applied sciences

Article

An Improved Confounding Effect Model for Software Defect Prediction

Yuyu Yuan [1,2,*], Chenlong Li [1,2] and Jincui Yang [1,2]

1 School of Computer Science (National Pilot Software Engineering School), Beijing University of Posts and Telecommunications, Beijing 100876, China
2 Key Laboratory of Trustworthy Distributed Computing and Service, Ministry of Education, Beijing 100876, China
* Correspondence: yuanyuyu@bupt.edu.cn

Abstract: Software defect prediction technology can effectively improve software quality. Depending on the code metrics, machine learning models are built to predict potential defects. Some researchers have indicated that the size metric could cause confounding effects and bias the prediction results. However, evidence shows that the real confounder should be the development cycle and number of developers, which could bring confounding effects when using code metrics for prediction. This paper proposes an improved confounding effect model, introducing a new confounding variable into the traditional model. On multiple projects, we experimentally analyzed the effect extent of the confounding variable. Furthermore, we verified that controlling confounding variables helps improve the predictive model's performance.

Keywords: defect prediction; code metrics; confounding effect; casual graph; half-sibling regression; logistic regression

Citation: Yuan, Y.; Li, C.; Yang, J. An Improved Confounding Effect Model for Software Defect Prediction. *Appl. Sci.* **2023**, *13*, 3459. https://doi.org/10.3390/app13063459

Academic Editors: Miltiadis G. Siavvas and Erol Gelenbe

Received: 11 February 2023
Revised: 6 March 2023
Accepted: 7 March 2023
Published: 8 March 2023

1. Introduction

Software defect prediction technology can effectively predict the defect proneness of software modules. Unlike the software preventive maintenance [1–3], which is applied after the delivery of software systems to detect and correct potential errors, software defect prediction aims to find potential software defects before software systems are delivered. Modules with a high defect proneness tend to be of greater concern for software testers, which significantly improves the efficiency of the software testing process [4]. Moreover, it also helps to find potential defects that could take more work to draw attention, thereby improving the quality of the software system [5,6]. In the past two decades, many researchers have used machine learning technology [7,8] to establish defect prediction models based on code metrics [9–13]. Software metrics objectively reflect the attributes of software modules, including Chidamber and Kemerer's object-oriented metrics, Halstead's complexity metrics and lines of code [9–13]. These metrics have been proven to be correlated with software defects and could be easily and automatically collected by software tools. So, researchers use code metrics to predict software defects.

In the prediction model based on code metrics, there are confounding effects among metrics, which will bring bias to the prediction results [14–16]. The confounding effect could be seen as multicollinearity between the confounder and code metrics. Emam et al. [14] strongly recommend removing confounding effects before building predictive models. Researchers point out that the size metric could be a significant confounder, affecting the predictive ability of other metrics. Zhou et al. [16] summarized the relevant research. Their paper described the confounding effect model of size in the form of a causal diagram [17], explaining the reasons for its confounding effects. Then they analyzed the extent of the size's confounding effects and proposed a linear-based method to remove them.

However, Kazman et al. [18] pointed out that existing factors affect the size, and the size is not the actual confounder. Under the analysis of multiple projects, they found that the software development cycle and the number of developers will affect the presentation of software code and directly affect the metrics, such as size. These factors are the actual confounders and are outside the scope of code metrics. These factors should be considered when analyzing the confounding effects in software defect prediction.

For the above problems, this paper proposes an improved confounding effect model and a predictive model under the controlling confounder, which could be adapted to the new model. Based on the causal graph theory [17], we make appropriate modifications to the traditional confounding model reported by Zhou et al. [16]. We introduce a new confounding variable, and the size variable is no longer considered the confounder. Due to Kazman et al.'s research [18], it is reasonably believed that edges should be added between the confounding variable and some software metrics variables. These edges satisfy the theoretical interpretability under the causal inference framework. Since the value of the confounding variable is unknown, we used the half-sibling regression method [19] to quantify its effect. The effect extent of the confounding variable was then analyzed with the help of logistic regression. Finally, we verified the effectiveness of the predictive model under controlling the confounding variable.

Our main contributions include the following aspects:

1. We noticed the new confounder in software defect prediction and proposed an improved confounding effect model.
2. We used half-sibling regression to quantify the confounding variable.
3. We experimentally analyzed the effect extent of the confounding variable, then verified the effectiveness of the proposed model for prediction.

The remainder of this paper is organized as follows. We first briefly review related work. After that, we introduce the traditional confounding effects in software defect prediction and propose both an improved confounding effect model and a prediction model based on it. Finally, comprehensive experiments are conducted for the analysis of confounding effects and prediction results.

2. Related Works

Code metrics have correlations with software defects, so researchers use these code metrics to train machine learning models for predicting [9–13].

In the past two decades, many researchers have applied different machine learning models as classifiers to obtain satisfactory prediction results [4,20,21]. Different classifiers perform well in software defect prediction content [22,23], such as logistic regression (LR) [24–27], neural network (NN) [28], support vector machines (SVMs) [29,30], naive Bayes [31], K-nearest neighbors [32], and Bayesian networks [22]. Logistic regression is the most commonly used classifier [33–35]. In recent years, Chi et al. [27] applied logistic regression to deal with software defect prediction, and they paid attention to the privacy protection issue in the prediction process. Combining logistic regression with homomorphic encryption technology, a method that can protect the privacy of datasets and models, was proposed. However, this method had no advantage over traditional logistic regression models in terms of predictive ability. Hailemelekot et al. [36] used logistic regression to deal with the problem of just-in-time defect prediction. The just-in-time defect prediction model classified potential defect changes at the check-in point. They identified six indicators from issue tracking systems to increase the indicator dataset and improve the performance of the JIT defect prediction model. Eivazpour et al. [37] used logistic regression technology to solve unbalanced data distribution challenge in the software defect prediction. Unbalanced data could cause the serious misclassification of software defects, thereby reducing the predictive model's performance. The authors combined logistic regression with cost-sensitive learning as the final classification model, effectively solving unbalanced data distribution. Rizal et al. [38] also used the logistic regression method in their article. Their paper was dedicated to solving unbalanced datasets and irrelevant features in software defect prediction. They used bagging techniques and genetic models, which can deal with class imbalance and feature selection issues. Their method improved the classification ability of the predictive model.

