Computers, Materials & Continua CMC, vol.63, no.2, pp.891-910, 2020

Within-Project and Cross-Project Software Defect Prediction Based on Improved Transfer Naive Bayes Algorithm

Kun Zhu[1], Nana Zhang[1], Shi Ying[1,*] and Xu Wang[2]

Abstract: With the continuous expansion of software scale, software update and maintenance have become more and more important. However, frequent software code updates will make the software more likely to introduce new defects. So how to predict the defects quickly and accurately on the software change has become an important problem for software developers. Current defect prediction methods often cannot reflect the feature information of the defect comprehensively, and the detection effect is not ideal enough. Therefore, we propose a novel defect prediction model named ITNB (Improved Transfer Naive Bayes) based on improved transfer Naive Bayesian algorithm in this paper, which mainly considers the following two aspects: (1) Considering that the edge data of the test set may affect the similarity calculation and final prediction result, we remove the edge data of the test set when calculating the data similarity between the training set and the test set; (2) Considering that each feature dimension has different effects on defect prediction, we construct the calculation formula of training data weight based on feature dimension weight and data gravity, and then calculate the prior probability and the conditional probability of training data from the weight information, so as to construct the weighted bayesian classifier for software defect prediction. To evaluate the performance of the ITNB model, we use six datasets from large open source projects, namely Bugzilla, Columba, Mozilla, JDT, Platform and PostgreSQL. We compare the ITNB model with the transfer Naive Bayesian (TNB) model. The experimental results show that our ITNB model can achieve better results than the TNB model in terms of accurary, precision and pd for within-project and cross-project defect prediction.

Keywords: Cross-project defect prediction, transfer Naive Bayesian algorithm, edge data, similarity calculation, feature dimension weight.

1 Introduction

As the software scale and its complexity increase, the number of defects generated will increase dramatically. Based on the software defect problem in the current software development field, some researchers have proposed software defect prediction technology,

[1] School of Computer Science, Wuhan University, Wuhan, 430072, China.

[2] Department of Computer Science, Vrije University Amsterdam, Amsterdam, 1081HV, The Netherlands.

[*] Corresponding Author: Shi Ying. Email: yingshl@whu.edu.cn.

Received: 26 July 2019; Accepted: 10 September 2019.

which is to mine and extract historical information and code information in the process of software project development, and analyze the strongly related information, and then establish a specific prediction model by mathematical statistics, machine learning and other methods to predict the software defects [Malhotra and Khanna (2017)].

Software defect prediction has the following research significance: (1) Reasonable allocation of test resource. Defect prediction technology can help testers to locate the position and number of software modules in advance, so as to allocate test resources reasonably, improve test efficiency and save cost [Li, Zhang, Wu et al. (2012)]. (2) Discover software defects in time and improve the quality of software products. (3) Help the new project to establish a reasonable cross-project prediction model through other project information.

Most of the current defect prediction methods are for the within-project defect prediction. Some defect data in the same project are used as the training set to build the prediction model, and the remaining small number of data are used as test set to test the performance of the prediction model. However, it is often a new project that needs to be predicted in the actual development. Due to relatively few historical information of the newly established project, the training data that can be extracted is too small to conduct accurate defect prediction. Therefore, we can use historical defect data from other projects to construct a reasonable defect prediction model for the new project by learning the idea of transfer learning, that is, cross-project defect prediction [Xu, Liu, Luo et al. (2018)]. Since application scenarios, development environment, developers and development languages between two different projects are not necessarily the same, the features of the dataset between the source project and the target project tend to be quite different [Herbold, Trautsch and Grabowski (2017)], so how to transfer effective feature from the source project to construct the prediction model of the target project will be a challenge.

Due to the similarity of data sets on the within-project defect prediction, it is relatively simple to extract relevant features, and can achieve better prediction result. However, in terms of cross-project defect prediction, because of the large difference between two different projects, it is particularly important to extract the common features between source project and target project [Zhang, Keivanloo and Zou (2017)]. The current transfer Naive bayesian model mainly extracts features from common feature spaces between different projects, and transfers effective information to reduce data differences, but sometimes discarded data may be useful data, resulting in relatively poor prediction effect [Ma, Luo, Zeng et al. (2012)].

In order to solve the problem that the transfer Naive bayesian model cannot fully reflect the data feature and the detection effect is not ideal enough, we propose a novel defect prediction model named ITNB (Improved Transfer Naive Bayes) based on improved transfer Naive Bayesian algorithm in this paper.

The main contributions of this paper are as follows:

(1) we propose a novel defect prediction model named ITNB based on improved transfer Naive Bayesian algorithm in this paper. The model removes the edge data in the test set when calculating the data similarity between the training set and the test set, and constructs the calculation formula of training data weight based on feature dimension weight and data gravity.

(2) We conduct a large number of within-project and cross-project software defect prediction experiments on six datasets from large open source projects, and compare the ITNB model with the transfer Naive Bayesian (TNB) model. The experimental results show that our ITNB model can achieve better results than the TNB model in terms of accurary, precision and pd.

The rest of this paper is organized as follows. Section 2 describes the background and related work. Section 3 details the proposed ITNB model. Section 4 shows the experimental setup, including data sets and evaluation metrics. Section 5 evaluates the performance of our ITNB model. Section 6 describes the threats to our work. We conclude this paper and describe future work in Section 7.

2 Background & related work

2.1 Within-project defect prediction

At present, some researchers mainly use the machine learning algorithm to construct the defect prediction model on the within-project defect prediction. In addition, how to optimize the data structure and extract effective feature are also the focus of current research. Some important research works will be summarized below.

Elish et al. [Elish and Elish (2008)] use support vector machine (SVM) to conduct defect prediction and compare its predictive performance with eight statistical and machine learning models on four NASA data sets. Lu et al. [Lu, Kocaguneli and Cukic (2014)] leverage active learning to predict defect, and they also use feature compression techniques to make feature reduction on defect data. Li et al. [Li, Zhang, Wu et al. (2012)] propose a novel semi-supervised learning method-ACoForest, which can sample the prediction modules that are most helpful for learning. Rodriguez et al. [Rodriguez, Herraiz, Harrison et al. (2014)] compare different methods for different data preprocessing problems, such as sampling method, cost sensitive method, integration method and hybrid method. The final experimental results show that the above different methods can effectively improve the accuracy of defect prediction after performing the class imbalance. Seiffert et al. [Seiffert, Khoshgoftaar, Van Hulse et al. (2014)] analyze 11 different algorithms and 7 different data sampling techniques, and find that class imbalance and data noise would have the negative impact on prediction performance.

