An Overview of Modern Approaches for Defect Prediction

Guilherme Ferreira
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
gferrei@ncsu.edu

ABSTRACT

The ability to predict whether a piece of code will be defective before it goes into deployment has become a very sought after commodity. Many different approaches have been proposed in order to either solve or improve current solutions to this matter. In this paper we provide a comprehensive overview of some of these approaches, how they relate to each other and how they changed over time. We also provide an overall critique of the approaches and recommendations for future work in the area.

Keywords

Fault prediction, Software mining, Machine Learning, Defect Prediction

1. INTRODUCTION

It is a well known fact that software defects are more expensive to fix after the code has been shipped to production and deployment[35]. A lot of effort has made on estimating the amount of time that will be spent on trying to predict the amount of time necessary to fix software defects [14], [37]. Since it is so expensive time and resource wise to fix defects after the product has already been shipped, it is advantageous to be able to predict if a certain piece of code have indications of being defective in the future, so that developers can fix them development, thus saving resources in the future.

Many different approaches have been proposed in the literature to solve or improve current solutions the defect prediction problem [7], ranging from using code metrics [27] (lines of code, complexity) to process metrics [2] (number of changes, recent activity) and analysis of previous defects [18].

On this paper we'll focus on a review of eight papers [3]-[36], spread consecutively over an 8-year period, that either propose a novel approach, improve an existing solution, or review current approaches and provide insights on defect prediction.

FSS '16 Raleigh, NC USA

© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00

DOI: 10.475/123_4

The rest of this paper is organized as follow: in Section 2, we will present a discussion on related work. Sections 3 to 10 analyzes the selected papers for this reviews. Section 11 contains our final observations and conclusions, and section 12 concludes with future work and potential directions for the field.

2. RELATED WORK

There have been many literature reviews on the topic of defect and fault prediction. In 1999 Fenton at al. [9] reviewed defect prediction models and evaluated and criticized them on their inability to cope with the unknown relationship between defects and failures. They claimed that there are fundamental statistical and data quality problems that can undermine model validity, and found that models for predicting software defects often have made many methodological and theoretical mistakes.

Koru et al. [21] tested building defect prediction models for a industry project dataset from NASA, trying to replicate a real-world setting. They were able to obtain better results on stratified portions of the data sets according to module size rather than predictiong on larger modules. With that they were able to develop guidelines that practitioners can follow in their defect-prediction efforts.

3. LOCAL VS GLOBAL MODELS

The paper by Bettenburg et al. [3] built upon the work of Menzies et al. [23] with regards to using local vs. global models for defect prediction. Menzies et al. claimed that most of software engineering data contains a great amount of variability, and that researchers have been using software engineering datasets for model building as is, without further considering such variability. This high amounts of variability can usually lead to poor fit of machine learning models to the underlying data. They were the first to propose that there might be potential benefit in partitioning software engineering datasets into smaller subsets of data with similar properties. The study showed that building models using such subsets (i.e. local) models lead to better fits when using specialized machine learning algorithms.

3.1 Contributions

Bettenburg et al. used those findings and compared local with global approaches, as well as a hybrid one, to see if those findings held for statistical techniques such as linear regression. They found that a local approach produces significantly better fits of statistical prediction models to the

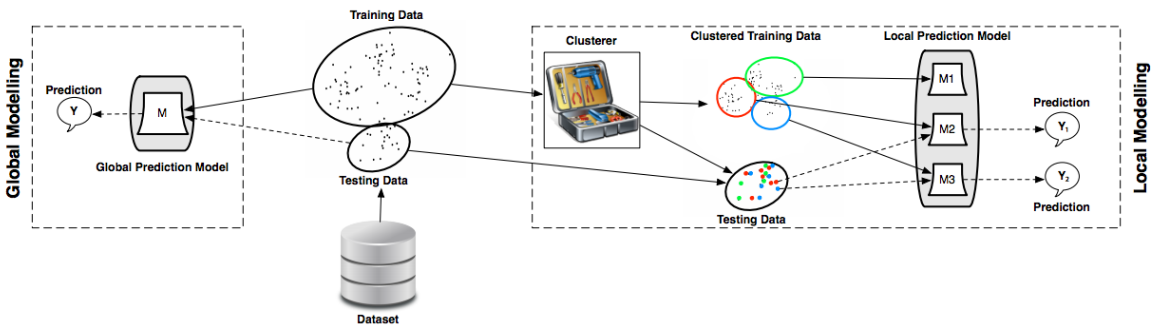

Figure 1: Overview of our approach for building global and local regression models.

underlying data, and that better fits do not overfit the prediction models. The local approach significantly increased the predictive power of statistical models, up to three times lower prediction error. They also found that while local models can distinguish the significant variables for each local region of the data, the recommendations between different regions can be conflicting.

