Towards Building a Universal Defect Prediction Model

Feng Zhang
School of Computing
Queen's University
Kingston, Ontario, Canada
feng@cs.queensu.ca

Audris Mockus
Department of Software
Avaya Labs Research
Basking Ridge, NJ 07920, USA
audris@avaya.com

Iman Keivanloo
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
iman.keivanloo@queensu.ca

Ying Zou
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
ying.zou@queensu.ca

ABSTRACT

To predict files with defects, a suitable prediction model must be built for a software project from either itself (within-project) or other projects (cross-project). A universal defect prediction model that is built from the entire set of diverse projects would relieve the need for building models for an individual project. A universal model could also be interpreted as a basic relationship between software metrics and defects. However, the variations in the distribution of predictors pose a formidable obstacle to build a universal model. Such variations exist among projects with different context factors (*e.g.*, size and programming language). To overcome this challenge, we propose context-aware rank transformations for predictors. We cluster projects based on the similarity of the distribution of 26 predictors, and derive the rank transformations using quantiles of predictors for a cluster. We then fit the universal model on the transformed data of 1,398 open source projects hosted on SourceForge and GoogleCode. Adding context factors to the universal model improves the predictive power. The universal model obtains prediction performance comparable to the within-project models and yields similar results when applied on five external projects (one Apache and four Eclipse projects). These results suggest that a universal defect prediction model may be an achievable goal.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product metrics*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software maintenance*

General Terms

Algorithms, Experimentation, Measurement

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597078

Keywords

Universal defect prediction model, defect prediction, context factors, rank transformation, large scale, quality, defect, bug

1. INTRODUCTION

A defect causes software to behave improperly or produce unexpected results. Attempts to anticipate the parts of source code that may have fixes for defects have a rich history. For example, D'Ambros *et al.* [10] evaluate over 30 different approaches to build defect prediction models that were published from 1996 to 2010. Unfortunately, such models could not be generalized to apply on other projects or even new releases of the same project [39, 27]. Refitting such models is non-trivial. It requires collecting and tagging defects for each file, and collecting sufficient history which may not be available in small or new projects [26]. We refer to a single model that is built from the entire set of diverse 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 [19]. Moreover, it might allow a more direct comparison of defect rates across projects and 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. Zimmermann *et al.* [39] apply defect prediction models learnt from one project on another, with a low ratio of successful predictions (*i.e.*, 3.4%). Zimmermann *et al.* [39] consider a prediction to be successful if all precision, recall, and accuracy are greater than 0.75. One difficulty for building cross-project defect prediction models may be related to the variations in the distribution of predictors [27]. To overcome this challenge, we consider two approaches: 1) only use the data from projects with similar distribution to the target project (*e.g.*, [35, 28, 21]); and 2) transform predictors in both training and target projects to make them more similar in their distribution (*e.g.*, [27, 18]). However, the first approach uses partial dataset and results in multiple models. The transformation approaches are typically specialized to a particular pair of training and testing

datasets. Our prior study [37] 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 to the universal defect prediction model. We use 21 code metrics, five process metrics, and six context factors as predictors (*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). 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.*, [1]) which use quantiles to derive thresholds for ranking software quality, we apply every tenth quantile of predictors on each cluster to formulate ranking functions. After transformation, the predictors from different projects have exactly the same scales. The universal model is then built based on the transformed predictors.

We apply our approach on 1,398 open source projects hosted on SourceForge and GoogleCode. We mine defect data solely from the commit messages of fixes for defects, since many subject projects have no issue tracking systems. The result of F-measures and area under curve (AUC) using rank-transformed predictors is comparable to the result of logarithmic transformation. The performance of the universal model can be improved by adding context factors as predictors. The universal model yields better recall and higher AUC than within-project models, possibly due to the fact that the defects in the files of similar properties are fixed in one project but overlooked in another. Moreover, the universal model achieves up to 70% of the successful predictions of within-project models, using the loose criteria suggested by He *et al.* [12] for determining the success of predictions (*i.e.*, recall is above 0.70, and precision is greater than 0.50).