2.2 Cross-project defect prediction

For cross-project software defect prediction, since application scenarios, development environment, developers and development languages between two different projects are not necessarily the same, the features of the dataset between the source project and the target project tend to be quite different, so how to transfer effective feature from the source project to construct the prediction model of the target project will be a challenge.

Briand et al. [Briand, Melo and Wust (2002)] first propose the earliest cross-project defect prediction, which uses logistic regression and MARS (Multivariate Adaptive Regression Splines) to construct the defect prediction model for the Xpose project, and conducts perform prediction for the Jwrite project. They find that the performance of cross-project defect prediction is lower than that of within-project defect prediction, and the main reason for this result is the feature difference between different projects.

Ma et al. [Ma, Luo, Zeng et al. (2012)] propose a transfer Naive Bayesian (TNB) model, which sets weight for instance in the source project, and effectively improves the accuracy of defect prediction. But the TNB model only uses the maximum and minimum values of the feature to construct the feature weight, and cannot fully reflect the features of the data.

Watanabe et al. [Watanabe, Kaiya and Kaijiri (2008)] propose a defect prediction method based on different metric elements, which corrects metric element values of the target project and the source project, and they find that both the accuracy and the recall are significantly improved. Jureczko et al. [Jureczko and Madeyski (2010)] use k-means and kohonen neural network to analyze multiple related projects. The experimental results show that predicting a project using multiple similar projects can achieve better prediction effect. Cheng et al. [Cheng, Wu and Yuan (2016)] calculate the difference between the source project and the target project, and then convert it into weight information, so as to establish the defect prediction model. Chen et al. [Chen, Fang, Shang et al. (2015)] argue that the distribution disparity between cross-company data and within-company data often makes it difficult to establish high-quality cross-project defect prediction model. They narrow the gap by reducing the negative samples for the cross-company data, thereby improving the performance of cross-project defect prediction. Abaei et al. [Abaei, Rezaei and Selamat (2013)] propose the self-organizing mapping (SOM) prediction model with the threshold, which can help testers to mark modules without the need of experts.

3 Methodolody

We propose the ITNB model based on improved transfer Naive Bayesian algorithm. The model removes the edge data in the test set when calculating the data similarity between the training set and the test set, and constructs the calculation formula of training data weight based on feature dimension weight and data gravity.

For within-project defect prediction, we group the data into the training set and the test set according to the time sequence. For cross-project defect prediction, we use the data sets constructed by different projects as the training set and test set respectively for experiments.

The workflow of the ITNB model is shown in Fig. 1. The model consists of the following four steps: (a) Feature extraction; (b) Data preprocessing; (c) Improved transfer Naive Bayesian model construction; (d) Within-project and cross-project defect prediction.

3.1 Feature extraction

Because the quantity and quality of features will directly affect the final prediction effect, we must first extract effective features from the software history repository.

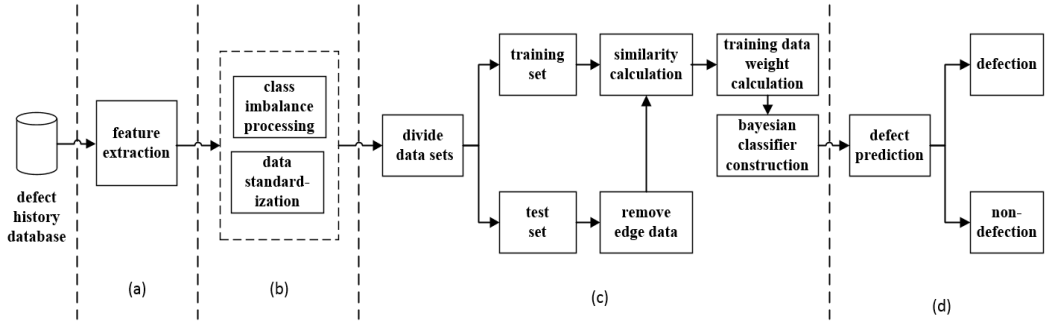

Figure 1: Framework of the ITNB model

We first extract various quantifiable change features from the software history repository (i.e., the change of known label), which can distinguish whether the change is defective or not. In this paper, we use 11 basic defect change features proposed by Kamei et al. [Kamei, Shihab, Adams et al. (2012)], these features can be classified into the following four dimensions: diffusion, size, purpose and experience, as shown in Tab. 1.

Table 1: Descriptions of 11 basic change features

Dimension	Name	Description
Diffusion	NS	The number of modified subsystems
	ND	The number of modified directories
	NF	The number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix
Experience	EXP	Developer experience
	REXP	Developer experience on a subsystem
	SEXP	Developer experience on a subsystem

3.2 Data preprocessing

The data preprocessing in this paper includes two parts: class imbalance processing and data standardization.

3.2.1 Class imbalance processing

Class imbalance is a common problem in software defect data sets. The distribution of software defect in the project is roughly in line with the pareto principle, that is, 20% of the program modules contain about 80% of the defects. For the defect prediction data set, the number of defective modules (a few class) is less than that of the non-defective modules (majority class). If there is the serious class imbalance problem in the software

defect data set, the prediction effect of the model is relatively poor.

In this paper, we adopt the SMOTE (Synthetic Minority Oversampling Technique) method [Chawla, Bowyer, Hall et al. (2002)] to conduct the class imbalance processing. This method is an improved scheme based on random oversampling algorithm. Because random oversampling adopts the strategy of simply copying samples to increase the number of a few class samples, the model is easy to overfit, and the SMOTE algorithm can avoid the problem of overfitting to a certain extent. The basic idea of SMOTE algorithm is to analyze a few class samples and manually synthesize new samples based on a few class samples, and add them to the dataset at the same time. In summary, the SMOTE algorithm synthesizes new samples for a few class based on interpolation.

This step is critical for software defect prediction, because it helps the trained classifier does not bias towards non-defective modules (majority class), thereby improving the performance of software defect prediction.

