Empir Software Eng
DOI 10.1007/s10664-015-9396-2

Towards building a universal defect prediction model with rank transformed predictors

Feng Zhang[1] · **Audris Mockus**[2] · **Iman Keivanloo**[3] · **Ying Zou**[3]

Abstract Software defects can lead to undesired results. Correcting defects costs 50 % to 75 % of the total software development budgets. To predict defective files, a prediction model must be built with predictors (e.g., software metrics) obtained from either a project itself (within-project) or from other projects (cross-project). A universal defect prediction model that is built from a large set of diverse projects would relieve the need to build and tailor prediction models for an individual project. A formidable obstacle to build a universal model is the variations in the distribution of predictors among projects of diverse contexts (e.g., size and programming language). Hence, we propose to cluster projects based on the similarity of the distribution of predictors, and derive the rank transformations

Communicated by: Sung Kim and Martin Pinzger

✉ Feng Zhang
feng@cs.queensu.ca

Audris Mockus
audris@utk.edu

Iman Keivanloo
iman.keivanloo@queensu.ca

Ying Zou
ying.zou@queensu.ca

[1] School of Computing, Queen’s University, Kingston, Ontario, Canada

[2] Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996-2250, USA

[3] Department of Electrical and Computer Engineering, Queen’s University, Kingston, Ontario, Canada

Published online: 30 August 2015

using quantiles of predictors for a cluster. We fit the universal model on the transformed data of 1,385 open source projects hosted on SourceForge and GoogleCode. The universal model obtains prediction performance comparable to the within-project models, yields similar results when applied on five external projects (one Apache and four Eclipse projects), and performs similarly among projects with different context factors. At last, we investigate what predictors should be included in the universal model. We expect that this work could form a basis for future work on building a universal model and would lead to software support tools that incorporate it into a regular development workflow.

Keywords Universal defect prediction model · Defect prediction · Context factors · Rank transformation · Large-scale · Software quality

1 Introduction

It is common for software to contain defects (Nguyen et al. 2011). A defect is an error in software behaviour that causes unexpected results. Software defects were estimated to cost U.S. economy \$59.5 billion annually (Tassey 2002). The cost of correcting defects ranges from 50 % to 75 % of the total software development cost (Hailpern and Santhanam 2002). There is a rich history of attempts to anticipate the parts of source code that are likely to have defects to fix. For example, D'Ambros et al. (2012) evaluate over 30 different approaches published from 1996 to 2010 for building defect prediction models. Unfortunately, such models could not be generalized to apply on other projects or even new releases of the same project (Zimmermann et al. 2009; Nam et al. 2013). Refitting such models is not a trivial task. It requires collecting and tagging defects for each file, and computing software metrics from historical data. Sufficient history may not be available in certain projects, e.g., small or new projects (Nagappan et al. 2006).

We refer to a single model that is built from the entire set of projects as a universal model. A universal defect prediction model would relieve the need for refitting project-specific or release-specific models for an individual project. A universal model would also help interpret basic relationships between software metrics and defects, potentially resolving inconsistencies among different studies (Mair and Shepperd 2005). Moreover, it might allow a more direct comparison of defect rates across projects, and enable a continuous evaluation of defect proneness of a project. Therefore, it is of significant interest to build a universal defect prediction model.

Cross-project prediction may be a step towards building a universal model. We refer to a prediction as a cross-project prediction if a model is learnt from one project and the prediction is performed on another project. Zimmermann et al. (2009) examine the performance of all 622 possible cross-project predictions using 28 versions from 12 projects, and find a low ratio of successful cross-project predictions (i.e., 3.4 %). They consider a prediction to be successful if all the three performance measures (i.e., precision, recall, and accuracy) are greater than 0.75. The first challenge in building successful cross-project defect prediction models is related to the variations in the distribution of predictors (Cruz and Ochimizu 2009; Nam et al. 2013). To overcome this challenge, we consider two approaches: 1) use data from projects with similar distributions of predictors to the target project as training data (e.g., Turhan et al. 2009; Menzies et al. 2011); or 2) transform predictors in both training and target projects to make them more similar in their distribution (e.g., Nam et al. 2013;

Ma et al. 2012). However, the first approach uses partial dataset and results in multiple models for different target projects. The transformation approaches are typically specialized to a particular pair of training and testing datasets. Our prior study (Zhang et al. 2013) found that the distribution of software metrics varies with project contexts (e.g., size and programming language). Therefore, we combine the three insights in an attempt to build a universal defect prediction model for a large set of projects with diverse contexts.

In this study, we propose a context-aware rank transformation to address the variations in the distribution of predictors before fitting them in the universal defect prediction model. There are six context factors investigated in this study, including programming language, issue tracking, the total lines of code, the total number of files, the total number of commits, and the total number of developers. The context-aware approach stratifies the entire set of projects by context factors, and clusters the projects with similar distribution of predictors. Inspired by metric-based benchmarks (e.g., Alves et al. 2010), which use quantiles to derive thresholds for ranking software quality, we apply every tenth quantile of predictors on each cluster to specify ranking functions. We use twenty-one code metrics and five process metrics as predictors. After rank transformation, the predictors from different projects will have exactly the same scale. The universal model is then built using the transformed predictors.

We apply our approach on 1,385 open source projects hosted on SourceForge and GoogleCode. We observe that the F-measures and area under curve (AUC) obtained using rank-transformed predictors is comparable to that of logarithmicly transformed predictors. The logrithmic transformation uses the logrithmic values of predictors, and is commonly used to build prediction models. After adding the six context factors as predictors, the performance of the universal model built using only code and process metrics can be further improved. On average, the universal model yields higher AUC than within-project models. Moreover, the universal model achieves up to 48 % of the successful predictions of within-project models using loose criteria (i.e., recall is above 0.70, and precision is greater than 0.50) suggested by He et al. (2012) to determine the success of defect prediction models.

We examine the generalizability of the universal model in two ways. First, we build the universal model using projects hosted on SourceForge and GoogleCode, and apply the universal model on five external projects, including Lucene, Eclipse, Equinox, Mylyn, and Eclipse PDE. The results show that the universal model provides a similar performance (in terms of AUC) as within-project models for the five projects. Second, we compare the performance of the universal model on projects of different context factors. The results indicate that the performance does not change significantly among projects with different context factors. These results suggest that the universal model is context-insensitive and generalizable.

In summary, the major contributions of our study are:

- **Propose an approach of context-aware rank transformation**. The rank transformation method addresses the problem of large variations in the distribution of predictors across projects from diverse contexts. The transformed predictors have exactly the same scales. This enables us to build a universal model for a large set of projects.
- **Improve the performance of the universal model by adding context factors as predictors**. We add the context factors to our universal prediction model, and find

that context factors significantly improve the predictive power of the universal defect prediction model (e.g., the average AUC increases from 0.607 to 0.641 comparing to the combination of code and process metrics).
- **Provide a universal defect prediction model**. The universal model achieves similar performance as within-project models for five external projects, and does not show significant difference in the performance for projects with different context factors. The universal model is context-insensitive and generalizable. We also provide the estimated coefficients of predictors for the universal model.

This work extends our previous work (Zhang et al. 2014) that was published in the proceedings of the 11th working conference on Mining Software Repositories (MSR) in three ways. First, we added the details of the approach and data processing steps, so that our study could be easily replicated. Second, we added RQ4 that examines the performance of the universal model when applied to projects from diverse contexts. Third, we added RQ5 that investigates the importance of different predictors in the universal model. Our findings provide insights of what predictors are more suitable to establish general relationships with defect-proneness.

The remainder of this paper is organized as follows. The related work is summarized in Section 2. Section 3 and Section 4 describe our approach and experiment design, respectively. Section 5 presents our results and discussions. The threats to validity of our work are discussed in Section 6. We conclude and provide insights for future work in Section 7.

2 Related Work

In this section, we review previous studies on defect prediction in general, cross-project defect prediction, and preprocessing techniques on predictors.

2.1 Defect Prediction

Defect prediction studies have a long history since 1970s (Akiyama 1971), and have become very active in the last decade (D'Ambros et al. 2012). The purpose of defect prediction models is to predict the defect proneness (i.e., buggy or clean) or the number of defects of a software artifact. The defect prediction is studied for artifacts with various granularities such as project, module, file, and method levels (e.g., Zimmermann et al. 2009; Hassan 2009). The impact of granularity on the performance of defect prediction models is studied by Posnett et al. (2011). This study chooses to predict defect proneness at file level.

Building a defect prediction model requires three major steps: 1) collect predictors; 2) label defect proneness; and 3) choose proper modelling techniques. Software metrics are commonly used as predictors in defect prediction models. Numerous software metrics have been investigated, including complexity metrics (e.g., lines of code and McCabe's cyclomatic complexity (Menzies et al. 2007b), structural metrics (Zimmermann and Nagappan 2008), process metrics (e.g., recent activities, number of changes, and the complexity of changes (Hassan 2009)), the number of previous defects (Zimmermann et al. 2007), and social network metrics (Bettenburg and Hassan 2010). D'Ambros et al. (2012) systematically compare the predictive power of different metric categories, and find that process metrics are superior in predicting the defect proneness. Arisholm et al. (2010) find large differences in terms of cost-effectiveness for defect prediction among different metric sets (e.g., process metrics significantly outperform structural metrics). Moreover, Radjenović et al. (2013) report that the performance of metric sets relates to several context factors (e.g., size and life cycles) of subject projects.

There are two major types of modelling techniques: statistical methods (e.g., Naive Bayes and logistic regression), and machine learning methods (e.g., decision trees, support vector machine (SVM), K-nearest neighbour, and artificial neural networks). Lessmann et al. (2008), Arisholm et al. (2010), and D'Ambros et al. (2012) propose different approaches to compare and evaluate different modelling techniques. Lessmann et al. (2008) find that there are no significant differences in the performance among different modelling techniques. Arisholm et al. (2010) also report that the choice of modelling techniques has only limited impact on the performance in terms of accuracy or cost-effectiveness. However, different observations are reported by Hall et al. (2012) that some modelling techniques (e.g., Naive Bayes and logistic regression) perform well in defect prediction, and some other modelling techniques (e.g., SVM) perform less well. Sarro et al. (2012) find that tuning the parameters of SVM using genetic algorithm can improve the performance of defect prediction. Our rank transformation approach is a step for data preprocessing, thus it is independent of the modelling techniques. Software organizations can choose the technique that best suits their needs.

2.2 Cross-Project Defect Prediction

Most of the aforementioned studies have been conducted under within-project settings. We refer to a prediction as a within-project prediction if the training and target projects are the same. Building within-project models requires enough historical data of the target project. However, some projects, such as small or new projects, may not have sufficient historical data (Nagappan et al. 2006). In response to such challenges, many researchers attempt to build cross-project defect prediction models. Most studies experience poor performance of cross-project defect predictions (Hall et al. 2012). For instance, Zimmermann et al. (2009) run cross-project predictions for all 622 possible pairs of 28 datasets from 12 projects, and find only 21 pairs (i.e., cross-project predictions) match their performance criteria (i.e., all precision, recall and accuracy are above 0.75). Turhan et al. (2009) observe that cross-project prediction not only underperforms within-project prediction, but also has excessive false alarms. Premraj and Herzig (2011) confirm the challenges in cross-project defect prediction through their replication study. Even for the same project, Shatnawi and Li (2008) report a decrease in performance of within-project defect prediction models from release to release.

Rahman et al. (2012) argue that cross-project defect prediction can yield the same performance as within-project prediction in terms of cost effectiveness, instead of standard measures (i.e., precision, recall, and F-measure). Nevertheless, the challenge of cross-project prediction still exists. It might be due to the fact that metrics from different projects may have significantly different distributions (Cruz and Ochimizu 2009; Nam et al. 2013). Denaro and Pezzè (2002) conclude that good predictive performance can be achieved only across homogeneous projects. Similar finding is reported by Nagappan et al. (2006). Hall et al. (2012) investigate 36 studies and report that some context factors (e.g., system size and application domain) affect the performance of cross-project predictions. Zimmermann et al. (2009) and Menzies et al. (2011) both suggest to consider project contexts for cross-project defect prediction. To deal with heterogeneous data from diverse projects, this study proposes context-aware rank transformation for predictors as a data preprocessing step. In addition, we find that adding context factors (see Section 3.2) as predictors can improve the predictive power for the universal defect prediction model.

