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@menzies.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 metrics on the classes in each cluster identified. We present an empirical investigation on data from 29 releases of 8 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, the architectural decomposition of software systems is not always explicit.

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 of the software. We conjecture that classes from such clusters are investigated together by developers, for example, when changing the software to add features or to fix defects.

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 of 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 8 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). The models build on the clustering results have been compared with a baseline: the fault prediction model build 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. The results indicate that our clustering-based approach is produces more accurate predictions than the baseline.

The paper is structured as follows. In Section II we discuss related work, while the clustering approach is presented in Section III. The design of our empirical study is presented in Section IV while the results are discussed in Section V. Final remarks and future directions for our research conclude.
the paper.

II. RELATED WORK

The related work falls in two categories and we treat them separately in this section: fault prediction and software clustering based on structural information in source code.

A. Fault Prediction Approaches

Fault prediction is a very active research field and many studies have addressed this issue using a variety of different methods and techniques. We restrict our discussion to fault prediction approaches that work with groups of classes, such as, packages, components, modules, or cross-system clusters.

Nagappan et al. 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 and predictors are likely to be accurate for similar projects.

Schroter et al. 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, as well as past failure history, can be successfully used in prediction. The models require relationships between components and this information is typically defined at design time, so helping the identification of failure-prone components early in the software process. Our intuition in clustering related classes is somewhat founded on the results presented in that study. We group connected classes in clusters to found components which implement related features of the software.

Zimmermann et al. conduct an empirical 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. extend the work of Wiggerts by presenting a comparative study of different hierarchical clustering algorithms and by analyzing their properties with regard to software remodularization. Differently from the
approach adopted in our work here, human decisions (e.g., cutting points of the dendrograms) are needed to partition software entities into clusters.

Mitchell and Mancoridis [17] present and analyze a clustering system, named Bunch. To produce a decomposition of a system in subsystems, Bunch uses search techniques to partition the graph representing software entities and their relations. The tool is based on several heuristics to navigate through the search space of all possible graph partitions. These heuristics strongly affect the overall quality of the clustering. Also, in [18] a structural approach based on genetic algorithms is proposed to group software entities in clusters. Similar to [17], the quality of clustering depends on the definition of fitness functions and search algorithms.

Clustering algorithms based on structural information in source code have been also successfully used in the analysis of the software architecture evolution [19], [20]. For example, Wu et al. [19] present a comparative study of a number of clustering algorithms (e.g., an agglomerative clustering algorithm based on the Jaccard coefficient and the complete linkage update rule using 0.75 and 0.90 as cutting points). To partition a software system into meaningful subsystems all algorithms need to be manually configured (e.g., the specification of cutting points and fitness functions). Similarly, Bittencourt et al. [20] present an empirical study to evaluate four widely known clustering algorithms on a number of software systems implemented in Java and C/C++. The algorithms are: edge betweenness clustering, k-means clustering, modularization quality clustering, and design structure matrix clustering.

The clustering approach we adopt here is different as it does not require any configuration to partition classes into clusters.

III. CLUSTERING APPROACH

The clustering process consists of the following steps:

1) Extracting dependencies among classes. Class dependencies are extracted statically from the source code. We considered static references in the classes. The software system is represented as a graph, where the nodes are the classes and the edges are the dependencies among them.

2) Clustering. Based on the graph representation, classes are grouped into clusters using the BorderFlow clustering algorithm [21].

In the remainder of the section, 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 [22]. Including additional relationships in the program dependence graph is subject of future work. We opted for this tool because it has been used in the past and showed good results in term of scalability (e.g., [23], [24]).

B. Clustering

To group together similar classes, we use the BorderFlow algorithm [21] on the 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 [23]. Conversely, on more dense graphs and on 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 [23]. Moreover, it was also applied on natural language processing problems and to extract concepts from large word similarity graphs [21]. We also choose BorderFlow because it does not need to specify the number of clusters to be found: clustering is automatically performed because no human intervention is need. The use of different clustering algorithms is subject of future work.