3.2.2 Data standardization

Since the distribution of the 11 basic defect change feature values extracted is of large difference, even not in the same order of magnitude, if the original measure value is used for analysis, the function of the higher value in the comprehensive analysis will be highlighted, and the function of the lower value is relatively weakened.

Therefore, in order to ensure the reliability of the results, we adopt the min-max standardization method [Nam, Pan and Kim (2013)], which can make the values of each metric in the same dimension and can be consistent with feature values of the original distribution.

The calculation equation for the min-max normalization method is as shown in Eq. (1):

$$\widetilde{S}_{\mathrm{i}}^{\mathrm{j}} = \frac{S_i^j - min(S^j)}{max(S^j) - min(S^j)} \tag{1}$$

where $max(\boldsymbol{s}^{\boldsymbol{j}})$ and $min(\boldsymbol{s}^{\boldsymbol{j}})$ are the maximum and minimum values in the vector $\boldsymbol{s}^{\boldsymbol{j}}$, respectively.

3.3 Improved transfer naive Bayesian model construction

After data preprocessing, we will construct a weighted Bayesian model for these numerical features. We first remove the edge data in the test set when calculating the similarity between the test set and the training set. Then, based on the feature dimension weight and the data gravity, we construct the calculation formula of the training data weight. Finally, we calculate the prior probability and conditional probability of training data based on the weight information, so as to construct the weighted Bayesian classifier for software defect prediction.

3.3.1 Similarity calculation

Because the features between the training set and the test set may be different, especially development languages, development process and developers among different projects are different, and the feature differences are even greater. Therefore, we first need to

calculate the similarity between the training set features and the test set features.

Before calculating the feature difference between the training set and the test set, we find that the proportion of edge data is often less by analyzing the data distribution of each feature, and the maximum and minimum vectors constructed by a small amount of data will affect the prediction effect, so we first need to remove these edge data.

The values of each feature in the test set need to be rounded approximately, and divided into [0,10] intervals, namely [0,1], (1,2], ..., (9,10]. Then we calculate the percentage of each feature in each interval, and we remove the intervals where the percentage is less than 5%. We construct the maximum vector and minimum vector by taking the maximum value and minimum value of each feature in the test set with edge data removed.

After obtaining the maximum and minimum vectors, we assume that the features of each dimension have the same effect on the classifier. For each training instance, we can obtain the similarity of the training set and the test set by calculating the position of each feature between the maximum value vector and the minimum value vector.

In this paper, we define each instance as $A_i=\{a_{i1},a_{i2},...,a_{ik}\}$, a_{ij} is the jth feature of A_i, and k is the number of features. For each instance A_i in the training set, we calculate the number of similar features, the equation is as shown in Eq. (2) [Ma, Luo, Zeng et al. (2012)]:

$$S_{\mathrm{i}} = \sum_{j=1}^{k} f(a_{ij}) \tag{2}$$

where $f(a_{ij}) = \begin{cases} 1 & \text{when } min_{\mathrm{j}} \le a_{ij} \le max_j \\ 0 & \text{other cases} \end{cases}$, a_{ij} is the jth feature of instance A_i.

Then we can calculate whether each feature value in each instance Ai is between the maximum vector and the minimum vector of each feature defined in the test set according to the above formula. If the feature value of the instance is between the maximum vector and the minimum vector, the similarity of the instance is increased by 1, otherwise the similarity is unchanged. In this way, we can calculate the similarity value of each instance in the training set.

The pseudo code of the similarity calculation algorithm between training set and test set is as shown in the algorithm 1.

Through the pseudo code of the algorithm 1, we can summarize the steps of similarity calculation as follows: (1) Remove a small amount of edge data in the test set; (2) Calculate the maximum vector and the minimum vector of each feature in the test set; (3) Calculate the similarity of each instance data in the training set by comparing the maximum vector and the minimum vector.

Algorithm 1 Similarity calculation algorithm between training set and test set

Input:

training set: $Tr=\{tr_1, tr_2, ..., tr_n\}$; test set: $T=\{t_1, t_2, ..., t_m\}$; the number of features in the training set and test set: k

Output:

D: the set of instance similarity in the training set: S[i]

```
len1←the length of training set Tr;
len2←the length of test set T;
Define the two-dimensional array arr1[len2][k] in the test set;
Define the two-dimensional array arr2[len2][k] of interval values into which the
 feature values in the test set are converted;
Define the array arrMax[k] of the largest vector in the test set;
Define the array arrMin[k] of the smallest vector in the test set;
Define the similarity array S[len1] for each instance in the training set;
# Calculate the interval value of the feature in the test set
for i←0 to len2-1 do
    for j←0 to k-1 do
        arr1[i][j]←the jth feature value in the test set;
        arr2[i][j]←the interval value into which the jth feature value in the test set
        is converted;
    end for
end for
# Remove edge data in array arr2[i][j]
if the percentage of the arr2[i][j] interval <5%  then
    Remove arr2[i][j] interval;
else
    Save arr2[i][j] interval;
end if
# Calculate the maximum vector and minimum vector of each feature in the test set
for i←0 to len2-1  do
    for j←0 to k-1 do
        if arr1[i][j]>arrMax[j] then
            arrMax[j]← arr1[i][j];
        end if
        if arr1[i][j]<arrMin[j] then
```

```
28:          arrMin[j]←arr1[i][j];
29:       end if
30:    end for
31: end for
32: # Calculate the similarity of each instance data in the training set
33: for i←0 to len1-1 do
34:    for j← 0 to k-1 do
35:       if arr[i][j] >=arrMin[j] && arr[i][j] <=arrMax[j] then
36:          S[i]=S[i]+1;
37:       end if
38:    end for
39: end for
40: return S[i]
```

3.3.2 Training data weight calculation

After calculating the similarity values of each instance in the training set, we convert these similarity values into the weights of training data, which are mainly constructed by feature dimension weight and data gravity.

Different from the traditional transfer Naive Bayesian model [Ma, Luo, Zeng et al. (2012)], we consider the feature dimension weight in this paper. We leverage feature dimension information to construct dimension-based weight. In the Section 3.1, we can distinguish the feature values extracted by different methods into four dimensions according to their features, each of which contains a part of the feature value information.