We examine the generalizability of the universal model by applying it on five external projects that are not hosted on SourceForge or GoogleCode (*i.e.*, one Apache project: Lucene, and four Eclipse projects: Eclipse, Equinox, Mylyn, and PDE). The results show that the universal model provides a similar performance (in terms of AUC) as within-project models for the five projects. In summary, the major contributions of our study are:

- **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.
- **Context factors as predictors of the universal model:** 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.*, AUC increases from 0.60 to 0.65 when comparing to the combination of code and process metrics).

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 four aspects of defect prediction: software metrics, data preprocessing, modelling techniques, and cross-project defect prediction.

2.1 Software Metrics

Software metrics are often used to evaluate software quality with proper thresholds and ranges of metric values [3]. For instance, McCabe [20] states that: the sub-functions are well structured if the value of his complexity metric is between 3 and 7; and sub-functions with the metric value beyond 10 are unmaintainable and untestable. 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 [23]), structural metrics [38], process metrics (*e.g.*, recent activities, number of changes, and the complexity of changes [11]), the number of previous defects [40], and social network metrics [4]. Arisholm *et al.* [2] find there exist large differences in terms of cost-effectiveness in defect prediction models among different metric sets (*e.g.*, process metrics significantly outperform structural metrics).

2.2 Data Preprocessing

The distribution of metric values sometimes varies significantly in projects of different contexts [37]. The varied scales of metrics are a challenge towards building a universal defect prediction model. Data preprocessing has been proved to improve the performance of defect prediction models by Menzies *et al.* [23]. Jiang *et al.*[14] evaluate the impact of log transformation and discretization on the performance of defect prediction models, and find different modelling techniques "prefer" different transformation techniques. For instance, Naive Bayes achieves better performance on discretized data, while logistic regression achieves better performance for both.

The state-of-the-art approaches to improve the performance of cross-project defect prediction mainly use two data preprocessing techniques: 1) only use the data from projects with similar distributions to the target project (*e.g.*, [35, 28, 21]); and 2) transform predictors in both training and target projects to make them more similar in their distribution (*e.g.*, [27, 18]). He *et al.* [12] propose to use the distributional characteristics (*e.g.*, median, mean, variance, standard deviation, skewness, and quantiles); Turhan *et al.* [35] and Peters *et al.* [28] propose different filters; and Li *et al.* [17] propose to use sampling. 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. The transformation approaches are typically specialized to a particular pair of training and testing datasets. For instance, Ma *et al.* [18] propose to weight training data by estimations on the distribution of testing data. Nam *et al.* [27] propose to transform both training and testing data to the same latent feature space, and build models on the latent feature space. Our previous study [37] finds that the distribution of metrics varies with project contexts. 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.

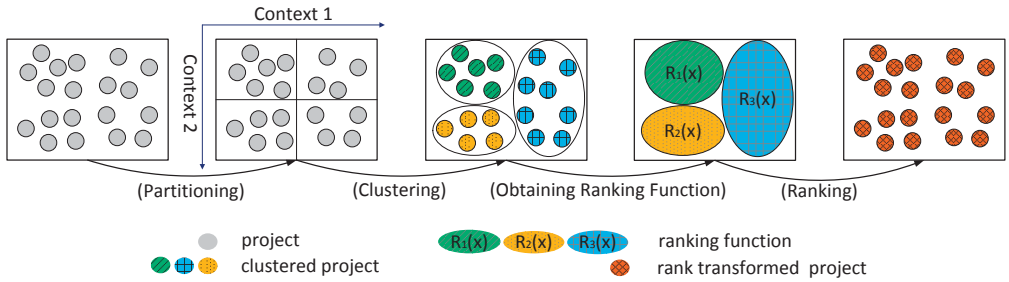

Figure 1: Our four-step rank transformation approach: 1) stratify the set of projects along different contexts into non-overlap groups; 2) cluster project groups; 3) derive ranking function for each cluster; and 4) perform rank transformation.

2.3 Modelling Techniques

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, K-nearest neighbour, and artificial neural networks). Lessmann *et al.* [16], Arisholm *et al.* [2], and D'Ambros *et al.* [10] propose different approaches to compare and evaluate different modelling techniques. Lessmann *et al.* [16] find that there are no significant differences in the performance among different modelling techniques. Arisholm *et al.* [2] also report that the choice of modelling techniques only has limited impact on the performance in terms of accuracy or cost-effectiveness. Our rank transformation approach is a step for data preprocessing, thus is independent from modelling techniques. Software organizations can choose the technique that best suits their needs.

2.4 Cross-Project Defect Prediction

To predict defects on the projects without sufficient training data, many researchers attempt to build cross-project defect prediction models. Zimmerman *et al.* [39] run cross-project predictions for 622 pairs of 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.* [35] observe that cross-project prediction not only underperforms within-project prediction, but also has excessive false alarms. Premraj and Herzig [29] confirm the big challenge in cross-project defect prediction through their replication study. However, Rahman *et al.* [30] 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). The challenge of cross-project prediction might be caused by the fact that metrics from different projects may have significantly different distributions [27]. Zimmerman *et al.* [39] and Menzies *et al.* [21] suggest to consider project contexts for cross-project defect prediction. We propose context-aware rank transformation for predictors, and find that adding context factors (see Section 3.1) can improve the predictive power for the universal defect prediction model.

3. APPROACH

It is very likely that predictors from different projects of various contexts exhibit different distribution [37]. To overcome this challenge towards building a universal defect prediction model, we propose a context-aware rank transformation approach, as illustrated in Figure 1. It consists of four steps:

1) Partition the entire set of projects to non-overlapped groups based on the six aforementioned context factors;
2) Cluster the project groups with the similar distribution of predictor values;
3) Derive a ranking function for each cluster using every $10th$ quantiles of predictor values, in order to address the large variations in the distribution of predictors;
4) Apply the ranking functions to convert the raw values of predictors to one of the ten levels.

The scales of the transformed predictors are exactly the same even from different projects. We then 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.

3.1 Context Factors

In this study, we choose six context factors based on their availability to open source projects and our previous work [37].
1) Programming Language (PL): describes the nature of programming paradigms. Due to the limitation of our metric computing tool, we only consider projects mainly written in C, C++, Java, C#, or Pascal. In this study, we divide the set of projects into five groups based on programming languages: G_c, G_{c++}, G_{java}, $G_{c\#}$, and G_{pascal}.
2) Issue Tracking (IT): describes whether a project uses an issue tracking system or not. The set of projects is separated into two groups based on the usage of an issue tracking system: G_{useIT} and G_{noIT}.
3) Total Lines of Code (TLOC): describes the project size in terms of source code. 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 four groups: $G_{leastTLOC}$, $G_{lessTLOC}$, $G_{moreTLOC}$, and $G_{mostTLOC}$.
4) Total Number of Files (TNF): describes the project size in terms of files. 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 four groups: $G_{leastTNF}$, $G_{lessTNF}$, $G_{moreTNF}$, and $G_{mostTNF}$.
5) Total Number of Commits (TNC): describes the project size in terms of commits. 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 four groups: $G_{leastTNC}$, $G_{lessTNC}$, $G_{moreTNC}$, and $G_{mostTNC}$.
6) Total Number of Developers (TND): describes the project size in terms of developers. 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 four groups: $G_{leastTND}$, $G_{lessTND}$, $G_{moreTND}$, and $G_{mostTND}$.

3.2 Partitioning Projects

We assume that projects with the same context factors have the 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. We get 5, 2, 4, 4, 4, and 4 groups, respectively. In total, we obtain 2560 (*i.e.*, $5 \times 2 \times 4 \times 4 \times 4 \times 4$) non-overlapped groups.

3.3 Clustering Similar Projects

To derive more accurate quantiles of a particular metric, we group the projects with the similar distribution of the metric. We consider two distributions are similar if their difference is neither statistically significant nor significantly large. The clusters may not be the same for different metrics. Therefore we yield different sets of clusters for each metric. Each cluster is described by a vector, *e.g.*, $< m$, $C++$, $useIT$, $moreTLOC >$. This cluster example is created for metric m, and contains $C++$ projects that use issue tracking systems, and has the TLOC between the second and third quartiles (see Section 3.1). For each metric m, the clusters of projects with the similar distribution of metric m are obtained using the Algorithm 1. It has two major steps.

1) Comparing the Distribution of Metrics. This step merges the groups of projects that do not have significantly different distribution of metric m. We apply Mann-Whitney U test [33] to compare the distribution of metric values between every two groups of projects, using the 5% 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 merges the groups of projects that have significantly different distributions of metric m, but the difference is not large. We calculate Cliff's δ as the effect size [32] 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 [32] to be more robust and reliable than Cohen's d [7]. Cliff's δ represents the degree of overlap between two sample distributions [32]. It ranges from -1 (if all selected values in the first group are larger than 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 [6]. To interpret the Cliff's δ, we map it to Cohen's standards (*i.e.*, small, medium, and large) using the percentage of non-overlap [32], as shown in Table 1. Cohen [8] 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 distribution differences.

3.4 Obtaining Ranking Functions

The ranking function transforms the raw metric values to relatively predefined values (*i.e.*, ranging from one to ten). The transformed metrics have exactly the same scales across projects. We use the quantiles of metric values to formulate our ranking functions. This is inspired by metric-based benchmarks (*e.g.*, [1]), which often use the quantiles to derive thresholds of metrics to distinguish files of different quality related to defects.

For metric m_i (*where* $i \in \{1, \ldots, M\}$, *and* M *is the number of metrics*), we denote the corresponding clusters as 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 functions for metric m_i following Equation (1).

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

where $R(m_i, C_{ij})$ *is the ranking function for metric* m_i *in cluster* C_{ij}, $V(m_i)$ *is the value of metric* m_i *to be converted,* $Q_{ij,k}(m_i)$ *is the* $k*10th$ *quantile of metric* m_i *in the 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. We then convert the value 27 of metric

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 δ is large then
   |  |  |  |  /* Put group i and j in different
   |  |  |  |     clusters.                       */
12 |  |  |  |  if clusterOfGroup[j] equals to
   |  |  |  |  clusterOfGroup[i] then
13 |  |  |  |  |  indexOfCluster = indexNewCluster;
14 |  |  |  |  |  clusterOfGroup[j] = indexOfCluster;
15 |  |  |  |  end
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

Table 1: Mapping Cliff's δ with Cohen's standards.

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

m_1 to 3 if the corresponding project belongs to cluster Cl_{12}. This is because the value 27 is greater than 22 (*i.e.*, the 20% quantile) and less than 33 (*i.e.*, the 30% quantile).

3.5 Building a Universal Defect Prediction Model

Choice of modelling techniques. As described in Section 2.3, there is no significant difference among different modelling techniques in the performance of defect prediction models [16, 2]. However, Kim *et al.* [15] 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 noises in defect data. Based on their findings, we apply Naive Bayes as the modelling technique in our experiments.

Steps to build the universal defect prediction model. First, we transform the raw values of each metric using Equation (1). Before transforming a metric m_i for project p_j, we identify context factors of project p_j and formulate a vector like $< m_i$, $C{+}{+}$, $useIT$, $moreTLOC$, $lessTNF$, $lessTNC$, $lessTND >$. In order to locate the ranking functions (see Section 3.4), we compare the vector of project p_j to the vectors of all clusters to determine which cluster project p_j belongs to. We apply the ranking functions of the identified cluster to transform the raw metric values of each file in project p_j to one of the ten levels. As a result, the transformed metrics have the scales ranging from one to ten. A universal defect prediction model is then built upon the entire set of projects using Weka[1] tool.

3.6 Measuring the performance

To evaluate the performance of prediction models, the confusion matrix (see Table 2) is computed. Using the confusion matrix, we calculate the following five measures: precision, recall, false positive rate, F-measure, and g-measure.

Precision ($prec$). Precision measures the proportion of actual defective entities that are predicted as defective against all predicted defective entities. It is defined as: $prec = \frac{TP}{TP+FP}$.

Recall (pd). Recall evaluates the proportion of actual defective entities that are predicted as defective against all actual defective entities. It is defined as: $pd = \frac{TP}{TP+FN}$.

False Positive Rate (fpr). False positive rate is the proportion of actual non-defective entities that are predicted as defective against all actual non-defective entities. It is defined as: $fpr = \frac{FP}{FP+TN}$.

F-measure. F-measure calculates the harmonic mean of precision and recall. It balances precision and recall. It is defined as: $F\text{-}measure = \frac{2\times pd\times prec}{pd+prec}$.

g-measure. 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.* [28], since Menzies *et al.* [22] show that precision can be unstable when datasets contain a low percentage of defects. It is defined as: $g\text{-}measure = \frac{2\times pd\times(1-fpr)}{pd+(1-fpr)}$.

Area Under Curve (AUC). 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. Rahman *et al.* [30] recommend to use AUC for cross-project defect prediction instead of traditional measures such as precision, and recall).

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

Table 2: 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)

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 [24] with his updates until October 2010. The dataset contains about 154K projects that are hosted on SourceForge and 81K projects that are hosted on GoogleCode. However, there are too many trivial projects. Many projects do not have enough history and defect data for evaluation. Hence, we clean the dataset for our experiments.

4.1.2 Cleaning the Dataset

Filtering out projects by programming languages. In this study, we use a commercial tool, called *Understand*[2], 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 thresholds 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 six months' period [40] 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 6 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 thresholds 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 com-

[2]http://www.scitools.com

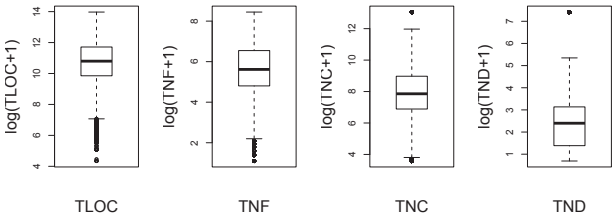

Figure 2: Boxplot of four numeric context factors (*i.e.*, TLOC, TNF, TNC, and TND) in our dataset.

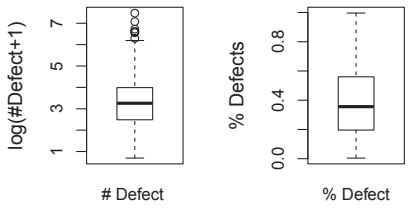

Figure 3: Boxplot of the number of defects and the percentage of defects in our dataset.

mits 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.* [28] 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.

Description of the final experiment dataset. In the cleaned dataset, there are 937 SourceForge projects, and 461 GoogleCode projects. Among them, 715 projects employ CVS as their version control system, 621 projects use Subversion, and 62 projects adopt Mercurial. The number of projects that are mainly written in C, C++, C#, Java, and Pascal are 288, 424, 84, 591, and 11, respectively. There are 817 projects using issue tracking systems, and 581 projects without using any issue tracking system. We show the boxplot of other four context factors in Figure 2.

4.2 Software Metrics

As shown in Table 3, this study covers 21 code metrics, and five process metrics that are often used in defect prediction models. The code metrics are computed by the tool called *Understand*. The process metrics are computed by our scripts. As mentioned in Section 4.1.2, we look 6 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 of 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 [40]. In our dataset, 42% of subject projects do not use issue tracking systems. For such projects, we mine defect data solely by tagging keywords of commit messages. A similar tagging method for mining defect data is used by Mockus and Votta [25] and in SZZ algorithm [34]. 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 [5, 15, 13]. However, Rahman *et al.* [31] report that increasing the sample size can leverage the possible bias in defect data. Our dataset contains 1,398 subject projects, and is around 140 to 280 times larger than most papers in this field [28]. In addition, the modelling technique (*i.e.*, Naive Bayes) used in this study is proved by Kim *et al.* [15] to have strong noise resistance with up to 20%-35% of false positive and false negative noises in defect data. The defect data is collected in the six months' period after the split date. We show the boxplot of the number of defects and the percentage of defects in our dataset in Figure 3.

5. CASE STUDY RESULTS

5.1 Project Clusters

In our dataset, there are 1,398 open source projects. The set of projects is stratified into non-overlapped groups along the six context factors: programming language, issue tracking, total lines of code, total number of files, total number of commits, and total number of developers, respectively. In total, we obtain 480 non-empty groups. For each metric, we perform $\binom{480}{2} = \frac{480!}{2!\times 478!} = 114,960$ 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 $0.05/114,960 = 4.35e\text{-}07$. 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 25, which is the number of clusters obtained for the metric total_CBO (*i.e.*, the sum of CBO values 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. The log transformation uses the logarithm of raw values, and has been proved to improve the predictive power in defect prediction approaches [23, 14]. Therefore, we compare the performance of defect prediction models built using rank transformations to the models built using log transformations.

Approach. We build two types of within-project defect prediction models for every project, using log transformations and rank transformations, respectively. 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 in the performance of defect prediction models built using log and rank transformations.*

Table 3: List of software metrics.

Type	Metric Level	Metric Name	Description	File Level
Code Metrics	File	LOC	Lines of Code	value
		CL	Comment Lines	value
		NSTMT	Number of Statements	value
		NFUNC	Number of Functions	value
		RCC	Ratio Comments to Codes	value
		MNL	Max Nesting Level	value
	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	value
		NFIX	Number of revisions a file was involved in bug-fixing	value
		ADDEDLOC	Lines added	avg, max, total
		DELETEDLOC	Lines deleted	avg, max, total
		MODIFIEDLOC	Lines modified	avg, max, total

Table 4: The results of Wilcoxon rank sum tests and mean values of six performance measures of log transformation (LogTran) and our context-aware rank transformation (RankTran) in the within-project settings. (* denotes statistical significance.)

Measures	LogTran	RankTran	p-value	Cohen's d
prec	**0.48**	**0.48**	0.71	-0.01
pd	0.57	**0.58**	0.31	0.03
fpr	0.36	**0.35**	0.43	0.04
F-measure	0.49	**0.50**	0.33	-0.03
g-measure	0.53	**0.54**	4.7e-03	-0.07*
AUC	0.61	**0.62**	0.14	-0.03

We conduct Wilcoxon rank sum test [33] to compare the six performance measures, using the 5% 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.* [39]; 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.* [12].

Findings. Table 4 presents the mean values of the six performance measures of both log and rank transformations, and the corresponding p-values of Wilcoxon rank sum test. We can not reject the hypothesis $H0_1$ that there is no difference between rank transformation and log transformation in within-project defect prediction in most measures, except g-measure. We further compute Cohen's d [7] as effect size to measure the mean difference. As shown in Table 4, the results show the difference between the two transformations is small (*i.e.*, less than 0.10). The proportion of successful predictions for both approaches is identical where it is 9% and 20% using the strict and loose criteria for successful prediction, respectively. 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.

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

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.* [21] report the 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. 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?*

Table 5: The performance measures for the universal models built using code metrics (CM), code + process metrics (CPM), and code + process + contexts (CPMC), respectively.

Measures	CM	CPM	CPMC
prec	0.36	0.38	**0.40**
pd	**0.91**	0.83	0.86
fpr	0.87	0.76	**0.70**
F-measure	0.51	0.51	**0.55**
g-measure	0.23	0.36	**0.42**
AUC	0.58	0.60	**0.65**

Table 6: The results for Wilcoxon rank sum tests and mean values of six performance measures of within-project models (WM) and the universal models (UM). (* denotes statistical significance.)

Measures	WM	UM	p-value	Cohen's d
prec	**0.48**	0.45	2.34e-03	0.10*
pd	0.58	**0.63**	2.81e-11	-0.25*
fpr	**0.35**	0.45	6.76e-42	-0.48*
F-measure	**0.50**	0.46	2.19e-05	0.15*
g-measure	**0.54**	0.52	7.58e-14	0.14*
AUC	0.62	**0.64**	3.33e-04	-0.16*

Approach. To address RQ2.1, we start to build the universal model using only code metrics, then adding process metrics, and finally including context factors. This provides the insights of the improvements on the performance of the universal model by adding context factors as predictors.

To address RQ2.2, we build a universal model using the set of projects except the testing projects and build within-project model for each project. We conduct 10-fold cross-validation to obtain the average predictive power of both the universal model and the within-project models. We apply Wilcoxon rank sum test (5% confidence level) to examine the following null hypothesis for each performance measure:

$H0_2$: *there is no difference in the performance of within-project and universal defect prediction models.*

We compute the proportion of acceptable predictions of both the universal model and the within-project models.

Findings. (RQ2.1) Table 5 provides a summary of the comparison steps. It shows that the AUC value keeps increasing when adding more metric sets to the model. The context factors increase the precision, recall, F-measure, g-measure, and the AUC value. Hence, the context factors are good predictors for building a universal defect prediction model.

(RQ2.2) Table 6 presents the Wilcox rank sum test results of six measures between within-project model and universal models built using rank transformations. We reject the null hypothesis $H0_2$ for all measures. The results show that the within-project model has 3% higher precision, 4% higher F-measure, and 2% higher g-measure. The universal model has 5% higher recall and 2% higher AUC, but experiences 10% higher false positive rate. The Cohen's d reports the difference among these measures are medium (*i.e.*, greater than 0.10, but less than 0.30), except g-measure. The higher recall and lower precision of universal models might be due to the fact that the defects of files with similar properties are fixed in one project but overlooked in another project. Nevertheless, the results show that the universal model can slightly outperform the within-project prediction models in the context of cross-project prediction by comparing the AUC values as Rahman *et al.* [30] show that AUC is a more reliable measure than precision and recall for cross-project prediction.

Moreover, the universal models yield similar percentage (*i.e.*, 3%) of successful predictions (see **RQ1**) as Zimmerman *et al.* [39] who report a 3.4% success rate. If using loose criteria, the universal model achieves 14% of successful predictions, much higher than He *et al.* [12] who report 0.32% of successful predictions. The universal model achieves up to 70% (*i.e.*, 14% against 20%) of the successful predictions by within-project model. We conclude that our approach for building a universal model is promising.

Table 7: The six performance measures for within-project model (wm) and the universal model (um). P1 is Eclipse, P2 is Equinox, P3 is PDE, P4 is Mylyn, P5 is Lucene.

Measures	P1	P2	P3	P4	P5	Type
prec	**0.47**	0.63	**0.28**	**0.28**	**0.21**	wm
	0.31	**0.66**	0.23	0.23	0.13	um
pd	0.57	**0.61**	0.47	0.42	0.34	wm
	0.79	0.54	**0.72**	**0.60**	**0.61**	um
fpr	0.17	**0.24**	0.20	0.17	0.13	wm
	0.46	0.19	**0.39**	**0.30**	**0.42**	um
F-measure	**0.52**	**0.62**	**0.35**	**0.34**	**0.26**	wm
	0.45	0.59	**0.35**	0.33	0.21	um
g-measure	**0.68**	**0.68**	0.59	0.56	0.49	wm
	0.64	0.65	**0.66**	**0.64**	**0.60**	um
AUC	0.76	0.78	**0.70**	0.68	**0.69**	wm
	0.77	**0.79**	**0.70**	**0.69**	0.67	um

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 this question, we choose to use the publicly available dataset[3] that was collected by D'Ambros *et al.* [9]. 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 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 1,398 SourceForge and GoogleCode projects. The six performance measures of within-project models are obtained via 10-folds cross-validation for each project.

Findings. Table 7 presents the mean of six performance measures of the universal model and within-project models. Similar to **RQ2**, the universal model achieves higher recall and better AUC values but has a higher false positive rate.

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

The results show 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.

6. THREATS TO VALIDITY

We now discuss the threats to validity of our study following common guidelines provided in [36].

Threats to conclusion validity concern the relation between the treatment and the outcome. Our conclusion validity threats are mainly due to 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.

Threats to internal validity concern our selection of subject systems and analysis methods. SourceForge and GoogleCode are considered to have a large proportion of not well managed projects. We believe our data cleaning step increases the data quality. The other threats to internal validity is possible biases 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[4].

7. CONCLUSION

In this study, we attempt to build a universal defect prediction model for a large set of projects from various contexts. We first propose a context-aware rank transformation method to pre-process the predictors. The transformed predictors have the same scales. We then build a universal model using the metrics after rank transformation, and find that the rank transformation performs as good as log transformation. By adding different metric sets (*i.e.*, code metrics, process metrics, and context factors) step by step, we find that the context factors increase the predictive power of the universal model. We further find that the universal model has higher AUC values and higher recall than within-project models. In the context of cross-project prediction, our approach is better based on the AUC measure. We also evaluate the generalizability of the universal model by evaluating its performance using five external projects that are not hosted on SourceForge and GoogleCode. The findings remain the same. 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.

[4]http://fengzhang.bitbucket.org/replications/unimodel.html

8. 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.

9. REFERENCES

[1] T. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1 –10, sept. 2010.

[2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[3] R. Baggen, J. Correia, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20:287–307, 2012.

[4] N. Bettenburg and A. E. Hassan. Studying the impact of social structures on software quality. In *Proceedings of the 18th IEEE International Conference on Program Comprehension*, pages 124–133, 2010.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the 12th European Software Engineering Conference and the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130, 2009.

[6] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, Nov. 1993.

[7] J. Cohen. *Statistical power analysis for the behavioral sciences : Jacob Cohen.* Lawrence Erlbaum, 2 edition, Jan. 1988.

[8] J. Cohen. A power primer. *Psychological Bulletin*, 112(1):155–159, 1992.

[9] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 31 – 41, 2010.

[10] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, Aug. 2012.

[11] A. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, pages 78 –88, 2009.

[12] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, June 2012.

[13] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction.

In *Proceedings of the 35th International Conference on Software Engineering*, pages 392–401, 2013.

[14] Y. Jiang, B. Cukic, and T. Menzies. 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, pages 16–20, 2008.

[15] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 481–490, 2011.

[16] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[17] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, June 2012.

[18] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, Mar. 2012.

[19] C. Mair and M. Shepperd. 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*, pages 10 pp.–, 2005.

[20] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308 – 320, Dec. 1976.

[21] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 343–351, 2011.

[22] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'". *IEEE Transactions on Software Engineering*, 33(9):637–640, 2007.

[23] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[24] A. Mockus. 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*, pages 11 –20, may 2009.

[25] A. Mockus and L. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 120–130, 2000.

[26] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, pages 452–461, 2006.

[27] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 35th International Conference on Software Engineering*, pages 382–391, 2013.

[28] F. Peters, T. Menzies, and A. Marcus. Better cross company defect prediction. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 409–418, 2013.

[29] R. Premraj and K. Herzig. Network versus code metrics to predict defects: A replication study. In *2011 International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 215–224, 2011.

[30] F. Rahman, D. Posnett, and P. Devanbu. 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, pages 61:1–61:11, 2012.

[31] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu. Sample size vs. bias in defect prediction. In *Proceedings of the 15th European Software Engineering Conference and the 21th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '13, 2013.

[32] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. 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*, pages 1–33, February 2006.

[33] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.

[34] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[35] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, Oct. 2009.

[36] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, 3 edition, 2002.

[37] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan. How does context affect the distribution of software maintainability metrics? In *Proceedings of the 29th IEEE International Conference on Software Maintainability*, pages 350 – 359, 2013.

[38] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540, 2008.

[39] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 12th European Software Engineering Conference and the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, 2009.

[40] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–15, 2007.