Figure 1 shows how the local and global models are built.

3.2 Model Building

For the global model, they simply split the dataset into 90% for training and 10% for testing. To build the model they used a statistical modelling approach, linear regression. In general, linear regression models attempt to find the best fit of a multi-dimensional line through the data, and they are of the form $Y = \epsilon_0 + \alpha_1 X_1 + \ + \alpha_n X_n$, with Y the dependent variable, ϵ called the Intercept of the model, α_i the i-th regression coefficient, and X_i the i-th independent variable. In particular, Y denotes the measure that one want to predict and X_i denotes the metrics that the predictions are based on.

For the local model, the training data is then partitioned into regions with local properties by a clustering algorithm. Since there is no prior knowledge as to the optimal number of subsets in the datasets, they employed a state-of-the-art model-based clustering technique called MCLUST [10]. This technique automatically derives all necessary parameters within the technique itself, and partitions a dataset into an approximately optimal number of subsets based on the variability in the data [11]. They applied the MCLUST clustering technique, to divide each of the four prediction datasets into smaller subsets, within which observations have similar properties. The final local model is obtained by creating individual regression models of the form $Y = \epsilon_0 + \alpha_1 X_1 + \ + \alpha_n X_n$ for each local region of the data (clusters produced by MCLUST). To carry out predictions on the testing data using local prediction models, we have to take the additional step of determining for each input the most similar cluster first. After the appropriate cluster has been determined for each entry in the testing data, we then subsequently use the local model that has been fitted to that particular cluster, and carry out the individual prediction.

For the hybrid model, they used Multivariate Adaptive Regression Splines [12], or MARS. A MARS model has the form $Y = \epsilon_0 + c_1 H(X_1) + \ + c_i H(X_n)$, with Y called the dependent variable (that is to be predicted), c_i the i-th hinge coefficient, and $H(X_i)$ the i-th âĂIJhinge functionâĂİ. Hinge functions are an integral part of MARS models, as they allow to describe non-linear relationships in the data. In particular, they partition the data into disjoint regions that can be then described separately (our notion of local considerations). In general, hinge functions used in MARS models take on the form of either $H(Xi) = max(c, X_i c)$, or $H(X_i) = max(c, cX_i)$, with c being some constant real value, and X_i an independent (predictor) variable.

3.3 Statistical Results

Figure 2 contains the results of the experiments. To compare the models, they used the following statistical measures: absolute sum of prediction error; median prediction error; variance of prediction errors; and correlation between predicted and actual values. We can see that the hybrid approach clearly outperforms both the local and the global models. Moreover, the local model also tended to outperform the global one in most scenarios.

		Global Models		
Dataset	**Error Sum**	**Median Error**	**Error Var**	**Cor**
Xalan 2.6	61.07	0.64	0.37	0.36
Lucene 2.4	49.72	1.15	**2.22***	0.71
CHINA	91,592.52	765.00	14,194,155.12	0.82
NasaCoc	48.75	3.26	31.63	0.95
		Local Models		
Dataset	**Error Sum**	**Median Error**	**Error Var**	**Cor**
Xalan 2.6	57.35	0.52	0.57	0.50
Lucene 2.4	55.15	1.15	217.63	0.67
CHINA	83,420.53	552.85	19,159,955.36	0.85
NasaCoc	41.49	2.14	703.19	0.95
		Global Models with Local Considerations		
Dataset	**Error Sum**	**Median Error**	**Error Var**	**Cor**
Xalan 2.6	**50.90***	**0.40***	**0.36***	**0.56***
Lucene 2.4	**43.61***	**0.94***	2.51	**0.72**
CHINA	**25,106.00***	**234.43***	**1,102,256.01***	**0.99***
NasaCoc	**26.95***	**1.63***	**25.46***	**0.97***

Figure 2: Summary of experimental results on prediction model performance. The best observations in each column are marked in bold font face. Stars denote that the best value is statistically significant from the others at p < 0.01.