2.3 Data Preprocessing

The distribution of metric values varies across projects (Cruz and Ochimizu 2009). Our previous study (Zhang et al. 2013) examines 320 diverse projects, and conclude that the distribution of metric values sometimes varies significantly in projects of different contexts. The variation in scales of metrics pose a challenge for building a universal defect prediction model. Data preprocessing has been proved to improve the performance of defect prediction models by Menzies et al. (2007b). These findings suggest that preprocessing predictors may be a mandatory step needed to build a successful cross-project defect prediction model. Jiang et al. (2008) evaluate the impact of log transformation and discretization on the performance of defect prediction models, and find different modelling techniques to "prefer" different transformation techniques. For instance, Naive Bayes achieves better performance on discretized data, while logistic regression benefits from both approaches. Cruz and Ochimizu (2009) also observe that log transformations can improve the performance of cross-project predictions, only if the data of target project is not as skewed as the data of the training project.

The state-of-the-art approaches to improve the performance of cross-project defect prediction mainly use two data preprocessing techniques: 1) use data from projects with similar distributions to the target project (e.g., Turhan et al. 2009; Menzies et al. 2011); or 2) transform predictors in both training and target projects to make them more similar in their distribution (e.g., Nam et al. 2013; Ma et al. 2012). To filter the training data, He et al. (2012) propose to use the distributional characteristics (e.g., median, mean, variance, standard deviation, skewness, and quantiles); Turhan et al. (2009) propose to use nearest neighbour filter; Li et al. (2012) propose to use sampling; and He et al. (2013) propose to use data similarity. The aforementioned approaches are able to improve the performance of cross-project defect prediction models. However, they use only partial dataset and end up with multiple models (i.e., one model per target project). On the other hand, the transformation approaches are typically specialized to a particular pair of training and testing datasets. For instance, Watanabe et al. (2008) propose to compensate the target project with the average values of predictors of both target and training projects. Similarly, Ma et al. (2012) weight training data by estimations on the distribution of target data. Nam et al. (2013) propose to transform both training and target data to the same latent feature space, and build models on the latent feature space. Our previous study (Zhang et al. 2013) suggests to consider project contexts, when deriving proper thresholds and ranges of metric values that are often used to evaluate software quality (Baggen et al. 2012). By combining these three insights, we propose a context-aware rank transformation approach which does not require or depend on the target data set. The target data set contains the projects on which to apply defect prediction models.

3 Approach

In this section, we present the details of our approach for building a universal defect prediction model.

3.1 Overview

The poor performance of cross-project prediction may be caused by the significant differences in the distribution of metric values among projects (Cruz and Ochimizu 2009; Nam et al. 2013). Therefore, to build a universal model using a large set of projects, it is essential to reduce the difference in the distribution of metric values across projects. Our previous work (Zhang et al. 2013) finds that context factors of projects can significantly affect the distribution of metric values. Therefore, we propose a context-aware rank transformation approach to preprocess metric values before fitting them to the universal model. As illustrated in Fig. 1, our approach consists of the following four steps:

1) ***Partitioning.*** We partition the entire set of projects into non-overlapping groups based on the six context factors (i.e., programming language, issue tracking, the total lines of code, the total number of files, the total number of commits, and the total number of developers). This step aims to reduce the number of pairwise comparisons. We compare the distribution of metric values across groups of projects instead of individual projects.
2) ***Clustering.*** We cluster the project groups with the similar distribution of predictor values. This step aims to merge similar groups of projects so that we could include more projects in each cluster for obtaining ranking functions.
3) ***Obtaining ranking functions.*** We derive a ranking function for each cluster using every $10th$ quantile of predictor values. This transformation removes large variations in the distribution of predictors by transforming them to exactly the same scale.
4) ***Ranking.*** We apply the ranking functions to convert the raw values of predictors to one of the ten levels. This step aims to remove the difference in the scales of metric values across projects. The scales of the transformed metric values are exactly the same for all projects.

After the preprocessing steps, we build the universal model based on the transformed predictors. The following subsections describe the context factors used in this study, and the details of each step.

3.2 Context Factors

Software projects have diverse context factors. However, it is still unclear what context factors best characterize projects. For instance, Nagappan et al. (2013) choose seven

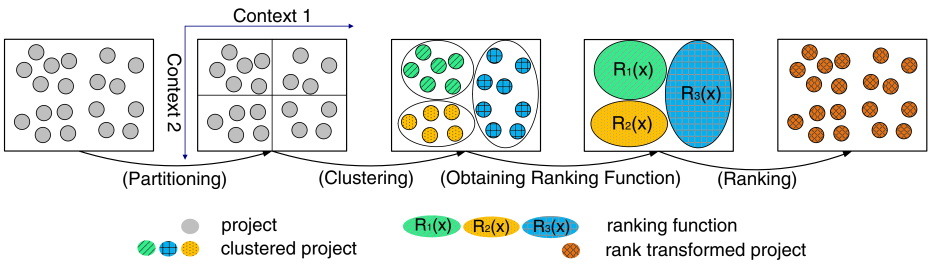

Fig. 1 Our four-step rank transformation approach: 1) stratify the set of projects along different contexts into non-overlap groups; 2) cluster project groups; 3) derive ranking function for each cluster; and 4) perform rank transformation

context factors based on their availability in Ohloh,[1] including main programming language, the total lines of code, the number of contributors, the number of churn, the number of commits, project age, and project activity. Along the same lines, our previous work (Zhang et al. 2013) also selects seven context factors, i.e., application domain, programming language, age, lifespan, the total lines of code, the number of changes, and the number of downloads. Our prior study (Zhang et al. 2013) finds that project age, lifespan, and the number of downloads do not significantly affect the distribution of metric values. Thus we exclude these three context factors from this study. The shared context factors of the aforementioned two studies are programming language, the total lines of code, and the number of commits. These factors are common to all projects with version control systems. Hence, we include these three context factors in this study. The information of application domain is unavailable to our subject projects that are hosted on GoogleCode. Therefore, we exclude application domain as well. Moreover, we add the number of developers as Nagappan et al. (2013), and the number of files as another size measurement. Eventually, we choose the following six context factors in this study.

1) ***Programming Language (PL)*** describes the nature of programming paradigms. There is a high chance for metric values of different programming languages to experience significantly different distributions. Moreover, it is interesting to investigate the possibility of inter language reuse of prediction models. Due to the limitation of our metric computing tool, we only consider projects mainly written in C, C++, Java, C#, or Pascal. A project is *mainly written* in programming language *pl* if the largest number of source code files are written in *pl*.
2) ***Issue Tracking (IT)*** describes whether a project uses an issue tracking system or not. The usage of an issue tracking system can reflect the quality of the project management process. It is likely that the distribution of metric values are different between projects with or without usage of issue tracking systems. A project uses an issue tracking system if the issue tracking system is enabled in the website of the project and there is at least one issue recorded.
3) ***Total Lines of Code (TLOC)*** describes the project size in terms of source code. Comment and blank lines are excluded when counting the total lines of code. Moreover, the lines of code of files that are not written in the main language of the project are also excluded. Such exclusion simplifies our approach for transforming metric values, as only one programming language is considered for each project.
4) ***Total Number of Files (TNF)*** describes the project size in terms of files. This context factor measures the project size from a different granulatiry to the total lines of code. Similar as the total lines of code measurement, we exclude files that are not written in the main language of each project.
5) ***Total Number of Commits (TNC)*** describes the project size in terms of commits. Different from the total lines of code and the total number of files, this context factor captures the project size from the process perspective. The total number of commits can describe how actively the project was developed.

[1] https://www.openhub.net (NOTE: ‘Ohloh’ was changed to ‘Open Hub’ in 2014.)

6) ***Total Number of Developers (TND)*** describes the project size in terms of developers. Teams of different sizes (e.g., small or large) may follow different development strategies, therefore the team size can impact the distribution of metric values.

3.3 Partitioning Projects

We assume that projects with the same context factors have similar distribution of software metrics, and projects with different contexts might have different distribution of software metrics. Hence, we stratify the entire set of projects based on the aforementioned six context factors.

1) ***PL.*** We divide the set of projects into 5 groups based on programming languages: G_c, G_{c++}, G_{java}, $G_{c\#}$, and G_{pascal}.
2) ***IT.*** The set of projects is separated into 2 groups based on the usage of an issue tracking system: G_{useIT} and G_{noIT}.
3) ***TLOC.*** We compute the TLOC of each project and the quartiles of TLOC. Based on the first, second, and third quartiles, we split the set of projects into 4 groups: $G_{leastTLOC}$, $G_{lessTLOC}$, $G_{moreTLOC}$, and $G_{mostTLOC}$.
4) ***TNF.*** We calculate TNF of each project, and the quartiles of TNF. Based on the first, second, and third quartiles, we separate the set of projects into 4 groups: $G_{leastTNF}$, $G_{lessTNF}$, $G_{moreTNF}$, and $G_{mostTNF}$.
5) ***TNC.*** We compute the TNC of each project, and the quartiles of TNC. Based on the first, second, and third quartiles, we break the entire set of projects into 4 groups: $G_{leastTNC}$, $G_{lessTNC}$, $G_{moreTNC}$, and $G_{mostTNC}$.
6) ***TND.*** We calculate the TND of each project, and the quartiles of TND. Based on the first, second, and third quartiles, we split the whole set of projects into 4 groups: $G_{leastTND}$, $G_{lessTND}$, $G_{moreTND}$, and $G_{mostTND}$.

In summary, we get 5, 2, 4, 4, 4, and 4 non-overlapping groups along each of the six context factors, respectively. In total, we obtain 2560 (i.e., $5 \times 2 \times 4 \times 4 \times 4 \times 4$) non-overlapping groups for the entire set of projects.

3.4 Clustering Similar Projects

In the previous step, we obtain non-overlapping groups of projects. However, the size of most groups is small. In some cases the non-overlapping groups of projects do not have significantly different distributions of metrics. In addition, clustering similar projects together helps obtain more representative quantiles of a particular metric. At this step, we cluster the projects with the similar distributions of a metric. We consider two distributions to be similar if neither their difference is statistically significant nor the effect size of their difference is large, as our previous study (Zhang et al. 2013).

For different metrics, the corresponding clusters are not necessarily the same. In other words, we produce a particular set of clusters for each individual metric. We describe a cluster using a vector. The first element shows for what metric the cluster is created, and the remaining elements characterize the cluster from the context factor perspective. For example, the cluster $< m, C{+}{+}, useIT, moreTLOC >$ is created for metric m, contains $C{+}{+}$ projects that use issue tracking systems, and has the TLOC between the second and third quartiles (see Section 3.2).

For each metric m, the clusters of projects with similar distribution of metric m are obtained using the Algorithm 1. The Algorithm 1 has two major steps:

Algorithm 1: Clustering Similar Projects

```
Input: m: the metric $m$
       N: the number of groups
Output: clusterOfGroup: the cluster index of projects
/* Initialize the array clusterOfGroup.                                  */
1  int indexOfCluster = 1;
2  for $i = 1$ to $N$ do
3      clusterOfGroup[i] = indexOfCluster;
4  end
   /* Do the clustering.                                                 */
5  for $i = 1$ to $N - 1$ do
6      int indexNewCluster = indexOfCluster+1;
7      for $j = i + 1$ to $N$ do
           /* Compare the distribution of metric values between two groups i
              and j.                                                     */
8          compareMetricDistribution(m, $i$, $j$);
9          if the difference is statistically significant then
               /* Quantify the importance of the difference.             */
10             computeCliffsDelta($i$, $j$);
11             if Cliff's $\delta$ is large then
                   /* Put group i and j in different clusters.           */
12                 if clusterOfGroup[j] equals to clusterOfGroup[i] then
                       /* Put group j in a new cluster.                  */
13                     clusterOfGroup[j] = indexNewCluster;
                       /* Update the base counter to compute new clusters. */
14                     indexOfCluster = indexNewCluster;
15                 end
16             end
17         end
18     end
19 end
```

1) ***Comparing the distribution of metrics***. This step (Line 8 in Algorithm 1) merges the groups of projects that do not have significantly different distribution of metric m. We apply Mann-Whitney U test (Sheskin 2007) to compare the distribution of metric values between every two groups of projects, using the 95 % confidence level (i.e., p-value<0.05). The Mann-Whitney U test assesses whether two independent distributions have equally large values. It is a non-parametric statistical test. Therefore it does not assume a normal distribution. As we conduct multiple tests to investigate the distribution of each metric, we apply Bonferroni correction to control family-wise errors. Bonferroni adjusts the threshold p-value by dividing the number of tests.
2) ***Quantifying the difference between distributions***. This step (Lines 10 to 16 in Algorithm 1) merges the groups of projects that have significantly different distributions of metric m, but the difference is not large. We calculate Cliff's δ (Line 10 in Algorithm 1) as the effect size (Romano et al. 2006) to quantify the importance of the difference between the distribution of every two groups of projects. Cliff's δ estimates non-parametric effect sizes. It makes no assumptions of a particular distribution, and is reported (Romano et al. 2006) to be more robust and reliable than Cohen's d (Cohen 1988). Cliff's δ represents the degree of overlap between two sample distributions (Romano et al. 2006). It ranges from -1 (if all selected values in the first

group are larger than in the second group) to +1 (if all selected values in the first group are smaller than the second group). It is zero when two sample distributions are identical (Cliff 1993). Cohen's standards (i.e., small, medium, and large) are commonly used to interpret effect size. Therefore, we map the Cliff's δ to Cohen's standards, using the percentage of non-overlap (Romano et al. 2006). The mapping between the Cliff's δ and Cohen's standards is shown in Table 1. Cohen (1992) states that a medium effect size represents a difference likely to be visible to a careful observer, while a large effect is noticeably greater than medium. In this study, we choose the large effect size as the threshold of the importance of the differences between the distributions (Line 11 in Algorithm 1).