In the BorderFlow clustering algorithm 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. The rationale behind is that 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 that cluster maximizes the border flow ratio:

\[\text{http://jripples.sourceforge.net/}\]
The worst-case time complexity of the BorderFlow algorithm is \(O(n^2)\), where \(n\) is the number of nodes of \(G\). The interested reader can found further details on the BorderFlow clustering algorithm in [27].

IV. EMPIRICAL INVESTIGATION

We conducted an empirical evaluation to evaluate whether our clustering approach can be effectively used for defect prediction. In this section, we present the design underlying our empirical investigation following the guidelines proposed by Wohlin et al. [25]. 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.

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

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:

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

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 \(C(X)\). This set contains all the nodes \(u \in X - V\) such that \(F(X \cup \{u\}) > F(X)\).
2) Select the candidates \(u \in C(X)\) to get the set \(C_f(X)\), this set contains all the nodes \(u\) that maximize \(\Omega(u, n(X))\).

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).

The worst-case time complexity of the BorderFlow algorithm is \(O(n^2)\), where \(n\) is the number of nodes of \(G\). The interested reader can found further details on the BorderFlow clustering algorithm in [27].

A. Definition and Context

Defect prediction approaches use software metrics and fault data to predict defects in source code [4], [26]. Our goal is to determine whether fault prediction in classes improves when building prediction models on a clusters. We build two kind 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 models using data from all the classes in the system (this is the baseline approach). In both the case, we built the prediction models by applying linear regression. We chose this technique because widely adopted in the fault prediction field (e.g., [26–28]).

Using the Goal Question Metrics (GQM) template [29], the goal of our study can be defined as follows:

**analyze** the use 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 8 open source software systems implemented in Java:

- **Ant.** It is a library and command-line tool to drive processes described in build files. Ant has been conceived to build Java applications. We studied the releases from 1.3 to 1.7.
- **JEdit.** It is a programmers text editor supporting more than 200 file types. We analyzed the releases from 3.2.1 to 4.3.
- **Lucene.** It is a cross-platform library for text search engines. For example, Lucene provides ranked searching and supports many powerful query types. In the study presented here we chose the releases from 2.2 to 2.4.
- **POI.** It has been developed to create and maintain Java APIs for manipulating various text file formats. We studied the releases from 1.5 to 3.0.
- **Synapse.** It is an enterprise service bus that provides support for XML and SOAP as well as other content interchange formats (e.g., plain text). In our study, we analyzed the releases from 1.0 to 1.2.
- **Velocity.** It is a template language engine based on a template language to reference objects defined in Java code. We studied the releases from 1.4 to 1.6.1.

• Synapse

• POI

• Ant

• JEdit

• Velocity

• Lucene

http://www2.unibas.it/gscanniello/ClusteringAndFaultPrediction/
• **Xalan.** It is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. The releases from 2.4 to 2.7 were studied.

• **Xerces.** It is a library for parsing, validating, and manipulating XML documents.

The defect data have been downloaded from the *popularity dataset* (available at [promisedata.googlecode.com](http://promisedata.googlecode.com)) of the PROMISE repository [2]. For each class the number of defects present is shown together with its software metrics. The source code of the releases was downloaded from the web (e.g., [sourceforge.net](http://sourceforge.net) and [apache.org](http://apache.org)). This was needed because our clustering approach works on source code and it is not available in the *popularity dataset*.

### 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 understudy 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. This information could not be available, while developing a predictive model, so affecting the validity of the prediction results.

### D. Selected Variables

The dependent variables is **ClassFault.** It denotes the number of faults in the classes. On the other hand, we chose as independent variables the eight software metrics shown in Table I. The first six metrics are those proposed by Chidamber and Kemerer (CK) in [30]. The next two are well known size metrics. All these metrics have been widely employed to measure software attributes of object oriented software systems and successfully used for fault prediction (e.g., [31]).