We obtain the dimension weight of the training instance by calculating the weights of four dimensions of a training instance respectively. The specific method is as follows: First, we calculate whether each feature value in the first dimension is between the maximum value and minimum value of the corresponding feature in the test instance. If it is between the maximum value and the minimum value, the similarity of the feature is 1, otherwise 0. Then, when obtaining the similarity of each feature value in one dimension, we calculate the weighted average value of these feature and take it as the weight value of these features in the dimension. Finally, we repeat the above method to obtain the weighted average values of other dimensions, so as to get the weight value of each dimension. The dimension weight equation of one training instance is as shown in Eq. (3):

$$w_{di} = \frac{w_{i1} + w_{i2} + \ldots + w_{in}}{n} \tag{3}$$

where i is the number of one instance, n is the number of dimensions, which is 4 in this paper, and w_{in} is the weight of one dimension.

In order to effectively transfer the information in the test set, we leverage the idea of data gravity to construct the data weight. Data gravity refers to the idea of universal gravitation used in data analysis to simulate the gravity between data [Peng, Yang, Chen et al. (2009)].

The weight between the training data and the test data is like the gravitation F in the universal gravitation. We assume that the quality of one feature in the instance is M, then the quality of the test data is kmM, and the quality of the training data between the maximum vector and the minimum vector is MS_i. Therefore, the weight w_i of the training instance Ai is inversely proportional to $r_2=(k-S_i+1)^2$, and is proportional to kS_imM^2 [Ma, Luo, Zeng et al. (2012)]. Combined with the information of the above dimension weights, we further conclude that it is proportional to $w_{di}S_i$. Therefore, the weighting equation of one training instance A_i is defined as shown in Eq. (4):

$$w_i = \frac{m_1 m_2}{r_i^2} = \frac{k S_i m M^2}{(k - S_i + 1)^2} \propto \frac{w_{di} S_i}{(k - S_i + 1)^2} \tag{4}$$

where S_i is the similarity value of each instance A_i, and k is the number of feature.

From the above formula, we can know that the more similar the feature data of the training instance A_i is to the test set, the greater the weight of the instance A_i will be. When the value of the similarity S_i is equal to the number k of features, all the features are located between the maximum vector and the minimum vector. At this time, the $k-S_i+1$ is equal to 1, and the feature weight will be the largest and only affected by the weight value of the dimension.

Then we need to calculate the prior probability based on the weighted data. The individual probability on the right side of the original bayesian prior probability equation is estimated based on the weighted data. In order to reflect the class distribution of the test data, we need to modify the original bayesian equation. If the training data is more similar to the test data, we need to give the training data a higher weight, and assign higher weight to the class of the training data. According to Frank et al. [Frank, Hall and Pfahringer (2002)], we rewrite the equation of prior probability as shown in Eq. (5):

$$P(y) = \frac{\sum_{i=1}^{n} w_i \lambda(y_i, y) + 1}{\sum_{i=1}^{n} w_i + n_y} \tag{5}$$

where w_i is the weight of the training instance, y_i is the class value of the training instance A_i, n is the total number of training instances, n_y is the total number of classes, and the function $\lambda(y_i, y)$ is an index function, when $y_i=y$, $\lambda(y_i,y)=1$, otherwise $\lambda(y_i,y)=0$.

The equation for the conditional probability is rewritten as shown in Eq. (6):

$$\mathrm{P}(a_j \mid y) = \frac{\sum_{i=1}^{n} w_i \lambda(a_{ij}, a_j) \lambda(y_i, y) + 1}{\sum_{i=1}^{n} w_i \lambda(y_i, y) + n_j} \tag{6}$$

where a_{ij} is the jth feature value in the ith training instance, a_j is the jth feature value, and n_j is the number of different values of the jth feature.

3.3.3 Weighted bayesian classifier construction

We define $\mathrm{Tr}=\{(x_1,y_1),(x_2,y_2),...,(x_n,y_n)\}$ as the source project data set, where x_i represents the ith instance, y_i is the feature of the instance x_i, and n is the number of the data, $y_i \in$

(true,false). We define the test data set as T=$\{t_1,t_2,...,t_m\}$, where m is the number of instances in the test dataset. Based on the prior probability P(y) and the conditional probability P($a_j \mid y$) in the above section, we can define the following bayesian classifier to classify the instance t in the test dataset [Ma, Luo, Zeng et al. (2012)]:

$$y(t) = arg \max_{y \in Y} P(y \mid t) = arg \max_{y \in Y} \frac{P(y)\prod_{j=1}^{k} P(a_j \mid y)}{\sum_{y \in Y} P(y)\prod_{j=1}^{k} P(a_j \mid y)} \tag{7}$$

where t=$\{a_1,a_2,...,a_k\}$, a_j is the jth feature of the test data instance t, k is the number of features, $P(y \mid u)$ represents the posterior probability, and $P(y)$ represents the priori probability, P($a_j \mid y$) represents the conditional probability.

3.4 Within-project and cross-project defect prediction

When constructing the defect prediction model, we need to verify the validity of the model. We conduct the within-project defect prediction and the cross-project defect prediction to verify our ITNB model, respectively.

For within-project defect prediction, we consider that many new technologies and development methods will affect the new code submitted after the update, and even introduce new defects in software development. Therefore, we first sort the data according to time sequence, and then divide the data of two years into a group, and select the data of the previous year as the training set and the data of the next year as the test set. This not only guarantees the integrity of a project development cycle, but also guarantees that the amount of data for two years in the training set is sufficient enough, so as to make the result of the within-project defect prediction more accurate.

For cross-project defect prediction, we combine six data sets in pairs as training set and test set respectively, and use our ITNB model to conduct defect prediction on any pair of combinations. This can not only make the amount of data sufficient in the training set and the test set, but also ensure the diversity of instances.

The pseudo code for our ITNB is shown in the algorithm 2:

4 Experimental setup

In this section, we will introduce the experimental setup, including data sets, evaluation metrics. We conduct the experiments on a 3.6 GHz i7-4790 CPU machine with 8 GB RAM.

4.1 Data sets

In this paper, we use six datasets from large open source projects, namely Bugzilla, Columba, Mozilla, JDT, Platform and PostgreSQL, which are large, well-known and long-term projects covering a wide range of fields and scales [Kamei, Shihab, Adams et al. (2012)].