Feature selection methods are also applied in the field of software defect prediction. (1) In using filter-based feature ranking techniques, the information gain method [39] is an entropy-based method; the gain ratio method [40] compensates for the bias of information gain by penalizing multivalued features. (2) In filter-based feature subset selection, correlation-based feature subset selection [41] aims to identify subsets of features with a high correlation to class labels and a low correlation. Consistency-based feature subset selection [42] uses a metric called consistency to measure the quality of feature subsets. (3) The extraction-based feature selection method, a principal component analysis (PCA) method [43], is an extraction-based dimensionality reduction method, which converts the original variables that may be correlated with each other into a new set of the orthogonal variable. These new variables are called principal components.

However, Emma et al. [14] first declared that there are correlations between code metrics, which can cause confounding effects and bias prediction results. They investigated, identified, and examined the confounding effect of the size metric and declared that the size metric is a strong confounder. Additionally, they recommended considering removing the confounding effect of size before building a prediction model. Zhou et al. [15,16] systematically analyzed the extent of the confounding effect of different size metrics and proposed a linear regression-based method to remove the confounding effect of size. Their work significantly advances our knowledge about code metrics and their use in fault prediction. However, this practice is rarely taken by other researchers. We believe that this is because the existing confounding effect model and removal method are of little help for defect prediction. Different from their research, we focus on the confounding effects caused by factors other than size metrics, such as the project cycle and the number of developers mentioned by Kazman et al. [18]. These factors are outside the scope of code metrics, and we abstract these factors as a confounder. In this paper, we analyze the extent of the confounding effect of the proposed confounder and the help of controlling the confounder for predictive models. The literature on the confounding effect in software defect prediction is shown in Table 1:

Table 1. The literature on the confounding effect in software defect prediction.

No.	Title	Topic	Study
1	The confounding effect of size on the validity of object-oriented metrics	investigate ,identify and examine the confounding effect of size in software defect prediction	Emam et al.[14] IEEE Transactions on Software Engineering 2001
2	Examining the Potentially Confounding Effect of size on the Associations between Object-Oriented Metrics and Change-Proneness	examine the potentially confounding effects of three size metrics on the associations between OO metrics and defects.	Zhou et al. [15] IEEE Transactions on Software Engineering 2009
3	An in-depth study of the potentially confounding effect of size in fault prediction	Systematically analyze the extent of confounding effect of seven size metrics; propose a linear regression-based method to remove the confounding effect of size	Zhou et al. [16] ACM Transactions on Software Engineering and Methodology 2014

This paper uses LR as the basic classifier for three reasons: (1) LR is easy to understand, and it is convenient to analyze the confounding effect using LR. (2) The previous work is based on LR, which is convenient to compare with. (3) This is a preliminary exploration to verify the applicability of the proposed improved confounding effect model. The follow-up work will focus on the confounding effect removal method for other complex classifiers.

LR is used as the classifier in all existing related papers about confounding problems in software defect prediction. So, this paper also uses logistic regression as the basic classifier.

The causal graph [17] is a powerful tool for analyzing the effects of confounding. Based on causal inference, Bernhard et al. [19] proposed HSR, a method for analyzing confounding noise from unobserved variables, and they applied HSR in a challenging astronomy application. Since this method has an excellent theoretical basis and strong applicability, the HSR method is also applicable for solving problems in other fields. Yang et al. [44] used HSR for denoise word embeddings. In this paper, we use half-siblingbrother regression to quantify the confounding noise of virtual metrics.

3. Confounding Effect Model for Software Defect Prediction

Researchers have paid attention to the confounding effect in software defect prediction, which will bias the prediction results. This section first introduces the traditional confounding effect model and proposes an improved model. The data analysis method for the confounder is then described. Finally, a predictive model is proposed.

3.1. Traditional Confounding Effect Model

In the traditional confounding effect model, size is considered a confounder, affecting the predictive ability of software metrics and bringing bias to the prediction results. In the study by Zhou et al. [16], a causal diagram was used to describe the confounding effect of size, as shown in Figure 1a. In this figure, variables represent the size, metric, and defect-proneness, respectively. The unidirectional edge represents causal belief, connecting the cause and effect of an event. Bidirectional edges represent general relationships. According to the graph's connectivity, there are paths m-s-d and m-d. It is known from the nature of the causal graph that when m is used to predict d, s will have a confounding effect and bias the results.

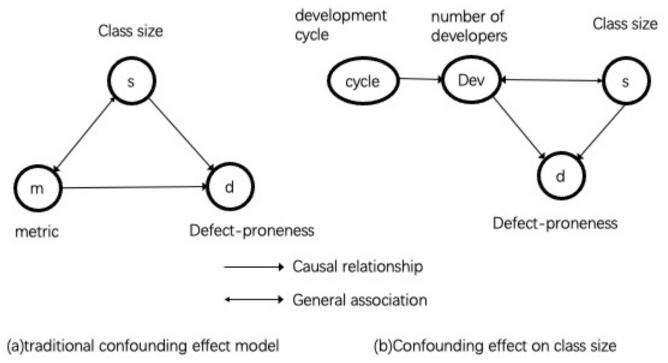

Figure 1. Confounding effect model in software defect prediction.

However, according to Kazman et al.'s paper [18], there are deeper affecting factors beyond size, as shown in Figure 1b. They concluded that those factors, such as the software development cycle and the number of developers, cause confounding in size and software defects. These factors affect the value of code metrics and are outside the scope of traditional code metrics.

3.2. Improved Confounding Effect Model

This paper proposes an improved confounding effect model, represented by a causal diagram, as shown in Figure 2.

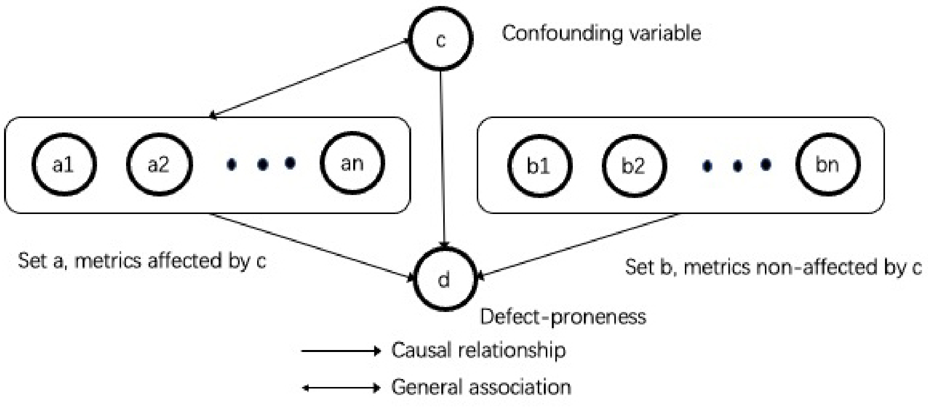

Figure 2. An improved confounding effect model.