3.5 Obtaining Ranking Functions

In Section 3.4, we create clusters of projects for each metric, independently. For a particular metric, projects within the same cluster exhibit similar distribution of values of the corresponding metric. To remove the variation in the scales of metric values, this step derives ranking functions for each cluster. The ranking function transforms the raw metric values to predefined values (i.e., ranging from one to ten). Therefore, the transformed metrics have exactly the same scale among the projects.

We use the quantiles of metric values to formulate our ranking functions. This is inspired by metric-based benchmarks (e.g., Alves et al. 2010), which often use the quantiles to derive thresholds of metrics. The thresholds of metrics are used to distinguish files of different levels of quality related to defects.

Let M denote the total number of metrics. For metric m_i (where $i \in \{1, \ldots, M\}$), the corresponding clusters are represented using Cl_{i1}, Cl_{i2}, ..., and Cl_{iN_i}, where N_i is the total number of clusters obtained for metric m_i. We formulate the ranking function for metric m_i in the jth cluster C_{ij} following (1).

$$Rank(m_i, Cl_{ij}, value) = \begin{cases} 1 & \text{if } value \in [0, Q_{ij,1}(m_i)] \\ k & \text{if } value \in (Q_{ij,k-1}(m_i), Q_{ij,k}(m_i)] \\ 10 & \text{if } value \in (Q_{ij,9}(m_i), +\infty) \end{cases} \quad (1)$$

where the variable $value$ denotes the value of metric m_i to be converted, $Q_{ij,k}(m_i)$ is the $k * 10th$ quantile of metric m_i in cluster Cl_{ij}, $j \in \{1, \ldots, N_i\}$, and $k \in \{2, \ldots, 9\}$.

For example, we assume that every tenth quantile for a metric m_1 in cluster Cl_{12} is: 11, 22, 33, 44, 55, 66, 77, 88, and 99, respectively. The ranking function for metric m_1 in cluster Cl_{ij} is shown in Table 2. If metric m_1 has a value of 27 in a file of a project that belongs to cluster Cl_{12}, then the metric value in the file will be converted to 3. This is because the value 27 is greater than 22 (i.e., the 20 % quantile) and less than 33 (i.e., the 30 % quantile).

Table 1 Mapping Cliff's δ to Cohen's standards

Cliff's δ	% of Non-overlap	Cohen's d	Cohen's Standards
0.147	14.7 %	0.20	small
0.330	33.0 %	0.50	medium
0.474	47.4 %	0.80	large

Table 2 An example of ranking functions

Range of Metric Value	[0, 11]	(11, 22]	(22, 33]	…	(99, $+\infty$)
$Rank(m_1, Cl_{12}, value)$	1	2	3	…	10

3.6 Building a Universal Defect Prediction Model

3.6.1 Choice of Modelling Techniques

As described in Section 2.1, Lessmann et al. (2008) and Arisholm et al. (2010) show that there is no significant difference among different modelling techniques in the performance of defect prediction models. However, Kim et al. (2011) find that Bayes learners (i.e., Bayes Net and Naive Bayes) perform better when defect data contains noises, even up to 20 %–35 % of false positive and false negative noise in defect data. Based on their findings, we apply Naive Bayes as the modelling technique in our experiments to evaluate the performance of the universal defect prediction model. When investigating the importance of different metrics in the universal model, we choose to apply logistic regression model as it is a common practice to compare the importance of different metrics (Zimmermann et al. 2012).

3.6.2 Steps to Build the Universal Defect Prediction Model

Our universal model is built upon the entire set of projects using rank transformed metric values. The first step is to transform metric values using ranking functions that are obtained from our dataset. In order to locate the ranking function for metric m_i in project p_j, we need to determine which cluster project p_j belongs to. We identify context factors of project p_j, and formulate a vector like $<m_i$, C++, $useIT$, $moreTLOC$, $lessTNF$, $lessTNC$, $lessTND>$ to present a cluster, where the first item specifies the metric, and the remaining items describe the corresponding context factors that projects in this cluster belong to. The vector of project p_j is then compared to the vectors of all clusters. The exactly matched cluster is the cluster that project p_j belongs to. After the transformation, the values of metrics will have the same scale ranging from one to ten.

The second and the last step is to build the model. We apply the Naive Bayes algorithm[2] implemented in Weka[3] tool to build a universal defect prediction model upon the entire set of projects.

3.7 Measuring the Performance

To evaluate the performance of prediction models, we compute the confusion matrix as shown in Table 3. In the confusion matrix, true positive (TP) is the number of defective files that are correctly predicted as defective files; false negative (FN) counts the number of defective files that are incorrectly predicted as clean files; false positive (FP) measures the number of files that are clean but incorrectly predicted as defective; and true negative (TN) represents the number of clean files that are correctly predicted as clean files.

[2]https://weka.sourceforge.net/doc.dev/weka/classifiers/bayes/NaiveBayes.html

[3]http://www.cs.waikato.ac.nz/ml/weka

Table 3 Confusion matrix used in defect prediction studies

Actual \ Predicted	defective	non-defective
defective	true positive (TP)	false negative (FN)
non-defective	false positive (FP)	true negative (TN)

We calculate the following six measures (i.e., precision, recall, false positive rate, F-measure, g-measure, and Matthews correlation coefficient) using the confusion matrix. We also compute the area under curve (AUC) as an additional measure.

Precision (*prec*) measures the proportion of actual defective entities that are predicted as defective against all predicted defective entities. It is defined as:

$$prec = \frac{TP}{TP + FP} \tag{2}$$

Recall (*pd*) evaluates the proportion of actual defective entities that are predicted as defective against all actual defective entities. It is defined as:

$$pd = \frac{TP}{TP + FN} \tag{3}$$

False Positive Rate (*fpr*) is the proportion of actual non-defective entities that are predicted as defective against all actual non-defective entities. It is defined as:

$$fpr = \frac{FP}{FP + TN} \tag{4}$$

F-measure calculates the harmonic mean of precision and recall. It balances precision and recall. It is defined as:

$$F\text{-}measure = \frac{2 \times pd \times prec}{pd + prec} \tag{5}$$

g-measure computes the harmonic mean of recall and 1-fpr. The 1-fpr represents *Specificity* (not predicting entities without defects as defective). We report g-measure as Peters et al. (2013b), since Menzies et al. (2007a) show that precision can be unstable when datasets contain a low percentage of defects. It is defined as:

$$g\text{-}measure = \frac{2 \times pd \times (1 - fpr)}{pd + (1 - fpr)} \tag{6}$$

Matthews Correlation Coefficient (MCC) is a balanced measure of true and false positives and negatives. It ranges from -1 to +1, where +1 indicates a perfect perdiction, 0 means the prediction is close to random prediction, and -1 represents total disaggrement between predicted and actual values. It is defined as:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}} \tag{7}$$

Area Under Curve (AUC) is the area under the receiver operating characteristics (ROC) curve. ROC is independent of the cut-off value that is used to compute the confusion matrix. We choose to apply AUC as another performance measure of prediction models, since AUC has been commonly reported in other studies of cross-project defect prediction. Rahman et al. (2012) find that traditional measures (e.g., precision, and recall) are not as effective as AUC when measuring the performance of cross-project defect prediction models.

Moreover, confusion matrix can be reconstructed from precision, recall, and d (Hall et al. 2012), where d represents the proportion of correct predictions (i.e., $d = TP + TN$). We can compute d using precision ($prec$), recall (pd), and false positive rate (fpr) as follows:

$$d = \frac{prec \times fpr}{pd \times (1 - prec) + prec \times fpr} \tag{8}$$

4 Experiment Setup

4.1 Data Collection

4.1.1 Subject Projects

SourgeForge and GoogleCode are two large and popular repositories for open source projects. We use the SourceForge and GoogleCode data initially collected by Mockus (2009) with his updates until October 2010. The dataset contains the full history of about 154K projects that are hosted on SourceForge and 81K projects that are hosted on GoogleCode to the date they were collected. The file contents of each revision and commit logs are stored separately and linked together using a universal unique identifier. The storage of file contents of SourceForge and GoogleCode projects spreads in 100 database files. Each database file is about 8 Giga bytes. The storage of commit logs spreads in 13 compressed files that have a total size of about 10 Giga bytes. Although we have 235K projects in total, there are too many trivial projects. Many projects do not have enough history and defect data for evaluation. Hence, we clean the dataset and obtain 1,385 projects for our experiments. Comparing to the 1,398 projects used in our previous work (Zhang et al. 2014), there are 13 projects removed due to an error in data pre-processing. The error is identified during this extension, and has been fixed. The cleaning process is detailed in the following subsection.

4.1.2 Cleaning the Dataset

Filtering Out Projects by Programming Languages In this study, we use a commercial tool, called *Understand* (SciTools 2015), to compute code metrics. Due to the limitation of the tool, we only investigate projects that are mainly written in C, C++, C#, Java, or Pascal. For each project, we determine its main programming languages by counting the total number of files per file type (i.e., *.c, *.cpp, *.cxx, *.cc, *.cs, *.java, and *.pas).

Filtering Out the Projects with a Small Number of Commits A small number of commits can not provide enough information for computing process metrics and mining defect data. We compute the quantiles of the number of commits of all projects throughout their history. We choose the 25 % quantile of the number of commits as the threshold to filter out projects. In our dataset, we filter out the projects with less than 32 (inclusive) commits throughout their histories.

Filtering Out the Projects with Lifespan Less Than One Year Most studies in defect prediction collect defect data from a six-months' period (Zimmermann et al. 2007) after the software release, and compute process metrics using the six months' data ahead. However, numerous projects on SourceForge or GoogleCode do not have clear release periods. Therefore, we simply determine the split date for each project by looking six months (i.e., 182.5 days) back from its last commit. We collect defect data in the six-months period after the split date, and compute process metrics using the change history in the six-months period before the split date. Thus we filter out the projects with a lifespan less than one year (i.e., 365 days).

Filtering Out the Projects with Limited Defect Data Defect data needs to be mined from enough commit messages. We count the number of fix-inducing and non-fixing commits from a one-year period. We choose the 75 % quantile of the number of fix-inducing (respectively non-fixing) commits as the threshold to filter out the projects with less defect data. For projects hosted on SourceForge, the 75 % quantile of the number of fix-inducing and non-fixing commits are: 152 and 1,689, respectively. For projects hosted on GoogleCode, the 75 % quantile of the number of fix-inducing and non-fixing commits are: 92 and 985, respectively.

Filtering Out the Projects Without Fix-Inducing Commits Subject projects in defect prediction studies usually contain defects. For example, the 56 projects used by Peters et al. (2013b) have at least one defect. We consider the projects that have no fix-inducing commits during six months as abnormal projects, therefore we filter out such projects. Moreover, there are 13 projects with few commits during the six-month period of collecting process metrics. We filter out these 13 projects since process metrics are not available to them.

Description of the Final Experiment Dataset In the cleaned dataset, there are 931 SourceForge projects, and 454 GoogleCode projects. Among them, 713 projects employ CVS as their version control system, 610 projects use Subversion, and 62 projects adopt Mercurial. The number of projects that are mainly written in C, C++, C#, Java, and Pascal are 283, 421, 84, 586, and 11, respectively. There are 810 projects using issue tracking systems, and 575 projects without using any issue tracking system. We show the boxplot of other four context factors in Fig. 2.