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC (Weighted Methods per Class)</td>
<td>It indicates the number of methods (assuming unity weights for all methods).</td>
</tr>
<tr>
<td>DIT (Depth Inheritance Tree)</td>
<td>It provides a measure of the inheritance levels from the object hierarchy top.</td>
</tr>
<tr>
<td>NOC (Number Of Children)</td>
<td>It measures the number of immediate descendants of the class.</td>
</tr>
<tr>
<td>CBO (Coupling Between Object classes)</td>
<td>It represents the number of classes coupled to a given class.</td>
</tr>
<tr>
<td>RFC (Response For Class)</td>
<td>It measures the number of methods that can be executed when an object of that class receives a message.</td>
</tr>
<tr>
<td>LCOM (Lack of Cohesion in Methods)</td>
<td>It counts the methods in a class that are not related through the sharing of some of the class fields.</td>
</tr>
<tr>
<td>NPM (Number of Public Methods)</td>
<td>It counts all the methods in a class that are declared as public.</td>
</tr>
<tr>
<td>LOC (Lines Of Code)</td>
<td>It is the number of instructions in each method of the class.</td>
</tr>
</tbody>
</table>

### E. Execution and Data Analysis

To perform the data analysis, we implemented a tool based on the above instantiation of our technique. The models have been built by using the R statistical environment (available at [www.r-project.org](http://www.r-project.org)).

#### 1) Estimation Techniques

In our investigation, we employed the StepWise Linear Regression (SWLR in the following) 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., [26], [27], [32]). In addition, a recent extensive literature review, in the context of fault prediction [33], 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 explores the relationship between a dependent variable and one or more independent variables, providing a model described by a linear equation:

\[ y = b_1x_1 + b_2x_2 + ... + b_nx_n + c \]

where \( y \) is the dependent variable, \( x_1, x_2, ..., x_n \) are the independent variables, \( b_i \) is the coefficient that represents the amount variable \( y \) changes when variables \( x_i \) changes 1 unit, and \( c \) is the intercept.

SWLR allows computing the equation above 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 [34]. 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 significant 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 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 applied to assess how the results of a statistical analysis can be generalized to an independent data set [35]. 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), where the original dataset is divided into $n$ different subsets of training and test sets, with each test set containing one observation [36].

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. This could happen only on joined clusters. 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 [37]. The smaller the values, the better the prediction of the faults is.

To compare our new approach and the baseline in terms of the prediction accuracy, we compute the following formula:

$$ error = \frac{MAR(Clustering) - MAR(Flat)}{StDAR(Flat)} $$

$MAR(Clustering)$ 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 $error$ assume values in between −1 and +1. Given a software release, a negative value indicate 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, in other work [38], we have found that while treatments on software engineering data may produce statistically significantly different results, the effect size of that difference is too small to be interesting. Effect size measures are also recommended by Kampenes et al. [39].

To verify whether our approach is significant better than the baseline, we tested the following null hypothesis:

H0. The errors made with our approach are not statistically less than to those made with the baseline approach.

This hypothesis is one-sided because we expect that our approach is more accurate in the prediction of faults. To test H0, we planned to apply the unpaired t-test on the two distributions of positive and negative $error$ values. The application of this test is possible if the positive and negative $error$ values are are normally distributed. This assumption is verified using the Shapiro-Wilk test [40]. In case this assumption is not verified (i.e., the $p$-value is less than a given $\alpha$ threshold), we will apply a non-parametric alternative to the unpaired t-test: the Wilcoxon rank-sum test, also known as the Mann-Whitney test [41]. For all the statistical tests performed, we decided to accept a probability of 5% (i.e., $\alpha = 0.05$) of committing a Type-I-Error [25].

Both the unpaired t-test and the Mann-Whitney statistical tests analyze the presence of a significant difference between two independent groups, but they do not provide any information about the magnitude of such a difference. Depending on the kind of the statistical test performed, we used Cohen’s $d$ or the point-biserial correlation $r$ [39] to measure the magnitude of the effect size of the possible presence of that difference. We will use Cohen’s $d$ if the data are normally distributed and the point-biserial $r$ otherwise. Depending on the data distribution, the statistical power for each performed test is also computed. The 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 [42].

F. Threats to Validity

We discuss here the threats that could affect the validity of the results. The internal validity threat is relevant in studies that try to establish a causal relationship. To this end, the pre-processing of the dataset has not been performed. This also allows other researchers to easily replicate our study. Construct validity refers to ability of establish a correct operational measures for the concepts considered in the empirical analysis [43]. We mitigated this threat by exploiting
a public dataset from the PROMISE repository, which have been employed for fault prediction in other empirical studies (e.g., [3], [4]). We used here all the systems in that dataset for which we were able to find source code.