In Figure 2, we introduce a new confounding variable signed by c, representing the confounding effect caused by unknown factors other than existing code metrics, including the development cycle and the number of developers mentioned above. Moreover, we divide the software metrics into two sets: a and b. variable a is pointed to by the confounding variable c, while variable b is not. Variable c also directly points to the software defect signed by d. Since there are paths a-d and a-c-d, the predictive power of variable a on d is confounded by c. We will analyze which software metrics belong to set a and which belong to set b in the following sections.

3.3. Data Analysis Method

A simple and parsimonious approach is as follows: a measured confounding variable can be controlled through a regression adjustment [45]. A regression adjustment entails including the confounder as another independent variable in a regression model. Suppose the regression coefficient of the metric changes dramatically with and without the size variable. In that case, this strongly indicates that there was indeed a confounding effect [46]. This is further elaborated below. The regression method used in this paper is logistic regression.

The general form of logistic regression(LR) is

$$p = \frac{1}{1 + e^{-(b + a_1 X_1 + \cdots + a_n X_n)}} \tag{1}$$

where p represents the probability that a software module contains at least one potential defect, and X_i are independent variables, which represent the software metrics. The parameters are estimated through the maximization of a log-likelihood [46].

Since the value of the confounding variable is unknown, we first quantified its confounding effect using half-sibling regression [19]. Half-sibling regression is a model used to quantify unobserved confounding noise under causal inference, which can incorporate causal beliefs and assumptions of empirical targets. Both the confounding variable and software metrics are considered responsible for software defects under causal belief. Therefore, the confounding effect can be reconstructed. We replace the value of the confounder with a confounding effect. When we utilize software metrics for defect prediction, the discrepancy between the predicted probability and the actual value is considered to be caused by the unknown confounding variable. The model assumes that the prediction noise is additive. Therefore, the following formula for variable C is obtained:

$$c = d - \frac{1}{1 + e^{-(b + a_1 X_1 + \cdots + a_n X_n)}} \tag{2}$$

We establish univariate and bivariate logistic regression, corresponding to uncontrolled and controlled variable c, respectively. The effect of the confounding variable is analyzed by analyzing the change of the X_1 coefficients of the two models:

$$p_1 = \frac{1}{1 + e^{-(b+a_1X_1)}} \tag{3}$$

$$p_2 = \frac{1}{1 + e^{-(b+a_1X_1+a_2C)}} \tag{4}$$

The odds ratio, as shown in the Formula (5), is the ratio of the probability that an event will happen (signed by p) to the probability that an event will not happen (signed by q). The extent of the confounding can be expressed by the change in the odds ratio when the X1 variable changes by one standard deviation($\varphi(X_1 + \sigma)$), denoted by $\Delta\varphi$ in Formula (7). Since we construct two models, as shown in Formula (3) and Formula (4), which are with and without control for the confounder, we denote the change in odds ratio as $\Delta\varphi_1$ and $\Delta\varphi_2$, respectively. As suggested in [47], we can evaluate the extent to which the change in odds ratio (shown as Formula (8)) changes as an indication of the extent of confounding. We operationalize this as follows:

$$\frac{p}{q} = \varphi(X) = e^{b+aX} \tag{5}$$

$$\varphi(X + \sigma) = e^{b+a(X+\sigma)} \tag{6}$$

$$\Delta\varphi = \frac{\varphi(X + \sigma)}{\varphi(X)} = e^{a\sigma} \tag{7}$$

$$\Delta^2\varphi = |\frac{\Delta\varphi_1 - \Delta\varphi_2}{\Delta\varphi_2}| * 100\% \tag{8}$$

3.4. Predictive Model with Controlling Confounder

We use Y to represent the software defect, and X represents the existing software metrics. Y has two values which are 0 and 1. Y = 1 means the relevant software model has one defect or more than one defect, and Y = 0 stands for the relevant software model with non-defect. When we use X to train the model for predicting Y. The prediction model parameters are distorted due to the confounding effect. In order to revise the model parameters, we train the prediction model under controlling the confounder. This way, the model parameters can more accurately and appropriately reflect the relationship between software metrics and software defects. As the formula shows,

$$E(Y|X, C') = \frac{1}{1 + e^{-(b+a_1X_1+\cdots+a_nX_n+a_{n+1}C')}} \tag{9}$$

where C' is the transformed value of the confounding noise C. For instances that contain at least one defect, we convert Q_1 to dQ_1, and for instances that do not contain defects, we convert it to nQ_1. Parameters d, n are constants. The grid search method is used to determine the values of d and n. Under the grid search, suitable values of d, n are determined, which helps the prediction model achieve high performance. Based on the above, we propose the Algorithm 1 presents the pseudocode.

Algorithm 1 Predictive model.

```
Input: Training data ($X$ represents software metrics; $Y$ represents defect-proneness; $C$ and
    $C'$ present confounding noise)
Output: Prediction model
1: Use logistic regression to fit $Y$ by $X$, obtain $Y := f(X)$
2: Predict Y by X, obtain $E[Y|X] := f(X)$
3: Calculate C based on half-sibling regression, obtain $C := Y - E[Y|X]$
4: Transform $C$ to $C'$. If $Y$==1, $C'$ equals to $d$*$C$; if $Y$==0, $C'$ equals to $n$*$C$
5: Determine values of d and n using the grid search method, which could get $f(X, C')$
    fitting $Y$ well.
6: Establish the prediction model by logistic regression, obtain $Y := f(X, C')$
```

4. Experiments and Results

4.1. Datasets

In this study, experimental datasets are from the Metric Data Program (MDP) repository. MDP is widely used in the field of software defect prediction. The original version of the datasets contains a large number of null values, wrong values. The preprocessing strategy is that the problem instances are discarded first, and then the instances that are not problematic but do not help improve the defect prediction are removed. For example, instances with either implausible values or conflicting feature values are logically erroneous, so they are removed. The preprocessing procedure is described in detail in the [48]. Martin et al. [48] provided a cleaned version, and this paper uses this clean version of the MDP datasets as the experimental datasets. Table 2 briefly describes the datasets. This table includes information, such as project name, number of instances, number of metrics, number of defective instances, number of non-defective instances, and defect rate. It is worth noting that the scales in different projects are very different, such as the number of instances ranging from 127 to 17001, and the defect rate ranging from 3 to 35%. Table 3 shows the metrics in the datasets.