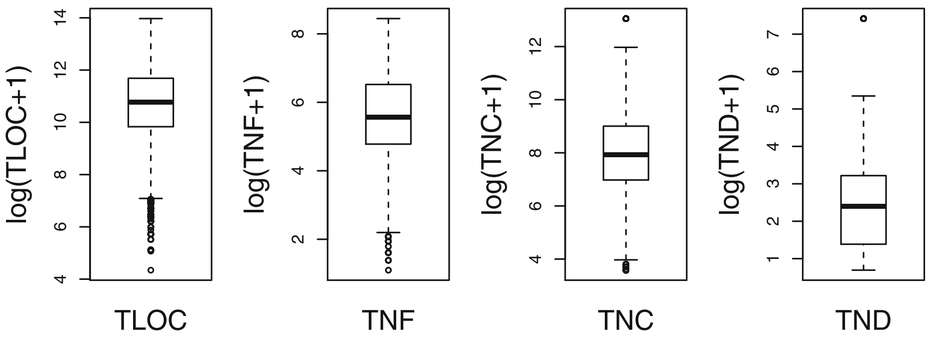

Fig. 2 Boxplot of four numeric context factors (i.e., TLOC, TNF, TNC, and TND) in our dataset

4.2 Software Metrics

Software metrics are used as predictors to build a defect prediction model. In this study, we choose 21 code metrics, and 5 process metrics that are often used in defect prediction models. The list of selected metrics is shown in Table 4. File and method level metrics are available to all the five studied programming languages. Class level metrics are only available to object-oriented programming languages, and are set to zero in files written in C. As defect prediction is performed at file level in this study, method level and class level metrics are aggregated to file level using three schemes, i.e., average (avg), maximum (max), and summation(total). The code metrics are computed by the *Understand* tool (SciTools 2015). Process metrics include the number of revisions and bug-fixing revisions (see Section 4.3), and lines of added/deleted/modified code. Process metrics are computed by our scripts. For each file, we extract all revisions that are performed during the period for collecting process metrics, and obtain the number of revisions and bug-fixing revisions. The number of

Table 4 List of software metrics. The last column refers to the aggregation scheme ("none means that aggregation is not performed for file level metrics)

Type	Metric Level	Metric Name	Description	Aggregation
Code Metrics	File	LOC	Lines of Code	none
		CL	Comment Lines	none
		NSTMT	Number of Statements	none
		NFUNC	Number of Functions	none
		RCC	Ratio Comments to Code	none
		MNL	Max Nesting Level	none
	Class	WMC	Weighted Methods per Class	avg, max, total
		DIT	Depth of Inheritance Tree	avg, max, total
		RFC	Response For a Class	avg, max, total
		NOC	Number of Immediate Subclasses	avg, max, total
		CBO	Coupling Between Objects	avg, max, total
		LCOM	Lack of Cohesion in Methods	avg, max, total
		NIV	Number of instance variables	avg, max, total
		NIM	Number of instance methods	avg, max, total
		NOM	Number of Methods	avg, max, total
		NPBM	Number of Public Methods	avg, max, total
		NPM	Number of Protected Methods	avg, max, total
		NPRM	Number of Private Methods	avg, max, total
	Methods	CC	McCabe Cyclomatic Complexity	avg, max, total
		FANIN	Number of Input Data	avg, max, total
		FANOUT	Number of Output Data	avg, max, total
Process Metrics	File	NREV	Number of revisions	none
		NFIX	Number of revisions a file was involved in bug-fixing	none
		ADDEDLOC	Lines added	avg, max, total
		DELETEDLOC	Lines deleted	avg, max, total
		MODIFIEDLOC	Lines modified	avg, max, total

added, deleted, and modified lines between each two consecutive revisions of each file are computed, and then aggregated to file level using the three aforementioned schemes. As mentioned in Section 4.1.2, we look six months (i.e., 182.5 days) back from the last commit to obtain the split date. The code metrics are computed using the files from the snapshot on the split date. The process metrics are computed using the change history in the six-months period before the split date.

4.3 Defect Data

Defect data are often mined from commit messages, and corrected using defect information stored in an issue tracking system (Zimmermann et al. 2007). In our dataset, 42 % of subject projects do not use issue tracking systems. For such projects, we mine defect data solely by analyzing the content of commit messages. A similar method for mining defect data is used by Mockus and Votta (2000) and in SZZ algorithm (Śliwerski et al. 2005). We first remove all words ending with "bug" or "fix" from commit messages, since "bug" and "fix" can be affix of other words (e.g., "debug" and "prefix"). A commit message is tagged as fixing defect, if it matches the following regular expression:

$$(bug|fix|error|issue|crash|problem|fail|defect|patch)$$

Using commit messages to mine defect information may be biased (Bird et al. 2009; Kim et al. 2011; Herzig et al. 2013). However, Rahman et al. (2013) report that increasing the sample size can leverage the possible bias in defect data. Our dataset contains 1,385 subject projects, and is around 140 to 280 times larger than most papers in this field (Peters et al. 2013b). In addition, the modelling technique (i.e., Naive Bayes) used in this study is proved by Kim et al. (2011) to have strong noise resistance with up to 20 %–35 % of false positive and false negative noises in defect data. Finally, the defect data is collected in the six-month' period after the split date. We show the boxplot of the number of defects and the percentage of defects in our dataset in Fig. 3.

5 Case Study Results

This section first describes the statistics of our project clusters, and then presents the motivation, approach, and findings of the following five research questions.

RQ1. *Can a context-aware rank transformation provide predictive power comparable to the power of log transformation?*

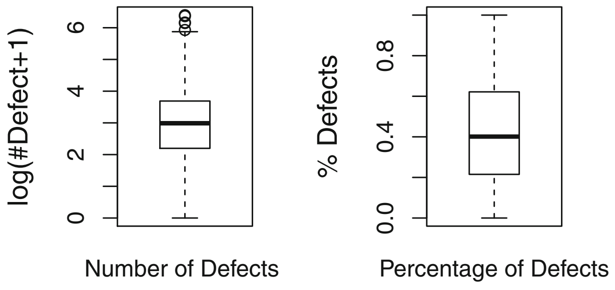

Fig. 3 Boxplot of the number of defects and the percentage of defects in our dataset

RQ2. *What is the performance of the universal defect prediction model?*
RQ3. *What is the performance of the universal defect prediction model on external projects?*
RQ4. *Do context factors affect the performance of the universal defect prediction model?*
RQ5. *What predictors should be included in the universal defect prediction model?*

5.1 Project Clusters

In our dataset, there are 1,385 open source projects. The set of projects is stratified into non-overlapped groups along the six context factors: programming language, issue tracking, the total lines of code, the total number of files, the total number of commits, and the total number of developers, respectively. In total, we obtain 478 non-empty groups. For each metric, we perform $\binom{478}{2} = \frac{478!}{2! \times 476!} = 114,003$ times of Mann-Whitney U tests to compare the difference of the distribution between any pair of groups. To control family-wise errors, we adjust the threshold p-value using Bonferroni correction to $0.05/114,003 = 4.39e$-07. Any pair of groups without statistically significant difference in their distribution are merged together. Moreover, the pair of groups without a large difference (measured by Cliff's δ) are also merged together. The maximum number of clusters observed for a metric is 32, which is the number of clusters obtained for the metric total_CBO (i.e., the sum of values of coupling between objects per file).

5.2 Research Questions

RQ1: ***Can a context-aware rank transformation provide predictive power comparable to the power of log transformation?***

Motivation We have proposed a context-aware rank transformation method to eliminate the impact of varied scales of metrics among different projects. The rank transformation converts raw values of all metrics to levels of the same scale. Before fitting the rank transformed metric values to a universal defect prediction model, it is necessary to evaluate the performance of our transformation approach. To achieve this goal, we compare the performance of defect prediction models built using rank transformations to the models built using log transformations. The log transformation uses the logarithm of raw metric values, and has been proved to improve the predictive power in defect prediction approaches (Menzies et al. 2007b; Jiang et al. 2008).

Approach For each project, we build two within-project defect prediction models using metrics listed in Table 4. One uses log transformed metric values, and the other uses rank transformed metric values. We call a model is a within-project defect prediction model if both training and testing data are from the same project. To evaluate the performance of predictions, we perform 10-fold cross validation on each project.

To investigate the performance of our rank transformation, we test the following null hypothesis for each performance measure:

$H0_1$: *there is no difference between the performance of defect prediction models built using log and rank transformations.*

Hypothesis $H0_1$ is two-tailed, since it investigates if rank transformation yields better or worse performance than log transformation. We conduct two-tailed and paired Wilcoxon rank sum test (Sheskin 2007) to compare the seven performance measures, using the 95 %

confidence level (i.e., p-value<0.05). The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical methods make no assumptions about the distribution of assessed variables. If there is a statistical significance, we reject the hypothesis and conclude that the performance of the two transformation techniques are different. Moreover, we compare the proportion of the successful predictions. The success of predictions is determined using two criteria: 1) strict criteria (i.e., precision and recall are greater than 0.75), as used by Zimmermann et al. (2009); and 2) loose criteria (i.e., precision is greater than 0.5 and recall is greater than 0.7), as applied by He et al. (2012).

Findings There are 99 projects that do not contain enough files to perform 10-fold cross validation. Hence, we compare the performance of log and rank transformations on the remaining 1,286 projects. Table 5 presents the mean values of the seven performance measures of both log and rank transformations, and the corresponding p-values of Wilcoxon rank sum test. We reject the hypothesis $H0_1$ for most measures (except recall), and conclude that there is significant difference between rank transformation and log transformation in within-project defect prediction in precision, false positive rate, F-measure, g-measure, MCC, and AUC. However, the differences between their average performance measures are negligible (i.e., the absolute value of Cliff's δ is less than 0.147), as shown in Table 5. To better illustrate the differences, we show the boxplots of performance measures of models built using log and rank transformation in Fig. 4.

Furthermore, the proportion of successful predictions for both approaches are identical where it is 13 % and 27 % using the strict and loose criteria for successful prediction, respectively. Therefore, we conclude that rank transformation achieves comparable performance to log transformation. It is reasonable to use the proposed rank transformation method to build universal defect prediction models.

Rank transformation achieves comparable performance to log transformation. The universal model can be built using rank transformed predictors.

RQ2: ***What is the performance of the universal defect prediction model?***

Table 5 The results of Wilcoxon rank sum tests and mean values of the seven performance measures of log transformation and our context-aware rank transformation in the within-project settings

Measures	Log transformation	Rank transformation	p-value	Cliff's δ
prec	0.519	**0.525**	2.42e-04*	−0.091
pd	0.576	**0.580**	0.08	−0.047
fpr	0.369	**0.359**	4.04e-04*	0.113
F-measure	0.527	**0.534**	8.19e-06*	−0.113
g-measure	0.511	**0.521**	1.37e-09*	−0.113
MCC	0.202	**0.214**	7.44e-06*	−0.120
AUC	0.609	**0.615**	8.90e-05*	−0.091

Bold font indicates a better performance

*Denotes statistical significance

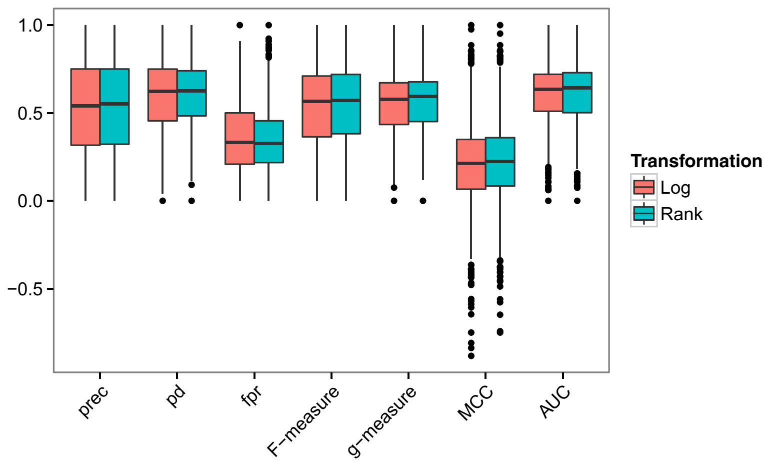

Fig. 4 Boxplots of performance measures of models built using log and rank transformations

Motivation The findings of **RQ1** support the feasibility of our proposed rank transformation method for building defect prediction models. However, building an effective universal model is still a challenge. For instance, Menzies et al. (2011) report to experience poor performance of a model built on the entire set of diverse projects. This research question aims to investigate the best achievable predictive power of the universal model. First, we evaluate if the predictive power of the universal model can be improved by adding context factors as predictors, together with code metrics and process metrics that are commonly used in prior studies for defect prediction. Second, we study if the universal model can achieve comparable performance as within-project defect prediction models. Accordingly, we split **RQ2** to two sub questions:

RQ2.1: *Can context factors improve the predictive power?*
RQ2.2: *Is the performance of the universal model defect prediction comparable to within-project models?*

Approach We describe approaches to address each sub question, respectively.