Algorithm 2 ITNB

Input:

training set: $Tr = \{tr_1, tr_2..., tr_n\}$; test set: $T = \{t_1, t_2, ..., t_m\}$

Output:

result set: R

1: **for** one training instance $tr_i \in Tr$ **do**
2: Calculate the data similarity of the instance A_i by the Eq. (2);
3: **end for**
4: **for** one training instance $tr_i \in Tr$ **do**
5: Calculate the dimension weight of the instance A_i by the Eq. (3);
6: Calculate the data weight of the instance A_i by the Eq. (4);
7: **end for**
8: Construct a weighted bayesian model by the data weight of the training instance A_i, Eqs. (5) and (6);
9: **for** one test instance $t_i \in T$ **do**
10: Conduct within-project defect prediction and cross-project defect prediction by Eq. (7);
11: Store the defect prediction result in R;
12: **end for**
13: **return** R

Tab. 2 lists the statistics for six datasets from these projects. As can be seen from Tab. 2, the first column and the second column are the project name and the time period for collecting changes, respectively. The third to sixth columns are the total number of changes, the percentage of defect-inducing changes, the average LOC for each change, and the number of files modified on average per change. The total number of changes in these six data sets are from 4,455 to 98,275, which are very helpful for us to conduct empirical research. All datasets in this paper are unbalanced, and the percentage of defect-inducing changes ranges from 5% to 36%. Therefore, both the training set and the test set need to perform class imbalance processing first.

To make our results more reliable, we use 10 times 10-fold cross-validation to evaluate the performance of the ITNB model, so each dataset is randomly divided into 10 folds, where 9 folds are used as the training dataset and the remaining 1 fold is used as the test dataset. To further reduce experimental error, we perform 10 times cross-validation and record the average performance.

Table 2: Statistics of the datasets

Project	Period	Total Changes	% of Defects	Avg LOC Per Change	#Modified Files Per Change
Bugzilla	08/1998-12/2006	4620	36%	37.5	2.3
Platform	05/2001-12/2007	64250	14%	72.2	4.3
Mozilla	01/2000-12/2006	98275	5%	106.5	5.3
JDT	05/2001-12/2007	35386	14%	71.4	4.3
Columba	11/2002-07/2006	4455	31%	149.4	6.2
PostgreSQL	07/1996-05/2010	20431	25%	101.3	4.5

4.2 Evaluation metrics

In order to evaluate the experimental results of the ITNB model, we use five metrics such as accurary, precision, recall, F1, and pf, which are widely used to evaluate the performance of software defect prediction [Menzies, Milton, Turhan et al. (2010); Monden, Hayashi, Shinoda et al. (2013); Zhong, Khoshgoftaar and Seliya (2004); Yang, Zhou, Liu et al. (2016); Zimmermann, Premraj and Zeller (2007)]. They can all be calculated from the confusion matrix of classification results. The confusion matrix is shown in Tab. 3. The calculation equation for these five metrics are as follows:

Table 3: Confusion matrix of classification results

Confusion matrix		Predicted	
		Positive(P)	Negative(N)
Actual	True(T)	TP	FN
	Flase(F)	FP	TN

accurary: In the defect prediction, the proportion of the correct result predicted by the predictor to the total program module, as shown in Eq. (8):

$$accurary = \frac{TP + TN}{TP + FN + FP + TN} \tag{8}$$

precision: The proportion of defective modules in the defective module predicted by the predictor, as shown in Eq. (9):

$$precision = \frac{TP}{TP + FP} \tag{9}$$

recall: In all defective modules, the proportion of defective modules predicted by the predictor, as shown in Eq. (10):

$$recall = \frac{TP}{TP + FN} \tag{10}$$

F1: The harmonic mean of precision and recall, as shown in Eq. (11):

$$F1 = \frac{2 \times precision \times recall}{precision + recall} \tag{11}$$

pf: In all non-defective modules, the proportion of defective modules predicted by the predictor. The smaller the pf value, the better the model prediction effect, as shown in Eq. (12):

$$pf = \frac{FP}{TN + FP} \tag{12}$$

5 Experimental results

In this section, we will introduce our experimental results. We evaluate and discuss the performance of our ITNB model by setting up the following three research questions (RQ).

RQ1: For within-project and cross-project defect prediction, can class imbalance processing improve the performance of the ITNB model?

For the class imbalance problem in the within-project and cross-project defect prediction, we use the SMOTE method to conduct class imbalance processing, as shown in Figs. 2 and 3. The evaluation metric values in the comparison figure is the average of the experimental results in the six data sets.

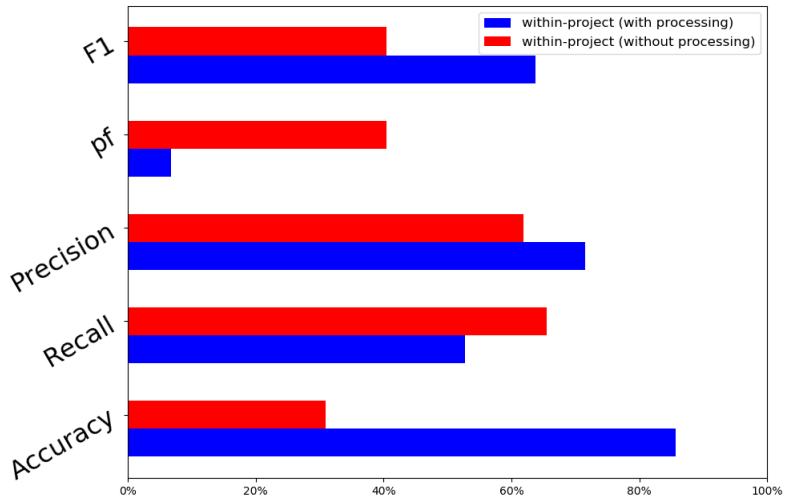

Figure 2: Comparison figure before and after class imbalance processing for within-project defect prediction

