Class Level Fault Prediction Using Software Clustering

Giuseppe Scanniello  
University of Basilicata, Italy  
giuseppe.scanniello@unibas.it

Carmine Gravino  
University of Salerno, Italy  
gravino@unisa.it

Andrian Marcus  
Wayne State University, USA  
amarcus@wayne.edu

Tim Menzies  
West Virginia University, USA  
tim@mnesies.us

Abstract—Defect prediction approaches use software metrics and fault data to learn which software properties associate with faults in classes. Existing techniques predict fault-prone classes in the same release (intra) or in a subsequent releases (inter) of a subject software system. We propose an intra-release fault prediction technique, which learns from clusters of related classes, rather than from the entire system. Classes are clustered using structural information and fault prediction models are built using the properties of the classes in each cluster. We present an empirical investigation on data from 29 releases of eight open source software systems from the PROMISE repository, with predictors built using multivariate linear regression. The results indicate that the prediction models built on clusters outperform those built on all the classes of the system.

Keywords - Empirical Study; Fault Prediction, Software Clustering

I. INTRODUCTION

Defect (or fault) prediction in software has experienced a surge in interest from researchers during the past few years [1]. Most fault predictors use knowledge from previous projects and products to predict fault-prone software entities in the current or future project [1]. While some studies (e.g., [2]) have suggested that fault predictors built on process metrics could outperform those built on product metrics, the research community still focuses heavily on using product metrics, as they are much easier to obtain.

Recent work ([3]–[5]) showed that learning from software components with similar characteristics is better than learning from entire systems, that is, it results in better defect prediction. The lesson here is that the best way to predict faults in a class is to learn from classes that have similar properties with it. Other work ([6], [7]) proposed the use of components or packages to group software metrics and to predict their faults. The conjecture is that the likelihood to fail of a component or of a package is dependent on its problem domain [6]. Furthermore, the results of an empirical study on open source software [8] suggested that the fault distribution within components is different than that over classes and might depend on the problem domain of the subject system. Unfortunately, components are not always documented.

We combine these two lessons and conjecture that the best way to predict faults in a class is to learn from the classes related to it. Based on our conjecture, we propose a defect prediction technique based on software clustering. The defect predictor models are built based on the classes within each cluster. Specifically, we propose clustering the software by using structural information (i.e., static dependencies among classes). This information is orthogonal to the code attributes used for building the predictor models and it is meant to produce clusters that group together classes that are related via dependencies. Reasoning about such groups of associated classes makes sense as they are likely implementing a small set of related features. We conjecture that classes from such clusters are investigated together by developers, for example, when changing software to add features or to fix faults.

Our approach is different from previous defect prediction work that is based on grouping together classes and data. With respect to the work mentioned above ([6], [7]) on using component or package level data for defect prediction, we perform defect prediction at class level, akin to most defect predictors. Given the granularity difference, comparison with these techniques is not feasible. With respect to the work on clustering to support defect prediction ([3]–[5]), we group together related classes, rather than classes with similar properties. Once again, direct comparison is not appropriate as these other techniques rely on learning data from multiple systems.

We present our approach and an empirical evaluation on 29 releases of eight open source software systems from the PROMISE repository [9]. For each release, we built predictors for faulty classes at intra-release level (i.e., learning from the same release on which prediction is made). We compared the models built at the cluster level with a baseline (i.e., the fault prediction model built at class level on the entire system). The predictor at class level is built in the same way as in each cluster using step wise linear regression. The validation of these predictors is performed through the leave-1-out cross validation method.

II. RELATED WORK

Fault prediction is a very active research field [1] and many studies have addressed this issue using a variety of different methods and techniques [10]. We restrict our discussion to fault prediction approaches that work with groups of classes, such as, packages, components, modules, or cross-system clusters. For example, Nagappan et al. [11] predict the likelihood of post-release faults at the module-level (i.e., binary file within Windows) using a regression model and principal component analysis. The results of the investigation
suggest that the chosen metrics can be used to successfully predict post-release defects. However, the authors also observe that there is no single set of metrics applicable to all projects. In addition, predictors are likely to be accurate for similar software projects.

Schroter et al. [6] present a study in which models are built to predict failure-prone components in new programs. The study is conducted on 52 Eclipse plug-ins. The results indicated that the software design artifacts, as well as past documented faults, can be successfully used in prediction.

Zimmermann et al. [7] conduct a study to assess complexity metrics for fault prediction in a dataset on three Eclipse releases (i.e., 2.0, 2.1, and 3.0). The results show that the combination of complexity metrics can predict defects, suggesting that the more complex code it, the more defects it has.

In contrast with the above approaches, Kamei et al. [12] evaluate the effect of prediction models on the costs of software quality assurance activities. The authors considered both process and software metrics at package level. The main results are: (i) process metrics still outperform software metrics at the file-level and (ii) package-level predictions are less effective than those at file-level.

Tan et al. [13] propose the use of both Latent Semantic Indexing (LSI) and a hierarchical clustering algorithm to group classes that are similar at the lexical level. Differently from our approach and those presented above, the authors present models to predict faulty clusters. The results suggest that the predictive models build on the clusters outperform those based on classes in terms of precision, recall, and accuracy of the faults predicted.

Our proposal borrows from these prior works the idea that defect prediction can be made at component level (which usually groups related classes). Where our work differs is that we perform learning on groups of related classes, but we predict defects at class level (as opposed to component level).

III. CLUSTERING APPROACH

The clustering process consists of the following steps: extracting dependencies among classes and clustering. In the following, we describe how we instantiated each step.

A. Extracting dependencies among classes

A software system is represented as an undirected graph \( G = (V, E) \), where \( V \) represents the classes in the system, while \( E \) is the set of edges (i.e., ordered pair of nodes of \( V \)). There is an edge \( e(c_i, c_j) \) if a static dependency between two methods and/or fields in the classes \( c_i \) and \( c_j \) is present. Depending on the availability of static analysis tools, more or less complex structural relationships could be extracted: direct method calls, common attribute reference, inheritance, etc. In this work, we take a conservative approach and only consider references between methods and attributes. In other words, \( e(c_i, c_j) \in E \) if there is a reference to \( c_j \) (i.e., class instantiation, method invocation, or field access) in the body of the class \( c_i \) or vice versa. To identify static references in Java code, we use JRipples[14]. Including additional relationships in the program dependence graph is subject of future work.

B. Clustering

To group together similar classes, we use the BorderFlow algorithm [15] on graph \( G \). This algorithm is a general-purpose graph-clustering algorithm, originally conceived for achieving a soft clustering of the input graph (i.e., a node can be in one or more clusters), but it can be also used for hard clustering (i.e., a node can be in exactly one cluster). The use of the hard variant on the same sparse graph may produce different clustering results [16]. Conversely, on dense and unweighted graphs the hard variant of the BorderFlow clustering algorithm is deterministic (this is our case).

We chose this algorithm because we have successfully applied it in the past in the context of concept location in source code [16]. We also choose BorderFlow because it does not need to specify the number of clusters to be found: clustering is automatically performed without human intervention. The use of different clustering algorithms is subject of future work.

In BorderFlow a cluster is defined as collection of nodes, such that each cluster has more links inside than links to the outside. Further on, a cluster has a set of nodes such that the flow is maximal in the cluster and the flow from the cluster to the other ones is minimal. The idea behind BorderFlow is to maximize the flow from the border of each cluster to the nodes within the cluster, while minimizing the flow from the cluster to the nodes outside. In this work, we reformulate that idea as follows: maximizing the number of dependencies from the border of each cluster to the nodes within the cluster, while minimizing the number of dependencies from the cluster to the nodes outside the cluster. We are interested in finding groups of strongly connected classes, which are likely to implement a set of related features.

A cluster \( X \) is a subset of \( V \) such that a cluster maximizes the border flow ratio:

\[
F(X) = \frac{\Omega(b(X), X)}{\Omega(b(X), n(X))}
\]

where \( b(X) \) is the set of border nodes of \( X \), while \( n(X) \) is a function used to identify the set of direct neighbors of \( X \). \( \Omega \) is a function that assigns the total number of the edges (i.e., dependencies) from a subset of \( V \) to another one to these subsets (i.e., the dependence between the first and the second subset). This function is computed as follows:

\[
\Omega(X, Y) = \sum_{c_i \in X \text{ and } c_j \in Y} e(c_i, c_j)
\]

The algorithm iteratively selects nodes from \( n(X) \) and then inserts them in \( X \) until \( F(X) \) is maximized. The selection of the nodes is performed according to the following two steps:

1) Computing the set \( C(X) \) that will contain all the nodes \( u \in V \setminus X \) such that \( F(X \cup \{u\}) > F(X) \).

2) Selecting the candidates \( u \in C(X) \) to get the set \( C'(X) \). This set contains all the nodes \( u \) that maximize \( \Omega(u, n(X)) \).

[http://jripples.sourceforge.net]
If $F(X \cup C_f(X)) \geq F(X)$, then the nodes of $C_f(X)$ are added to the set $X$. The iterative selection of nodes concludes when $|n(X)|$ equals to 0 for each set of nodes $X$ identified by the BorderFlow algorithm. Each set of nodes forms a cluster. The algorithm may produce very small clusters. Since we plan to build a prediction model using the classes from each cluster, too small clusters are not ideal. To address this concern, we introduced the following heuristic: all the clusters with size less than $\sqrt{n}$, where $n$ is the total number of the classes in the system, are merged in a single cluster (which we will call the joined cluster from here on).

IV. EMPIRICAL INVESTIGATION

In this section, we present the design underlying our empirical investigation. To conduct our empirical evaluation, we implemented a prototype of a supporting system based on the above instantiation of our technique. This prototype has been developed as an Eclipse plug-in. For replication purposes, a release of this plug-in and the experimental data are available on the web.

A. Definition and Context

Defect prediction approaches use software metrics and fault data to predict defects in source code. Our goal is to determine whether fault prediction in classes improves when building prediction models on a cluster. We build two kinds of predictors: (1) we use data from a cluster to train the model and to predict faults for the classes in the clusters; (2) we build the model using data from all the classes in the system (this is the baseline approach). In both cases we built the prediction models using linear regression. We chose this technique because it is widely adopted in defect prediction literature (e.g., [17]–[19]).

Using the Goal Question Metrics (GQM) template [20], the goal of our study can be defined as follows: analyze the used software clustering (using the BorderFlow algorithm), for the purpose of evaluating fault prediction in OO classes, with respect to the quality of the predictions, from the point of view of the researcher and the professional developer, in the context of object-oriented open source software systems.

Accordingly, we have formulated and investigated the following research question:

RQ: Does the new clustering-based approach improve fault prediction results compared with the baseline?

B. Context

The empirical investigation has been conducted on 29 releases of the following eight open source software systems implemented in Java: Ant (releases from 1.3 to 1.7), JEdit (releases from 3.2.1 to 4.3), Lucene (releases from 2.2 to 2.4), POI (releases from 1.5 to 3.0), Synapse (releases from 1.0 to 1.2), Velocity (releases from 1.4 to 1.6.1), Xalan (releases from 2.4 to 2.7), and Xerces (releases 1.2 and 1.3).

The defect data have been downloaded from the popularity dataset (available at promisedata.googlecode.com) of the PROMISE repository [9].

C. Planning

Our empirical evaluation uses past changes to perform intra-release fault prediction. Past changes in software provide us with a number of faults (if present) in the classes of a given release. A software engineer can use the number of faults obtained from the past changes to predict faults in a class of a given release (i.e., the baseline) or to predict faults in a class of a cluster for that release. To verify that the prediction is correct, a subset of the documented faults can be used (i.e., test set). This subset is different from the one used to build the predictive model (i.e., training set). If a predictive model returns the number of faults present in the test set, then we can conclude that the prediction succeeded.

Many predictive approaches (included the one presented in this paper) depend on past information about software products or artifacts. In consequence, this information might affect the model built and the overall quality of its prediction. For example, if a given system contains a low number of documented faults, then we can conclude that: (i) the previous release of that system has been largely tested and minor changes have been performed to get the considered release; (ii) the system has been adequately tested before its deployment; or (iii) the lifetime of a software release was too short and then a low number of faults was detected during its normal use. While developing a predictive model this information may be not available, which would affect the validity of predictions.

D. Selected Variables

The dependent variable is ClassFault. It denotes the number of faults in the classes. On the other hand, we chose as independent variables the eight software metrics: WMC (Weighted Methods per Class), DIT (Depth Inheritance Tree), NOC (Number Of Children), CBO (Coupling Between Object classes), RFC (Response For Class), LCOM (Lack of Cohesion in Methods), NPM (Number of Public Methods), and LOC (Lines Of Code). These metrics are widely employed for fault prediction (e.g., [24]).

E. Execution and Data Analysis

1) Estimation Techniques: In our investigation, we employed the StepWise Linear Regression (SWLR) technique. SWLR allows computing linear regression in stages. We opted for linear regression because it has been widely used in the context of effort and fault prediction with good results (e.g., [17], [18], [23]). In addition, a recent extensive literature review, in the context of fault prediction [24], reports that supposedly better learners are not giving us better defect models. We conjecture that the research has expended all the benefit that can be gained from more elaborate learning and future progress will be achieved by exploring the structure of the data. This is why we exploited linear regression and
structural relationships among source code classes to improve accuracy of class level fault prediction.

SWLR allows computing an equation in stages in which the choice of the independent variables is carried out by an automatic procedure. These variables can be chosen applying three approaches: forward, backward, or a combination of both. The forward approach starts with no variables in the model, trying out the variables one by one and including them in the model if they are statistically significantly correlated with the dependent variable. The backward approach starts with all the variables and testing them one by one. The variables that are not statistically significant correlated with the dependent variable are removed from the model. We used here a combination of the forward and backward approaches. At each step, this combined approach includes or removes variables one by one in the prediction model if they are or are not statistically significant correlated with the dependent variable, respectively.

2) Prediction Validation: To assess the quality of the predictions of the models obtained with the SWLR, we performed a k-fold cross validation. The k-fold cross validation is widely used to assess how the results of a statistical analysis can be generalized to an independent data set. In particular, when the goal is the prediction, the k-fold cross validation is used to estimate how accurately a predictive model will perform in practice. The validation goes through k rounds. Each round of the validation involves the splitting the original dataset into training and test sets. The training set is used to build the fault prediction model, while the test set is exploited to validate the model. The results are averaged over the rounds. In the data analysis, we used a leave-one-out cross validation (i.e., $k = n$ where $n$ is the size of the dataset). The original dataset is divided into $n$ different subsets of training and test sets, with each test set containing one observation.

It could be possible that SWLR fails to build fault prediction models. This happens when lessons formed from the classes in a cluster are few (e.g., faults are not present or are very few in these classes). SWLR fails to build prediction models also when the number of classes is low. When prediction models are not built, the estimated faults in the test set are obtained by considering the mean of the faults in the training set.

To evaluate the quality of the fault predictions achieved with SWLR, we computed: sum (S), median (Md), mean (M), and standard deviation (StD) of Absolute Residuals (AR). Given the predicted and actual values $a$ and $p$, then the absolute residual is equal to $|p - a|$. This is a widely used performance measure. The smaller the value, the better the prediction of the faults is.

To compare our new approach and the baseline, we computed: $error = \frac{MAR(Clustering) - MAR(Flat)}{StDAR(Flat)}$. $MAR(Clustering)$ is the median of the mean values of the absolute residuals computed for the models built at cluster level. $MAR(Flat)$ and $StDAR(Flat)$ are the mean and the standard deviation values of the absolute residuals obtained from the fault prediction model built on the entire system. The variable $error$ assumes values in between $-1$ and $+1$. Given a software release, a negative value indicates that our approach outperforms the baseline, while a positive value indicates that the baseline is better. The absolute value of $error$ shows the magnitude of how much better or worse is our approach compared to the baseline. If $error$ is equal to 0 the two approaches are equivalent: the prediction error is the same. Formally, this is an effect size test and we use it since it has been found that, while treatments on software engineering data may produce statistically significant different results, the effect size of that difference is too small to be interesting.

We tested the following null hypothesis:

$H_{n0}$. The errors made with our approach are not statistically less than to those made with the baseline.

To test $H_{n0}$, we apply a non-parametric alternative to the unpaired t-test: the Wilcoxon rank-sum test, also known as the Mann-Whitney test. For all the tests performed, we decided to accept a probability of 5% (i.e., $\alpha = 0.05$) of committing a Type-I-Error.

The statistical tests used to analyze the presence of a significant difference do not provide any information about the magnitude of such a difference. To measure the magnitude of such a difference, the point-biserial correlation $r$ is used. We also compute the statistical power.

V. RESULTS

The descriptive statistics of the absolute residuals are summarized in Table I. In this table, we also report the values for $error$ and the number of classes and faults for each release. We can note that our approach outperforms the baseline in terms of accuracy of predicted faults in all the releases of the following systems: Ant, Lucene, Synapse, and Xalan. As for JEdit and POI, the baseline outperforms the approach proposed here only on a single release, while on all the other releases of these systems our approach achieves better prediction accuracy. As for Velocity, the baseline produced more accurate predictions in 2 out of 3 releases. There is no difference in the accuracy of the predictions in both the releases of Xerces, i.e., $error = 0$. For Xerces 1.3 this result was expected since our approach identified a single cluster containing all the classes.

The results of the Mann-Whitney test indicated that $H_{n0}$ could be rejected ($p$-value < 0.001): our approach made fewer errors in prediction that the chosen baseline. The effect size is large since the value of the point-biserial $r$ is 0.598 and a high statistical power was obtained (0.993).

VI. DISCUSSION

The achieved results suggest that program dependencies among source code classes improve fault prediction when

---

3The statistical power is the probability that a test will reject a null hypothesis when it is actually false. Researchers assess the power of a test using 0.80 as the standard for adequacy.

4We could not apply parametric analyses because the data were not normally distributed for the positive $error$ values as the results of the Shapiro-Wilk test show ($p$-value = 0.037).

5The effect size can be classified as: small ($0 < r < 0.193$), medium ($0.193 < r < 0.456$), and large ($0.456 < r < 0.868$).
using SWLR. That is, grouping classes by exploiting static dependencies among them and using a well established technique for fault prediction (e.g., \cite{18, 27}) improves the accuracy of fault prediction in these classes. This is one of the most important contributions of our research work.

The results of our empirical investigation allowed us to positively answer our research question: the new clustering approach improves fault prediction results. Our answer is cautious though, given that the empirical study is conducted on source code developed in open-source projects and uses past information about these projects. For example, we observed that our approach is more accurate on some systems (i.e., Ant, Lucene, Synapse, and Xalan) than on others independently from the releases considered. Indeed, for both the releases of Xerces the baseline and our approach are equivalent in terms of fault prediction accuracy. There are cases where the clustering algorithm cannot find more than one or two clusters. This happens when the classes in a system are strongly connected, which means high coupling in the system. In such cases, our approach does not improve the accuracy of defect prediction.

### A. Implications

We adopted a perspective-based approach to judge the practical implications of our study, taking into consideration the practitioner/consultant (simply practitioner, from here on) and researcher perspectives \cite{35}.

- Our clustering approach yields an average reduction of error in predicting faults of approximately 13.4%. The effect is statistically significant, the effect size is large, and the statistical power is high. For practitioners, we would report Table I in the following, more direct format: very rarely our method produces worse estimates; usually it produces better estimates; and sometimes it produces much better estimates. A researcher could also be interested in our results.
- The adoption of our approach does not require a complete and radical process change in an interested company. This is relevant for the practitioner.
- Using our approach allows the software engineer to focus only on a part of the system (i.e., a cluster at a time). This potentially could reduce the effort to analyze the source code as a consequence of fault finding and fixing tasks. This is relevant for the practitioner. From the researcher perspective, it could be interesting to study that potential benefit.
- The faults in the joined cluster are either many or very few. It seems that structural dependencies are useful to group classes that are faulty or not. This point is relevant for the researcher.
- Our results indicate that local models built on a subset of the classes of the entire system show a significantly better fit to the data compared to the global model built on all the classes together, that is, cluster rules do better than rules learned across the entire system. Our findings complement those reported in earlier work \cite{3–5}, but at smaller level of granularity. This is relevant for the researcher.
- Structural dependencies among source code classes proved to be relevant for improving fault prediction. Our work is the first in that direction. A researcher could be interested to investigate how structural dependencies affect fault prediction.
- All the systems have been developed in open-source software projects. The magnitude of the benefits deriving from the use of clustering suggests that the obtained results could be

<table>
<thead>
<tr>
<th>ID</th>
<th>Dataset</th>
<th>Baseline</th>
<th>Clustering Based Approach</th>
<th>% error</th>
<th>#classes</th>
<th>#faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SAR</td>
<td>MAR</td>
<td>SLD</td>
<td>SAR</td>
<td>MAR</td>
</tr>
<tr>
<td>1</td>
<td>Ant 1.3</td>
<td>46.4</td>
<td>0.17</td>
<td>0.37</td>
<td>45.4</td>
<td>0.12</td>
</tr>
<tr>
<td>2</td>
<td>Ant 1.4</td>
<td>71.6</td>
<td>0.29</td>
<td>0.43</td>
<td>69.2</td>
<td>0.17</td>
</tr>
<tr>
<td>3</td>
<td>Ant 1.5</td>
<td>54.4</td>
<td>0.09</td>
<td>0.19</td>
<td>51.9</td>
<td>0.05</td>
</tr>
<tr>
<td>4</td>
<td>Ant 1.6</td>
<td>185.7</td>
<td>0.27</td>
<td>0.53</td>
<td>180.19</td>
<td>0.24</td>
</tr>
<tr>
<td>5</td>
<td>Ant 1.7</td>
<td>356.2</td>
<td>0.22</td>
<td>0.48</td>
<td>362.22</td>
<td>0.22</td>
</tr>
<tr>
<td>6</td>
<td>JEdit 3.2</td>
<td>402.66</td>
<td>1.48</td>
<td>2.78</td>
<td>293.24</td>
<td>0.24</td>
</tr>
<tr>
<td>7</td>
<td>JEdit 4.0</td>
<td>243.8</td>
<td>0.35</td>
<td>1.34</td>
<td>209.44</td>
<td>0.16</td>
</tr>
<tr>
<td>8</td>
<td>JEdit 4.1</td>
<td>201</td>
<td>0.28</td>
<td>0.64</td>
<td>177.35</td>
<td>0.08</td>
</tr>
<tr>
<td>9</td>
<td>JEdit 4.2</td>
<td>136.15</td>
<td>0.37</td>
<td>0.67</td>
<td>125.7</td>
<td>0.11</td>
</tr>
<tr>
<td>10</td>
<td>JEdit 4.3</td>
<td>23.4</td>
<td>0.02</td>
<td>0.16</td>
<td>26.51</td>
<td>0.03</td>
</tr>
<tr>
<td>11</td>
<td>Lucene 2.0</td>
<td>239.71</td>
<td>1.22</td>
<td>1.7</td>
<td>218.23</td>
<td>0.56</td>
</tr>
<tr>
<td>12</td>
<td>Lucene 2.2</td>
<td>397.93</td>
<td>1.61</td>
<td>2.92</td>
<td>400.7</td>
<td>0.8</td>
</tr>
<tr>
<td>13</td>
<td>Lucene 2.4</td>
<td>455.1</td>
<td>1.34</td>
<td>1.38</td>
<td>425.8</td>
<td>1.03</td>
</tr>
<tr>
<td>14</td>
<td>POI 1.5</td>
<td>303.81</td>
<td>1.28</td>
<td>1.48</td>
<td>244.22</td>
<td>0.57</td>
</tr>
<tr>
<td>15</td>
<td>POI 2.0</td>
<td>65.2</td>
<td>0.11</td>
<td>0.21</td>
<td>58.92</td>
<td>0.11</td>
</tr>
<tr>
<td>16</td>
<td>POI 2.5</td>
<td>328.74</td>
<td>0.85</td>
<td>0.8</td>
<td>238.94</td>
<td>0.355</td>
</tr>
<tr>
<td>17</td>
<td>POI 3.0</td>
<td>368.44</td>
<td>0.83</td>
<td>1.26</td>
<td>351</td>
<td>0.43</td>
</tr>
<tr>
<td>18</td>
<td>Synapse 1.0</td>
<td>38.7</td>
<td>0.14</td>
<td>0.25</td>
<td>44.2</td>
<td>0.1</td>
</tr>
<tr>
<td>19</td>
<td>Synapse 1.1</td>
<td>120</td>
<td>0.35</td>
<td>0.56</td>
<td>123.17</td>
<td>0.33</td>
</tr>
<tr>
<td>20</td>
<td>Synapse 1.2</td>
<td>151</td>
<td>0.4</td>
<td>0.59</td>
<td>148.47</td>
<td>0.263</td>
</tr>
<tr>
<td>21</td>
<td>Velocity 1.4</td>
<td>192</td>
<td>0.44</td>
<td>0.62</td>
<td>156.1</td>
<td>0.48</td>
</tr>
<tr>
<td>22</td>
<td>Velocity 1.5</td>
<td>244</td>
<td>0.72</td>
<td>1.14</td>
<td>284</td>
<td>0.83</td>
</tr>
<tr>
<td>23</td>
<td>Velocity 1.6</td>
<td>223</td>
<td>0.59</td>
<td>1.98</td>
<td>244.5</td>
<td>0.63</td>
</tr>
<tr>
<td>24</td>
<td>Xalan 2.4</td>
<td>219</td>
<td>0.13</td>
<td>0.3</td>
<td>219.3</td>
<td>0.14</td>
</tr>
<tr>
<td>25</td>
<td>Xalan 2.5</td>
<td>519</td>
<td>0.51</td>
<td>0.65</td>
<td>426.8</td>
<td>0.4</td>
</tr>
<tr>
<td>26</td>
<td>Xalan 2.6</td>
<td>563</td>
<td>0.49</td>
<td>0.64</td>
<td>465.3</td>
<td>0.34</td>
</tr>
<tr>
<td>27</td>
<td>Xalan 2.7</td>
<td>378</td>
<td>0.25</td>
<td>0.52</td>
<td>335.7</td>
<td>0.23</td>
</tr>
<tr>
<td>28</td>
<td>Xerces 1.2</td>
<td>187</td>
<td>0.29</td>
<td>0.43</td>
<td>187</td>
<td>0.29</td>
</tr>
<tr>
<td>29</td>
<td>Xerces 1.3</td>
<td>304.31</td>
<td>0.34</td>
<td>1.42</td>
<td>312.2</td>
<td>0.27</td>
</tr>
</tbody>
</table>
generalized for other kinds of projects. This point is of interest for both the practitioner and the researcher.

- The benefits deriving from software clustering appear to be dependent on the subject software system. The researchers may be interested on identifying the properties of systems where clustering does not improve defect prediction.

VII. CONCLUSION

We presented and evaluated a new approach for defect prediction based on software clustering. The clustering is based on structural relations among classes, i.e., static references. We conclude that in order to predict faults in a class it is better to learn from classes related to it than from the entire system. We experimentally observed that this result holds at intra-release level, which is the most important result of our study.

VIII. ACKNOWLEDGMENT

Marcus and Menzies were supported in part by grants from NSF (CCF-1017263 and CCF-1017330).

REFERENCES