To address RQ2.1, we build the universal model using five combinations of metrics: 1) code metrics; 2) code and process metrics; 3) code metrics and context factors; 4) process metrics and context factors; and 5) code, process metrics, and context factors. All metrics are transformed using the context-aware rank transformation. To evaluate the performance of predictions, we perform 10-fold cross validation on the entire set of projects. To compare the performance of the universal model among different combinations of metrics, we test the following null hypothesis for each pair of metric combinations:

$H0_{21}$: *there is no difference between the performance of the universal defect prediction models built using two metric combinations.*

To address RQ2.2, we obtain the performance of within-project and universal models for each project, respectively. The predictive power of within-project models is obtained using 10-fold cross-validation (same as **RQ1**). The performance of universal models on a particular project is evaluated by applying a universal model built upon the remaining set of projects on the project. We compute and compare the proportion of acceptable predictions of both

Table 6 The seven performance measures (mean± std.dev) of comparing the universal models built using code metrics (CM), code + process metrics (CPM), code metrics and context (CM-C), process metrics and context (PM-C), and code + process metrics + contexts (CPM-C), respectively

Measures	CM	CPM	CM-C	PM-C	CPM-C
prec	0.431 ± 0.067	0.437 ± 0.069	0.445 ± 0.061	0.438 ± 0.063	**0.455** ± 0.065
pd	0.551 ± 0.020	0.548 ± 0.015	**0.602** ± 0.048	0.557 ± 0.038	0.591 ± 0.040
fpr	0.404 ± 0.025	**0.392** ± 0.020	0.419 ± 0.070	0.401 ± 0.073	0.396 ± 0.065
F-measure	0.480 ± 0.046	0.484 ± 0.048	0.508 ± 0.043	0.488 ± 0.048	**0.510** ± 0.045
g-measure	0.572 ± 0.016	0.577 ± 0.013	0.587 ± 0.027	0.574 ± 0.029	**0.594** ± 0.022
MCC	0.141 ± 0.032	0.150 ± 0.029	0.175 ± 0.047	0.150 ± 0.060	**0.186** ± 0.045
AUC	0.600 ± 0.020	0.607 ± 0.019	0.636 ± 0.041	0.628 ± 0.046	**0.641** ± 0.038

Bold font indicates a better performance

the universal model and the within-project models. To compare the performance of within-project and universal models, we test the following null hypothesis for each performance measure:

$H0_{22}$: *there is no difference between the performance within-project and universal defect prediction models.*

Hypotheses $H0_{21}$ and $H0_{22}$ are two-tailed, since they investigate if one prediction model yields better or worse performance than the other prediction model. We apply two-tailed and paired Wilcoxon rank sum test at 95 % confidence level to examine each hypothesis. If there is significance, we reject the null hypothesis and compute Cliff's δ (Cliff 1993) to measure the difference.

Findings We report our findings for two sub questions, respectively.

(RQ2.1) Table 6 provides the performance measures of the universal model using each combination of metrics. In general, adding context factors increases five performance measures (i.e., precision, F-measure, g-measure, MCC, and AUC value). AUC value is the only measure that is independent of the cut-off value. As the space is limited, Table 7 only presents Cliff's δ and p-value of Wilcoxon rank sum tests on comparisons of AUC values. We observe that adding context factors significantly improves the performance than using only code metrics. The Cliff's δ is -0.734, indicating a large improvement (i.e., the absolute value of Cliff's δ is greater than 0.474). In addition, adding context factors yields significant improvement (Cliff's δ is -0.707) in the performance than using code

Table 7 The Cliffs δ and p-value of Wilcoxon rank sum tests on the comparison of AUC values across the universal model built using different combinations of metrics

Metric Sets	CPM		CM-C		PM-C		CPM-C	
CM	−0.572	0.07	−0.734	9.15e-03*	−0.491	0.16	−0.804	5.89e-03*
CPM	–	–	−0.606	0.04*	−0.392	0.23	−0.707	8.00e-03*
CM-C	–	–	–	–	0.244	0.49	−0.631	0.02*
PM-C	–	–	–	–	–	–	−0.406	0.13

*Denotes statistical significance

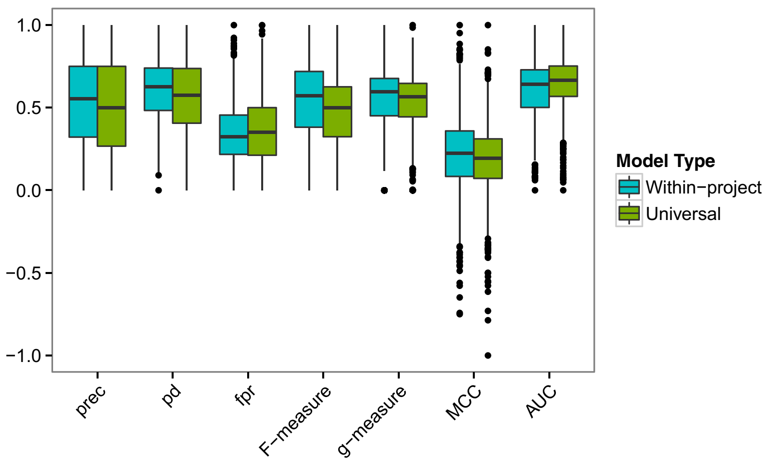

Fig. 5 Boxplots of performance measures of within-project and universal models

and process metrics. Hence, we conclude that the context factors are good predictors for building a universal defect prediction model.

(RQ2.2) The boxplots of performance measures of within-project and universal models are shown in Fig. 5. Table 8 presents the Wilcoxon rank sum test results of performance measures between within-project model and universal models built using rank transformations. We reject the null hypothesis $H0_{22}$ for all measures except precision and g-measure. The results show that the universal model and the within-project model have similar precision, recall, false positive rate, g-measure, and MCC. The differences in these five performance measures are neither significant (i.e., p-value is greater than 0.05) nor observable (i.e., the absolute value of Cliff's δ is less than 0.147). There is observable small (i.e., the absolute value of Cliff's δ is greater than 0.147, but less than 0.330) difference in F-measure and AUC value. The universal model has lower F-measure but higher AUC value than within-project model. F-measure is computed based on the confusion matrix (see Section 3.7) that is obtained using a cut-off value. On the other hand, calculating AUC does not require a cut-off value. The possible cause of lower F-measure but

Table 8 The results for Wilcoxon rank sum tests and average values of the seven performance measures of within-project models and universal models

Measures	Within-project models	Universal models	p-value	Cliff's δ
prec	**0.525**	0.518	0.47	0.047
pd	**0.580**	0.570	6.60e-03*	0.031
fpr	**0.359**	0.365	0.04*	−0.016
F-measure	**0.534**	0.474	< 2.2e-16*	0.291
g-measure	0.521	**0.534**	0.19	−0.054
MCC	**0.214**	0.184	1.32e-03*	0.113
AUC	0.615	**0.655**	< 2.2e-16*	−0.219

Bold font indicates a better performance

*Denotes statistical significance

Table 9 The descriptive statistics of the five external projects used in this study

Project	TLOC	TNC	# Classes	# Defects	Percentage of defects
Eclipse	224, 055	45, 482	997	206	20.7 %
Equinox	39, 534	3, 691	324	129	39.8 %
Lucene	73, 184	4, 329	691	64	9.3 %
Mylyn	156, 102	20, 451	1,862	245	13.2 %
PDE	146, 952	20, 228	1,497	209	14.0 %

higher AUC value of the universal model is that different cut-off values may be needed for different projects when applying the universal model. Understanding how to choose the best cut-off values might help improve the F-measure of the universal model.

Moreover, the universal models yield similar percentage (i.e., 3.6 %) of successful predictions (see **RQ1**) as Zimmermann et al. (2009) who report a 3.4 % success rate. If using loose criteria, the universal model achieves 13 % of successful predictions, much higher than He et al. (2012) who report 0.32 % of successful predictions. The universal model achieves up to 48 % (i.e., 13 % against 27 %) of the successful predictions by within-project model. We conclude that our approach for building a universal model is promising.

The universal model yields similar predictive performance as within-project models.

RQ3: ***What is the performance of the universal defect prediction model on external projects?***

Motivation In **RQ2**, we successfully build a universal model for a large set of projects. The universal model slightly outperforms within-project models in terms of recall and AUC. Although our experiments involve a large number of projects from various contexts, the projects are selected from only two hosts: SourceForge and GoogleCode. It is still unclear if the universal model is generalizable, i.e., whether it works well for external projects that are not managed on the aforementioned two hosts. This research question aims to investigate the capability of applying the universal model to predict defects for external projects that are not hosted on SourceForge or GoogleCode.

Approach To address the question, we choose to use the publicly available dataset[4] that was collected by D'Ambros et al. (2010). The dataset contains four Eclipse projects (i.e., Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, and Mylyn), and one Apache project (i.e., Lucene). We present the descriptive statistics of the five external projects in Table 9.

We calculate the six context factors of the five aforementioned projects, and apply related ranking functions to convert their raw metric values to one of the ten levels. We predict defects on each project using the universal model which is learnt from 1,385 SourceForge

[4]http://bug.inf.usi.ch/download.php

and GoogleCode projects. The seven performance measures of within-project models are obtained via 10-fold cross validation for each project.

Findings Table 10 presents the average values of the seven performance measures of the universal model and within-project models, respectively. Overall, there are clear differences in the performance (all measures except AUC value) of the universal model and within-project models. In particular, the universal model yields lower precision, larger false positive rate, but higher recall than within-project models. However, these performance measures depend on the cut-off value that is used to determine if an entity is defective or not (see Section 3.7). Such performance measures can be significantly changed by altering the cut-off value. The AUC value is independent of the cut-off value and is preferred for cross-project defect prediction (Rahman et al. 2012). As the universal model achieves similar AUC values to within-project models on the five subject projects, we conclude that the universal model is as effective as within-project defect prediction models for the five subject projects. However, various cut-off values may be needed to yield high precision or low false positive rate, when applying the universal model on different projects. We present further discussions on dealing with high false positive rate as follows.

Discussions on False Positive Rate In practice, high false positive is unacceptable, e.g., false positive rate is greater than 0.64 (Turhan et al. 2009). As shown in Table 10, the universal model experiences high false positive rates in three projects (i.e., Eclipse, Lucene, and PDE). In **RQ2**, we observe that the universal model exhibits similar false positive rate to within-project defect prediction models in general. Hence, we conjecture that the high false positive rate in external projects is due to the different percentages of defects in the training set (e.g., the median percentage of defects is 40 %) and in the five external projects (e.g., the median percentage of defects is 14 %). Nevertheless, it is of significant interest to seek insights on how to determine cut-off values to reduce false positive rate.

Table 10 The performance measures for within-project model and the universal model

Measures	Eclipse	Equinox	Lucene	Mylyn	PDE	Type
prec	**0.323**	**0.621**	**0.197**	**0.252**	**0.220**	within-project
	0.222	0.607	0.128	0.168	0.155	universal
pd	0.782	0.775	0.531	0.473	0.732	within-project
	0.937	**0.899**	**0.922**	**0.767**	**0.914**	universal
fpr	**0.427**	**0.313**	**0.222**	**0.213**	**0.422**	within-project
	0.853	0.385	0.643	0.578	0.806	universal
F-measure	**0.457**	0.690	**0.287**	**0.329**	**0.338**	within-project
	0.359	**0.725**	0.224	0.275	0.266	universal
g-measure	**0.661**	0.728	**0.631**	**0.591**	**0.646**	within-project
	0.254	**0.731**	0.515	0.545	0.320	universal
MCC	**0.287**	0.453	**0.207**	**0.204**	**0.216**	within-project
	0.101	**0.512**	0.172	0.131	0.098	universal
AUC	0.764	0.804	0.727	**0.677**	0.700	within-project
	0.766	**0.821**	**0.750**	0.664	**0.704**	universal

Bold font indicates a better performance

1) *Effort-aware estimation of the cut-off value.* It is time consuming to examine all entities that are predicted as defective. If a development team has limited resources or a tight schedule, it is more realistic to inspect only the top X % of entities that are predicted as defective. To this end, we choose the minimum predicted probability among the top X % of entities as the cut-off value for each project. We recalculate the performance measures, and present the detailed results in Table 11. We observe that the median false positive is reduced to 0.053, if considering only the top 10 % of entities as defective. When considering only the top 20 % and 30 % of entities as defective, the median false positive rate becomes 0.137 and 0.230, respectively. As cut-off values change, the other performance measures are also updated. For instance, the median recall becomes 0.270, 0.444, and 0.578, respectively, if considering only the top 10 %, 20 %, and 30 % of entities as defective. The AUC values remain the same when altering cut-off values. Therefore, we conclude that high false positive rate can be tamed by considering only the top 10 %, 20 %, or 30 % of entities that are predicted as defective by the universal model.