Table 2. Datasets from MDP.

Dataset	Instances	Metrics	Defect	Non-Defect	Defect Rate
CM1	344	37	42	302	12%
JM1	9593	21	1759	7834	18%
KC1	2096	21	325	1771	16%
MC2	127	39	44	83	35%
PC3	1125	37	140	985	12%
PC5	17,001	36	503	16,498	3%

Table 3. Code metrics.

No.	MetricName	Description
1	LOC_BLANK	number of blank lines
2	BRANCH_COUNT	number of branches
3	CALL_PAIRS	pairs of call
4	LOC_CODE_AND_COMMENT	number of lines of code and comments
5	LOC_COMMENTS	nomber ofcomment lines
6	CONDITION_COUNT	number of conditional statement
7	CYCLOMATIC_COMPLEXITY	cyclomatic complexity
8	CYCLOMATIC_DENSITY	circle density

Table 3. *Cont.*

No.	MetricName	Description
9	DECISION_COUNT	number of decisions
10	DECISION_DENSITY	decision complexity
11	DESIGN_COMPLEXITY	design complexity
12	DESIGN_DENSITY	design density
13	EDGE_COUNT	number of boundary
14	ESSENTIAL_COMPLEXITY	Intrinsic complexity
15	ESSENTIAL_DENSITY	Intrinsic density
16	LOC_EXECUTABLE	number of executable lines
17	PARAMETER_COUNT	number of parameters
18	GLOBAL_DATA_COMPLEXITY	global data complexity
19	GLOBAL_DATA_DENSITY	global data density
20	HALSTEAD_CONTENT	content metric
21	HALSTEAD_DIFFICULTY	complexity
22	HALSTEAD_EFFORT	programming efficiency
23	HALSTEAD_ERROR_EST	misprediction
24	HALSTEAD_LENGTH	program length
25	HALSTEAD_LEVEL	programming language class
26	HALSTEAD_PROG_TIME	how long wrote the program
27	HALSTEAD_VOLUME	program capacity
28	MAINTENANCE_SEVERITY	maintenance severity
29	MODIFIED_CONDITION_COUNT	modify the number of conditional statements
30	MULTIPLE_CONDITION_COUNT	number of conditional statements
31	NODE_COUNT	number of nodes
32	NORMALIZED_CYLOMATIC_ COMPLEXITY	canonical cyclomatic complexity
33	NUM_OPERANDS	number of operands
34	NUM_OPERATORS	number of operators
35	NUM_UNIQUE_OPERANDS	number of special operands
36	NUM_UNIQUE_OPERATORS	number of special operators
37	NUMBER_OF_LINES	number of rows
38	PERCENT_COMMENTS	percent of comments
39	LOC_TOTAL	total lines of code

4.2. Presence and Extent of Confounding Effect

We analyze the extent of the confounding effect on each metric using the method introduced in Section 3.3. The data results are shown in Table 4. The indexes in Table 4 correspond to those of Table 3. Since the number of metrics collected by each project varies, we use #NA to represent empty items, which means the metric is not included in this project. It can be observed that the range of change in the odds ratio is widely distributed.

The greater the change in the odds ratio, the greater the effect of the confounding variable on the metric's predictive ability, and the greater the change in the odds ratio, the stronger the effect is considered to be.Conversely, the smaller the change in the odds ratio, the smaller the extent of the confounding variable on the metric's predictive ability, and it is considered that the confounding effect is weaker. For example, in project CM1, we declare that the confounding variable has a stronger effect on BRANCH_COU than LOC_BLA. We set the threshold value to 20%. That is, when the odds ratio change is greater than 20% before and after the confounding variable is controlled, we consider that the confounder will have a significant effect on the metric. For example, in project CM1, the confounding variable significantly affects HALSTEAD_DIFFICU but not HALSTEAD_EFFO. It can be concluded from Table 4 that the confounding variable generally has confounding effects on different metrics.

Table 4. Rate of change in odds ratio for each metric in different project.

No. of Metrics	CM1	JM1	KC1	MC2	PC3	PC5
1	33.0%	100.0%	88.2%	100.0%	211.3%	48.6%
2	100.0%	6.3%	90.3%	100.0%	53.7%	134.1%
3	32.7%	#N/A	#N/A	96.3%	3.8%	89.2%
4	100.0%	100.0%	149.4%	100.0%	99.7%	100.0%
5	99.7%	100.0%	63.0%	100.0%	43.6%	99.9%
6	99.9%	#N/A	#N/A	100.0%	95.2%	133.1%
7	100.0%	18.6%	90.6%	100.0%	55.0%	119.9%
8	2.3×10^{34}%	#N/A	#N/A	73,222.6%	42.6%	73.7%
9	99.9%	#N/A	#N/A	100.0%	274.5%	124.7%
10	1.3×10^{17}%	#N/A	#N/A	70.0%	100.0%	#N/A
11	27.8%	114.0%	85.7%	100.0%	61.0%	134.0%
12	73.3%	#N/A	#N/A	295.7%	70.4%	1165.0%
13	100.0%	#N/A	#N/A	100.0%	109.3%	76.5%
14	100.0%	35.6%	83.1%	100.0%	12.3%	139.0%
15	4.9×10^{5}%	#N/A	#N/A	93.9%	63.0%	37.0%
16	100.0%	41.7%	95.9%	100.0%	145.0%	28.2%
17	100.0%	#N/A	#N/A	99.6%	50.8%	2.1×10^{5}%
18	#N/A	#N/A	#N/A	100.0%	#N/A	129.4%
19	#N/A	#N/A	#N/A	94.7%	#N/A	81,595.0%
20	100.0%	100.0%	99.4%	100.0%	757.4%	12.3%
21	23.0%	15.9%	99.2%	100.0%	47.0%	99.7%
22	4.7%	69.4%	91.6%	100.0%	1.6×10^{7}%	100.0%
23	11.1%	99.9%	91.7%	100.0%	248.7%	10.5%
24	10.9%	91.3%	95.3%	100.0%	276.4%	4.4%
25	4160.7%	122.7%	2.4×10^{23}%	1.7×10^{49}%	10.6%	2.6×10^{33}%
26	4.7%	69.4%	91.6%	100.0%	1.6×10^{7}%	100.0%
27	11.2%	99.9%	91.6%	100.0%	250.3%	10.6%
28	11,250.5%	#N/A	#N/A	99.8%	66.3%	96.5%
29	99.9%	#N/A	#N/A	100.0%	50.9%	137.3%
30	99.9%	#N/A	#N/A	100.0%	23.2%	123.6%
31	100.0%	#N/A	#N/A	100.0%	155.6%	67.2%
32	51.5%	#N/A	#N/A	6086.0%	54.9%	#N/A
33	11.4%	99.9%	94.0%	100.0%	184.6%	8.1%
34	11.0%	82.0%	94.3%	100.0%	471.5%	6.8%
35	100.0%	100.0%	97.4%	100.0%	205.6%	#N/A
36	14.2%	1.4%	99.9%	100.0%	64.7%	100.0%
37	4.5%	#N/A	#N/A	100.0%	138.0%	18.3%
38	100.0%	#N/A	#N/A	88.9%	100.0%	100.0%
39	100.0%	100.0%	92.7%	100.0%	129.8%	28.8%