3.4 Paper conclusions and discussion

With their experiments, Bettenburg et al. were able to confirm the results of Menzies et al., who observed a similar effect of data localization on their WHICH machine-learning algorithm. These increased fits have practical implications for researchers concerned in using regression models for understanding: local models are more insightful than global models, which report only general trends across the whole dataset, whereas they have demonstrated that such general trends may not hold true for particular parts of the dataset.

As a future direction, they could see if the results hold for when they use different machine learning techniques, other than just linear regression.

4. ECOLOGICAL INFERENCE

Posnet et al. [32] defined a conceptual framework of ecological inference risk in software engineering, and empirically demonstrated the existence of this risk, while studying the incidence of ecological inference in various open source projects.

4.1 Definitions and Theory

Empirical software engineering is mainly concerned with running experiments that can gather data regarding outcomes that are observable, like quality and productivity. Such outcomes are subject to large-sample studies, so that statistical methods can be brought to bear for hypothesis testing, and machine learning methods on past data can be built into tools that support programming tasks.

Ecological inference is when an empirical finding at an aggregated level (for instance, packages) can be applied at the disaggregated level (files). When this inference is mistaken, we have the ecological fallacy. Software systems can be decomposed hierarchically, for instance, into modules, packages and files. This hierarchical decomposition has an immense influence on software evolvability, maintainability and work assignment. Hierarchical decomposition is thus clearly of central concern for empirical software engineering researchers. They also defined, for the purposes of empirical software engineering, ecological fallacy, which is when an empirical finding at an aggregated level cannot be applied at the disaggregated level, therefore only being true for the specific aggregated level.

4.2 Contributions

The effect in the defect prediction area is that it might also have Ecological Fallacy, where findings at the aggregated level are expected to be valid at a disaggregated level. Based on this concept Posnett et al. wanted to prove whether or not the Ecological Fallacy applied to defect prediction. Posnett et al proposed two main overarching conclusions in their paper: That aggregated models when evaluated only at the aggregated level may have deceivingly strong performance, and furthermore inferences from aggregated models may not apply to their disaggregated parts.

4.3 Paper conclusions and discussion

Posnett at al. showed that, because of aggregate phenomena, and observational resolution, it is often necessary to study phenomena and/or gather data at aggregated levels of products, teams, or processes. It is also possible that the resulting findings are only actionable at the disaggregated level, at the level of files, individual people, or steps of a process. Therefore, it is unlikely that ecological inference, risks notwithstanding, can be completely avoided in empirical software engineering. When making the inference, however, the risk of ecological fallacy needs to be considered and discussed. They also showed that there are construct validity issues; it is not always clear how to translate a finding relating to an aggregated metric into a concrete action that can be applied to a disaggregated product, process, or team. There are also internal validity issues, as we discussed above; factors that influence aggregation, such as intentional or unintentional assortativity can confound the results, and threaten internal validity when ecological inferences are made. Also, one must always be mindful while aggregating of the loss of statistical power due to reduction in sample size

Based on the results of Posnnet at al., Rahman et al. [33] in 2012 made considerations regarding the aggregation level of their models, in that they decided to examine code at the file level. They went that direction because they were worried that evaluation of a model on a coarser level of aggregation may not be a good predictor at the file level.

5. DEFECT PREDICTION BENCHMARK

As we have previously stated, there have been many approaches proposed to defect prediction. Although many had promising results, they often used different metrics and were tested on different datasets, thus making it hard to fully compare one another. Given that, D'Ambros et al. [6] set up to conduct an analysis of the most promising approaches at the time and create a public benchmark to which new techniques could be tested against.