Table 11 The performance measures for the universal model on external projects with cut-off values determined by the minimum predicted probability by the universal model among the top 10 %, 20 %, and 30 % of defective entities

Top %	Measure	Eclipse	Equinox	Lucene	Mylyn	PDE	Median	Average
10 %	Cut-off	0.968	0.950	0.948	0.942	0.964	0.950	0.954
	prec	0.722	0.844	0.309	0.404	0.331	0.404	0.522
	pd	0.316	0.209	0.328	0.302	0.196	0.302	0.270
	fpr	0.032	0.026	0.075	0.067	0.064	0.064	0.053
	F-measure	0.439	0.335	0.318	0.346	0.246	0.335	0.337
	g-measure	0.476	0.345	0.484	0.456	0.324	0.456	0.417
	MCC	0.401	0.301	0.246	0.266	0.166	0.266	0.276
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
20 %	Cut-off	0.958	0.912	0.905	0.903	0.946	0.912	0.925
	prec	0.538	0.746	0.234	0.276	0.312	0.312	0.421
	pd	0.510	0.364	0.500	0.412	0.435	0.435	0.444
	fpr	0.114	0.082	0.167	0.164	0.156	0.156	0.137
	F-measure	0.524	0.490	0.318	0.331	0.363	0.363	0.405
	g-measure	0.647	0.522	0.625	0.552	0.574	0.574	0.584
	MCC	0.404	0.349	0.242	0.211	0.244	0.244	0.290
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
30 %	Cut-off	0.944	0.879	0.825	0.844	0.924	0.879	0.883
	prec	0.444	0.680	0.198	0.225	0.274	0.274	0.364
	pd	0.641	0.512	0.641	0.510	0.584	0.584	0.578
	fpr	0.209	0.159	0.265	0.267	0.252	0.252	0.230
	F-measure	0.525	0.584	0.303	0.312	0.373	0.373	0.419
	g-measure	0.708	0.636	0.685	0.602	0.656	0.656	0.657
	MCC	0.383	0.377	0.238	0.180	0.252	0.252	0.286
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741

2) *Other insights on selecting the cut-off value.* Inspired by transfer learning (Pan and Yang 2010), we suppose that appropriate cut-off values may be learnt from target projects. Intuitively, we conjecture that the appropriate cut-off values might depend on the percentage of defective entities in target projects. Alternatively, minimizing the total error rates (i.e., FP+FN) may help reduce the false positive rate while maintain the recall. Therefore, we examine if it is feasible to reduce false positive rate by inferring the cut-off value based on: 1) the percentage of defects in the target project; and 2) the minimized total error rates (i.e., FP+FN). Table 12 shows the detailed results of the two methods. In particular, using the percentage of defects in the target project results in median false positive rate as 0.323 along with 0.727 of recall. Minimizing error rates yields median false positive rate as 0.255 along with 0.609 of recall. In both cases, the performances are similar as the work by Turhan et al. (2009) that successfully reduces high false positive rate to 0.33 along with 0.68 of recall by filtering the training set.

However, it is impractical to obtain the defectiveness of all entities in the target project. Otherwise, a within-project defect prediction model can be constructed. Therefore, it is necessary to investigate how many entities are required to estimate cut-off values. As using less entities may yield unstable cut-off values, we average the cut-off values determined by the aforementioned two methods as the final cut-off values. We perform an exploratory experiment by randomly sampling 10, 20, and 30 entities from each project. We further repeat the experiment 100 times for each project, and report the average performance measures in Table 13. In general, we can observe that increasing the number of randomly sampled entities improves the performance of the universal model in terms of five measures (i.e., precision, false positive rate, F-measure, g-measure, and MCC). When randomly sampling 10 entities, the universal model achieves the median false positive as 0.344 and the median recall as 0.723. Apart from the work by Turhan et al. (2009) that customizes the prediction

Table 12 The performance measures for the universal model on external projects with cut-off values obtained by using ratio of defects or minimizing the error rates on the entire set of entities

	Measure	Eclipse	Equinox	Lucene	Mylyn	PDE	Median	Average
Percentage of defects	Cut-off	0.793	0.602	0.907	0.868	0.860	0.860	0.806
	prec	0.265	0.625	0.235	0.241	0.219	0.241	0.317
	pd	0.879	0.814	0.500	0.482	0.727	0.727	0.680
	fpr	0.635	0.323	0.166	0.230	0.420	0.323	0.355
	F-measure	0.407	0.707	0.320	0.321	0.337	0.337	0.418
	g-measure	0.516	0.739	0.625	0.593	0.645	0.625	0.624
	MCC	0.213	0.481	0.244	0.193	0.214	0.214	0.269
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
Minimized error rates	Cut-off	0.936	0.519	0.851	0.893	0.923	0.893	0.824
	prec	0.415	0.616	0.210	0.270	0.274	0.274	0.357
	pd	0.699	0.884	0.609	0.449	0.593	0.609	0.647
	fpr	0.257	0.364	0.234	0.184	0.255	0.255	0.259
	F-measure	0.521	0.726	0.312	0.337	0.375	0.375	0.454
	g-measure	0.721	0.740	0.679	0.579	0.661	0.679	0.676
	MCC	0.376	0.514	0.245	0.217	0.256	0.256	0.322
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741

Table 13 The performance measures for the universal model on external projects with cut-off values learnt from both the ratio of defects and the minimized error rates, using a subset of randomly sampled entities

# Entities	Measure	Eclipse	Equinox	Lucene	Mylyn	PDE	Median	Average
10	prec	0.298	0.618	0.195	0.220	0.221	0.221	0.310
	pd	0.832	0.815	0.644	0.575	0.723	0.723	0.718
	fpr	0.537	0.344	0.298	0.342	0.445	0.344	0.393
	F-measure	0.434	0.696	0.291	0.307	0.332	0.332	0.412
	g-measure	0.573	0.712	0.647	0.585	0.596	0.596	0.623
	MCC	0.249	0.472	0.225	0.174	0.202	0.225	0.264
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
20	prec	0.308	0.621	0.212	0.235	0.231	0.235	0.321
	pd	0.816	0.824	0.593	0.545	0.687	0.687	0.693
	fpr	0.498	0.338	0.259	0.310	0.402	0.338	0.361
	F-measure	0.442	0.705	0.297	0.310	0.336	0.336	0.418
	g-measure	0.608	0.727	0.629	0.578	0.607	0.608	0.630
	MCC	0.265	0.483	0.228	0.182	0.210	0.228	0.274
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741
30	prec	0.310	0.620	0.226	0.238	0.231	0.238	0.325
	pd	0.814	0.836	0.559	0.529	0.690	0.690	0.686
	fpr	0.488	0.344	0.231	0.290	0.399	0.344	0.350
	F-measure	0.445	0.709	0.299	0.313	0.336	0.336	0.420
	g-measure	0.618	0.730	0.615	0.583	0.614	0.615	0.632
	MCC	0.270	0.488	0.232	0.185	0.212	0.232	0.277
	AUC	0.766	0.821	0.750	0.664	0.704	0.750	0.741

model (i.e., filtering the training set) based on the target project, the universal model is not altered for a particular target project. Software organizations do not need to provide their data for customizing prediction models, but tune cut-off values for their goals. The universal model can help address the concern on sharing data or model across companies (Peters et al. 2013a). Although inspection on the defectiveness in the target project is needed, the proportion of required entities is relatively low, such as 1 % (10/997) for Eclipse, 3 % (10/324) of Equinox, 1 % (10/691) for Lucene, 1 % (10/1862) for Mylyn, and 1 % (10/1497) for PDE.

As a summary, the results show that our universal model can provide comparable performances to within-project defect prediction models for the five subject projects. Considering the five projects might conduct different development strategies than SourceForge or GoogleCode projects, there is a high chance to apply the universal model on more external projects with acceptable predictive power.

The universal model can predict defects for external projects with acceptable false positive rate.

RQ4: ***Do context factors affect the performance of the universal defect prediction model?***

Motivation In **RQ3**, we verified the capability of the universal model to predict defects for five external projects. The universal model can interpret general relationships between metrics and defect proneness, regardless of the place where projects are hosted. However, the five external projects have limited diversity. For example, they are all written in Java. The generalizability of the universal model is not deeply examined in **RQ3**. This threatens the applicability of the universal model to projects from different contexts (Nagappan et al. 2013). Hence, it is essential to investigate whether the performance of the universal model varies across projects with different context factors.

Approach To address this question, we compare the performance of the universal model across projects with different context factors (see Section 3.2). To train a universal model with the largest number of projects, we apply leave-one-out cross validation (e.g., Zhou and Leung 2007). For a particular project, we use all other projects to build a universal model and apply the universal model on the project. We repeat this step for every project and obtain the predictive power of the universal model on each project. As the findings of **RQ2** and **RQ3** suggest that different projects may prefer different cut-off values, we choose to only compare AUC values to avoid the impact of cut-off values on our observations.

We divide the entire set of projects along each context factor, respectively. There are three types of context factors: categorical factor (i.e., programming language), boolean factor (i.e., issue tracking), and numerical factors (e.g., the total lines of code). For categorical factor, we obtain a group per category. We get two groups for boolean factor. For numerical factors, we compute quantiles of the numbers and then derive four groups. All groups are listed in Table 14. The details on these groups are described in Section 3.3. Please note that these groups are created solely based on context factors, other than the distribtuion of software metrics.

For each pair of groups on a particular context factor, we test the following null hypothesis:

$H0_4$: *there is no difference in the performance of the universal model between projects of the group-pair.*

Hypothesis $H0_4$ is two-tailed, since it investigates if the universal model yields better or worse performance in one project group than the other project group of a group-pair. As the size of two groups may be different, we apply two-tailed and unpaired Wilcoxon rank sum test at 95 % confidence level to examine the hypothesis.

Table 14 Groups of projects split along different context factors

Context factor	Groups
Programming language (PL)	G_c, G_{c++}, $G_{c\#}$, G_{java}, and G_{pascal}
Issue Tracking (IT)	G_{useIT} and G_{noIT}
Total Lines of Code (TLOC)	$G_{leastTLOC}$, $G_{lessTLOC}$, $G_{moreTLOC}$, and $G_{mostTLOC}$
Total Number of Files (TNF)	$G_{leastTNF}$, $G_{lessTNF}$, $G_{moreTNF}$, and $G_{mostTNF}$
Total Number of Commits (TNC)	$G_{leastTNC}$, $G_{lessTNC}$, $G_{moreTNC}$, and $G_{mostTNC}$
Total Number of Developers (TND)	$G_{leastTND}$, $G_{lessTND}$, $G_{moreTND}$, and $G_{mostTND}$

Findings For each context factor, we present our findings on the generalizability of the universal model.

1) Programming language (PL): The average AUC values for groups G_c, G_{c++}, $G_{c\#}$, G_{java}, and G_{pascal} are 0.64, 0.65, 0.61, 0.64, and 0.64, respectively. There are 5 groups of projects divided by programming languages, and the number of pairwise comparisons is 10. Hence, the threshold p-value is 5.00e-03 after Bonferroni correction. The p-values of Wilcoxon rank sum test are always greater than 5.00e-03. We do not find enough evidence to support that there are significant difference across projects with different programming languages. In other word, the universal model yields similar performance for projects written in any of the five studied programming languages. There exist common relationships between software metrics and defect proneness, no matter whether projects are developed using C, C++, C#, Java, or Pascal. Future work is needed to understand such common relationships more deeply.
2) Issue tracking (IT): The average AUC values for groups G_{useIT} and G_{noIT} are 0.64 and 0.65, respectively. There is only one pair of groups of projects divided by the usage of issue tracking systems, and therefore the threshold p-value is 0.05. The p-value of Wilcoxon rank sum test is 0.14, indicating that there is no significant difference between projects with or without usage of issue tracking systems.
3) Total lines of code (TLOC): The average AUC values for groups $G_{leastTLOC}$, $G_{lessTLOC}$, $G_{moreTLOC}$, and $G_{mostTLOC}$ are 0.63, 0.65, 0.65, and 0.64, respectively. There are 4 groups of projects divided by the total lines of code, and the number of pairwise comparisons is 6. Hence, the threshold p-value is 8.33e-03 after Bonferroni correction. The p-values of Wilcoxon rank sum test are always greater than 8.33e-03. The universal model can reveal general relationships between software metrics and defect proneness for small, medium, or large projects.
4) Total number of files (TNF): The average AUC values for groups $G_{leastTNF}$, $G_{lessTNF}$, $G_{moreTNF}$, and $G_{mostTNF}$ are 0.62, 0.65, 0.64, and 0.64, respectively. Similarly, the threshold p-value is 8.33e-03 after Bonferroni correction. The p-values of Wilcoxon rank sum test are always greater than 8.33e-03. We conclude that no matter how many number of files a project has, the universal model can predict defect proneness without significant difference in its performance.
5) Total number of commits (TNC): The average AUC values for groups $G_{leastTNC}$, $G_{lessTNC}$, $G_{moreTNC}$, and $G_{mostTNC}$ are 0.64, 0.65, 0.65, and 0.63, respectively. There are 4 groups and 6 pair-wise comparisons. We correct the threshold p-value to 8.33e-03. The p-values of Wilcoxon rank sum test are always greater than 8.33e-03. There is no significant difference in the performance of the universal model across projects with different total number of commits.
6) Total number of developers (TND): The average AUC values for groups $G_{leastTND}$, $G_{lessTND}$, $G_{moreTND}$, and $G_{mostTND}$ are 0.63, 0.65, 0.64, and 0.64, respectively. There are 4 groups and 6 pair-wise comparisons. We correct the threshold p-value to 8.33e-03. The p-values of Wilcoxon rank sum test are always greater than 8.33e-03. The performance of the universal model does not change significantly across projects with different number of developers.