For different projects, confounders have a general effect on metrics. This bias results when using code metrics for prediction. The scope of confounding effects varies from project to project. We take projects CM1 and JM1 as examples, combined with our confounding effect model proposed in Section 3.2. The confounder significantly affects 28 metrics on project CM1, which belong to set a corresponding to the model; 9 metrics have little effect and belong to set b. In project JM1, 17 metrics have a significant impact and belong to set a corresponding to the model; 4 indicators have little influence and belong to set b. Numbers of metrics with and without confounding effects in each project are present in Figure 3.

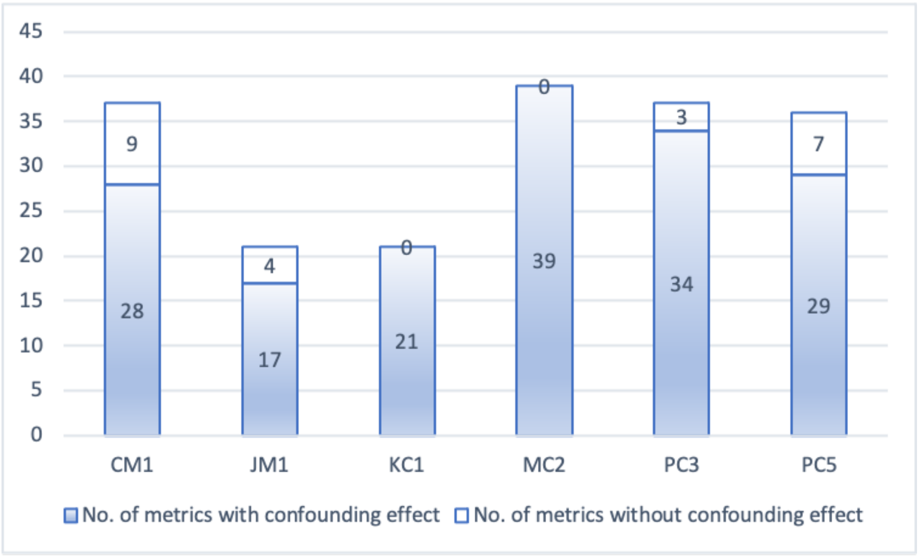

Figure 3. No. of metrics with and without confounding effect.

The same metric is affected differently in each project. For example, LOC_BLANK is affected in all six projects, and BRANCH_COUNT is significantly affected in the five projects, except project JM1. Different metrics have distinct confounding effects in different projects. This situation also commonly exists in the research of Kazman et al., which is related to the metrics sets collected by each project. The metrics statistics in the different projects are shown in Figure 4, in which the indexes correspond to Table 3. In our experiments, the size is strongly affected by confounding variables in every project, which is also consistent with the research conclusions of the work of Zhou et al. [16] and Kazman et al. [18].

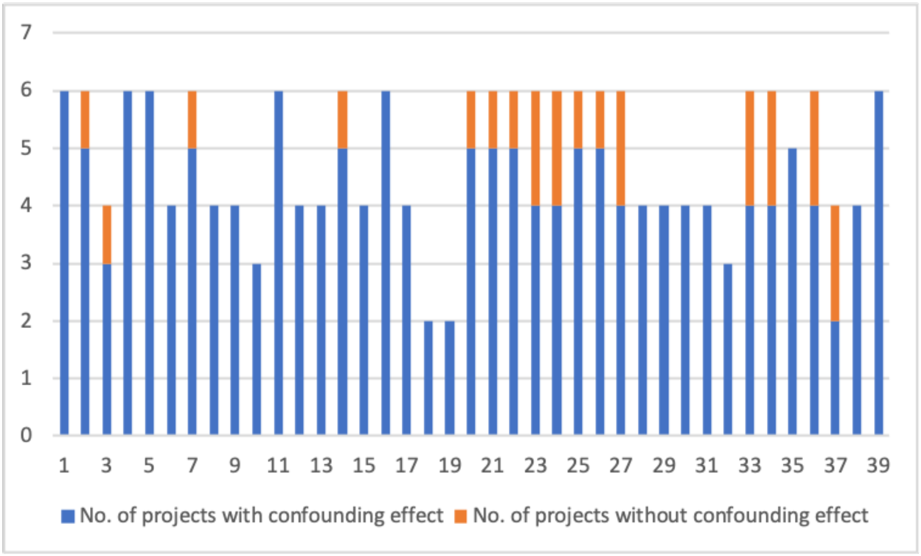

Figure 4. No. of projects for each metric with and without confounding effect.

4.3. Experiments for the Proposed Prediction Model

4.3.1. Experiments Set

To verify the effectiveness of the LR model under controlling the confounding variable, we compared it with traditional LR and LR with the confounding effect removal method, SVM and NN. The removal method was proposed by Zhou et al. and can remove the confounding effects of size. We randomly selected 70% of the instances for training and the remaining 30% for testing. Precision and recall rates are indicators for evaluating classification models. The F1 score considers both precision and recall rates, which can be regarded as a harmonic average of them. We employ the widely used F1 score as the main indicator for objectively evaluating each predictive model. Its maximum value is 1, and its minimum value is 0. The criterion for the F1 score is that the larger the F1 score, the better the model performance. We run the experiments on the R platform.