5.1 Types Defect Prediction Approaches

The different types defect prediction approaches can be group and explained as follow:

Change Log Approaches use information extracted from the versioning system, assuming that recently or frequently changed files are the most probable source of future defects. Nagappan and Ball performed a study on the influence of code churn (i.e., the amount of change to the system) on the defect density in Windows Server 2003. They found that relative code churn was a better predictor than absolute churn [26]. Hassan introduced the entropy of changes, a measure of the complexity of code changes [17]. Entropy was compared to amount of changes and the amount of previous defects, and was found to be often better. The entropy metric was evaluated on six open-source systems: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL. Moser et al. used metrics (including code churn, past defects and refactorings, number of authors, file size and age, etc.), to predict the presence/absence of defects in files of Eclipse [24].

The mentioned techniques do not make use of the defect archives to predict defects, while the following ones do. Hassan and HoltâĂŹs top ten list approach validates heuristics about the defect-proneness of the most and most recently changed and defect-fixed files, using the defect repository data [18]. The approach was validated on six open-source case studies: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL. They found that recently modified and fixed entities were the most defect-prone. Ostrand et al. predict faults on two industrial systems, using change and defect data [31]. The defect cache approach by Kim et al. uses the same properties of recent changes and defects as the top ten list approach, but further assumes that faults occur in bursts [20]. The defect-introducing changes are identified from the SCM logs. Seven open-source systems were used to validate the findings (Apache, PostgreSQL, Subversion, Mozilla, JEdit, Columba, and Eclipse). Bernstein et al. use defect and change information in non-linear prediction models [2]. Six eclipse plugins were used to validate the approach.

Single-version approaches assume that the current design and behavior of the program influences the presence of future defects. These approaches do not require the history of the system, but analyze its current state in more detail, using a variety of metrics. One standard set of metrics used is the Chidamber and Kemerer (CK) metrics suite [5]. Basili et al. used the CK metrics on eight medium-sized information management systems based on the same requirements [1]. Ohlsson et al. used several graph metrics including McCabe‘s cyclomatic complexity on an Ericsson telecom system [30]. El Emam et al. used the CK metrics in conjunction with Briand‘s coupling metrics [4] to predict faults on a commercial Java system [8]. Subramanyam et al. used CK metrics on a commercial C++/Java system [34]; Gyimothy et al. performed a similar analysis on Mozilla [16]. Nagappan and Ball estimated the pre-release defect density of Windows Server 2003 with a static analysis tool [25]. Nagappan et al. used a catalog of source code metrics to predict post release defects at the module level on five Microsoft systems, and found that it was possible to build predictors for one individual project, but that no predictor would perform well on all the projects [27]. Zimmermann et al. applied a number of code metrics on Eclipse [40].

Other Approaches Zimmermann and Nagappan used dependencies between binaries in Windows server 2003 to predict defect [38]. Marcus et al. used a cohesion measurement based on LSI for defect prediction on several C++ systems, including Mozilla [22]. Neuhaus et al. used a variety of features of Mozilla (past defects, package imports, call structure) to detect vulnerabilities [29].

5.2 Defect Prediction Approaches

The individualized approaches can be classified according to certain metrics. Figure 3 summarizes said metrics.

Type	Rationale	Used by
Change metrics	Bugs are caused by changes.	Moser [11]
Previous defects	Past defects predict future defects.	Kim [13]
Source code metrics	Complex components are harder to change, and hence error-prone.	Basili [1]
Entropy of changes	Complex changes are more error-prone than simpler ones.	Hassan [10]
Churn (source code metrics)	Source code metrics are a better approximation of code churn.	Novel
Entropy (source code metrics)	Source code metrics better describe the entropy of changes.	Novel

Figure 3: Categories of Bug Prediction Approaches

Below we categorize each particular approach, give them a definition and the how they'll be used to form the benchmark.

A. Change Metrics

The approach of Moser et al. was selected as a representative, and based on it three additional variants are described.

MOSER. We use the catalog of file-level change metrics introduced by Moser et al. [24] listed in Figure 4. The metric NFIX represents the number of bug fixes as extracted from the versioning system, not the defect archive. It uses a heuristic based on pattern matching on the comments of every commit. To be recognized as a bug fix, the comment must match the string '%fix%' and not match the strings '%prefix%' and '%postfix%'. The bug repository is not needed, because all the metrics are extracted from the CVS/SVN logs, thus simplifying data extraction. For systems versioned using SVN (such as Lucene) we perform some additional data extraction, since the SVN logs do not contain information about lines added and removed.

NR	Number of revisions
NREF	Number of times file has been refactored
NFIX	Number of times file was involved in bug-fixing
NAUTH	Number of authors who committed the file
LINES	Lines added and removed (sum, max, average)
CHURN	Codechurn (sum, maximum and average)
CHGSET	Change set size (maximum and average)
AGE	Age and weighted age

Figure 4: Change metrics used by et al.

NFIX: Zimmermann et al. showed that the number of past defects has the highest correlation with number of future defects [18]. We inspect the accuracy of the bug fix approximation in isolation.

NR: In the same fashion, since Graves et al. showed that the best generalized linear models for defect prediction are based on number of changes [15], we isolate the number of revisions as a predictive variable.

NFIX+NR: We combine the previous two approaches.

B. Previous Defects

This approach relies on a single metric to perform its prediction. We also describe a more fine-grained variant exploiting the categories present in defect archives.

BUGFIXES. The bug prediction approach based on previous defects, proposed by Zimmermann et al. [39], states that the number of past bug fixes extracted from the repository is correlated with the number of future fixes. They then use this metric in the set of metrics with which they predict future defects. This measure is different from the metric used in **NFIX-ONLY** and **NFIX+NR**: For **NFIX**, we perform pattern matching on the commit comments. For **BUGFIXES**, we also perform the pattern matching, which in this case produces a list of potential defects. Using the defect id, we check whether the bug exists in the bug database, we retrieve it and we verify the consistency of timestamps (i.e., if the bug was reported before being fixed).

Variant: BUG-CATEGORIES. We also use a variant in which as predictors we use the number of bugs belonging to five categories, according to severity and priority. The categories are: All bugs, non trivial bugs (severity>trivial), major bugs (severity>major), critical bugs (critical or blocker severity) and high priority bugs (priority>default).

C. Source Code Metrics

Many approaches in the literature use the CK metrics. We compare them with additional object-oriented metrics, and LOC. Figure 5 lists all source code metrics used here.

Type	Metric	
CK	WMC	Weighted Method Count
CK	DIT	Depth of Inheritance Tree
CK	RFC	Response For Class
CK	NOC	Number Of Children
CK	CBO	Coupling Between Objects
CK	LCOM	Lack of Cohesion in Methods
OO	FanIn	Number of other classes that reference the class
OO	FanOut	Number of other classes referenced by the class
OO	NOA	Number of attributes
OO	NOPA	Number of public attributes
OO	NOPRA	Number of private attributes
OO	NOAI	Number of attributes inherited
OO	LOC	Number of lines of code
OO	NOM	Number of methods
OO	NOPM	Number of public methods
OO	NOPRM	Number of private methods
OO	NOMI	Number of methods inherited

Figure 5: Class level source code metrics by et al.

CK. Many bug prediction approaches are based on metrics, in particular the Chidamber and Kemerer suite [5].

OO. An additional set of object-oriented metrics.

CK+OO. The combination of the two sets of metrics.

LOC. Gyimothy et al. showed that lines of code (**LOC**) is one of the best metrics for fault prediction [16]. We treat it as a separate predictor.

D. Entropy of Changes

Hassan predicts defects using the entropy (or complexity) of code changes [17]. The idea consists in measuring, over a time interval, how distributed changes are in a system. The more spread, the higher is the complexity. The intuition is that one change affecting one file only is simpler than one affecting many different files, as the developer who has to perform the change has to keep track of all of them.

With that, there are two metrics: **HCM**, where every file modified in the considered period gets the entropy of the system in the considered time interval, and **WHCM**, where each modified file gets the entropy of the system weighted with the probability of the file being modified.

Variants. We define three further variants based on **HCM**, with an additional weight for periods in the past. In **ED-HCM** (Exponentially Decayed **HCM**, introduced by Hassan), entropies for earlier periods of time, i.e., earlier modifications, have their contribution exponentially reduced over time, modelling an exponential decay model. Similarly, **LD-HCM** (Linearly Decayed) and **LGDHCM** (LoGarithmically decayed), have their contributions reduced over time in a respectively linear and logarithmic fashion.

E. Churn of Source Code Metrics

Using churn of source code metrics to predict post release defects is novel. The intuition is that higher-level metrics may better model code churn than simple metrics like addition and deletion of lines of code. We sample the history of the source code every two weeks and compute the deltas of source code metrics for each consecutive pair of samples.

For each source code metric, we create a matrix where the rows are the classes, the columns are the sampled versions, and each cell is the value of the metric for the given class at the given version. If a class does not exist in a version, we indicate that by using a default value of -1. We only consider the classes which exist at release x for the prediction.

We generate a matrix of deltas, where each cell is the absolute value of the difference between the values of a metric âĂŞfor a classâĂŞ in two subsequent versions. If the class does not exist in one or both of the versions (at least one value is -1), then the delta is also -1.

Variants. We define several variants of the partial churn of source code metrics (PCHU): The first one weights more the frequency of change (i.e., delta >0) than the actual change (the delta value). We call it **WCHU** (weighted churn).

Other variants are based on weighted churn (**WCHU**) and take into account the decay of deltas over time, respectively in an exponential (**EDCHU**), linear (**LDCHU**) and logarithmic manner (**LGDCHU**)

F. Entropy of Source Code Metrics

In the last bug prediction approach we extend the concept of code change entropy [17] to the source code metrics listed in Figure 5. The idea is to measure the complexity of the variants of a metric over subsequent sample versions. The more distributed over multiple classes the variants of the metric is, the higher the complexity. For example, if in the system the WMC changed by 100, and only one class is involved, the entropy is minimum, whereas if 10 classes are involved with a local change of 10 WMC, then the entropy is higher. To compute the entropy of source code metrics, we start from the matrices of deltas computed as for the churn metrics.

Compared to the entropy of changes, the entropy of source code metrics has the advantage that it is defined for every considered source code metric. If we consider âĂIJlines of codeâĂİ, the two metrics are very similar: HCM has the benefit that it is not sampled, i.e., it captures all changes recorded in the versioning system, whereas HHLOC , being sampled, might lose precision. On the other hand, HHLOC is more precise, as it measures the real number of lines of code (by parsing the source code), while HCM measures it from the change log, including comments and whitespace

From these definitions, we define several prediction models using several object-oriented metrics: **HH**, **HWH**, **ED-HHK**, **LDHH** and **LGDHH**.

5.3 Experiments

We compare different bug prediction approaches in the following way: *Given a release x of a software system s, released at date d, the task is to predict, for each class of x, the number of post release defects, i.e., the number of defects reported from d to six months later.* We chose the last release of the system in the release period and perform class-level defect prediction, and not package- or subsystemlevel defect prediction, for the following reasons:

- Predictions at the package-level are less helpful since packages are significantly larger. The review of a defect-prone package requires more work than a class.
- Classes are the building blocks of object-oriented systems, and are self-contained elements from the point of view of design and implementation.
- Package-level information can be derived from classlevel information, while the opposite is not true. We pre-

dict the number of bugs in each class âĂŞnot the presence/absence of bugsâĂŞ as this better fits the resource allocation scenario, where we want an ordered list of classes.

We use post-release defects for validation (i.e., not all defects in the history) to emulate a real-life scenario. As in [39] we use a six months time interval for post-release defects.

Figure 6 has the results for all the metrics.

5.4 Discussion

6. USING CODE CHANGES

Predicting the incidence of faults in code has been commonly associated with measuring code complexity. While managing the complexity of a project is a paramount goal while striving to meet user needs, little attention has been paid to measuring and controlling the complexity of the code change process. A software system with a complex code change process is undesirable since it will likely produce a system which has many faults and the project will face delays.

It is also believed that a complex code change process negatively affects its product, the software system. The more complex changes to a file, the higher the chance the file will contain faults.

In [17], Hassan et al. shows that by using the complexity of code changes one can better predict the incidence of faults in a software system.

7. PERSONALIZED DEFECT PREDICTION

Different developers have different coding styles, commit frequencies, and experience levels, all of which cause different defect patterns. When the defects of different developers are combined, such differences are obscured, hurting the prediction performance. Therefore, it is desirable to build personalized defect prediction models.

Because of that, Jiang et al. [19] showed that by using personalized defect prediction and looking at the file change level, it's possible to discover up to 155 more bugs than the traditional change classification (210 versus 55) if developers inspect the top 20% lines of code that are predicted buggy. In addition, this approach improves the F1-score by 0.01âĂŞ0.06 when compared to the traditional change classification.