As a summary, we can not find enough evidence to support the hypothesis that the universal model performs significantly different for projects with different context factors.

Hence, we conclude that the universal model is applicable to projects with different context factors.

> *The universal model yields similar performance when it is applied to projects with different context factors, and therefore it is context-insensitive.*

RQ5: ***What predictors should be included in the universal defect prediction model?***

Motivation The purpose of **RQ3** and **RQ4** is to show that our universal model is applicable to external projects, and is context-insensitive, respectively. Therefore in earlier experiments, we use all metrics together for building Naive Bayes models. However, many of these metrics may be strongly correlated. To build an interpretable model, we need to select a subset of these metrics that are not strongly correlated. In within-project settings, the importance of various metrics has been examined in depth (e.g., Shihab et al. 2010). For the universal model, we aim to find an uncorrelated interpretable set of predictors that are associated with the chance that a file will have a fix in the future. We do this to understand the general relationship between predictors and defect proneness for the entire set of projects.

Approach To make the model more interpretable, we chose to use logistic regression to build the universal model. Our choice was motivated by the ease with which logistic regression coefficients can be interpreted (Zimmermann et al. 2012). For instance, the sign of a coefficient presents the direction of the impact, i.e., positive or negative. The magnitude indicates the strength of the impact, i.e., how much the probability of defect proneness is affected by a one-unit change in the corresponding predictor. Further details on the convention from coefficients to exact probabilities can be found in the book by Hosmer et al. (2013).

Because predictors may be highly correlated, we first need to select an uncorrelated subset of predictors. We use the following rules to select predictors:

1) Select well-known simplest predictors that are uncorrelated. We choose lines of code (LOC) as code metrics and number of revisions (NREV) as process metrics that have been often associated with future fixes.
2) We analyze the correlation among context factors, and find that context factors are strongly associated. Hence, we choose the total number of files (TNF) as a context measure of project size and the total number of developers (TND) as a context measure of project activity. Using the first, second, and third quartiles, we convert the two context measures to four levels, respectively. We treat them as categorical variables (same as programming language) in the model because the odds of future fixes may not increase by the same amount as we go from one level (defined by quartiles) to the next.
3) We perform hierarchical clustering for all predictors using distance defined as $1 - \|cor(p_1, p_2)\|^2$, where p_1 and p_2 are two predictors. We use R function *hclust*[5] to get clusters of predictors as shown in Fig. 6. We then apply R function *cutree*[6] to get eight distinct clusters.
4) For each cluster not containing the aforementioned predictors, we choose the first predictor that uses the simplest aggregation (avg).

[5] http://stat.ethz.ch/R-manual/R-patched/library/stats/html/hclust.html

[6] http://stat.ethz.ch/R-manual/R-patched/library/stats/html/cutree.html

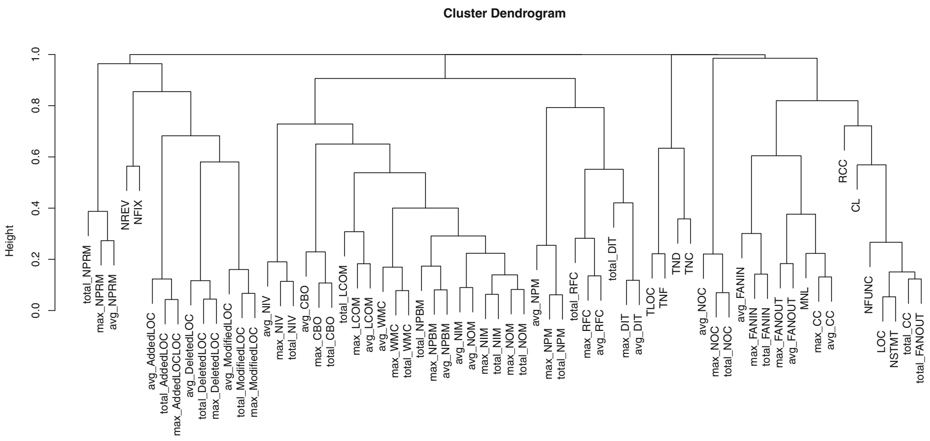

Fig. 6 Cluster dendrogram of predictors

We then use these selected predictors to build the universal model. The *glm*[7] method in R is use to build the logistic regression model. We then inspect the coefficients for each metric to interpret the universal model.

Findings There are eight code and process metrics selected, i.e., lines of code (Loc), average number of immediate subclasses (avgNoc), average number of instance variables (avgNiv), average number of protected methods (avgNpm), average number of private methods (avgNprm), average number of input data (avgFanin), the number of revisions (Nrev), and average added lines of code (avgAddedloc). The correlation matrix of the selected eight predictors in Table 15 does not show any correlations above 0.5.

Table 16 presents the coefficient for each predictor and the amount of deviance that each predictor explains. The final model explains 7 % of deviance of the probability of fixes for the entire set of projects. The null deviance is 177,497 on 136,160 degrees of freedom, while residual deviance is 165,003 on 136,142 degrees of freedom. It is important to note that each predictor should be considered as a representative of all tightly correlated predictors within the cluster. In particular, avgNiv represents a very large number of metrics, including Cbo, Lcom, Wmc, Npbm, Nim, and Nom. Also, avgNprm is not significantly different from zero, suggesting that all predictors in the small cluster are not helping model defect proneness. Furthermore, coefficients for avgAddedLoc and avgNoc do not explain as much variance as the remaining predictors and have coefficient values that are barely significantly different from zero. All of these three predictors should be removed from the final model used for prediction in practice.

In models where each predictor is measured in different units, it is difficult to compare coefficient magnitudes among predictors. In our study, coefficient magnitudes of code and process metrics can be compared, as they have exactly the same units after the rank transformation. In the resulting model, The most important code metric is lines of code (Loc), followed by average number of input data (avgFanin) and average number of private methods (avgNpm). The three code metrics can explain 4,876 of deviance for the probability of

[7]http://stat.ethz.ch/R-manual/R-patched/library/stats/html/glm.html

Table 15 The Pearson correlation among selected code and process metrics

	avgNOC	avgNIV	avgNPM	avgNPRM	avgFANIN	NREV	avgADDEDLOC
LOC	−0.08*	0.18*	0.03*	0.15*	0.49*	0.32*	0.27*
avgNOC	–	0.02*	0.00	0.01*	−0.02*	0.03*	0.02*
avgNIV	–	–	0.20*	0.27*	0.07*	0.09*	0.04*
avgNPM	–	–	–	0.05*	−0.02*	0.04*	−0.05*
avgNPRM	–	–	–	–	0.05*	0.08*	0.06*
avgFANIN	–	–	–	–	–	0.18*	0.12*
NREV	–	–	–	–	–	–	0.28*

*Denotes statistical significance

defect proneness for the entire set of projects. The most important process metrics is the number of revisions (NREV), which can explain 1,493.7 of deviance.

Among context factors, the most important predictor is the total number of files, followed by the programming languages and the total number of developers. The three context factors explain 6,027.4 of deviance in total. The R tool treats the alphabetically earliest category of each categorical factor as the reference level, and folds it into the intercept term. The

Table 16 The coefficents of each predictor in the logistic regression model. The numerical intercept is -2.68. For context factors PL, TNF and TND, “C”, “$leastTNF$” and “$leastTND$” are folded into the intercept term, respectively

Type	Metric Name		Coefficients	p-value	Deviance explained	p-value of χ^2-test
	(Intercept)		−2.68	$< 2.2e$-16*		
Context Factors	TNF	$lessTNF$	−0.24	$< 2.2e$-16*	4969.5	$< 2.2e$-16*
		$moreTNF$	−0.44	$< 2.2e$-16*		
		$mostTNF$	−1.33	$< 2.2e$-16*		
	TND	$lessTND$	0.05	9.03e-03*	508.9	$< 2.2e$-16*
		$moreTND$	0.15	1.60e-15*		
		$mostTND$	0.36	$< 2.2e$-16*		
	PL	C++	−0.28	$< 2.2e$-16*	549.0	$< 2.2e$-16*
		C#	0.07	0.01*		
		Java	−0.29	$< 2.2e$-16*		
		Pascal	−0.91	$< 2.2e$-16*		
Code Metrics	LOC		0.10	$< 2.2e$-16*	4555.8	$< 2e$-16*
	avgNOC		0.06	1.23e-03*	11.4	7.53e-04*
	avgNIV		−0.04	$< 2.2e$-16*	65.7	5.15e-16*
	avgNPM		0.10	$< 2.2e$-16*	89.5	$< 2.2e$-16*
	avgNPRM		−0.01	0.29	7.7	5.40e-03*
	avgFANIN		0.04	$< 2.2e$-16*	230.7	$< 2.2e$-16*
Process Metrics	NREV		0.10	$< 2.2e$-16*	1493.7	$< 2.2e$-16*
	avgADDEDLOC		0.01	1.35e-04*	12.0	5.27e-04*

*Denotes statistical significance

intercept represents the base probability of defect proneness when all categorical factors are at reference levels. In our case, “C” programming language, “leastTNF”, and “leastTND” are the reference levels, therefore not shown in Table 16. There are differences among languages with “C” code having more fixes than “Java”, “C++”, and “Pascal” code. Projects with more developers involved (relatively to other projects in the cluster) are also more likely to contain a fix. But projects with more files (relatively to other projects in the cluster) are less likely to contain a fix. Future research is needed to fully understand the mechanisms and causes that affect both the predictor values and the chances of future fixes.

The final universal model can be obtained by the following equation:

$$P = 1 - \frac{1}{1 + exp^{(\beta_0 + \beta_1 * m_1 + \ldots + \beta_k * m_k)}} \tag{9}$$

where P is the probability of a file to be defective, k is the total number of predictors (in our case k=11, including three context factors and eight metrics), β_0 is the intercept, and β_i is the coefficient of metric m_i $(i = 1, \ldots, k)$. For instance, if m_1 is metric Loc, then β_1 is 0.10. This model can be implemented in an integrated development environment (IDE) for instant evaluation of defect proneness, and be used to compare defect proneness across projects.

*Our universal model explains 7% of variance of the probability of defect proneness for the entire set of projects. The most important code and process metrics are lines of code (*LOC*) and the number of revisions (*NREV*), respectively. The most influential context factor is the total number of files, followed by programming languages and the total number of developers.*

6 Threats to Validity

We now discuss the threats to validity of our study following common guidelines provided by Yin (2002).

Threats to Conclusion Validity concern the relation between the treatment and the outcome. One conclusion validity threat comes from data cleaning methods. For instance, we remove the projects with negligible fix-inducing or non-fixing commits (both using 75 % quantile as the threshold). We plan to investigate the impact of different thresholds in future study. Another threat is due to the extraction of defect data. We mine defect data solely based on commit messages, since 42 % of our subject projects do not use issue tracking systems. To deal with this threat, we use a large set of subject projects (Rahman et al. 2013) and apply Naive Bayes as the modelling technique that have strong noise resistance with defect data (Rahman et al. 2013).

Threats to Internal Validity concern our selection of subject projects and analysis methods. SourceForge and GoogleCode are considered to have a large proportion of not well managed projects. We believe that our data cleaning step increases the data quality. The other threat to internal validity is the possible bias in the defect data. We plan to include well managed projects (e.g., Linux projects, Eclipse projects, and Apache projects) in future study.

Threats to External Validity concern the possibility to generalize our results. Although we demonstrate the capability of the universal model on predicting defects for four Eclipse projects and one Apache project, it is unclear if the universal model also performs well for commercial projects. Future validation on commercial projects is welcome.