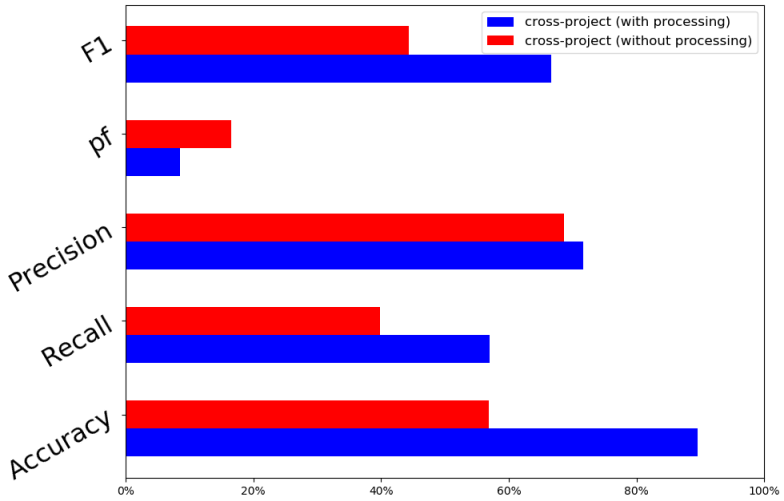

Figure 3: Comparison figure before and after class imbalance processing for cross-project defect prediction

From Figs. 2 and 3, we can find that the experimental effect on the data sets with class imbalance processing is significantly better in terms of accuracy, precision and F1 for within-project defect prediction and cross-project defect prediction. Only on the recall of within-project defect prediction, the effect on the data sets without class imbalance processing is better. This is because the similarity of the data sets for within-project is high, so it is better on the recall without class imbalance processing. However, due to the large difference between data sets, the effect is better on the recall with class imbalance processing for cross-project defect prediction.

Through the QR1, we can find that because the defection class is significantly less than the non-defection class, the final experimental results often cannot reflect the true prediction result of a few class. Therefore, it is very necessary for us to conduct class imbalance processing before defect prediction.

RQ2: Is the performance of our ITNB model better than the transfer naive Bayesian (TNB) model for within-project defect prediction?

For within-project defect prediction, we first sort the data according to time sequence, and then divide the data of two years into a group, and select the data of the previous year as the training set and the data of the next year as the test set. This not only guarantees the integrity of a project development cycle, but also guarantees that the amount of data for two years in the training set is sufficient enough.

We use the ITNB model and the TNB model to conduct within-project defect prediction based on the partitioned data. The experimental results are shown in Tab. 4.

From Tab. 4, we can find that the ITNB model is better than the TNB model on the accuracy, precision and pf. However, consider that the accuracy and precision may be more important in the defect prediction, so our ITNB model is very meaningful. We also find that in these experiments with a small amount of data, the final results are often poor.

For example, the prediction interval of Bugzilla is 2000.01-2001.12 and the prediction interval of Platform is 2007.01-2007.12, where the former has a small amount of data in the training set and the latter has a small amount of data in the test set.

Table 4: Comparison table of the experiment results for within-project defect prediction

Data sets	Time interval	precision		recall		accurary		pf		F1	
		ITNB	TNB	ITNB	TNB	ITNB	TNB	ITNB	TNB	ITNB	TNB
Bug	2000.01-2001.12	**0.489**	0.442	0.072	**0.124**	**0.378**	0.333	**0.123**	0.201	0.126	**0.194**
	2002.01-2003.12	**0.687**	0.631	0.098	**0.201**	**0.421**	0.398	**0.071**	0.104	0.173	**0.305**
	2004.01-2005.12	**0.777**	0.753	0.253	**0.341**	**0.489**	0.457	**0.120**	0.198	0.382	**0.469**
	2006.01-2006.12	**0.845**	0.841	0.417	**0.453**	**0.617**	0.598	**0.106**	0.156	0.559	**0.589**
Col	2004.01-2005.12	**0.911**	0.900	0.434	**0.487**	**0.633**	0.623	**0.064**	0.089	0.588	**0.632**
	2006.01-2006.12	**0.925**	0.912	0.586	**0.611**	**0.743**	0.719	**0.060**	0.109	0.717	**0.732**
JDT	2003.01-2004.12	0.920	**0.921**	**0.582**	0.578	**0.744**	0.698	**0.060**	0.167	**0.713**	0.710
	2005.01-2006.12	**0.913**	0.894	**0.635**	0.619	**0.772**	0.749	**0.069**	0.098	**0.749**	0.732
	2007.01-2007.12	**0.915**	0.887	**0.646**	0.639	**0.784**	0.778	**0.066**	0.145	**0.757**	0.743
Moz	2002.01-2003.12	**0.913**	0.902	0.811	**0.849**	**0.862**	0.849	**0.083**	0.102	0.859	**0.860**
	2004.01-2005.12	**0.930**	0.891	0.825	**0.883**	0.878	**0.883**	**0.066**	0.089	**0.874**	0.869
	2006.01-2006.12	**0.910**	0.901	0.753	**0.812**	**0.837**	0.812	**0.077**	0.092	**0.824**	0.813
Pla	2003.01-2004.12	**0.923**	0.921	0.588	**0.598**	**0.749**	0.698	**0.058**	0.113	0.718	**0.725**
	2005.01-2006.12	**0.926**	0.897	0.646	**0.663**	**0.776**	0.753	**0.064**	0.103	0.761	**0.762**
	2007.01-2007.12	0.302	**0.321**	0.191	**0.201**	**0.876**	0.795	**0.048**	0.093	0.234	**0.247**
Pos	1998.01-1999.12	**0.937**	0.895	0.527	**0.601**	**0.700**	0.649	**0.051**	0.078	0.675	**0.719**
	2000.01-2001.12	**0.951**	0.921	0.511	**0.532**	**0.693**	0.632	**0.039**	0.067	0.664	**0.674**
	2002.01-2003.12	**0.955**	0.945	0.583	**0.641**	**0.737**	0.702	**0.040**	0.059	0.724	**0.764**
	2004.01-2005.12	**0.930**	0.887	0.602	**0.652**	**0.745**	0.667	**0.061**	0.108	0.731	**0.752**
	2006.01-2007.12	**0.935**	0.921	**0.600**	0.598	**0.752**	0.694	**0.054**	0.106	**0.731**	0.725
	2008.01-2009.12	0.925	**0.931**	**0.671**	0.643	**0.792**	0.734	**0.064**	0.113	**0.778**	0.761
	2010.01-2010.05	**0.930**	0.927	0.554	**0.559**	**0.745**	0.721	**0.046**	0.087	0.695	**0.697**
	Average	**0.857**	0.838	0.527	**0.552**	**0.715**	0.680	**0.068**	0.113	0.638	**0.658**

RQ3: Is the performance of our ITNB model better than the TNB model for cross-project defect prediction?