**External validity** refers to the approximate truth of conclusions involving generalizations within different contexts. External validity threats are always present when exploiting data from a specific context. We mitigated this threat considering systems from different domains. However, replications with other systems are needed to confirm or contradict the results. Replications with data from commercial software systems are also needed.

**Conclusion validity** concerns issues that affect the ability of drawing a correct conclusion from the analysis of the gathered data. We chose a set of statistical tests to ensure that our observations are not occurring by chance.

V. RESULTS

The descriptive statistics of the absolute residuals are summarized in Table II. In this table, we also report the values for error and the number of classes and faults for each release. In the following, we first summarize the clustering results and then we detail the prediction results obtained with the baseline and our clustering based approach.

A. Clustering

In Table III we report information about the identified clusters and the faults for each release of the studied systems. In particular, for each release we report in the second and third columns the number of classes in the joined cluster and the total number of faults in these classes, respectively. The number of classes in the identified clusters are shown within parentheses in the fourth column. For example, our clustering approach on the dataset 4 (i.e., Ant 1.6) identified 2 clusters plus the joined one. The size of the two clusters was 19 and 312, respectively. Finally, the last column reports the number of faults contained in the classes of the identified clusters (also between parentheses). For example, the cluster with 19 classes of the dataset 4 contains 2 faults, while that with 312 classes contains 171 faults.

The cases where SWLR failed to build fault prediction models are highlighted with the “∗” symbol in Table III. In general, this happened when clusters (included the joined ones) contained 0 faults. In a few cases, SWLR failed to build prediction models due to the small size of relevant data, namely the classes in the clusters. For example, see the joined cluster of Xerces 1.3 (ID=29), where the number of classes in that cluster is 3, while the faults are 6. For the dataset 28, our clustering approach was not able to partition the classes into clusters. This implies that the fault prediction model built at the cluster level coincides with the one built on the entire system (i.e., the baseline).