Threats to Reliability Validity concern the possibility of replicating this study. The subject projects are publicly available from SourceForge and GoogleCode. We attempt to provide all necessary details to replicate our study.[8]

7 Conclusion

In this study, we attempt to build a universal defect prediction model using a large set of projects from various contexts. We first propose a context-aware rank transformation method to pre-process the predictors. This step makes predictors (i.e., software metrics) from the entire set of projects have the same scales. We compare our rank transformation with widely used log transformation, and find that the rank transformation performs as good as log transformation in within-project settings. We then build a universal model using the rank-transformed metrics. For building a universal model, we add different metric sets (i.e., code metrics, process metrics, and context factors) step by step. The studied context factors include programming languages, presence of issue tracking systems, the total lines of code, the total number of files, the total number of commits, and the total number of developers. The results show that the context factors further increase the predictive power of the universal model besides code and process metrics.

To evaluate the performance of the universal defect prediction model, we compare with within-project models. We find that the universal model has higher AUC values but lower F-measures than within-project models, suggesting that different cut-off values may be needed for different projects. We also study the generalizability of the universal model. First, we apply the universal model that are built using projects from SourceForge and GoogleCode on five external projects from Eclipse and Apache repositories. We observe that the AUC values of the predictions by the universal model are very close to within-project models built from each project. Moreover, we provide several insights on how to select appropriate cut-off values to control false positive rate. For instance, the median false positive rate is reduced to 0.053, if considering only the top 10 % of entities as defective.

We further investigate if the universal model performs differently for projects with different contexts, and find that there is no statistically significant difference for all context factors. Based on our findings, we conclude that our universal model is context-insensitive and applicable to external projects. Finally, we investigate the importance of different metrics using logistic regression model and present coefficients of each metric in the universal model. The universal model not only relieves the need for training defect prediction models for different projects, but also helps interpret basic relationships between software metrics and defects.

In future, we plan to evaluate the feasibility of the universal model for commercial projects. We will also evaluate the possibility to embed the universal model as a plugin for a version control system or an integrated development environment (IDE) to provide developers with an immediate feedback on risk.

[8] http://fengzhang.bitbucket.org/replications/universalModel.html

Acknowledgments The authors would like to thank Professor Ahmed E. Hassan from Software Analysis and Intelligence Lab (SAIL) at Queen's University for his strong support during this work. The authors would also like to thank Professor Daniel German from University of Victoria for his insightful advice. The authors are appreciated for the great help of Mr. Shane McIntosh from Software Analysis and Intelligence Lab (SAIL) at Queen's University during the improvement of this work. The authors are also grateful to the anonymous reviewers of MSR and EMSE for valuable and insightful comments.

References

Akiyama F (1971) An example of software system debugging. In: Proceedings of the international federation of information processing societies congress, pp 353–359

Alves T, Ypma C, Visser J (2010) Deriving metric thresholds from benchmark data. In: Proceedings of the 26th IEEE international conference on software maintenance, pp 1–10

Arisholm E, Briand LC, Johannessen EB (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J Syst Softw 83(1):2–17

Baggen R, Correia J, Schill K, Visser J (2012) Standardized code quality benchmarking for improving software maintainability. Softw Qual J 20:287–307

Bettenburg N, Hassan AE (2010) Studying the impact of social structures on software quality. In: Proceedings of the 18th IEEE international conference on program comprehension, ICPC '10, pp 124–133

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, pp 121–130

Cliff N (1993) Dominance statistics: ordinal analyses to answer ordinal questions. Psychol Bull 114(3):494–509

Cohen J (1988) Statistical power analysis for the behavioral sciences: Jacob Cohen, 2nd edn. Lawrence Erlbaum

Cohen J (1992) A power primer. Psychol Bull 112(1):155–159

Cruz A, Ochimizu K (2009) Towards logistic regression models for predicting fault-prone code across software projects. In: 3rd international symposium on empirical software engineering and measurement, 2009. ESEM 2009, pp 460–463

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: Proceedings of the 7th IEEE working conference on mining software repositories, MSR'10, pp 31–41

D'Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir Softw Eng 17(4-5):531–577

Denaro G, Pezzè M (2002) An empirical evaluation of fault-proneness models. In: Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002, pp 241–251

Hailpern B, Santhanam P (2002) Software debugging, testing, and verification. IBM Syst J 41(1):4–12

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. IEEE Trans Softw Eng 38(6):1276–1304

Hassan A (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st IEEE international conference on software engineering, ICSE'09, pp 78–88

He Z, Shu F, Yang Y, Li M, Wang Q (2012) An investigation on the feasibility of cross-project defect prediction. Autom Softw Eng 19(2):167–199

He Z, Peters F, Menzies T, Yang Y (2013) Learning from open-source projects: an empirical study on defect prediction. In: 2013 ACM / IEEE international symposium on empirical software engineering and measurement, pp 45–54

Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the 35th international conference on software engineering, ICSE '13, pp 392–401

Hosmer D W Jr, Lemeshow S, Sturdivant RX (2013) Interpretation of the Fitted Logistic Regression Model. Wiley, pp 49–88

Jiang Y, Cukic B, Menzies T (2008) Can data transformation help in the detection of fault-prone modules? In: Proceedings of the 2008 workshop on defects in large software systems, DEFECTS '08, pp 16–20

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Proceedings of the 33rd international conference on software engineering, ICSE '11, pp 481–490

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Softw Eng (TSE) 34(4):485–496

Li M, Zhang H, Wu R, Zhou ZH (2012) Sample-based software defect prediction with active and semi-supervised learning. Autom Softw Eng 19(2):201–230

Ma Y, Luo G, Zeng X, Chen A (2012) Transfer learning for cross-company software defect prediction. Inf Softw Technol 54(3):248–256

Mair C, Shepperd M (2005) The consistency of empirical comparisons of regression and analogy-based software project cost prediction. In: Proceedings of the 2005 international symposium on empirical software engineering, pp 509–518

Menzies T, Dekhtyar A, Distefano J, Greenwald J (2007a) Problems with precision: a response to comments on 'data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng (TSE) 33(9):637–640

Menzies T, Greenwald J, Frank A (2007b) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng (TSE) 33(1):2–13

Menzies T, Butcher A, Marcus A, Zimmermann T, Cok D (2011) Local vs. global models for effort estimation and defect prediction. In: Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering, ASE '11, pp 343–351

Mockus A (2009) Amassing and indexing a large sample of version control systems: towards the census of public source code history. In: Proceedings of the 6th IEEE international working conference on mining software repositories, MSR'09, pp 11–20

Mockus A, Votta L (2000) Identifying reasons for software changes using historic databases. In: Proceedings of the 16th international conference on software maintenance, ICSM '00, pp 120–130

Nagappan M, Zimmermann T, Bird C (2013) Diversity in software engineering research. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, vol 2013. ACM, New York, pp 466–476

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th international conference on software engineering, ACM, ICSE '06, pp 452–461

Nam J, Pan SJ, Kim S (2013) Transfer defect learning. In: Proceedings of the 2013 international conference on software engineering, ICSE '13, pp 382–391

Nguyen TT, Nguyen TN, Phuong TM (2011) Topic-based defect prediction (nier track). In: Proceedings of the 33rd international conference on software engineering, ICSE '11. ACM, New York, pp 932–935

Pan SJ, Yang Q (2010) A survey on transfer learning. IEEE Trans Knowl Data Eng 22(10):1345–1359

Peters F, Menzies T, Gong L, Zhang H (2013a) Balancing privacy and utility in cross-company defect prediction. IEEE Trans Softw Eng 39(8):1054–1068

Peters F, Menzies T, Marcus A (2013b) Better cross company defect prediction. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pp 409–418

Posnett D, Filkov V, Devanbu P (2011) Ecological inference in empirical software engineering. In: Proceedings of the 26th IEEE/ACM international conference on automated software engineering, ASE '11. IEEE Computer Society, Washington, pp 362–371

Premraj R, Herzig K (2011) Network versus code metrics to predict defects: a replication study. In: 2011 international symposium on empirical software engineering and measurement (ESEM), pp 215–224

Radjenović D, Heričko M, Torkar R, Živkovič A (2013) Software fault prediction metrics: A systematic literature review. Inf Softw Technol 55(8):1397–1418

Rahman F, Posnett D, Devanbu P (2012) Recalling the "imprecision" of cross-project defect prediction. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, FSE '12, pp 61:1–61:11

Rahman F, Posnett D, Herraiz I, Devanbu P (2013) Sample size vs. bias in defect prediction. In: Proceedings of the 21th ACM SIGSOFT symposium and the 15th European conference on foundations of software engineering, ESEC/FSE '13

Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? In: Annual meeting of the Florida association of institutional research, pp 1–33

Sarro F, Di Martino S, Ferrucci F, Gravino C (2012) A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In: Proceedings of the 27th annual ACM symposium on applied computing, SAC '12. ACM, New York, pp 1215–1220

SciTools (2015) Understand 3.1 build 726. https://scitools.com, [Online; accessed 15-June-2015]

Shatnawi R, Li W (2008) The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. J Syst Softw 81(11):1868–1882

Sheskin DJ (2007) Handbook of parametric and nonparametric statistical procedures, 4th edn. Chapman & Hall/CRC

Shihab E, Jiang ZM, Ibrahim WM, Adams B, Hassan AE (2010) Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In: Proceedings of the 2010 ACM/IEEE international symposium on empirical software engineering and measurement, ESEM '10. ACM, New York, pp 4:1–4:10

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2nd international workshop on mining software repositories, MSR '05, pp 1–5

Tassey G (2002) The economic impacts of inadequate infrastructure for software testing. Tech. Rep. Planning Report 02-3, National Institute of Standards and Technology

Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empir Softw Eng 14(5):540–578

Watanabe S, Kaiya H, Kaijiri K (2008) Adapting a fault prediction model to allow inter languagereuse. In: Proceedings of the 4th international workshop on predictor models in software engineering, PROMISE '08. ACM, New York, pp 19–24

Yin RK (2002) Case study research: design and methods, 3rd edn. SAGE Publications

Zhang F, Mockus A, Zou Y, Khomh F, Hassan AE (2013) How does context affect the distribution of software maintainability metrics? In: Proceedings of the 29th IEEE international conference on software maintainability, ICSM '13, pp 350–359

Zhang F, Mockus A, Keivanloo I, Zou Y (2014) Towards building a universal defect prediction model. In: Proceedings of the 11th working conference on mining software repositories, MSR '14, pp 41–50

Zhou Y, Leung H (2007) Predicting object-oriented software maintainability using multivariate adaptive regression splines. J Syst Softw 80(8):1349–1361

Zimmermann T, Nagappan N (2008) Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th international conference on software engineering, ICSE '08. ACM, New York, pp 531–540

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Proceedings of the international workshop on predictor models in software engineering, PROMISE '07, p 9

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, pp 91–100

Zimmermann T, Nagappan N, Guo PJ, Murphy B (2012) Characterizing and predicting which bugs get reopened. In: 34th International Conference on Software Engineering (ICSE), 2012, pp 1074–1083

Feng Zhang received his B.S. degree in electronic information engineering from Nanjing University of Science and Technology (China) in 2004, and his M.S. degree in control science and engineering from Nanjing University of Science and Technology (China) in 2006. He is currently pursuing the Ph.D. degree in computer science from Queen's University in Canada. His research interests include empirical software engineering, software re-engineering, mining software repositories, and source code analysis.

Audris Mockus received the BS and MS degrees in applied mathematics from the Moscow Institute of Physics and Technology in 1988, the MS degree in 1991 and the PhD degree in statistics from Carnegie Mellon University in 1994. He studies software developers' culture and behavior through the recovery, documentation, and analysis of digital remains. These digital traces reflect projections of collective and individual activity. He reconstructs the reality from these projections by designing data mining methods to summarize and augment these digital traces, interactive visualization techniques to inspect, present, and control the behavior of teams and individuals, and statistical models and optimization techniques to understand the nature of individual and collective behavior. He is Harlan Mills Chair Professor of Digital Archeology in the Department of Electrical Engineering and Computer Science of the University of Tennessee. He also continues to work part-time at Avaya Labs Research. Previously he worked in the Software Production Research Department at Bell Labs. He is a member of the IEEE and ACM.

Iman Keivanloo is currently a Post-doctoral researcher in the Department of Electrical and Computer Engineering at Queens University, Canada. He received his MSc in Computer Science from Sharif University of Technology, Iran in 2008. He graduated with his PhD from Concordia University, Montreal, Canada in 2013. His research focuses on the area of source code similarity search and clone detection. He has published over 20 papers in major refereed international journals, conferences and workshops. Dr. Keivanloo has also served as a committee member on several international conferences and workshops in the area of clone detection and program comprehension.

Ying Zou is a Canada Research Chair in Software Evolution. She is an associate professor in the Department of Electrical and Computer Engineering, and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture.