult to measure directly [4–6]. Thus, it is generally measured and predicted based on other quality attributes like defect density and fault-proneness identified as direct reflections of reliability [7]. These attributes are directly measurable through software metrics and widely used to validate different suites software metrics. Examples include complexity, cohesion, defect density, and fault-proneness [6, 8–10].

Software metrics are quantitative measures used to make evaluation, improvement, and prediction of different software quality attributes [11–15]. Numerous software metrics (or combination of them) are proposed to assess internal attributes like complexity, cohesion and size, faults count, and defect density. Concerning external attributes like reliability and maintainability, they are generally expressed through internal attributes using the metrics proposed to measure them. For instance, reliability can be reflected through cohesion, coupling, and complexity using their related metrics [5].

Continuous research studies focusing on the assessment and prediction of software reliability are needed. Therefore, a suite of four semantic metrics (language-independent) are proposed in [16] to assess programs redundancy in order to monitor their reliability. However, these metrics are theoretically presented and manually computed for basic arithmetic operations. Furthermore, it is unclear how they can be computed for more complex programs (functions and classes). Using basic programs containing simple operations, we cannot assume that the metrics are related to programs' redundancy. Thus, the authors need to perform the empirical assessment based on more complex software projects to convincingly show that the proposed metrics are related to program redundancy. On the contrary, concerning the hypothesis of these metrics as useful indicators of software reliability, there are no concrete examples or evidence of how these metrics can be linked to reliability. Therefore, empirical studies focusing on concrete relationship between these metrics and various reliability factors are required.

The authors in [17, 18] focused on the empirical validation of the redundancy metrics as measures of programs mutants' rate. The manual assessment of these metrics for basic arithmetic operations and certain types of computation-based programs represent the main limitation of these studies.

Software metrics validation consists on identifying their utility by studying the relationship between these metrics and a quality attribute. The most used quality attributes as reliability indicators are defect density and fault-proneness [6, 8–10, 14, 19]. This relationship can be exploited to propose a quantitative model helping to predict a quality attribute through these metrics [9, 14].

To perform this validation, we used an empirical dataset including the different metrics values and defect density attribute. Metrics values are computed using a set of classes taken from Apache projects. Defect density attribute is obtained using the fault injection procedure on the same classes. Using the linear regression technique, this relationship is exploited to propose a quantitative model that reflects programs defect density through these metrics. This model is also useful to predict defect density for further datasets.

This paper is organized as follows: Section 2 presents the related works and motivation. Section 3 discusses the proposed empirical assessment approach. Section 4 outlines the empirical validation of redundancy metrics as defect density predictors. Section 5 analyses the experimental results. Conclusion and perspectives are reported in Section 6.

2. Related Works and Motivation

This section focuses on software reliability measurement, software redundancy, and entropy concepts. Furthermore, it describes the redundancy metrics suite.

2.1. Software Reliability. Software reliability is one of the most important software quality attributes [11, 20]. It is defined from two main sides:

(i) The mathematical side considers reliability as the probability of failure-free operation for a specified period of time in a specified environment [3].

(ii) The broader side considers reliability as the degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time [2]. It is described by combining different subcharacteristics which are maturity, availability, fault tolerance, and recoverability [3].

Different techniques, namely, fault prevention, fault elimination, fault tolerance, and fault forecasting, are defined to reflect software reliability [11, 21, 22]. Software prevention mechanisms help to prevent faults; however, they cannot guarantee the avoidance of all software faults [23]. Consequently, further protective mechanisms like fault removal are required. For [22, 24], fault removal techniques are important and critical for software reliability. However, these techniques cannot guarantee the elimination of all faults since they are based on software testing and formal inspections which in turn have their problems. Besides, Mili and Tchier [22] argued that while removing one fault, other ones can appear. They also note that some faults cannot be removed since they are not sensitized or masked (actual states are equal to the stated ones), but these faults can cause software failure. Consequently, fault tolerance techniques and forecasting are required to enhance the performance of fault avoidance and removal to reduce faults and to improve reliability [23–25].

Bansyia et al. [19] noted that reliability is one of the high-level quality attributes that are abstract concepts and cannot be directly observed and measured. Different models based on direct metrics are proposed to predict various external quality attributes including reliability. Chidamber and Kemerer (C&K) [19, 26–29] proposed a suite of metrics used for reliability prediction. The proposed reliability prediction models use software metrics (called independent variables) to evaluate measurable reliability attributes (called dependent variable) like defect density, fault-proneness, and defect count [4, 8–10, 14, 30]. Mili et al. [16] also proposed a suite of four metrics to monitor programs reliability based on their redundancy.

2.2. Software Redundancy. Redundancy concept was firstly used in hardware systems. It provides more physical copies of components, i.e., processors and memories, to improve their reliability [31]. Different forms of redundancy were defined in software systems like information redundancy (code redundancy), functional redundancy, and time redundancy [22, 32, 33]:

(i) Information redundancy indicates the excess of information expressed in Shannon bits used to represent the state of a program [34].

(ii) Functional redundancy consists on using the same program specification to generate different algorithm or program versions performing the same functionality [33].

(iii) Time redundancy represented by the time was used to repeat the execution of the failed process [24, 31].

Literature review shows that redundancy can be used in different applications. For instance, in 1976, redundancy was exploited through N-version programming technique to achieve reliable systems [35]. This technique was also used to compare the probability of failure between single and N-version systems. Furthermore, in [21], the authors noted that redundancy is useful for self-checking programs. In addition, in 2003, Lyu [25] showed that redundancy structured in mutation testing is exploited through the injection of faults in numerous program versions. Furthermore, in 2015, redundancy was also used to identify the semantic similarity of code fragments [36].

The redundancy metrics proposed by Mili et al. [16, 22] assess program information redundancy provided by the different states of the program [34]. Program states are the set of variables manipulating related data and reflect the uncertainty about the outcome of these variables. To develop a better understanding of the redundancy metrics, we present the terminology related to program states:

(i) State space is defined by a set of values that the declared program variables may take [16].

(ii) Initial state space is the state of the program (function/class) represented by the input variables [22].

(iii) Current state (actual state) is the set of states that the program may be in at any given point [16, 37].

(iv) Final state space is the state of the program that is produced by the output variables [22].

(v) State redundancy arises when the representation of the program state allows a wider range of values than needed to represent the different states [32].

These definitions are illustrated in the following example:

Let the following program:

```
{
int s, x, result; //state space of the program
int s = 2; //initial state of the program
s = s + 1; //internal state 1 of the program
s = 2 * s; //internal state 2 of the program
s = s%3; //internal state 3 of the program
s = s + 12; //final state of the program
}
```

2.3. Entropy Concept.

2.3. Entropy Concept. As mentioned, the redundancy metrics were defined based on the Shannon entropy measure. Thus, for a given random variable X that takes its values in a finite set, its entropy is the function denoted by $H(X)$ and defined by [34, 38]

$$H(X) = -\sum_{x_i \in X} p(x_i)\log(p(x_i)), \tag{1}$$

where $p(x_i)$ is the probability of the variable $X = x_i$.

Intuitively, this function measures (in bits) the uncertainty pertaining to the outcome of X and takes its maximum value $H(X) = \log_2(N)$ when the probability distribution is uniform. N is the cardinality of X.

2.4. Redundancy Metrics Suite. Redundancy metrics were defined based on Shannon entropy measure of programs code. Mili et al. [16] supported the hypothesis of uniform probability distribution in order to perform the analytical evaluation of these metrics. The authors in [16, 17] argued that, for a random variable X that takes values of a 32-bit integer, N equals 2^{32} and $\log_2(N)$ is then merely equal to 32 bits. They assumed the uniform probability throughout; then, the entropy of any program variable is basically the number of bits in that variable. The entropy of a given variable X of value N is given as

$$H(X) = \log_2(N). \tag{2}$$

Four metrics were defined which are initial state redundancy, final state redundancy, functional redundancy, and noninjectivity. To define these metrics, the following assumptions were made by Mili et al. [16]:

(i) Probability distribution of the different variables is uniform.

(ii) Variables are 32 bits size.

(iii) Metrics were computed at method level. These methods manipulate input and output' variables of integer type. This means programs with input states were represented by the declared variables and output states were represented by the modified states of these variables.

2.4.1. Initial and Final State Redundancy Metrics. We recall that the state of a given program g is defined by its declared variables. It is very common to declare the range of values related to these variables much more than it is really required. For instance, the age of an employee is generally declared as an integer variable type. However, only a restrict range, i.e., between 0 and 120, is really required. This means that 7 bits are sufficient to store the age variable, but the typical 32 bits size of an integer variable is used. The unused bits measure the code redundancy. Thus, state redundancy represents the gap between the declared state and the actual state (really used) of this program [16, 18, 22].

Mathematically, let S be the declared state of the program (its variables) and σ be its actual state (the actual values of these variables), and then, the state redundancy is given by the difference between their respective entropies denoted by $H(S)$ and $H(\sigma)$. The program moves from its initial states (σ_1) to its final states (σ_f), and then, two state redundancy measures, namely, initial state redundancy (ISR) and final state redundancy (FSR), were defined by the following equations [18] (Table 1):

$$\text{ISR} = \frac{H(S) - H(\sigma_1)}{H(S)}, \tag{3}$$

$$\text{FSR} = \frac{H(S) - H(\sigma_f)}{H(S)}. \tag{4}$$

Table 1: Entropy for basic data type.

Data type	Entropy
Boolean	1
Byte	8
Char, short	16
Int, float	32
Double, long	64

(i) ISR is the gap between the declared state and the initial state of the program.

(ii) FSR is the gap between the declared state and the final state of the program.

(iii) $H(S)$ is the entropy of the individual variable declarations (number of bits required to store these variables) of the program. For instance, for Java language, the entropy of variable declarations of basic data types is illustrated in Table 1.

(iv) σ_1 and σ_f are, respectively, the initial and final actual states of g.

(v) $H(\sigma_1)$ and $H(\sigma_f)$ are, respectively, the entropies of these states.

(vi) To compute SR metric (ISR and FSR), each data type has to be mapped to its width in bits.

Example 1. Consider the following program:

int x, y, z; //the program state is represented by x, y, and z variables

{

int $x = 21$; //initial state of x

int $y = 90$; //initial state of y

$z = (x + y)/2$//final state

}

The declared space of this program is defined by three integer variables: x, y, and z; hence, using the metrics definitions provided in [18], we have stated that $H(S) = 96$ bits since 3 integer variables are used. Its initial state is defined by three variables: x, y, and z. The input variables x and y require, respectively, 5 and 7 bits to be stored. The output variable z has a free range (32 bits). Hence, $H(\sigma_1) = 5 + 7 + 32 = 44$ bits. For the final state, it is determined by the value of the variable z; then, $H(\sigma_f) = 6$ bits, and we find

$$\text{ISR} = \frac{96-44}{96} = 0.54,$$
$$\text{FSR} = \frac{96-6}{96} = 0.93. \tag{5}$$

2.4.2. Functional Redundancy (FR) Metric. According to [18], the functional redundancy metric is a function from initial states to final states. It reflects how initial states are mapped to final states. It also identifies how initial states are affected by input data and how final states are projected onto output data. Mathematically, for a program (function), FR is the ratio of the output data delivered by g prorated to the input data received by the program and given by

$$\text{FR} = \frac{H(Y)}{H(X)}. \tag{6}$$

(i) $H(S)$ is the entropy of the program' declared space defined by the declared variables.

(ii) $H(Y)$ is the output data delivered by the program.

(iii) $H(X)$ is the entropy of input data passed through parameters, global variables, and read statements.

Considering the same previous example, $H(S) = 96$ bits. The random variable Y is defined by the integer variable z represented by 32 bits. Then, $H(Y) = \log_2(2^{32}) = 32$ bits. $H(X)$ is the input data received by g and represented by the two integer variables x and y. Then, $H(X) = 2 * \log_2(2^{32}) = 64$, and FR is given by

$$\text{FR} = \frac{32}{64} = 0.5. \tag{7}$$

2.4.3. Noninjectivity (NI). According to [28], a major source of program (function) redundancy is its noninjectivity. An injective function is a function whose value changes whenever its argument does. A function is noninjective when it maps several distinct arguments (initial states σ_1) into the same image (final states σ_f). Mathematically, NI is the conditional entropy of the initial state given the final state: if we know the final state, how much uncertainty do we have about the initial state? According to [34], this conditional entropy equals the difference between entropies of these two states. Hence, NI was defined as

$$\text{NI} = \frac{H(\sigma_1/\sigma_f)}{H(\sigma_1)} = \frac{H(\sigma_1) - H(\sigma_f)}{H(\sigma_1)}. \tag{8}$$

Using the previous example, NI is equal to $44 - 6/44 = 0.86$. A sum up of the presented metrics is illustrated in Table 2.

2.5. Motivation and Objective. As mentioned, the main objective of redundancy metrics defined in [16] is to monitor product reliability. This makes them important measures since software reliability is one of the most important quality attributes. However, the different metrics composing this suite are theoretically presented and manually computed for basic arithmetic operations. Furthermore, it is unclear how they can be computed for more complex programs. With very simple operation programs, we cannot assume that the metrics are related to programs' redundancy. Therefore, we need to perform the empirical assessment based on more complex software programs (functions/classes) to convincingly show that the proposed metrics are related to program redundancy. In addition, there are no concrete examples or evidence of how these metrics can be linked to reliability. Thus, empirical studies focusing on concrete

Table 2: Sum up of redundancy metrics.

Metric	Purpose	Equation	Metric hypothesis
ISR	Measures the redundancy (excess bits) of the program' initial state	$H(S) - H(\sigma_1)/H(S)$	The proposed redundancy metrics can be exploited to monitor software reliability
FSR	Measures the redundancy (excess bits) of the program' final state	$H(S) - H(\sigma_f)/H(S)$	
FR	Measures redundancy of the program functionality	$H(Y)/H(X)$	
NI	Mapping between initial states to the same final ones	$H(\sigma_1) - H(\sigma_f)/H(\sigma_1)$	

relationship between these metrics and various reliability factors are required. Given these statements, the main objectives of this study can be stated into two parts:

(i) First, we aim to propose an approach to empirically assess the mentioned redundancy metrics. Solving this issue consists first on considering complex programs taken from real-world software projects rather than be limited to basic operations and certain types of computation-based programs [29, 39]. The basic idea is to generate an empirical database including for each program (function/class) the values of the redundancy metrics.

(ii) Second, we aim to propose an empirical validation of the proposed metrics as reliability indicators by considering defect density attribute as direct reflection of software reliability [5, 7]. The basic idea is to exploit the generated database to study the relationship between the metrics and defect density using regression techniques [8, 28, 40].

Therefore, we propose in Section 3 an empirical assessment approach to calculate the different metrics. In addition, we present in Section 4 an empirical validation approach to demonstrate the concrete relationship between these metrics and software reliability.

3. Empirical Assessment of Entropy-Based Redundancy Metrics

The proposed redundancy metrics were computed manually in [16] at function level for simple examples and for specific data type and values, i.e., greatest common division of two integer variables. Thus, in this paper, we will consider more complex examples taken from real-world software projects to automatically compute these metrics at the class level since software projects are organized in classes. Three main steps are used:

(1) Selection of software classes: in this step, we have selected different classes from which the metrics will be generated.

(2) Compute redundancy metrics: once the different classes are selected, we have used appropriate scripts to compute these metrics.

(3) Construct the database: the implementation of the two previous steps helps to obtain an empirical database that contains for each class, the values of the different metrics. The presented steps will be detailed in the following sections.

3.1. Selection of Software Classes. According to Radjenović et al. [29] and Kumar et al. [39], software repositories used to validate most of software metrics are of three main types:

(i) Private/commercial repositories: this type of repositories was used and maintained by companies within the organizational use. In these repositories, source code and other related information like fault datasets are not available [29].

(ii) Partially public repositories: in these repositories, Radjenović et al. [29] noted that only the product source code and the related software faults are available, whereas the values of software metrics are usually unavailable so, they need to be calculated from the available source code and then mapped with their fault information. Regarding to [29], the mapping may lead to biased results.

(iii) Public repositories: in these repositories, the values of software metrics and other information like software faults are usually available, and this justifies their uses in many research projects [27]. Some examples of these public repositories include PRedictOr Models In Software Engineering (PROMISE (http://promise.site.uottawa.ca/SERepository/datasets-page.html)) repository of NASA projects, Software-artifact Infrastructure Repository (SIR (http://sir.csc.ncsu.edu/portal/index.php)), and Bug Prediction Dataset (BPD (http://bug.inf.usi.ch/index.php)).

To perform the empirical assessment of redundancy metrics, we have focused on the repository containing programs of input/output type as explained above [16].

Given that computing redundancy metrics requires the availability of the source code, private and commercial repositories were not considered since the programs' source codes are not available. Therefore, we have focused on partially public and public repositories. Literature review [7, 41] shows that most of studies focused on metrics validation have used programs (classes or methods) taken from NASA projects like CM1, JM1, and KC1. In this context, the authors in [40] showed that, from 64 metrics' validation studies performed from 1991 to 2013, NASA projects were the most used (60%), followed by PROMISE repository datasets (15%) and other open-source projects (12%).

In our research project, we have proceeded as follows:

(1) First, we have used NASA then PROMISE projects for which different information are available including the values of software metrics. Furthermore, faults datasets of these projects are also available; in addition, for each class, we can identify if it is fault-free or not (true/false). However, we have decided to not use this repository as we have identified the unavailability of the source code mandatory to our study as we need to compute the redundancy metrics using this code.

(2) Next, we have focused on a set of open-source projects, some of them are not of input/output type like SIR$'$ velocity and Camel projects. Others do not include source code such as BPD repository. Literature review [39] shows that Apache Common library including different java projects with available source code of input/output type was also used to validate software metrics. Besides, in this repository, the unit tests related to the classes are available.

Consequently, to select the needed repository, we have considered Apache Commons products library which respects all our requirements and hypothesis. Then, from the selected repository, we have considered a set 43 classes (see Table 3) containing functions manipulating variables in the input and the output state.

A description of each class and its related function is available at http://commons.apache.org/proper/commons-math/javadocs/api-3.6/.

3.2. Automatic Metrics Computing and Database Construction. The process we used to compute redundancy metrics (ISR, FSR, FR and NI) is summarized in Figure 1.

Figure 1 presents the different steps used to compute redundancy metrics. These steps are the same for the different selected classes presented in Table 3. To compute these metrics, we have used the Eclipse development environment (version: Neon.3 Release (4.6.3)). As the selected source code, we have used to compute these metrics organized in classes, and we present under here how redundancy metrics are computed at the class level. The computing process was developed using the following steps.

3.2.1. Compute Program State Space $H(S)$. To compute the state space $H(S)$, we have first identified for each class the input/output functions manipulating the different states of the class variables. Next, we have computed $H(S)$ as the maximum entropy of all function variables (input/output). For a better understanding of $H(S)$ and the other metrics computing, an example of the used script is illustrated in Figure 2.

In Figure 2, $H(S)$ is computed as shown in line 33, and its value is equated with the maximum entropy of the input and output variables used in lines 24 to 31. The input data related to these variables are randomly generated as shown in lines 34 to 36.

3.2.2. Compute $H(\sigma_1)$ and ISR Metric. $H(\sigma_1)$ reflects the initial entropy of the input variables and the maximum entropy of the output ones. To compute the entropy of a variable (exact number of the used bits) presented by equation (2), a Java function called *sizeOfBits* is used. More details about this function are presented in Appendix A (see Figure 3). $H(\sigma_1)$ is computed as illustrated in lines 48 and 49 of Figure 2. Using equation (3), ISR metric value is deduced as illustrated in line 64 of Figure 2.

3.2.3. Compute $H(\sigma_f)$ and FSR Metric. As mentioned, $H(\sigma_f)$ reflects the entropy of the variables used to produce the output ones. The *sizeOfBits* function is used to provide their values as illustrated in lines 76, 78, and 79 of Figure 2. Using equation (4), the FSR metric value is deduced as illustrated in line 81 of Figure 2.

3.2.4. Compute FR and NI Metrics. To compute FR and NI metrics, we have used equations (6) and (8). We have first computed $H(Y)$ and $H(X)$ values as shown in lines 84 and 85 of Figure 2. Then, FR and NI values are deduced as illustrated in lines 86 and 88 of Figure 2.

3.2.5. Generate Metrics Values to Excel Files. We have noted that the metrics values were generated using 1000 iterations to test different possibilities of random inputs as shown in lines 19 and 20 of Figure 2. To store the values of these metrics, we have generated them in .xsl files as shown in lines 16 and 91. Then, we have computed for each class the average of the 1000 generated metrics values to construct the final database. A part of this database is illustrated in Figure 4.

The presented process can be also used to compute the redundancy metrics at a function level. Thus, an example of the used script is illustrated in Figure 5 of Appendix A. The source of the metrics generator is available at https://gitlab.com/dalilaamara/redundancymetrics.

4. Empirical Validation of Semantic Metrics as Reliability Indicators

According to [5], a valid metric is one whose values are statistically associated with a quality attribute. The empirical validation aims not only to identify the utility of a proposed metric and to make comparisons with other metrics but also to identify which metrics are not useful. Reliability can be reflected by other measurable attributes including defect density and fault-proneness. In our research, we have focused on the defect density attribute as fault-proneness attribute indicates whether a class contains faults (1) or not (0), and in our research, all classes in the constructed redundancy database contain faults.

Table 3: Selected software classes.

Class name		
Beta	EuclideanDistance	GammaDistribution
BesselJ	BinomialDistribution	GeometricDistribution
Gamma	Primes	LogisticDistribution
Erf	SmallPrimes	LevyDistribution
Vector2D	PollarRho	LaplaceDistribution
BetaDistribution	MathUtils	WeibullDistribution
CombinatoricUtils	FastMathCalc	FDistribution
Covariance	Precision	ParetoDistribution
ArithmeticUtils	Gaussian	PolynomialFunction
StatUtils	NormalDistribution	Skewness
CanberraDistance	SimpleRegression	Variance
ChebyshevDistance	SpearmansCorrelation	PolynomialsUtils
EarthMoversDistance	PearsonsCorrelation	Percentile
ManhattanDistance	ChiSquareDistribution	StepFunction
		DfpDec

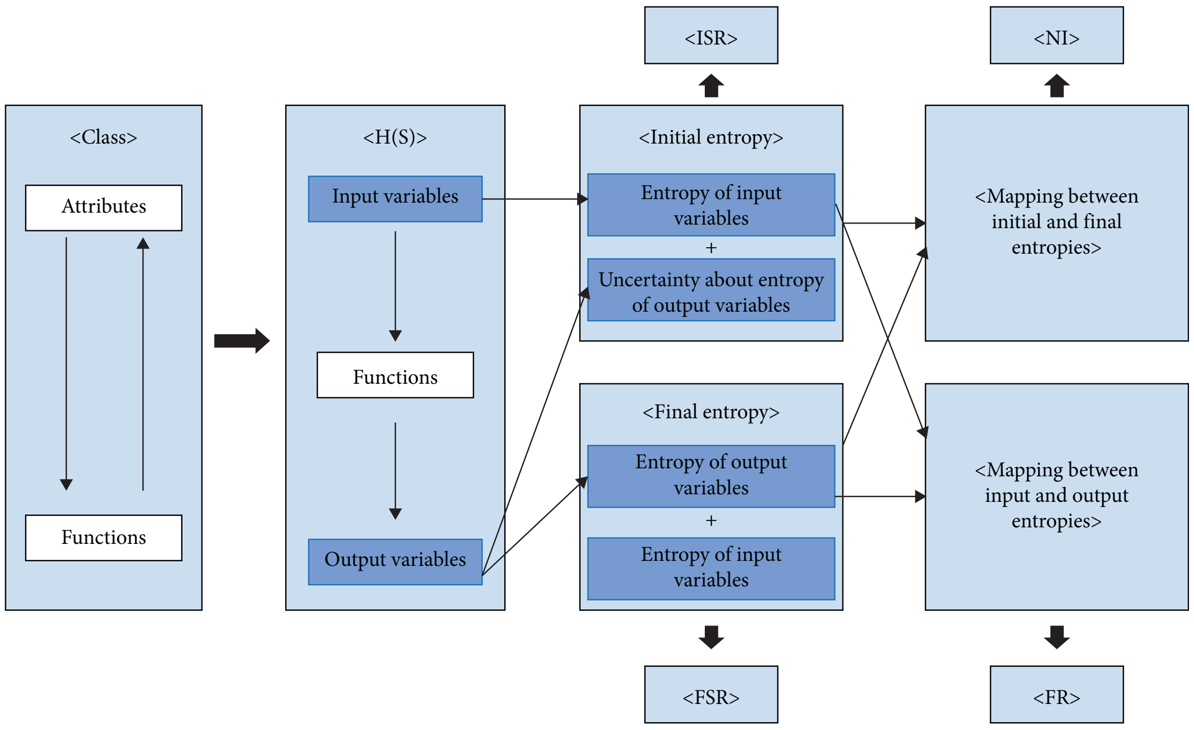

<Class>
Attributes
Functions
<H(S)>
Input variables
Functions
Output variables
<ISR>
<Initial entropy>
Entropy of input variables
+
Uncertainty about entropy of output variables
<Final entropy>
Entropy of output variables
+
Entropy of input variables
<FSR>
<NI>
<Mapping between initial and final entropies>
<Mapping between input and output entropies>
<FR>

Figure 1: Metrics computing process.

4.1. Formulation of Research Hypotheses. To study the relationship between redundancy metrics and software defect density attribute, the following hypotheses are designed:

(i) H1: ISR redundancy metric is significant as software defect density indicator.

(ii) H2: FR redundancy metric is significant as software defect density indicator.

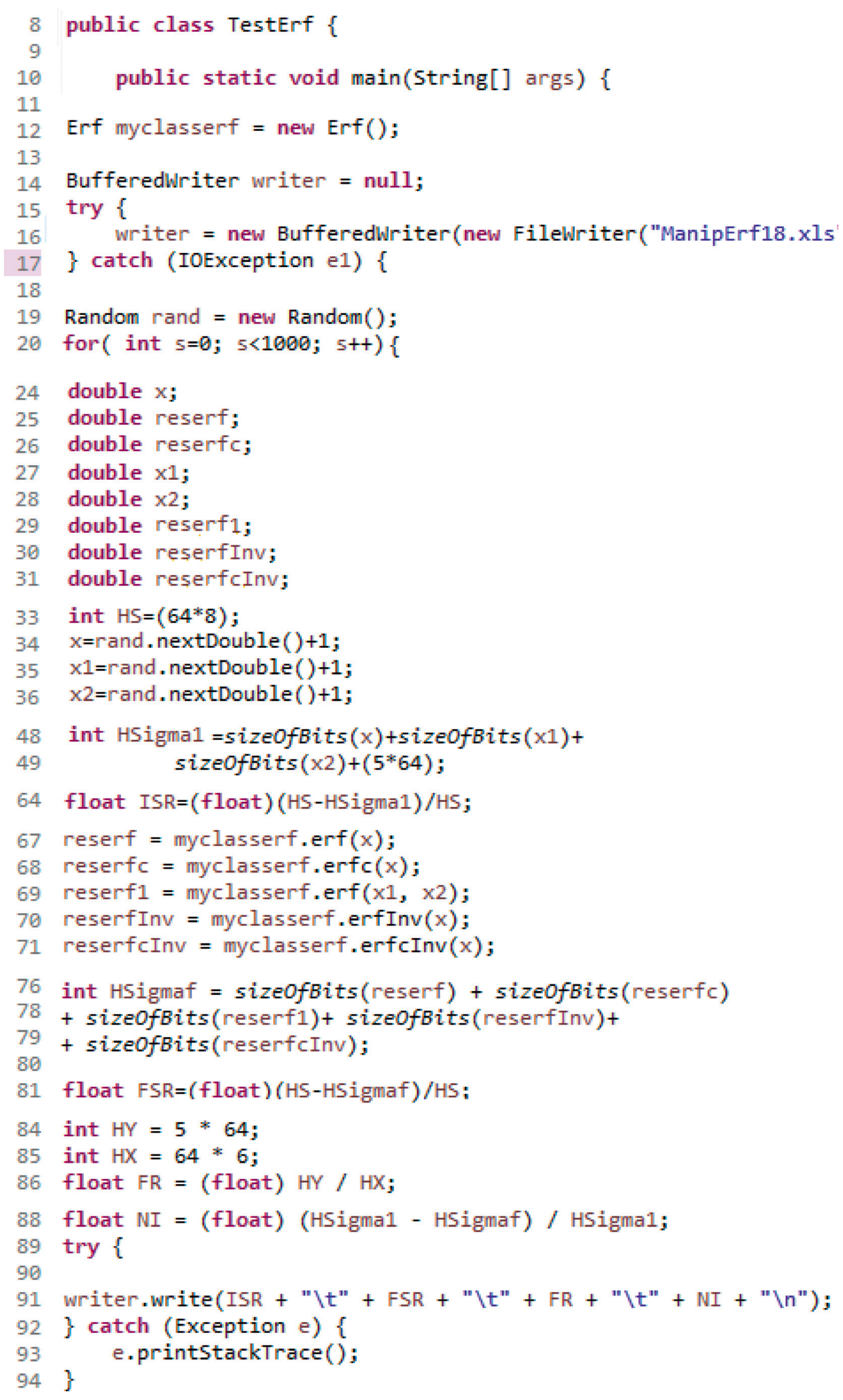
(iii) H3: NI redundancy metric is significant as software defect density indicator.

(iv) H4: ISR, FR, and NI (or combination of them) are jointly indicators of software defect density.

Through these hypotheses, we aim to verify if a relationship between the different metrics and defect density attribute exists. Once, a significant correlation between redundancy metrics and the defect density is identified, and it can be stated that these metrics are useful to monitor software reliability.

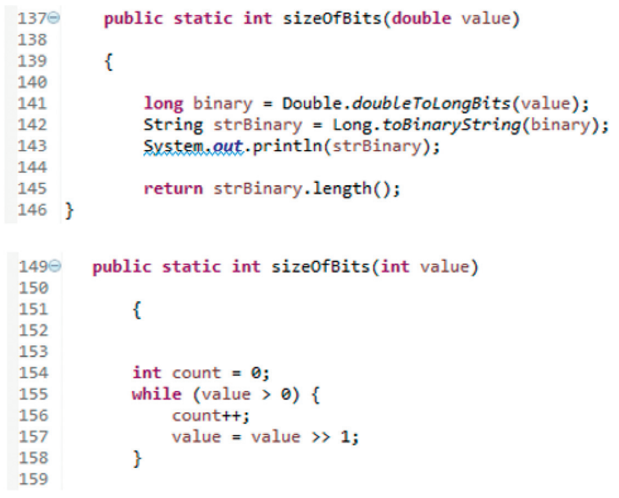
4.2. Software Defect Density. Defect density (DD) was defined as the number of defects divided by thousand lines of a delivered code [5, 42, 43]. It is given as follows [5, 7]:

$$\mathrm{DD} = \frac{\text{number of defects (defect count)}}{\textit{product size}\ (\textit{KLOC})}, \qquad (9)$$



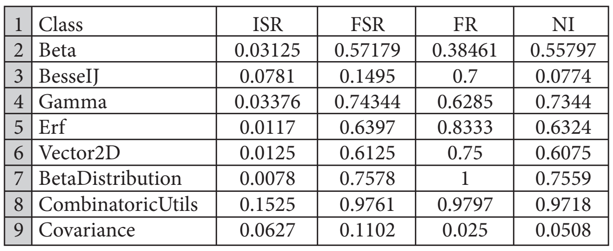
```
public class TestErf {

    public static void main(String[] args) {

Erf myclasserf = new Erf();

BufferedWriter writer = null;
try {
    writer = new BufferedWriter(new FileWriter("ManipErf18.xls
} catch (IOException e1) {

Random rand = new Random();
for( int s=0; s<1000; s++){

double x;
double reserf;
double reserfc;
double x1;
double x2;
double reserf1;
double reserfInv;
double reserfcInv;

int HS=(64*8);
x=rand.nextDouble()+1;
x1=rand.nextDouble()+1;
x2=rand.nextDouble()+1;

int HSigma1 =sizeOfBits(x)+sizeOfBits(x1)+
        sizeOfBits(x2)+(5*64);

float ISR=(float)(HS-HSigma1)/HS;

reserf = myclasserf.erf(x);
reserfc = myclasserf.erfc(x);
reserf1 = myclasserf.erf(x1, x2);
reserfInv = myclasserf.erfInv(x);
reserfcInv = myclasserf.erfcInv(x);

int HSigmaf = sizeOfBits(reserf) + sizeOfBits(reserfc)
+ sizeOfBits(reserf1)+ sizeOfBits(reserfInv)+
+ sizeOfBits(reserfcInv);

float FSR=(float)(HS-HSigmaf)/HS;

int HY = 5 * 64;
int HX = 64 * 6;
float FR = (float) HY / HX;

float NI = (float) (HSigma1 - HSigmaf) / HSigma1;
try {

writer.write(ISR + "\t" + FSR + "\t" + FR + "\t" + NI + "\n");
} catch (Exception e) {
    e.printStackTrace();
}
```

Figure 2: Example of metrics computing for *Erf* class.



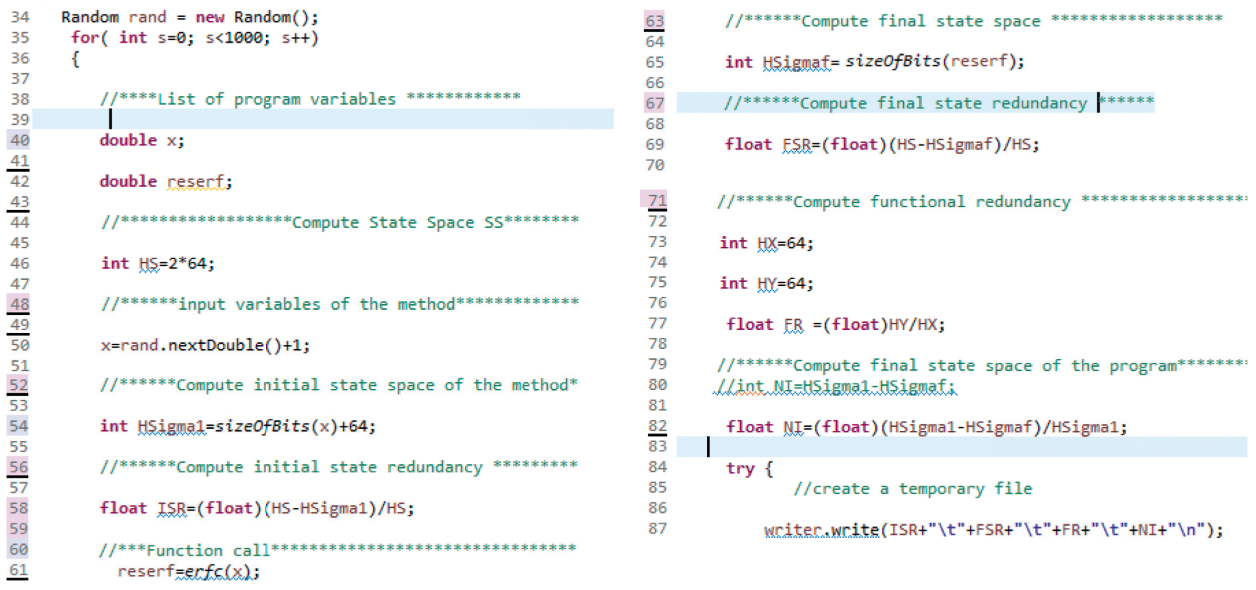
```
    public static int sizeOfBits(double value)

    {

        long binary = Double.doubleToLongBits(value);
        String strBinary = Long.toBinaryString(binary);
        System.out.println(strBinary);

        return strBinary.length();
}

    public static int sizeOfBits(int value)

        {


        int count = 0;
        while (value > 0) {
            count++;
            value = value >> 1;
        }

```

Figure 3: Part of the used script to compute the used entropy.



1	Class	ISR	FSR	FR	NI
2	Beta	0.03125	0.57179	0.38461	0.55797
3	BesselJ	0.0781	0.1495	0.7	0.0774
4	Gamma	0.03376	0.74344	0.6285	0.7344
5	Erf	0.0117	0.6397	0.8333	0.6324
6	Vector2D	0.0125	0.6125	0.75	0.6075
7	BetaDistribution	0.0078	0.7578	1	0.7559
8	CombinatoricUtils	0.1525	0.9761	0.9797	0.9718
9	Covariance	0.0627	0.1102	0.025	0.0508

Figure 4: Example of the generated metrics in *.xsl* file.



```
Random rand = new Random();
 for( int s=0; s<1000; s++)
 {

    //****List of program variables ************


    double x;

    double reserf;

    //******************Compute State Space SS********

    int HS=2*64;

    //******input variables of the method*************

    x=rand.nextDouble()+1;

    //******Compute initial state space of the method*

    int HSigma1=sizeOfBits(x)+64;

    //******Compute initial state redundancy *********

    float ISR=(float)(HS-HSigma1)/HS;

    //***Function call********************************
      reserf=erfc(x);
```

(a)

```
    //******Compute final state space ******************

    int HSigmaf= sizeOfBits(reserf);

    //******Compute final state redundancy ******

    float FSR=(float)(HS-HSigmaf)/HS;

   //******Compute functional redundancy *****************

   int HX=64;

   int HY=64;

    float FR =(float)HY/HX;

   //******Compute final state space of the program*******
   //int NI=HSigma1-HSigmaf;

    float NI=(float)(HSigma1-HSigmaf)/HSigma1;

    try {
         //create a temporary file

        writer.write(ISR+"\t"+FSR+"\t"+FR+"\t"+NI+"\n");
```

(b)

Figure 5: Metrics computing at function level.

(i) The product size is in general measured in terms of thousand lines of code (KLOC) [42, 43].

(ii) According to [5], defect counts can include post-release failures, residual faults (all faults discovered after release), all known faults, and the set of faults discovered after some arbitrary fixed points in the software life cycle (after unit testing).

Once the quality attribute is identified, the next step consists on studying the existence of a relationship between this attribute as dependent variable and the different redundancy metrics as independent variables.

4.3. Empirical Validation Approach. A software metric shall be validated [26] as the validation helps to identify the best metrics providing the required information leading to the metrics' purpose [14].

Different studies [4, 19, 26] detail various software metric suites' validation. Table 4 shows a comparison between the common validation approaches based on their objectives, process validation, and used repositories.

As illustrated in Table 4, different studies were proposed to validate software metrics as appropriate indicators of various quality factors like defect density, maintainability, and fault-proneness. We have stated the following:

(i) The different validation approaches were based on three main steps: dataset collection, dataset analysis and models building, and models' performance evaluation.

(ii) The data related to software metrics and the considered quality attribute to validate them are available in public datasets including NASA projects.

Therefore, our proposed methodology to validate redundancy metrics consists of the following steps:

(1) First, we have collected data related to dependent and independent variables represented, respectively, by defect density and redundancy metrics. As explained above, redundancy metrics are computed for a set of classes selected from Commons Apache library. In these classes, the defect density was not available. Thus, we have used defect injection procedure to compute this attribute for the same used classes.

(2) Second, we have studied the independent and joint impact of the different redundancy metrics on software defect density using data analysis tools.

(3) Third, we have proposed a defect density predictive model based on redundancy metrics.

4.4. Defect Density Data Collection. Based on equation (9), defect density is derived from the number of faults in the source code divided by thousands of line of code (KLOC) from the classes presented in Table 3.

4.4.1. KLOC Computing. To compute the KLOC measure, we have used the *Metrics* tool [47]. Within Eclipse environment, this tool provides for each of the used classes the number of lines of code illustrated in Figure 6.

4.4.2. Defect Count Computing. As mentioned, we have used the Apache commons Math library including only the source code and the associated unit tests. Thus, we used fault injection procedure to obtain the values of this measure. One of the well-known fault injection techniques is mutation testing consisting on automatically seeding into each class' code a number of faults (or mutations). The new classes are called mutants. Then, tests are run, and two possible cases are presented [48, 49]:

(i) If tests fail: the injected fault is detected (or killed) since the test gives different results between the original program (function in the class) and the faulty one.

(ii) If tests pass: the injected fault is masked (or lived) since the original program and the faulty one (mutant) give the same results when tests are run.

Fault injection procedure is performed based on automated mutation tools like MuJava, MuEclipse, PiTest, and much more [50]. In our research work, PiTest (https://pitest.org/) is used within Maven (https://maven.apache.org/) environment. To inject faults, we have proceeded as follows:

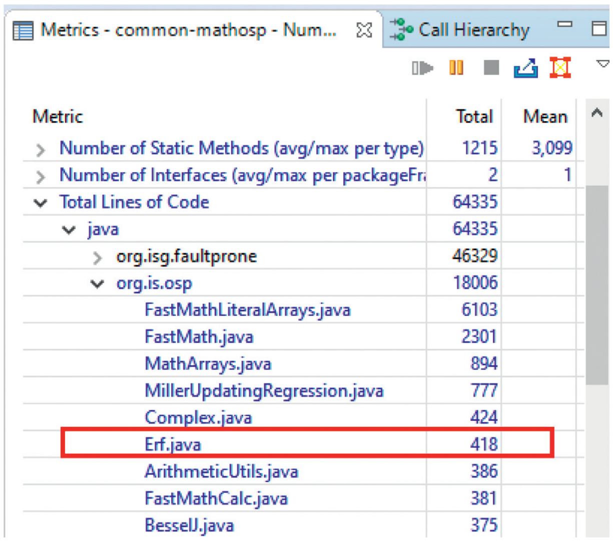
(i) All possible faults which are active by default in PiTest are injected in the considered classes [50]. These faults include the replacement of binary arithmetic operations by another ones (+ by -, - by +, * by /, / by *, and % by *), replacement of increments with decrements and vice versa, etc.

(ii) PiTest runs, and related reports are generated. PitReport indicates for each function of the different classes the type and the location of the injected fault. Moreover, it indicates whether the injected fault is detected or masked. An example of PitReport of *Erf class* is illustrated, respectively, in Figures 7(a) and 7(b). Further details of fault injection process are presented in Appendix B.

Figures 7(a) indicates the line coverage and mutation coverage for each class. This measure represents the rate of the source code executed by a program when a test suite is launched. For Figure 7(b), it presents a report of the injected faults in the *Erf* class especially for *erf (double, double)* function. The green lines indicate that the injected fault is detected, whereas the pink one indicates that it is masked. Based on these reports, the number of the injected faults is used to compute defect density measure for each class. The final structure of the obtained database contains for each class the values of the four metrics and the density attribute. This database is available at https://gitlab.com/dalilaamara/redundancymetrics/.

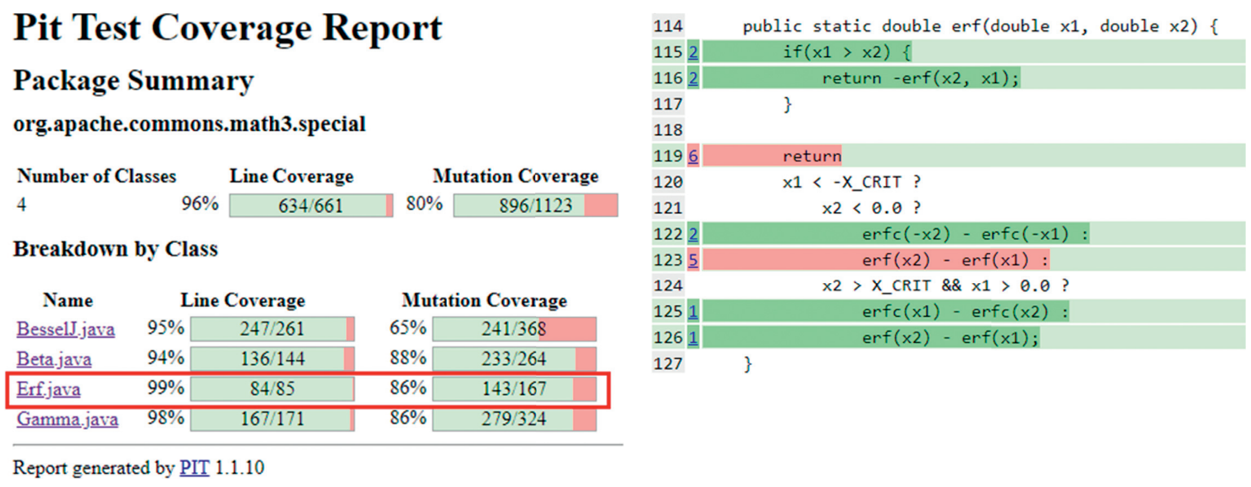
Table 4: Common software metrics' validation approaches.

Approach	Objective and used repositories	Common steps of the validation process
Cited in [4, 12, 13, 26, 28, 39, 44]	Validate different metrics like C&K, MOOD, and QMOOD suites as indicators of fault-proneness attribute The used data were extracted from different repositories including graphical user interface (GUI), Bugzilla Database, Mozilla Rhino open-source project, and NASA projects	(1) Collect data related to software metrics and the considered quality attribute (faults found during tests, maintenance effort, etc); historical data related to the selected quality attribute are available in the used repository (2) Study the relationship between the two variables (software metrics and the quality attribute) based on different machine learning techniques like random forest, Naïve Bayes, logistic regression, decision tree, and neural network (3) Evaluate results based on different performance evaluation measures like accuracy, F-measure, and precision
Cited in [7, 10, 42, 45]	Validate different software metrics including size as indicators of defect density attribute The data collected from the PROMISE and other public datasets	(1) Historical data related to software metrics and static code attributes (size and number of methods) were collected from these projects (2) Defect density attribute was predicted using the simple and multiple linear regression techniques applied to static metrics (3) Results were evaluated based on *R*-squared performance evaluation measure
Cited in [46]	C&K metrics were used to predict software maintainability attribute; the number of lines changed per class was considered as a criterion in determining the maintainability of a class User interface system (UIMS) and quality evaluation system (QUES) were used to extract the needed information (number of lines changed per class)	(1) Historical data of three years related to the number of lines changed per class in the selected software systems was used; also, the C&K metrics were extracted using metrics extraction tools (2) Neurogenetic algorithm (hybrid approach of neural network and genetic algorithm) was applied to estimate the maintainability attribute based on these metrics (3) The performance of this technique was evaluated using the mean absolute error (MAE), mean absolute relative error (MARE), root mean square error (RMSE), and standard error of the mean (SEM) evaluation measures



Metric	Total	Mean
Number of Static Methods (avg/max per type)	1215	3,099
Number of Interfaces (avg/max per packageFr	2	1
Total Lines of Code	64335	
java	64335	
org.isg.faultprone	46329	
org.is.osp	18006	
FastMathLiteralArrays.java	6103	
FastMath.java	2301	
MathArrays.java	894	
MillerUpdatingRegression.java	777	
Complex.java	424	
Erf.java	418	
ArithmeticUtils.java	386	
FastMathCalc.java	381	
BesselJ.java	375	

Figure 6: Example of LOC computing.



Pit Test Coverage Report

Package Summary

org.apache.commons.math3.special

Number of Classes	Line Coverage		Mutation Coverage	
4	96%	634/661	80%	896/1123

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
BesselJ.java	95%	247/261	65%	241/368
Beta.java	94%	136/144	88%	233/264
Erf.java	99%	84/85	86%	143/167
Gamma.java	98%	167/171	86%	279/324

Report generated by PIT 1.1.10

(a)

```
     public static double erf(double x1, double x2) {
2        if(x1 > x2) {
2            return -erf(x2, x1);
         }

6        return
         x1 < -X_CRIT ?
             x2 < 0.0 ?
2                erfc(-x2) - erfc(-x1) :
5                erf(x2) - erf(x1) :
             x2 > X_CRIT && x1 > 0.0 ?
1                erfc(x1) - erfc(x2) :
1                erf(x2) - erf(x1);
     }
```

(b)

Figure 7: (a) Example of PitReport; (b) example of injected faults in the *Erf* class.

4.5. Redundancy Metrics Data Collection. We have presented in the previous section the procedure to inject faults into the source code of the different selected classes from the Commons Math library. To test whether the metrics values can be affected by the injected faults, we have computed the redundancy metrics after the fault injection process. Thus, we have adopted the following steps (Table 5):

(i) Step 1: we have selected five mutators, namely, Conditionals Boundary (Mut 1), Negate conditionals (Mut 2), Return values (Mut 3), Inverts negatives (Mut 4), and Math (Mut 5) as different mutators were active by default in the PiTest tool used for fault injection. A brief description of these mutators is illustrated in Table 5. Further details are available at https://pitest.org/quickstart/mutators/.

(ii) Step 2: we have computed the values of the redundancy metrics using formulas 3 to 6 presented in Section 2 for each injected type of mutators (faults).

(iii) Step 3: the constructed dataset contains the values of the redundancy metrics for each type of the selected mutators. It will be used to determine whether the values of the redundancy metrics are still unchangeable when faults are injected.

4.6. Dataset Analysis. Dataset analysis phase requires other important steps like data normalization/standardization and correlation analysis. Normalization and standardization are feature scale transformation. According to [51], in case we have a large difference between the maximum and minimum values, e.g., 0.01 and 1000, we should rescale them in the range [0, 1]. In this study, the used metrics are defined in such a way that they range between 0 and 1 [18].

Correlation analysis is required to identify the association between the different independent variables (redundancy metrics) in order to consider only significant ones (not intercorrelated). To test the redundancy metrics correlation, we have used Python language. The results are illustrated in Figure 8.

Correlation coefficients between the independent variables are analyzed based on Hopkin's statements [52]:

(i) Correlation coefficients which are greater than 0.5 are considered as large.

(ii) Correlation coefficients which are between 0.3 and 0.5 are considered as moderate.

(iii) Correlation coefficients which are between 0.1 and 0.3 are considered as small.

(iv) Correlation coefficients which are smaller than 0.1 are considered as insubstantial.

Figure 8 indicates a strong significant correlation between NI and FSR metrics since their correlation coefficient is equal to 0.99. Thus, one of these variables will be omitted.

5. Experiments and Results

In this section, we have evaluated the usefulness of redundancy metrics as predictors of defect density to test the stated hypotheses. According to [7, 30], regression techniques are used since the defect density attribute we aim to predict is represented by quantitative values.

5.1. Experiments. As mentioned, linear (simple and multiple) can be used to predict defect density based on redundancy metrics. Thus, we present the general form of each regression type as follows:

(i) The general form of the simple linear regression model is presented as

$$Y = \beta_0 + \beta_1 X + \ldots. \qquad (10)$$

(ii) The general form of the multiple linear regression model is presented as

Table 5: Description of the five selected mutators.

Mutators	Description
Mut 1: Conditionals Boundary	Replaces the relational operators <, ≤, >, and ≥
Mut 2: Negate conditionals	Mutates all conditionals found (< by ≥, ≥ by <, etc.)
Mut 3: Return values	Mutates the return values of method calls (true by false, false by true, 0 by 1, and x by $x+1$)
Mut 4: Inverts negatives	Inverts negation of integer and floating-point numbers
Mut 5: Math	Replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation (+ by -, - by +, * by /, etc.).

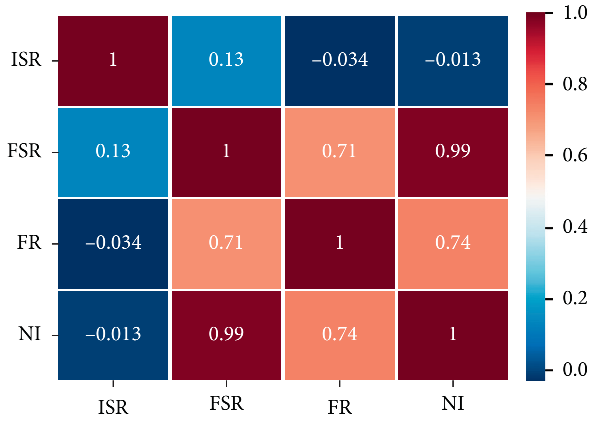


Figure 8: Correlation matrix of dependent and independent variables.

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots \beta_i X_i + \ldots . \quad (11)$$

In the presented equations, the Y represents the dependent variable, the X_i represent the independent variables, β_i are the estimated parameters, and Epsilon (ε) is the random error.

Using the presented formulas (10) and (11), three main experiments are performed:

(iii) Experiment 1: before studying the linear regression between the redundancy metrics and the defect density, we have tested whether the values of the redundancy metrics are affected by the injected faults. Therefore, for each metric, we have represented its variation with the five selected mutators described in Table 4.

(iv) Experiment 2: in this experiment, we have used the univariate linear regression to test the hypotheses H1 to H3 presented in Section 4.1.

(v) Experiment 3: in this experiment, the multivariate linear regression is used to test the hypothesis H4.

5.2. Results

5.2.1. Experiment 1: Variation in the Redundancy Metrics with the Injected Faults.

We have studied the variation in the ISR, FSR, and NI redundancy metrics for each type of the injected mutators for a set of classes selected from the constructed dataset (see Figure 9). We have only focused on these metrics since FR redundancy metric is computed using the maximum entropy of the input and output data insensitive to any change in the variables' states (see equation (6)). Results are depicted in Figure 9.

Figure 9 shows some variation in the values of the redundancy metrics for the different mutators. This variation indicates that change in mutators of different types affects the redundancy of the source code. Thus, the state of the variables used to assess this redundancy changes with the injected faults.

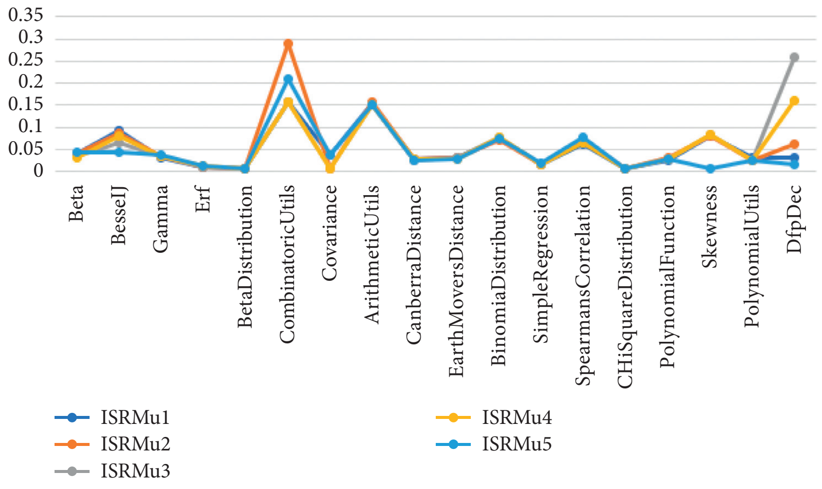
Figure 10 illustrates a part of the mutated source code of the BesselJ class.

In line 446, 2 injected faults of Math mutators are survived, and these faults consist on replacing, respectively, the multiplication with a division and the subtraction with an addition. Therefore, the state of the p variable will be modified and will affect the state of the subsequent instructions. In line 464, other types of mutators were injected, which are Mut 1, Mut 5, and Mut 2. These mutators affect the state of the different variables used by the program showing a variation in the redundancy metrics as their values depend on the variables' states.