<table>
<thead>
<tr>
<th>ID</th>
<th># classes in joined cluster</th>
<th># faults in joined cluster</th>
<th># classes in clusters</th>
<th># faults in clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9*</td>
<td>0</td>
<td>(116)</td>
<td>(33)</td>
</tr>
<tr>
<td>2</td>
<td>21*</td>
<td>0</td>
<td>(157)</td>
<td>(47)</td>
</tr>
<tr>
<td>3</td>
<td>33*</td>
<td>0</td>
<td>(260)</td>
<td>(35)</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>11</td>
<td>(19, 312)</td>
<td>(2, 171)</td>
</tr>
<tr>
<td>5</td>
<td>118</td>
<td>42</td>
<td>(600, 27)</td>
<td>(292, 4)</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>2</td>
<td>(27*, 73, 162)</td>
<td>(0, 13, 367)</td>
</tr>
<tr>
<td>7</td>
<td>3*</td>
<td>1</td>
<td>(204, 70, 29)</td>
<td>(220, 4, 1)</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>4</td>
<td>(72, 198, 27)</td>
<td>(5, 208, 0)</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>7</td>
<td>(28*, 226, 83)</td>
<td>(0, 98, 1)</td>
</tr>
<tr>
<td>10</td>
<td>74</td>
<td>1</td>
<td>(98*, 26, 294)</td>
<td>(0, 1, 10)</td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>18</td>
<td>(15, 16, 142)</td>
<td>(49, 2, 199)</td>
</tr>
<tr>
<td>12</td>
<td>88</td>
<td>160</td>
<td>(65, 24, 20, 18, 32)</td>
<td>(96, 66, 28, 37, 27)</td>
</tr>
<tr>
<td>13</td>
<td>117</td>
<td>116</td>
<td>(27, 114, 22, 33, 27)</td>
<td>(41, 285, 17, 51, 122)</td>
</tr>
<tr>
<td>14</td>
<td>64</td>
<td>52</td>
<td>(22, 132, 19)</td>
<td>(7, 203, 80)</td>
</tr>
<tr>
<td>15</td>
<td>103</td>
<td>13</td>
<td>(23, 24*, 19, 42, 17, 86)</td>
<td>(3, 0, 6, 9, 6, 2)</td>
</tr>
<tr>
<td>16</td>
<td>107</td>
<td>138</td>
<td>(44, 22, 42, 43, 23, 44, 60)</td>
<td>(89, 2, 42, 70, 12, 21, 122)</td>
</tr>
<tr>
<td>17</td>
<td>86</td>
<td>61</td>
<td>(21*, 27, 31, 277)</td>
<td>(29, 14, 10, 386, 186, 24, 37)</td>
</tr>
<tr>
<td>18</td>
<td>6*</td>
<td>0</td>
<td>(112, 39)</td>
<td>(13, 3)</td>
</tr>
<tr>
<td>19</td>
<td>4*</td>
<td>1</td>
<td>(103, 59, 14*, 42*)</td>
<td>(73, 12, 6, 7)</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>1</td>
<td>(42, 93, 91)</td>
<td>(7, 42, 95)</td>
</tr>
<tr>
<td>21</td>
<td>1*</td>
<td>1</td>
<td>(92, 53, 27, 23)</td>
<td>(110, 44, 23, 32)</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>0</td>
<td>(104, 110)</td>
<td>(121, 210)</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>0</td>
<td>(116, 113)</td>
<td>(117, 73)</td>
</tr>
<tr>
<td>24</td>
<td>52*</td>
<td>1</td>
<td>(109, 91, 58, 99, 123, 40, 66, 85)</td>
<td>(23, 12, 12, 26, 49, 12, 7, 14)</td>
</tr>
<tr>
<td>25</td>
<td>56</td>
<td>14</td>
<td>(85, 80, 91, 55, 98, 60, 76, 57, 72)</td>
<td>(99, 65, 51, 18, 35, 59, 50, 59, 36, 45)</td>
</tr>
<tr>
<td>26</td>
<td>48</td>
<td>17</td>
<td>(87, 69, 110, 77, 50, 52, 62, 76, 72, 78, 113)</td>
<td>(112, 39, 44, 42, 22, 31, 64, 61, 34, 42, 117)</td>
</tr>
<tr>
<td>27</td>
<td>379</td>
<td>497</td>
<td>(120, 42, 215, 121, 32)</td>
<td>(188, 51, 253, 153, 71)</td>
</tr>
<tr>
<td>28</td>
<td>-</td>
<td>-</td>
<td>(440)</td>
<td>(115)</td>
</tr>
<tr>
<td>29</td>
<td>3*</td>
<td>6</td>
<td>(124, 122, 114, 90)</td>
<td>(43, 54, 60, 36)</td>
</tr>
</tbody>
</table>

B. Prediction Validation

While analyzing the data in Table II we can note that our approach outperforms the baseline in terms of accuracy of predicted faults in all the releases of the 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 not 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 the approach identified a single cluster containing all the classes of that system (see Table III).