For cross-project defect prediction, we combine six data sets in pairs as training set andtest set respectively, and use the ITNB model and the TNB model to conduct defect prediction on any pair of combinations. This can not only make the amount of data sufficient in the training set and the test set, but also ensure the diversity of instances. The experimental results are shown in Tab. 5.

From Tab. 5, we can find that our ITNB model is better than the TNB model on accuracy, precision and pf. The TNB model is better than the ITNB model in terms of recall and F1. Considering that the ITNB model removes edge data when calculating similarity, it is likely to remove some useful data, so ITNB model is not able to find as many defects as possible on recall. Although we remove some test set data for similarity calculation, the removal of some useless information and the reassigned weight of each dimension make the weight of each feature no longer the same, which improves the accuracy and precision of the final

prediction results.

Table 5: Comparison table of the experimental results for cross-project defect prediction results

Data sets	Test sets	precision		recall		accurary		pf		F1	
		ITNB	TNB	ITNB	TNB	ITNB	TNB	ITNB	TNB	ITNB	TNB
Bug	Col	**0.860**	0.833	0.390	**0.401**	**0.603**	0.601	**0.091**	0.112	0.537	**0.541**
	JDT	**0.883**	0.879	0.546	**0.637**	0.716	**0.718**	**0.085**	0.125	0.675	**0.739**
	Moz	**0.881**	0.867	0.739	**0.751**	**0.815**	0.759	**0.106**	0.156	0.804	**0.805**
	Pla	0.873	**0.881**	0.552	**0.589**	**0.715**	0.711	**0.095**	0.123	0.676	**0.706**
	Pos	**0.889**	0.859	0.515	**0.549**	**0.686**	0.669	**0.085**	0.091	0.652	**0.670**
Col	Bug	**0.929**	0.893	0.216	**0.398**	**0.510**	0.459	**0.026**	0.067	0.350	**0.551**
	JDT	**0.923**	0.916	0.534	**0.558**	**0.725**	0.659	**0.052**	0.119	0.677	**0.694**
	Moz	**0.911**	0.897	0.734	**0.749**	**0.827**	0.763	**0.075**	0.121	0.813	**0.816**
	Pla	**0.931**	0.882	0.530	**0.601**	**0.725**	0.698	**0.046**	0.068	0.675	**0.715**
	Pos	**0.901**	**0.901**	0.505	**0.583**	**0.688**	0.682	**0.068**	0.070	0.650	**0.708**
JDT	Bug	**0.922**	0.887	0.390	**0.491**	**0.606**	0.574	**0.052**	0.099	0.548	**0.632**
	Col	**0.870**	0.849	0.415	**0.487**	**0.618**	0.601	**0.089**	0.128	0.562	**0.619**
	Moz	**0.869**	0.801	0.837	**0.886**	**0.851**	0.831	**0.133**	0.157	**0.852**	0.832
	Pla	**0.931**	0.873	0.674	**0.721**	**0.797**	0.772	**0.059**	0.089	0.782	**0.790**
	Pos	**0.875**	0.821	0.631	**0.649**	**0.737**	0.668	**0.121**	0.149	**0.733**	0.725
Moz	Bug	**0.958**	0.923	0.470	**0.512**	**0.663**	0.641	**0.032**	0.095	0.631	**0.659**
	Col	**0.837**	0.801	0.444	**0.491**	**0.621**	0.606	**0.124**	0.198	0.580	**0.609**
	JDT	**0.894**	0.881	0.692	**0.721**	**0.790**	0.779	**0.096**	0.154	0.780	**0.793**
	Pla	**0.914**	0.904	0.708	0.701	**0.807**	0.799	**0.078**	0.134	**0.798**	0.790
	Pos	**0.870**	0.819	0.673	**0.695**	**0.756**	0.715	**0.135**	0.211	**0.759**	0.752
Pla	Bug	**0.939**	0.907	0.484	**0.501**	0.665	**0.667**	**0.050**	0.094	0.638	0.645
	Col	**0.850**	0.825	0.431	**0.443**	0.619	0.601	**0.109**	0.192	0.572	**0.576**
	JDT	**0.902**	0.891	0.667	**0.712**	**0.782**	0.751	**0.085**	0.102	0.767	**0.792**
	Moz	**0.864**	0.808	0.840	**0.841**	**0.850**	0.825	**0.140**	0.159	**0.852**	0.824
	Pos	**0.865**	0.851	0.638	**0.641**	**0.736**	0.609	**0.133**	0.156	**0.734**	0.731
Pos	Bug	**0.943**	0.928	0.379	**0.401**	**0.606**	0.582	**0.036**	0.079	0.036	**0.560**
	Col	**0.864**	0.812	0.453	**0.562**	**0.635**	0.617	**0.103**	0.198	0.594	**0.664**
	JDT	**0.908**	0.894	0.608	**0.649**	**0.755**	0.744	**0.072**	0.101	0.728	**0.752**
	Moz	**0.884**	0.853	0.788	**0.796**	**0.838**	0.795	**0.109**	0.143	**0.833**	0.824
	Pla	0.920	**0.921**	0.613	**0.623**	**0.762**	0.733	**0.063**	0.099	0.736	**0.743**
Average		**0.896**	0.869	0.570	**0.611**	**0.717**	0.688	**0.085**	0.126	0.667	**0.709**

6 Threats to validity

In this section, we discuss three kinds of validity threats that may affect our experimental results, namely internal validity, external validity and construct validity.

6.1 Internal validity

Internal validity is related to uncontrolled aspects that may affect our experimental results, such as errors in the experiment. We examined our experiment process carefully. However,

there may still be errors that we have not noticed.

6.2 External validity

External validity is related to the quality and universality of the datasets. In this paper, we use six open source projects, which belong to different application fields, cover a long time, and are written with different programming languages. We analyze six datasets, and think that the number of change instances used in the paper is large enough and has a certain universality. In the future, we also plan to further reduce this threat by analyzing more change instances in other open source and commercial projects.