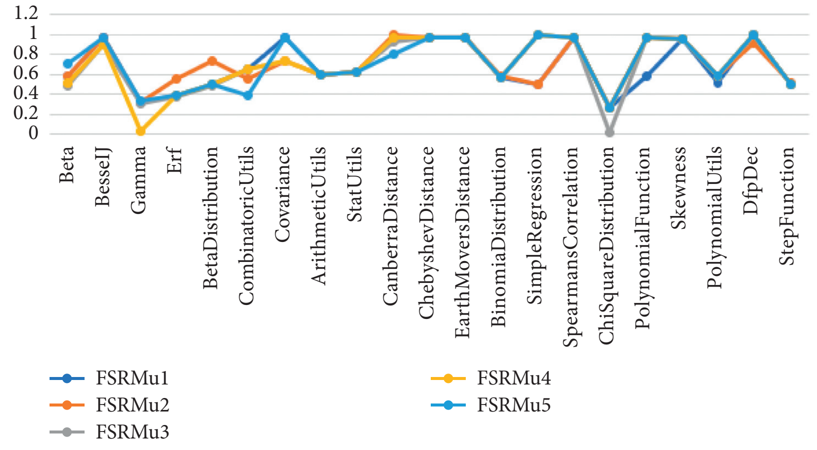
In experiment 1, the variability of the redundancy metrics for the different mutators is negligible in some classes, as these faults cause a little change in the number of needed bits representing the variable states. For instance, in Figure 10 line 446, if we consider the variables $en = 2$, $plast = 1$, $x = 1$, and $pold = 1$, then the value of p variable before and after the injection of Math mutators (replace multiplication with division and subtraction with addition) is equal to 1. Thus, the redundancy provided by the state of the p variable is still unchangeable. In line 465, if we consider $l = 7$, then the states of $ncalc$ variable before and after the Math mutators (replace subtraction with division) are, respectively, 6 and 8. The required entropy to represent these states is 3 bits. Therefore, there is no variation in the redundancy metrics between these two states (6 and 8). This explains the similar values of redundancy metrics for the different mutators presented in Figure 9.

5.2.2. Experiment 2: Univariate Linear Regression.

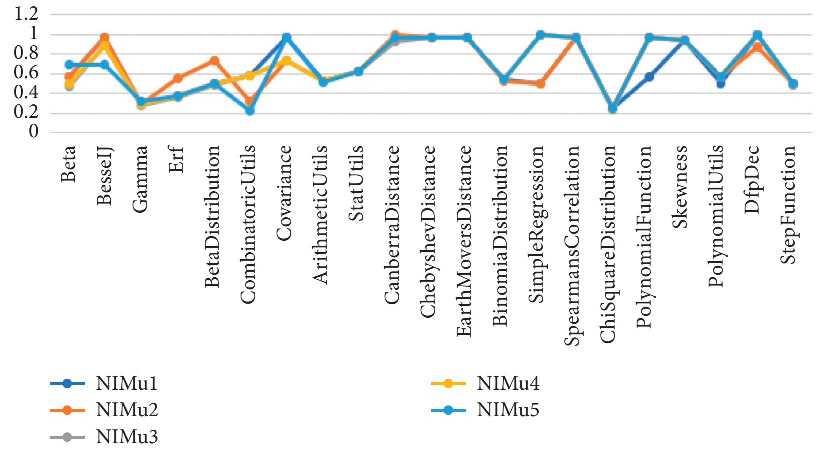
The univariate linear regression uses only one independent variable (one of the presented redundancy metrics) to predict defect density. Thus, we perform univariate linear regression to test separately the three first hypotheses presented above. Results of this experiment are illustrated in Table 6.



(a)

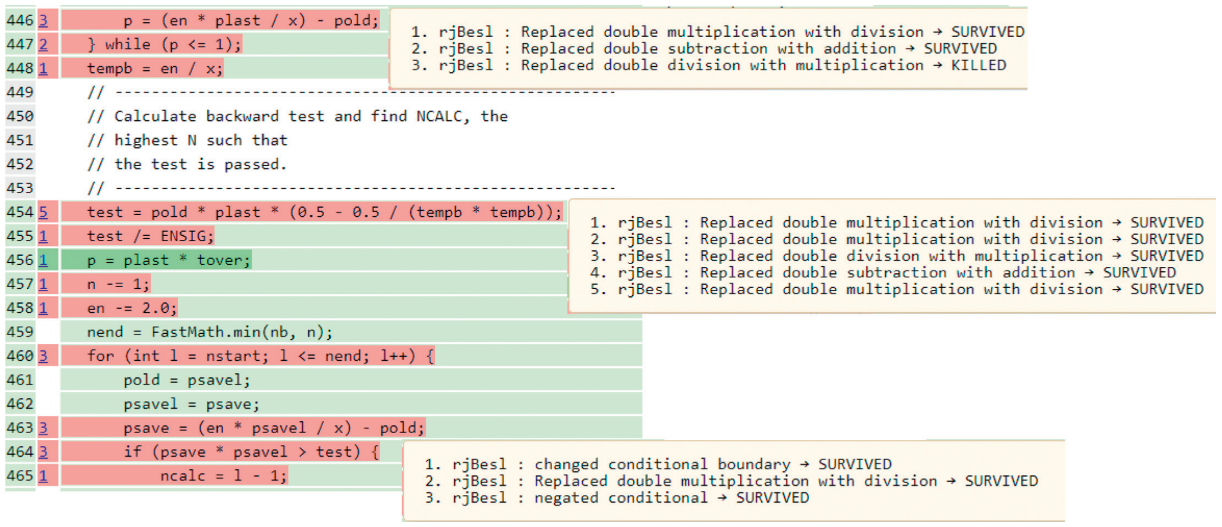


(b)



(c)

Figure 9: Variation in the (a) ISR, (b) FSR, and (c) NI metric values for the different mutators.



```
3    p = (en * plast / x) - pold;
2  } while (p <= 1);
1  tempb = en / x;
   // -----------------------------------------------------
   // Calculate backward test and find NCALC, the
   // highest N such that
   // the test is passed.
   // -----------------------------------------------------
5  test = pold * plast * (0.5 - 0.5 / (tempb * tempb));
1  test /= ENSIG;
1  p = plast * tover;
1  n -= 1;
1  en -= 2.0;
   nend = FastMath.min(nb, n);
3  for (int l = nstart; l <= nend; l++) {
       pold = psavel;
       psavel = psave;
3      psave = (en * psavel / x) - pold;
3      if (psave * psavel > test) {
1          ncalc = l - 1;
```

```
1. rjBesl : Replaced double multiplication with division → SURVIVED
2. rjBesl : Replaced double subtraction with addition → SURVIVED
3. rjBesl : Replaced double division with multiplication → KILLED
```

```
1. rjBesl : Replaced double multiplication with division → SURVIVED
2. rjBesl : Replaced double multiplication with division → SURVIVED
3. rjBesl : Replaced double division with multiplication → SURVIVED
4. rjBesl : Replaced double subtraction with addition → SURVIVED
5. rjBesl : Replaced double multiplication with division → SURVIVED
```

```
1. rjBesl : changed conditional boundary → SURVIVED
2. rjBesl : Replaced double multiplication with division → SURVIVED
3. rjBesl : negated conditional → SURVIVED
```

Figure 10: Part of the mutated source code of the BesselJ class.

Table 6: Regression results of experiment 2.

Hypothesis	Dependent variable	Independent variable	Coefficients	$p > \|t\|$ (p values)
H1	Defect density	ISR	1.566	0.000
H2	Defect density	FR	0.306	0.000
H3	Defect density	NI	0.398	0.000

Results shown in Table 6 are analyzed based on p value measure. This measure is defined as the probability of error which is the significance level that is used to accept or reject the hypothesis [7]. Two possible cases are presented:

(i) To accept the hypothesis, the p value must be less than or equal to 0.05.

(ii) Otherwise, reject the hypothesis.

Taking defect density as dependent variable and redundancy metrics as predictors and based on previous statements, the results in Table 6 are summarized as follows:

(i) For H1, p value is 0.000 and less than 0.05. So the H1 hypothesis is accepted which means that ISR metric can be considered as a significant defect density predictor.

(ii) For FR, p value is 0.000. Thus, we can accept the hypothesis H3 which indicates that FR redundancy metric can be considered as a significant predictor of defect density attribute.

(iii) For NI, p value is 0.000. Using previous statements, the hypothesis H3 is also accepted which means that NI can be considered as a significant predictor of defect density attribute.

5.2.3. Experiment 3: Multivariate Linear Regression. Once univariate linear regressions are performed to identify the relationship between each redundancy metric and defect density separately, we aim in this experiment to join these metrics and study their common effect on defect density. For this, we have tested the multivariate regression for the hypothesis H4 based on equation (11). Results are summarized in Figure 11.

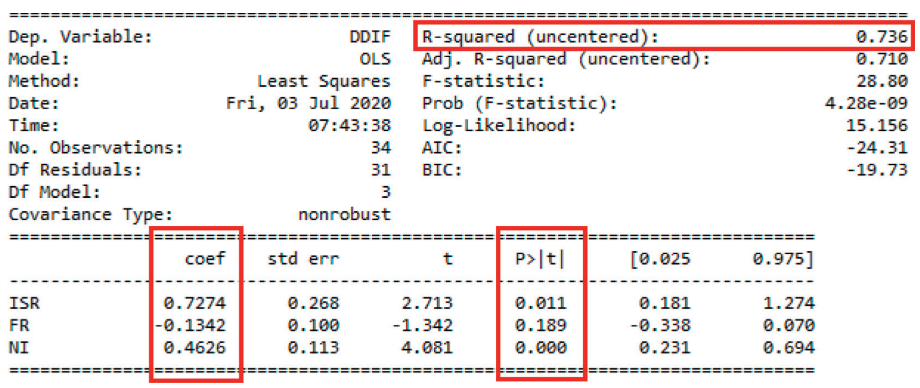
Taking defect density as dependent variable and redundancy metrics as predictors, results in Figure 9 show the following:

(i) For ISR and NI, the p values are less than 0.05 and equal, respectively, to 0.011 and 0.000. Consequently, the hypothesis H5 which supposes that redundancy metrics are useful as defect density predictors is accepted for ISR and NI.

(ii) For FR metric, p value is greater than 0.05 and equal to 0.189. For this, this metric is omitted from the multivariate regression and only ISR and NI are considered as significant predictors of defect density attribute.

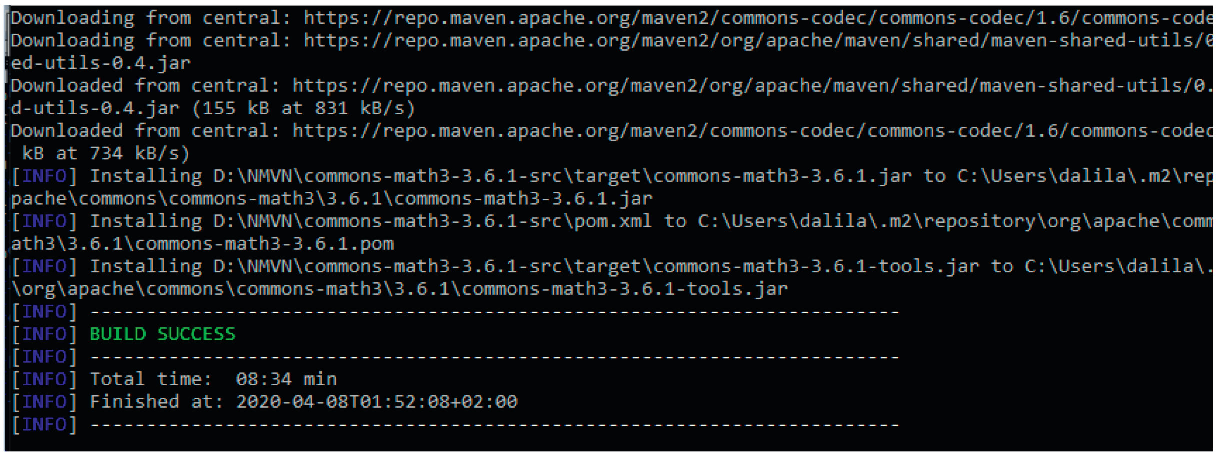
5.3. Model Performance Evaluation. We present in this section the overall evaluation of the linear regression model and summary of results:

(i) Model performance evaluation: model evaluation is required to evaluate the significance of the model to identify whether it fits well the data. Among the performance evaluation measures, the coefficient of determination (R-squared score) is the most used [41, 53]. This measure represents the proportion of variance in the dependent variable that can be predicted from the independent variables.