The error values are summarized in Figure I. All the points below the horizontal lines are the releases for which our approach has a better prediction accuracy. Most of these
<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>SAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ant 1.5</td>
<td>46.3</td>
<td>45.4</td>
<td>-33%</td>
<td>125</td>
<td>33</td>
</tr>
<tr>
<td>2</td>
<td>Ant 1.4</td>
<td>71.6</td>
<td>69.2</td>
<td>-50%</td>
<td>178</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>Ant 1.5</td>
<td>54.4</td>
<td>51.9</td>
<td>-38%</td>
<td>293</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>Ant 1.6</td>
<td>185.02</td>
<td>180.19</td>
<td>-1%</td>
<td>351</td>
<td>184</td>
</tr>
<tr>
<td>5</td>
<td>Ant 1.7</td>
<td>356.22</td>
<td>362.22</td>
<td>-7%</td>
<td>745</td>
<td>338</td>
</tr>
<tr>
<td>6</td>
<td>JEdit 3.2.1</td>
<td>402.66</td>
<td>293.24</td>
<td>-34%</td>
<td>272</td>
<td>382</td>
</tr>
<tr>
<td>7</td>
<td>JEdit 4.0</td>
<td>243.8</td>
<td>209.44</td>
<td>5%</td>
<td>306</td>
<td>226</td>
</tr>
<tr>
<td>8</td>
<td>JEdit 4.1</td>
<td>201.28</td>
<td>177.35</td>
<td>-46%</td>
<td>312</td>
<td>217</td>
</tr>
<tr>
<td>9</td>
<td>JEdit 4.2</td>
<td>136.15</td>
<td>125.7</td>
<td>-11%</td>
<td>367</td>
<td>106</td>
</tr>
<tr>
<td>10</td>
<td>JEdit 4.3</td>
<td>23.402</td>
<td>25.61</td>
<td>-1%</td>
<td>492</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>Lucene 2.0</td>
<td>239.71</td>
<td>218.23</td>
<td>-16%</td>
<td>195</td>
<td>268</td>
</tr>
<tr>
<td>12</td>
<td>Lucene 2.2.0</td>
<td>397.93</td>
<td>400.7</td>
<td>-15%</td>
<td>247</td>
<td>414</td>
</tr>
<tr>
<td>13</td>
<td>Lucene 2.4.0</td>
<td>455.102</td>
<td>425.8</td>
<td>-9%</td>
<td>340</td>
<td>632</td>
</tr>
<tr>
<td>14</td>
<td>POI 1.5</td>
<td>303.81</td>
<td>244.22</td>
<td>-18%</td>
<td>237</td>
<td>342</td>
</tr>
<tr>
<td>15</td>
<td>POI 2.0</td>
<td>65.20</td>
<td>58.92</td>
<td>11%</td>
<td>314</td>
<td>39</td>
</tr>
<tr>
<td>16</td>
<td>POI 2.5</td>
<td>328.74</td>
<td>238.94</td>
<td>-39%</td>
<td>385</td>
<td>496</td>
</tr>
<tr>
<td>17</td>
<td>POI 3.0</td>
<td>368.44</td>
<td>351.43</td>
<td>-26%</td>
<td>442</td>
<td>500</td>
</tr>
<tr>
<td>18</td>
<td>Synapse 1.0</td>
<td>38.714</td>
<td>44.2</td>
<td>-14%</td>
<td>157</td>
<td>21</td>
</tr>
<tr>
<td>19</td>
<td>Synapse 1.1</td>
<td>120.35</td>
<td>123.17</td>
<td>-6%</td>
<td>222</td>
<td>99</td>
</tr>
<tr>
<td>20</td>
<td>Synapse 1.2</td>
<td>151.4</td>
<td>148.47</td>
<td>-20%</td>
<td>256</td>
<td>145</td>
</tr>
<tr>
<td>21</td>
<td>Velocity 1.4</td>
<td>122.04</td>
<td>136.1</td>
<td>-12%</td>
<td>196</td>
<td>210</td>
</tr>
<tr>
<td>22</td>
<td>Velocity 1.5</td>
<td>244.72</td>
<td>284.83</td>
<td>15%</td>
<td>214</td>
<td>331</td>
</tr>
<tr>
<td>23</td>
<td>Velocity 1.6</td>
<td>223.59</td>
<td>244.55</td>
<td>7%</td>
<td>229</td>
<td>190</td>
</tr>
<tr>
<td>24</td>
<td>Xalan 2.4</td>
<td>219.03</td>
<td>219.3</td>
<td>-8%</td>
<td>723</td>
<td>156</td>
</tr>
<tr>
<td>25</td>
<td>Xalan 2.5</td>
<td>519.51</td>
<td>426.8</td>
<td>29%</td>
<td>803</td>
<td>531</td>
</tr>
<tr>
<td>26</td>
<td>Xalan 2.6</td>
<td>563.49</td>
<td>465.3</td>
<td>32%</td>
<td>885</td>
<td>625</td>
</tr>
<tr>
<td>27</td>
<td>Xalan 2.7</td>
<td>378.25</td>
<td>335.7</td>
<td>-15%</td>
<td>909</td>
<td>1213</td>
</tr>
<tr>
<td>28</td>
<td>Xercles 1.2</td>
<td>187.29</td>
<td>187.29</td>
<td>0%</td>
<td>440</td>
<td>115</td>
</tr>
<tr>
<td>29</td>
<td>Xercles 1.3</td>
<td>304.31</td>
<td>312.21</td>
<td>0%</td>
<td>453</td>
<td>193</td>
</tr>
</tbody>
</table>

Table II: Absolute Residuals, Information on the Dataset, and Error Made on the Predictions

1) Testing Hₙ₀: The data is not normally distributed for the positive error values (p-value = 0.037). Hence, we applied the Mann-Whitney test, whose result indicated that H₀ 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. A high statistical power was obtained. That is, the obtained value was 0.993.