6.3 Construct validity

Construct validity involves the applicability of our evaluation methods. In this paper, we use five evaluation metrics, namely accurate, precision, recall, F1, and pf. These metrics have been used in previous studies [Menzies, Milton, Turhan et al. (2010); Monden, Hayashi, Shinoda et al. (2013); Zhong, Khoshgoftaar and Seliya (2004); Yang, Zhou, Liu et al. (2016); Zimmermann, Premraj and Zeller (2007)], so we believe that the construct validity should be acceptable. The experimental design may also affect our experimental results. Recent studies have pointed out that defect prediction models with different parameter settings may produce different results. In order to reduce the threat to the experimental design of parameter settings, we plan to use parameter optimization techniques for more experiments.

7 Conclusions and future work

As the software scale and its complexity increase, the number of defects generated will increase dramatically. However, the current defect prediction methods cannot fully reflect the data feature and the detection effect is not ideal enough, we propose a novel defect prediction model named ITNB based on improved transfer Naive Bayesian algorithm in this paper. We first remove the edge data in the test set when calculating the similarity between the test set and the training set. Then, based on the feature dimension weight and the data gravity, we construct the calculation formula of the training data weight. Finally, we calculate the prior probability and conditional probability of training data based on the weight information, so as to construct the weighted bayesian classifier for software defect prediction. We use six datasets from large open source projects, namely Bugzilla, Columba, Mozilla, JDT, Platform and PostgreSQL. The experimental results show that our ITNB model can achieve better results than the TNB model in terms of accurary, precision and pd for within-project and cross-project defect prediction.

In future work, we will evaluate our models in more open source and commercial projects. In addition, we will use parameter optimization techniques to adjust the parameter settings of our model.

Acknowledgement: This work is supported in part by the National Science Foundation of China (Nos. 61672392, 61373038), and in part by the National Key Research and Development Program of China (No. 2016YFC1202204).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Abaei, G.; Rezaei, Z.; Selamat, A. (2013): Fault prediction by utilizing self-organizing map and threshold. *IEEE International Conference on Control System, Computing and Engineering*, pp. 465-470.

Briand, L. C.; Melo, W. L.; Wust, J. (2002): Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 706-720.

Chawla, N. V.; Bowyer, K. W.; Hall, L. O.; Kegelmeyer, W. P. (2002): Smote: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, vol. 16, no. 1, pp. 321-357.

Chen, L.; Fang, B.; Shang, Z.; Tang, Y. (2015): Negative samples reduction in cross-company software defects prediction. *Information and Software Technology*, vol. 62, no. 1, pp. 67-77.

Cheng, M.; Wu, G.; Yuan, M. (2016): Transfer learning for software defect prediction. *Acta Electronica Sinica*, vol. 44, no. 1, pp. 115-122.

Elish, K. O.; Elish, M. O. (2008): Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, vol. 81, no. 5, pp. 649-660.

Frank, E.; Hall, M.; Pfahringer, B. (2002): Locally weighted naive bayes. *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, pp. 249-256.

Herbold, S.; Trautsch, A.; Grabowski, J. (2017): A comparative study to benchmark cross-project defect prediction approaches. *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 811-833.

Jureczko, M.; Madeyski, L. (2010): Towards identifying software project clusters with regard to defect prediction. *Proceedings of the Sixth International Conference on Predictive Models in Software Engineering*, pp. 1-10.

Kamei, Y.; Shihab, E.; Adams, B.; Hassan, A. E.; Mockus, A. et al. (2012): A largescale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757-773.

Li, M.; Zhang, H.; Wu, R.; Zhou, Z. (2012): Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, vol. 19, no. 2, pp. 201-230.

Lu, H.; Kocaguneli, E.; Cukic, B. (2014): Defect prediction between software versions with active learning and dimensionality reduction. *IEEE 25th International Symposium on Software Reliability Engineering*, pp. 312-322.

Ma, Y.; Luo, G.; Zeng, X.; Chen, A. (2012): Transfer learning for cross-company software defect prediction. *Information and Software Technology*, vol. 54, no. 3, pp. 248-256.

Malhotra, R.; Khanna, M. (2017): An empirical study for software change prediction using imbalanced data. *Empirical Software Engineering*, vol. 22, no. 6, pp. 2806-2851.

Menzies, T.; Milton, Z.; Turhan, B.; Cukic, B.; Jiang, Y. et al. (2010): Defect

prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, vol. 17, no. 4, pp. 375-407.

Monden, A.; Hayashi, T.; Shinoda, S.; Shirai, K.; Yoshida, J. et al. (2013): Assessing the cost effectiveness of fault prediction in acceptance testing. *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1345-1357.

Nam, J.; Pan, S. J.; Kim, S. (2013): Transfer defect learning. *35th International Conference on Software Engineering*, pp. 382-391.

Peng, L.; Yang, B.; Chen, Y.; Abraham, A. (2009): Data gravitation based classification. *Information Sciences*, vol. 179, no. 6, pp. 809-819.

Rodriguez, D.; Herraiz, I.; Harrison, R.; Dolado, J.; Riquelme, J. C. (2014): Preliminary comparison of techniques for dealing with imbalance in software defect prediction. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, pp. 1-10.

Seiffert, C.; Khoshgoftaar, T. M.; Van Hulse, J.; Folleco, A. (2014): An empirical study of the classification performance of learners on imbalanced and noisy software quality data. *Information Sciences*, vol. 259, no. 2, pp. 571-595.

Watanabe, S.; Kaiya, H.; Kaijiri, K. (2008): Adapting a fault prediction model to allow inter languagereuse. *Proceedings of the Fourth International Workshop on Predictor Models in Software Engineering*, pp. 19-24.

Xu, Z.; Liu, J.; Luo, X.; Zhang, T. (2018): Cross-version defect prediction via hybrid active learning with kernel principal component analysis. *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 209-220.

Yang, Y.; Zhou, Y.; Liu, J.; Zhao, Y.; Lu, H. et al. (2016): Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 157-168.

Zhang, F.; Keivanloo, I.; Zou, Y. (2017): Data transformation in cross-project defect prediction. *Empirical Software Engineering*, vol. 22, no. 6, pp. 3186-3218.

Zhong, S.; Khoshgoftaar, T. M.; Seliya, N. (2004): Unsupervised learning for expert based software quality estimation. *IEEE International Symposium on High Assurance Systems Engineering*, pp. 149-155.

Zimmermann, T.; Premraj, R.; Zeller, A. (2007): Predicting defects for eclipse. *Third International Workshop on Predictor Models in Software Engineering*, pp. 1-9.