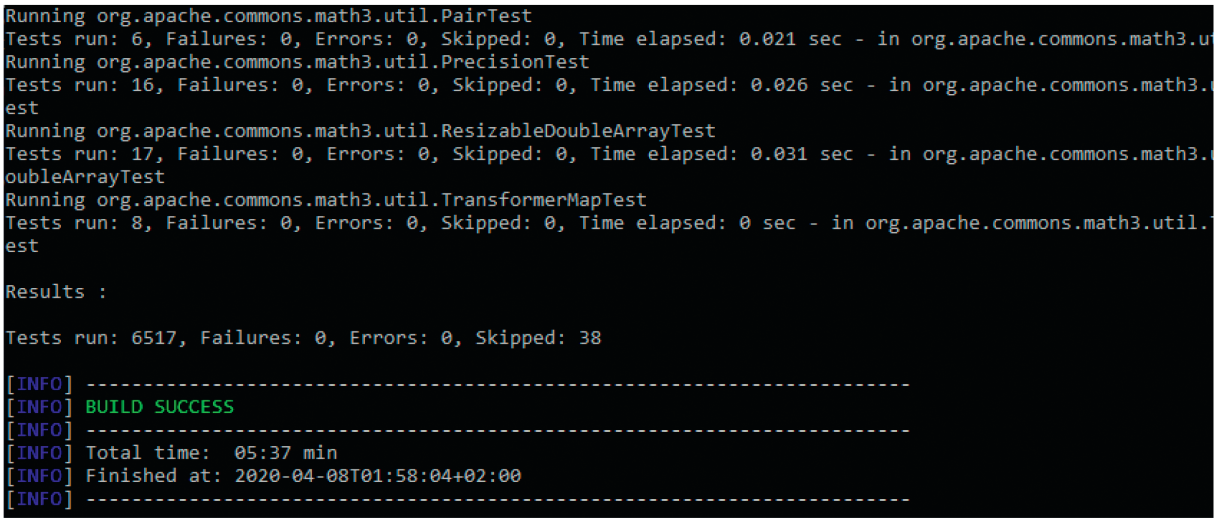
```
==============================================================================
Dep. Variable:                   DDIF   R-squared (uncentered):          0.736
Model:                            OLS   Adj. R-squared (uncentered):     0.710
Method:                 Least Squares   F-statistic:                     28.80
Date:                Fri, 03 Jul 2020   Prob (F-statistic):           4.28e-09
Time:                        07:43:38   Log-Likelihood:                 15.156
No. Observations:                  34   AIC:                            -24.31
Df Residuals:                      31   BIC:                            -19.73
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
ISR            0.7274      0.268      2.713      0.011       0.181       1.274
FR            -0.1342      0.100     -1.342      0.189      -0.338       0.070
NI             0.4626      0.113      4.081      0.000       0.231       0.694
==============================================================================
```

Figure 11: Results of the multivariate linear regression.



```
Downloading from central: https://repo.maven.apache.org/maven2/commons-codec/commons-codec/1.6/commons-code
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-shared-utils/0
ed-utils-0.4.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-shared-utils/0.
d-utils-0.4.jar (155 kB at 831 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/commons-codec/commons-codec/1.6/commons-codec
 kB at 734 kB/s)
[INFO] Installing D:\NMVN\commons-math3-3.6.1-src\target\commons-math3-3.6.1.jar to C:\Users\dalila\.m2\rep
pache\commons\commons-math3\3.6.1\commons-math3-3.6.1.jar
[INFO] Installing D:\NMVN\commons-math3-3.6.1-src\pom.xml to C:\Users\dalila\.m2\repository\org\apache\comm
ath3\3.6.1\commons-math3-3.6.1.pom
[INFO] Installing D:\NMVN\commons-math3-3.6.1-src\target\commons-math3-3.6.1-tools.jar to C:\Users\dalila\.
\org\apache\commons\commons-math3\3.6.1\commons-math3-3.6.1-tools.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  08:34 min
[INFO] Finished at: 2020-04-08T01:52:08+02:00
[INFO] ------------------------------------------------------------------------
```

Figure 12: Maven install phase.



```
Running org.apache.commons.math3.util.PairTest
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 sec - in org.apache.commons.math3.u
Running org.apache.commons.math3.util.PrecisionTest
Tests run: 16, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.026 sec - in org.apache.commons.math3.
est
Running org.apache.commons.math3.util.ResizableDoubleArrayTest
Tests run: 17, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.031 sec - in org.apache.commons.math3.
oubleArrayTest
Running org.apache.commons.math3.util.TransformerMapTest
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec - in org.apache.commons.math3.util.
est

Results :

Tests run: 6517, Failures: 0, Errors: 0, Skipped: 38

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  05:37 min
[INFO] Finished at: 2020-04-08T01:58:04+02:00
[INFO] ------------------------------------------------------------------------
```

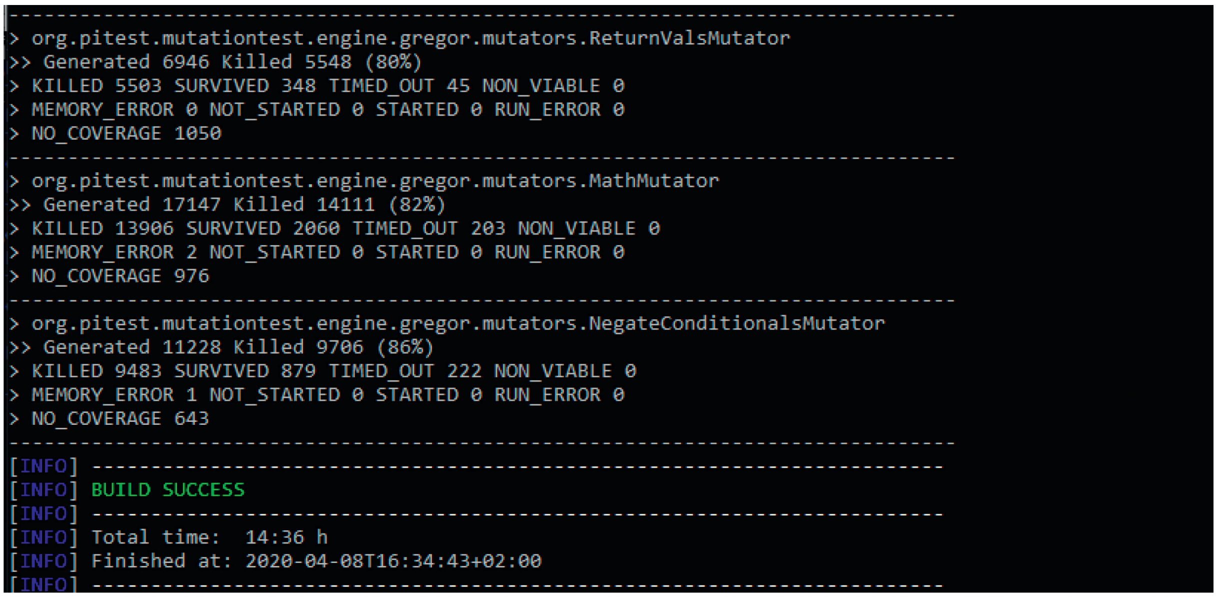
Figure 13: Maven test phase.

(ii) Model parameters are regression beta coefficients (β_i) that describe the degree of the response variable for each 1-unit change in the independent variables.

A summary of results for the different above hypothesis based on the presented performance measures and model parameters is illustrated in Table 7.

For the different previous hypotheses and based on the presented performance evaluation measures, Table 7 shows the following:

(i) For H1, H2, and H3, R-squared values indicate, respectively, that 38.8%, 47.3%, and 65.8% of the variability of defect density is predicted separately



```
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.ReturnValsMutator
>> Generated 6946 Killed 5548 (80%)
> KILLED 5503 SURVIVED 348 TIMED_OUT 45 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 1050
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.MathMutator
>> Generated 17147 Killed 14111 (82%)
> KILLED 13906 SURVIVED 2060 TIMED_OUT 203 NON_VIABLE 0
> MEMORY_ERROR 2 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 976
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator
>> Generated 11228 Killed 9706 (86%)
> KILLED 9483 SURVIVED 879 TIMED_OUT 222 NON_VIABLE 0
> MEMORY_ERROR 1 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 643
--------------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  14:36 h
[INFO] Finished at: 2020-04-08T16:34:43+02:00
[INFO] ------------------------------------------------------------------------
```

Figure 14: Mutations run phase.

Table 7: Evaluation measures and model parameters for regression techniques.

Hypotheses		*p* value	Beta coefficient	Result	*R*-squared
H1 (ISR)		0.000	1.566	Accepted	0.388
H2 (FR)		0.000	0.306	Accepted	0.473
H3 (NI)		0.000	0.398	Accepted	0.658
H4 (multiple linear regression)	ISR	0.011	0.727	Accepted for ISR and NI but rejected for FR	0.736
	FR	0.189	−0.134		
	NI	0.000	0.462		

by, respectively, ISR, FR, and NI. The obtained *R*-squared values for these hypotheses are moderate and indicate that using the redundancy metrics as separate independent variables explains moderately the variation in defect density as a dependent variable.

(ii) For H4, adjusted *R*-squared shows that ISR and NI jointly predict 73.6% variability of defect density. This indicates that, overall, the performed multiple regression can significantly predict the defect density. So, this multiple regression between redundancy metrics and defect density is justified.

To sum up, the combined impact of ISR and NI redundancy metrics is analyzed, and results show that this multiple regression is justified. So, using these metrics jointly gives more improvement in predicting defect density. The application of multiple linear regression provides a model that reflects the relationship between the redundancy metrics and defect density and can be depicted by the following equation based on beta coefficients presented in Table 7:

$$\mathrm{DD} = 0.727\,\mathrm{ISR} + 0.462\,\mathrm{NI}. \tag{12}$$

Using previous statements, we can note the following:

(i) For ISR, the regression coefficient is positive and equal to 0.727. This means that, for each 1-unit increase in the ISR metric, there will be an increase in defect density by 0.727 units.

(ii) For NI, the regression coefficient is positive and equal to 0.462. Consequently, for each 1-unit increase in the NI variable, the defect density variable will increase by 0.462 units.

5.4. Discussion and Threats to Validity. Reliability is in general predicted based on predictive models which are developed using two basic elements: software metrics and software faults [8, 54]. The proposed linear regression model is justified and can be used to predict the defect density for new datasets based on their redundancy measures. This defect prediction model can serve as early quality indicator for developers and testing teams to manage and control test execution activities. For different code alternations, different values of the ISR and NI redundancy metrics are obtained. The variation in these values can affect the defect density attribute.