VI. Discussion

Our approach proves to predict faulty classes with fewer errors than the baseline. The statistical test we performed confirmed that this is not by chance.

This result suggests that program dependencies among source code classes might improve fault prediction when using SWLR. That is, grouping classes by exploiting static dependencies among them and using a well established technique for fault prediction (e.g., [27], [36]) improves the dependencies among them and using a well established technique for fault prediction (e.g., [27], [36]) improves the accuracy of fault prediction in these classes. This is one of the most important contribution of our research work.

The analysis of the distribution of the faults among clusters suggests that our approach is able to group in the

![Figure 1. Distribution of the error values for the datasets considered](image-url)
most cases either classes with a large number of faults or classes with few faults. This result supports the premise of our work that faults should be analyzed among classes related via structural dependencies.

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. To this end, we took advantage of the checklists proposed by Kitchenham et al. [44]. The main practical implications of our study can be summarized as follows:

- Our clustering approach yields an average reduction of error in predicting faults of approximately 13.4% (i.e., \(\sum_{i}^{29} \frac{\text{error}_i}{29}\)). The effect is statistically significant, the effect size is large, and the statistical power is high. The statistical results were reported above. 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. This result is relevant from the practitioner perspective because he/she could use the proposed approach in his/her company to enhance intra-release fault prediction. The researcher could also be interested in our study. In particular, it could be interesting to investigate whether the use of different clustering approaches would enhance the accuracy of intra-release fault prediction.
- The adoption of our approach does not require a complete and radical process change in an interested company provided that it uses SWLR for fault prediction. 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 (see Table III). It seems that structural dependencies are useful to group classes that are faulty or not. This point is relevant from the researcher perspective.
- 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 whole system. Our findings complement those reported in earlier work [3]–[5], but at smaller level of granularity. This is relevant for the researcher, who should investigate how subsets of the entire data should be selected to define rules that do better than rules learned across the whole data.
- Structural dependencies among source code classes proved to be a relevant source of information for improving the accuracy of fault prediction. Our work is the first in that direction. From the researcher perspective, it could be interesting to investigate whether a less conservative approach in the identification of structural dependencies could lead to different result.
- The study is focussed on desktop applications and development frameworks, tools, and libraries. From the researcher perspective, the effect of using our clustering based approach on different kinds of applications (e.g., commercial and web based) represents a possible future direction.
- All the systems have been developed in open-source software projects. The magnitude of the benefits deriving from the use of clustering suggest that the obtained results could be generalized for other kinds of software development projects. This point is of interest for both the practitioner and the researcher.
- The benefits deriving from software clustering appear to be dependent from the subject software system. For example, on all the releases of Ant, Lucene, Synapse, and Xalan our approach outperforms the baseline in terms of accuracy of the predictions. A different result was obtained on the releases of Velocity and Xerces. This result is practically relevant for the practitioner and it is clearly relevant for the researcher. The researchers may be interested on identifying the properties of systems where clustering does not improve defect prediction.

VII. CONCLUSION AND FUTURE WORK

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. The results complement previous related work [3]–[5]: cluster rules do better than rules learned across the whole data. Specifically, we conclude that in order to predict
faults in a class 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.

As future work, we plan to modify the clustering approach to take into consideration also lexical information from the source code. To this end, we plan to use an approach similar to that used in [23]. Although our empirical investigation has been conducted on a large number of systems (29 releases of 8 open source software systems), we plan to extend the dataset to further assess the validity of our observations.

ACKNOWLEDGMENT

We would like to thank the Maria La Becca, who developed some of the software modules of the prototype implementing the clustering approach presented here. Special thanks are due to Alex-Cyrille Ngonga Ngomo for his support with BorderFlow clustering algorithm.

REFERENCES