The description of baselines is as follows:

1. LR: A two-step logistic regression is widely used in software defect prediction content. First, for each code metric, build a univariate logistic regression against defect proneness; second, metrics with significant correlations (p-value < 0.05) are used to establish a multivariate logistic regression to predict the defects.
2. LCERM+LR: Before applying the two-step LR model mentioned above, the LCERM, a method which could remove the confounding effects of the size metric, is applied. The LCERM uses linear regression to fit the size metric and one other metric, and this linear relationship between them is seen as the confounding effect. The removal of the confounding effect is achieved by subtracting the fitted value from the metric value. More descriptions can be seen in [16]. This paper uses the LOC(line of code) metric as the size metric.
3. SVM: To improve the predictive ability of the SVM model, we oversample the defect instances, standardize the original data, and perform principal component analysis transformation. The first five components are applied to the SVM model.
4. NN: To improve the predictive ability of the NN model, we oversample the defect instances, standardize the original data, and perform principal component analysis transformation. The first five components are applied to the NN model.

4.3.2. Parameters Selection

When controlling the confounding variables, we need to make appropriate changes to the effect values. The values of parameters d and n will affect the transformation, and different parameter values will lead to different model prediction performance results. In order to choose the value of d and n reasonably, we conducted experiments. On each dataset, we perform gridded experiments with d and n ranging from 0 to 1 with a step size of 0.1, respectively. The experimental results for project JM1 are presented in Figure 5, which is a heatmap, where the darker color means the larger F1 score. The abscissa represents the value of n, and the ordinate represents the value of d. By observation, we conclude the general rule that when n is larger than d, the models generally achieve better predictive ability. Without loss of generality, parameter d is set to equal the defect rate, and parameter n equals the non-defect rate in the following experiment.

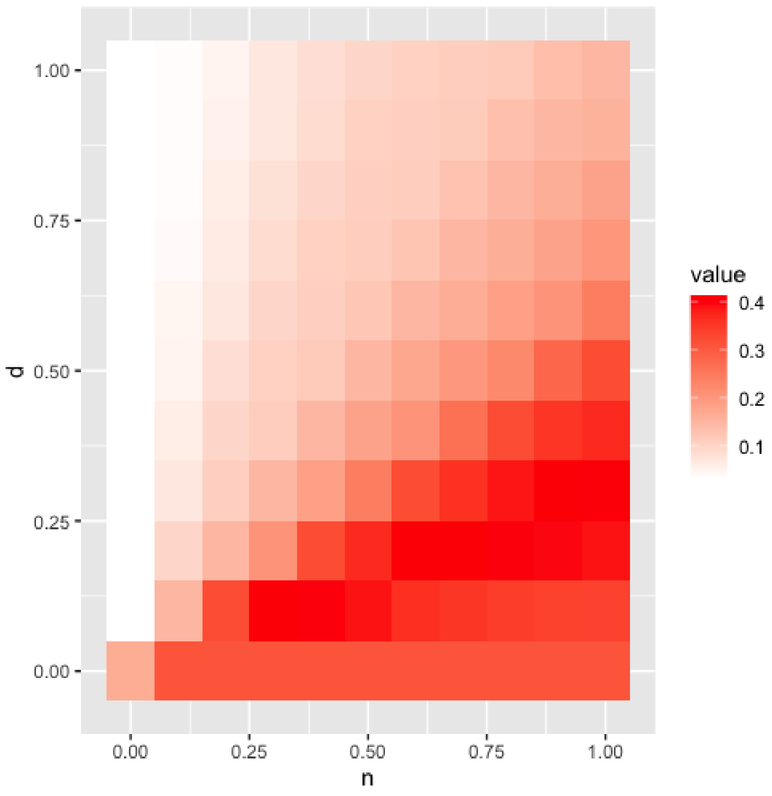

Figure 5. Heatmap for JM1 for different d and n values in strategy 2.

4.3.3. Presents of Prediction Result

This section shows the prediction results of our proposed model and the baseline models. LR, CM-LR, SVM, and NN represent the baseline models, and HSR-LR represents the proposed model.

Case 1: CM1

CM1 contains 344 instances with a defect rate of 12%. The prediction results of different models on CM1 are shown in Table 5. Our proposed model outperformed the baseline models under the F1 score.

Table 5. Results on CM1.

Model	Precision	Recall	F1
LR	50.0%	23.1%	0.316
CM-LR	50.0%	23.1%	0.316
SVM	31.0%	62.7%	0.415
NN	29.6%	65.8%	0.408
HSR-LR	40.0%	46.2%	0.429 *

The * indicates the best performance of the models.

Case 2: JM1

JM1 contains 9593 instances with a defect rate of 18%. The prediction results of different models on JM1 are shown in Table 6. Our proposed model outperformed the baseline models under the F1 score.

Table 6. Results on JM1.

Model	Precision	Recall	F1
LR	55.7%	8.3%	0.145
CM-LR	55.7%	8.3%	0.145
SVM	29.3%	61.7%	0.397
NN	28.6%	64.2%	0.395
HSR-LR	28.4%	72.0%	0.407 *

The * indicates the best performance of the models.

Case 3: KC1

KC1 contains 2096 instances with a defect rate of 16%. The prediction results of different models on KC1 are shown in Table 7. Our proposed model outperformed the baseline models under the F1 score.

Table 7. Results on KC1.

Model	Precision	Recall	F1
LR	58.6%	22.0%	0.319
CM-LR	58.6%	22.0%	0.319
SVM	29.7%	76.1%	0.427
NN	29.0%	80.0%	0.424
HSR-LR	30.7%	72.3%	0.430 *

The * indicates the best performance of the models.

Case 4: MC2

MC2 contains 127 instances with a defect rate of 35%. The prediction results of different models on MC2 are shown in Table 8. Our proposed model outperformed the baseline models under the F1 score.

Table 8. Results on MC2.

Model	Precision	Recall	F1
LR	48.4%	52.1%	0.498
CM	47.5%	50.0%	0.484
SVM	52.1%	50.7%	0.502
NN	47.7%	53.6%	0.493
HSR-LR	51.7%	58.2%	0.541 *

The * indicates the best performance of the models.

Case 5: PC3

PC3 contains 1125 instances with a defect rate of 12%. The prediction results of different models on PC3 are shown in Table 9. Our proposed model outperformed the baseline models in the F1 score.

Table 9. Results on PC3.

Model	Precision	Recall	F1
LR	52.0%	22.6%	0.312
CM-LR	52.0%	22.6%	0.312
SVM	27.7%	77.0%	0.407
NN	25.1%	77.4%	0.375
HSR-LR	34.1%	61.5%	0.436 *

The * indicates the best performance of the models.

Case 6: PC5

PC5 contains 17,001 instances with a defect rate of 3%. The prediction results of different models on PC5 are shown in Table 10. Our proposed model outperformed the baseline models in the F1 score.

Table 10. Results on PC5.

Model	Precision	Recall	F1
LR	56.0%	29.4%	0.384
CM-LR	56.1%	29.4%	0.384
SVM	20.5%	91.4%	0.334
NN	28.1%	91.7%	0.429
HSR-LR	32.7%	80.0%	0.463 *

The * indicates the best performance of the models.

The above results present the performance of our proposed model and baseline models. We have the following observations and analysis:

(1) Compared with all baseline models, our proposed HSR-LR model performs best under the F1 score, and it does not obtain the best performance in precision and recall rate. Compared with the best baseline model, the F1 score of HSR-LR increases 1% in CM, 1.1% in JM1, 0.3% in KC1, 3.9% in MC2, 2.9% in KC3 and 3.4% in KC5, with an average of 2.2%.

(2) HSR-LR performs better than LR, which verifies that the confounding effects mentioned in this paper affect the LR prediction ability. With the help of HSR, we can quantify the confounding effect. Under controlling the confounding effect, LR significantly improves.

(3) Tables 5–10 show that HSR-LR and LR have similar precision rates, and each has wins and losses but not much difference. However, HSR-LR has better recall performance than that of LR. These points are why HSR-LR has better F1 values; that is to say, controlling the confounder can effectively increase the recall rate of LR, thereby increasing the F1 value of the LR model.

(4) HSR-LR performs better than CM-LR, indicating that our proposed confounding effect model is more suitable for software defect prediction content than the traditional confounding effect model. CM-LR performs similarly to lR, indicating that the existing confounding removal methods are unsuitable for software defect prediction, which further illustrates the necessity of this paper. Based on the proposed model, HSR-LR could help solve this issue.

(5) Compared to SVM and NN, our model also achieves better performance. Both SVM and NN are commonly used classifiers in the field of software defect prediction. On six projects, our model outperforms SVM by an average of 3.7% and outperforms NN by an average of 2.5% under F1 scores.

5. Internal and External Validity

Here, we consider threats to our study's internal and external validity. Internal validity is the degree to which confounding noise accurately measures the intended concept of confounding effect and the degree to which conclusions can be drawn about its causal effect on code metrics. External validity is the degree to which research findings can be generalized to the study population and other research settings.

The most important threat to the internal validity of this study is the impact of the method of computing confounding noise on the analysis of confounding effects. Before analyzing the confounding effect, we quantify its value using half-sibling regression. Under the definition of half-sibling regression, there are other existing noise effects, whereas these noises are ignored, and their effects are assumed to be caused by the confounder. This allows the quantified value to correlate with the code metrics' predictive ability, affecting the conclusions obtained.In future work, we will investigate more appropriate methods for quantifying the effect of confounding.

The most important threat to the external validity of this study is that our results may not generalize to other systems. Although our research investigates some systems, the definition of the effects of confounding noise beyond code metrics in this paper is relatively simple, and it can be more specific when facing different project contexts and human factors. In subsequent research, we will focus on this issue.

6. Conclusions and Future Work

This confounding effects analysis is intended as a first step and certainly not a final decision. With an in-depth understanding of the confounding effect, the prediction model's performance is bound to be further improved. Moreover, we could unravel the underlying causal dynamics if we collect longitudinal data on similar variables. However, the critical point we made here is the introduction of unknown confounding effects in code metrics-based software defect prediction for the first time.

In future, we will further analyze the confounding effects in software defect prediction and focus on suitable methods to quantify and remove the confounding effects for different classifiers. Additionally, we will attempt to apply the research conclusions to other applications.

Author Contributions: Conceptualization, Y.Y. and J.Y.; methodology, C.L. and Y.Y.; software C.L.; writing—original draft preparation, C.L., Y.Y. and J.Y.; writing—review and editing, C.L. and J.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by the National Social Science Fund of China (No. 21CGJ006).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Jiang, Y.; Li, X. Broadband cancellation method in an adaptive co-site interference cancellation system. *Int. J. Electron.* **2022**, *109*, 854–874. [CrossRef]
2. Lei, W.; Hui, Z.; Xiang, L.; Zelin, Z.; Xu-Hui, X.; Evans, S. Optimal remanufacturing service resource allocation for generalized growth of retired mechanical products: Maximizing matching efficiency. *IEEE Access* **2021**, *9*, 89655–89674. [CrossRef]
3. Ban, Y.; Liu, M.; Wu, P.; Yang, B.; Liu, S.; Yin, L.; Zheng, W. Depth estimation method for monocular camera defocus images in microscopic scenes. *Electronics* **2022**, *11*, 2012. [CrossRef]
4. Wahono, R.S. A systematic literature review of software defect prediction. *J. Softw. Eng.* **2015**, *1*, 1–16.
5. Kitchenham, B.; Pfleeger, S.L. Software quality: The elusive target [special issues section]. *IEEE Softw.* **1996**, *13*, 12–21. [CrossRef]
6. Gruhn, V. Validation and verification of software process models. In *European Symposium on Software Development Environments*; Springer: Berlin/Heidelberg, Germany, 1991; pp. 271–286.
7. Heckman, J.J. Sample selection bias as a specification error. *Econom. J. Econom. Soc.* **1979**, *47*, 153–161. [CrossRef]
8. Huang, J.; Gretton, A.; Borgwardt, K.; Schölkopf, B.; Smola, A. Correcting sample selection bias by unlabeled data. *Adv. Neural Inf. Process. Syst.* **2006**, *19*, 601–608.
9. Catal, C.; Diri, B. A systematic review of software fault prediction studies. *Expert Syst. Appl.* **2009**, *36*, 7346–7354. [CrossRef]
10. Catal, C. Software fault prediction: A literature review and current trends. *Expert Syst. Appl.* **2011**, *38*, 4626–4636. [CrossRef]
11. Radjenović, D.; Heričko, M.; Torkar, R.; Živkovixcx, A. Software fault prediction metrics: A systematic literature review. *Inf. Softw. Technol.* **2013**, *55*, 1397–1418. [CrossRef]
12. Malhotra, R. A systematic review of machine learning techniques for software fault prediction. *Appl. Soft Comput.* **2015**, *27*, 504–518. [CrossRef]
13. Pandey, S.K.; Mishra, R.B.; Tripathi, A.K. Machine learning based methods for software fault prediction: A survey. *Expert Syst. Appl.* **2021**, *172*, 114595. [CrossRef]
14. Emam, K.E.; Benlarbi, S.; Goel, N.; Rai, S.N. The confounding effect of size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.* **2001**, *27*, 630–650. [CrossRef]
15. Zhou, Y.; Leung, H.; Xu, B. Examining the potentially confounding effect of size on the associations between object-oriented metrics and change-proneness. *IEEE Trans. Softw. Eng.* **2009**, *35*, 607–623. [CrossRef]