We have obtained promising results proposing validated ISR and NI redundancy metrics as significant reliability indicators. However, we have noted several threads to validity. First, the proposed redundancy metrics are semantic as they depend on the program functionality; each program (function or class) has its state represented by the manipulated variables. Hence, each time the used variables in the program input state change, the output state will change, and the values of the redundancy metrics will change too. Therefore, the proposed computing process described in Section 3 is not automated, and it is implemented separately for each program. Second, the more the larger training datasets and optimizing model parameters are used, the better the model prediction performance [55], and our dataset can be extended to enhance the performance of the proposed prediction model. Third, literature review related to software metrics validation [7, 10, 12, 13] shows that usually numerous quality attributes can be used to validate a software metric. Hence, we can use other reliability subcharacteristics like fault-proneness to show the utility of the redundancy metrics as reliability indicators.

6. Conclusion and Perspectives

Initial state redundancy, final state redundancy, non-injectivity, and functional redundancy metrics were proposed to assess the code' redundancy in order to monitor software reliability. However, all of these metrics are manually computed and theoretically presented. In this research, we aim at empirically assessing and validating these metrics as significant reliability indicators. We have used the defect density attribute as a direct reflection of software reliability to reach our objective.

We have built an empirical database including a set of Java classes taken from the Commons Math Library, all related redundancy metrics' values, and the defect density as a direct reliability indicator. This database has allowed us to empirically assess and validate the redundancy metrics as reliability indicators.

Regression techniques have been used to propose a predictive model based on the defect density attribute as a dependent variable and initial state redundancy and non-injectivity metrics as independent variables.

The proposed model is useful for testers and developers and can be used to predict defect density and to monitor software reliability for further datasets.

As the initial state redundancy metric only measures the program redundancy in its initial and final states without considering the redundancy of its internal states, we propose in the future work, to improve this metric by considering its internal states in order to reflect the overall program redundancy. In addition, we envision to develop an automated support tool computing the redundancy metrics leading to ameliorate the performance of the computing process.

Appendix

A

In this Appendix, we present examples of the used scripts to perform the empirical assessment and the validation of redundancy metrics as useful indicators of software reliability.

Figure 3 presents an example of the sizeOfBits function identifying the needed entropy of the used variables in the different program (function/class) states.

In Figure 5, the different redundancy metrics are computed at method level. The computing process is the same as for the class level. However, at function level, only one function and the associated input and output variables are considered.

B

This appendix presents the process of mutations (faults) injection. Thus, three main steps were adopted to seed faults into java programs. These steps consist on executing three Maven command lines:

(1) mvn install: consists of installing Maven packages into the local repository; various actions are printed which end with build success result as shown in Figure 12.

(2) mvn test: required to compile test classes result as shown in Figure 13.

(3) mvn org.pitest:pitest-maven:mutationCoverage: used to run mutations; build success result is obtained as shown in Figure 14.

Data Availability

Datasets used to perform our empirical research work are available through https://gitlab.com/dalilaamara/redundancymetrics. This link is also included in the manuscript.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] G. A. García-Mireles, M. Á. Moraga, F. García, and M. Piattini, "Approaches to promote product quality within software process improvement initiatives: a mapping study," *Journal of Systems and Software*, vol. 103, pp. 150–166, 2015.

[2] I. S. O. Iso, "Iec25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models," *International Organization for Standardization*, vol. 34, p. 2910, 2011.

[3] F. Febrero, C. Calero, and M. Ángeles Moraga, "Software reliability modeling based on ISO/IEC SQuaRE," *Information and Software Technology*, vol. 70, pp. 18–29, 2016.

[4] L. C. Briand and J. Wüst, "Empirical studies of quality models in object-oriented systems," *Advances in Computers*, vol. 56, pp. 97–166, 2002.

[5] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*, CRC Press, Boca Raton, FL, USA, 2014.

[6] R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," *Empirical Software Engineering*, vol. 20, no. 3, pp. 640–693, 2015.

[7] D. K. Verma and S. Kumar, "Prediction of defect density for open source software using repository metrics," *Journal of Web Engineering*, vol. 16, no. 3, pp. 294–311, 2017.

[8] C. Catal, "Software fault prediction: a literature review and current trends," *Expert Systems with Applications*, vol. 38, no. 4, pp. 4626–4636, 2011.

[9] K. P. Srinivasan, "Unique fundamentals of software measurement and software metrics in software engineering," *International Journal of Computer Science & Information Technology*, vol. 7, no. 4, 2015.

[10] D. Verma and S. Kumar, "An improved approach for reduction of defect density using optimal module sizes," *Advances in Software Engineering*, vol. 2014, Article ID 803530, 2014.

[11] M. R. Lyu, *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[12] N. Vanitha and R. ThirumalaiSelvi, *A Report on the Analysis of Metrics and Measures on Software Quality Factors–A Literature Study*, International Journal of Computer Science and Information Technologies, vol. 5, no. 5, , pp. 6591–6595, 2014.

[13] H. Nakai, N. Tsuda, K. Honda, H. Washizaki, and Y. Fukazawa, "Initial framework for software quality evaluation based on iso/iec 25022 and iso/iec 25023," in *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 410-411, IEEE, Vienna, Austria, August 2016).

[14] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Software metrics fluctuation: a property for assisting the metric selection process," *Information and Software Technology*, vol. 72, pp. 110–124, 2016.

[15] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: a systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 164–197, 2017.

[16] A. Mili, A. Jaoua, M. Frias, and R. G. M. Helali, "Semantic metrics for software products," *Innovations in Systems and Software Engineering*, vol. 10, no. 3, pp. 203–217, 2014.

[17] I. Marsit, M. N. Omri, and A. Mili, "Estimating the survival rate of mutants," in *Proceedings of the 2th International Conference on Software Technologies (ICSOFT)*, pp. 208–213, Madrid, Spain, July 2017.

[18] A. Ayad, I. Marsit, N. M. Omri, J. Loh, and A. Mili, "Using semantic metrics to predict mutation equivalence," in *Proceedings of the International Conference on Software Technologies*, pp. 3–27, Porto, Portugal, July 2018.

[19] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[20] D. Amara, E. Fatnassi, and L. Rabai, "An automated support tool to compute state redundancy semantic metric," in *Proceedings of the International Conference on Intelligent Systems Design and Applications*, pp. 262–272, Delhi, India, December 2017.

[21] A. Jaoua and A. Mili, "The use of executable assertions for error detection and damage assessment," *Journal of Systems and Software*, vol. 12, no. 1, pp. 15–37, 1990.

[22] A. Mili and F. Tchier, *Software Testing: Concepts and Operations*, John Wiley & Sons, Hoboken, NJ, USA, 2015.

[23] M. R. Lyu, "Software reliability engineering: a roadmap," in *Future of Software Engineering (FOSE'07)*, pp. 153–170, IEEE, 2007.

[24] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*, Artech House, Norwood, MA, USA, 2001.

[25] M. R. Lyu, Z. Huang, S. K. Sze, and X. Cai, "An empirical study on testing and fault tolerance for software reliability engineering," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pp. 119–130, Denver, Colorado, November 2003.

[26] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[27] C. S. Gall, S. Lukins, L. Etzkorn et al., "Semantic software metrics computed from natural language design specifications," *IET Software*, vol. 2, no. 1, pp. 17–26, 2008.

[28] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[29] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: a systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[30] N. Kalaivani and R. Beena, "Overview of software defect prediction using machine learning algorithms," *International Journal of Pure and Applied Mathematics*, vol. 118, no. 20, pp. 3863–3873, 2018.

[31] E. Dubrova, *Fault-Tolerant Design*, pp. 55–65, Springer, New York, NY, USA, 2013.

[32] A. Mili, L. Wu, F. T. Sheldon, M. Shereshevsky, and J. Desharnais, "Modeling redundancy: quantitative and qualitative models," in *Proceedings of the IEEE International Conference on Computer Systems and Applications*, pp. 1–8, Sharjah, United Arab Emirates, March 2006.

[33] S. A. Asghari, M. Binesh Marvasti, and A. M. Rahmani, "Enhancing transient fault tolerance in embedded systems through an OS task level redundancy approach," *Future Generation Computer Systems*, vol. 87, pp. 58–65, 2018.

[34] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[35] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[36] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," in *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pp. 156–166, IEEE, Florence, Italy, May 2015.

[37] P. A. Laplante, Ed., *Dictionary of Computer Science, Engineering and Technology*, CRC Press, Boca Raton, FL, USA, 2000.

[38] V. B. Singh and K. K. Chaturvedi, "Improving the quality of software by quantifying the code change metric and predicting the bugs," in *Proceedings of the International Conference on Computational Science and its Applications*, pp. 408–426, Ho Chi Minh City, Vietnam, June 2013.

[39] S. Kumar and S. S. Rathore, *Software Fault Prediction: A Road Map*, Springer, Singapore, 2018.

[40] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[41] S. Reddivari and J. Raman, "Software quality prediction: an investigation based on machine learning," in *Proceedings of the 2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pp. 115–122, IEEE, Los Angeles, CA, USA, July 2019.

[42] S. M. A. Shah, M. Morisio, and M. Torchiano, "An overview of software defect density: a scoping study," in *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference*, pp. 406–415, IEEE, Hong Kong, China, December 2012.

[43] H. B. Yadav and D. K. Yadav, "Early software reliability analysis using reliability relevant software metrics," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 4, pp. 2097–2108, 2017.

[44] Y. Zhou, B. Xu, and H. Leung, "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems," *Journal of Systems and Software*, vol. 83, no. 4, pp. 660–674, 2010.

[45] N. Mandhan, D. K. Verma, and S. Kumar, "Analysis of approach for predicting software defect density using static metrics," in *Proceedings of the International Conference on Computing, Communication & Automation*, pp. 880–886, IEEE, Noida, India, May 2015.

[46] L. Kumar, D. K. Naik, and S. K. Rath, "Validating the effectiveness of object-oriented metrics for predicting maintainability," *Procedia Computer Science*, vol. 57, pp. 798–806, 2015.

[47] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pp. 131–142, Seattle, WA, USA, July 2008.

[48] I. Pill, I. Rubil, F. Wotawa, and M. Nica, "SIMULTATE: a toolset for fault injection and mutation testing of simulink models," in *Proceedings of the 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 168–173, IEEE, Chicago, IL, USA, April 2016.

[49] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–55, 2016.

[50] M. Delahaye and L. Du Bousquet, "A comparison of mutation analysis tools for java," in *Proceedings of the 2013 13th International Conference on Quality Software*, pp. 187–195, IEEE, Nanjing, China, July 2013.

[51] T. Chen and K. Honda, "Solving data preprocessing problems in existing location-aware systems," *Journal of Ambient Intelligence and Humanized Computing*, vol. 9, no. 2, pp. 253–259, 2018.

[52] L. H. Etzkorn, S. Gholston, and W. E. Hughes Jr., "A semantic entropy metric," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 4, pp. 293–310, 2002.

[53] G. Abaei and A. Selamat, "A survey on software fault detection based on different prediction approaches," *Vietnam Journal of Computer Science*, vol. 1, no. 2, pp. 79–95, 2014.

[54] H. Turabieh, M. Mafarja, and X. Li, "Iterated feature selection algorithms with layered recurrent neural network for software fault prediction," *Expert Systems with Applications*, vol. 122, pp. 27–42, 2019.

[55] A. Singh, R. Bhatia, and A. Singhrova, "Taxonomy of machine learning algorithms in software fault prediction using object oriented metrics," *Procedia Computer Science*, vol. 132, pp. 993–1001, 2018.