16. Zhou, Y.; Xu, B.; Leung, H.; Chen, L. An in-depth study of the potentially confounding effect of size in fault prediction. *ACM Trans. Softw. Eng. Methodol.* **2014**, *23*, 1–51. [CrossRef]
17. Helmert, M. A planning heuristic based on causal graph analysis. In Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), Whistler, BC, Canada, 3–7 June 2004.
18. Kazman, R.; Stoddard, R.; Danks, D.; Cai, Y. Causal modeling, discovery, & inference for software engineering. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 20–28 May 2017; pp. 172–174.
19. Schölkopf, B.; Hogg, D.W.; Wang, D.; Foreman-Mackey, D.; Janzing, D.; Simon-Gabriel, C.J.; Peters, J. Modeling confounding by half-sibling regression. *Proc. Natl. Acad. Sci. USA* **2016**, *113*, 7391–7398. [CrossRef]
20. Pachouly, J.; Ahirrao, S.; Kotecha, K.; Selvachandran, G.; Abraham, A. A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Eng. Appl. Artif. Intell.* **2022**, *111*, 104773. [CrossRef]
21. Jorayeva, M.; Akbulut, A.; Catal, C.; Mishra, A. Machine learning-based software defect prediction for mobile applications: A systematic literature review. *Sensors* **2022**, *22*, 2551. [CrossRef]
22. Okutan, A.; Yıldız, O.T. Software defect prediction using bayesian networks. *Empir. Softw. Eng.* **2014**, *19*, 154–181. [CrossRef]
23. Wang, S.; Yao, X. Using class imbalance learning for software defect prediction. *IEEE Trans. Reliab.* **2013**, *62*, 434–443. [CrossRef]
24. Emam, K.E.; Melo, W.; Machado, J.C. The prediction of faulty classes using object-oriented design metrics. *J. Syst. Softw.* **2001**, *56*, 63–75. [CrossRef]
25. Basili, V.R.; Briand, L.C.; Melo, W.L. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.* **1996**, *22*, 751–761. [CrossRef]
26. Olague, H.M.; Etzkorn, L.H.; Gholston, S.; Quattlebaum, S. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Trans. Softw. Eng.* **2007**, *33*, 402–419. [CrossRef]
27. Yu, C.; Ding, Z.; Chen, X. Hope: Software defect prediction model construction method via homomorphic encryption. *IEEE Access* **2021**, *9*, 69405–69417. [CrossRef]
28. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software defect prediction via convolutional neural network. In Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 25–29 July 2017; pp. 318–328.
29. Goyal, S. Effective software defect prediction using support vector machines (svms). *Int. J. Syst. Assur. Eng. Manag.* **2022**, *13*, 681–696. [CrossRef]
30. He, C.; Xing, J.; Zhu, R.; Li, J.; Yang, Q.; Xie, L. A new model for software defect prediction using particle swarm optimization and support vector machine. In Proceedings of the 2013 25th Chinese Control and Decision Conference (CCDC), Guiyang, China, 25–27 May 2013; pp. 4106–4110.
31. Zhu, K.; Zhang, N.; Ying, S.; Wang, X. Within-project and cross-project software defect prediction based on improved transfer naive bayes algorithm. *Comput. Mater. Contin.* **2020**, *63*, 891–910.
32. Goyal, S. Handling class-imbalance with knn (neighbourhood) under-sampling for software defect prediction. *Artif. Intell. Rev.* **2022**, *55*, 2023–2064. [CrossRef]
33. Goyal, J.; Sinha, R.R. Software defect-based prediction using logistic regression: Review and challenges. In *Second International Conference on Sustainable Technologies for Computational Intelligence: Proceedings of ICTSCI 2021*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 233–248.
34. Hall, T.; Beecham, S.; Bowes, D.; Gray, D.; Counsell, S. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.* **2011**, *38*, 1276–1304. [CrossRef]
35. Shatnawi, R.; Li, W. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *J. Syst. Softw.* **2008**, *81*, 1868–1882. [CrossRef]
36. Tessema, H.D.; Abebe, S.L. Enhancing just-in-time defect prediction using change request-based metrics. In Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 9–12 March 2021; pp. 511–515.
37. Eivazpour, Z.; Keyvanpour, M.R. Cssg: A cost-sensitive stacked generalization approach for software defect prediction. *Softw. Testing, Verif. Reliab.* **2021**, *31*, e1761. [CrossRef]
38. Bahaweres, R.B.; Suroso, A.I.; Hutomo, A.W.; Solihin, I.P.; Hermadi, I.; Arkeman, Y. Tackling feature selection problems with genetic algorithms in software defect prediction for optimization. In Proceedings of the 2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), Jakarta, Indonesia, 19–20 November 2020; pp. 64–69.
39. Cover, T.M. *Elements of Information Theory*; John Wiley & Sons: Hoboken, NJ, USA, 1999.
40. Quinlan, J.R. *C4. 5: Programs for Machine Learning*; Elsevier: Amsterdam, The Netherlands, 2014.
41. Hall, M.A. *Correlation-Based Feature Selection of Discrete and Numeric Class Machine Learning*; University of Waikato, Department of Computer Science: Hamilton, New Zealand, 2000.
42. Dash, M.; Liu, H. Consistency-based search in feature selection. *Artif. Intell.* **2003**, *151*, 155–176. [CrossRef]
43. Bro, R.; Smilde, A.K. Principal component analysis. *Anal. Methods* **2014**, *6*, 2812–2831. [CrossRef]

44. Yang, Z.; Liu, T. Causally denoise word embeddings using half-sibling regression. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 9426–9433.
45. Schlesselman, J.J. *Case-Control Studies: Design, Conduct, Analysis*; Oxford University Press: Oxford, UK, 1982; Volume 2.
46. Hosmer, D.W., Jr.; Lemeshow, S.; Sturdivant, R.X. *Applied Logistic Regression*; John Wiley & Sons: Hoboken, NJ, USA, 2013; Volume 398.
47. Sangi-Haghpeykar, H.; Poindexter, A.N., III. Epidemiology of endometriosis among parous women. *Obstet. Gynecol.* **1995**, *85*, 983–992. [CrossRef] [PubMed]
48. Shepperd, M.; Song, Q.; Sun, Z.; Mair, C. Data quality: Some comments on the nasa software defect datasets. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1208–1215. [CrossRef]