8. HETEROGENEOUS DEFECT PREDICTION

Crossproject Defect Prediction is normally used for new projects lacking in historical data by reusing prediction models built by other project datasets. By using models and data from other, similar projects, one can sucessfully predict defects for a project that doesn't have enough project history. Crossproject Defect Prediction requires projects to have a similar metric set, meaning the metric sets should be identical between projects. As a result, current techniques for Crossproject Defect Prediction are difficult to apply across projects with heterogeneous metric sets.

To address this limitation, Nam et al. [28] proposes heterogeneous defect prediction (HDP) to predict defects across projects with heterogeneous metric sets. The HDP approach conducts metric selection and metric matching to build a prediction model between projects with heterogeneous metric sets.

9. AUTOMATIC FEATURE LEARNING

To build accurate prediction models, studies usually focus on manually designing features that encode the characteristics of programs and exploring different machine learning algorithms. Existing traditional features often fail to capture the semantic differences of programs, and such a capability is needed for building accurate prediction models.

To bridge the gap between programsâĂŹ semantics and defect prediction features, Wang et al. [36] proposes to leverage a powerful representation-learning algorithm, deep learning, to learn semantic representation of programs automatically from source code. Specifically, we leverage Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programsâĂŹ Abstract Syntax Trees (ASTs).

10. CLASSIFICATION PERFORMANCE

Recent research on the NASA dataset suggests that the performance of a defect prediction model is not significantly impacted by the classification technique that is used to train it. However, the dataset that is used in the prior study is both: (a) noisy, i.e., contains erroneous entries and (b) biased, i.e., only contains software developed in one setting.

Ghotra et al. [13] that the choice of classification technique indeed has an impact on the performance of defect prediction models, suggesting that some classification techniques tend to produce defect prediction models that outperform others. These classification techniques can be further clustered in statistically distinct groups of techniques, where one group clearly performs better than the other.

To compare the performance of defect prediction models, the Area Under the receiver operating characteristic Curve (AUC) was used, which plots the false positive rate (i.e., FP/FP+TN) against the true positive rate (i.e., TP/FN+TP). Larger AUC values indicate better performance. AUC values above 0.5 indicate that the model outperforms random guessing. In order to group classification techniques into statistically distinct ranks, the Scott-Knott test was used. The Scott-Knott test uses hierarchical cluster analysis to partition the classification techniques into ranks. The Scott-Knott test was used to overcome the confounding issue of overlapping groups that are produced by several other post hoc tests.

Figure 7 explains each algorithm, and Figure 8 shows the results of their experiments:

11. CONCLUSIONS

12. REFERENCES

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[2] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

[3] N. Bettenburg, M. Nagappan, and A. E. Hassan. Think locally, act globally: Improving defect and

	Adjusted R^2 - Explanative power						Spearman correlation - Predictive power					
Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	Score	Eclipse	Mylyn	Equinox	PDE	Lucene	Score
Change metrics (Section IV-A)												
MOSER	0.454	**0.206**	0.596	0.517	**0.57**	9	0.323	**0.284**	**0.534**	0.165	0.238	6
NFIX-ONLY	0.143	0.043	0.421	0.138	0.398	-3	0.288	0.148	0.429	0.113	0.284	-1
NR	0.38	0.128	0.52	0.365	0.487	2	0.364	0.099	**0.548**	0.245	0.296	5
NFIX+NR	0.383	0.129	0.521	0.365	0.459	2	0.381	0.091	**0.567**	0.255	0.277	4
Previous defects (Section IV-B)												
BF (short for BUGFIXES)	0.487	0.161	0.503	0.539	**0.559**	5	**0.41**	0.159	0.492	**0.279**	**0.377**	10
BUG-CAT	0.455	0.131	0.469	0.539	**0.559**	5	**0.434**	0.131	0.513	**0.284**	**0.353**	9
Source code metrics (Section IV-C)												
CK+OO	0.419	0.195	**0.673**	**0.634**	0.379	8	0.39	**0.299**	0.453	**0.284**	0.214	8
CK	0.382	0.115	0.557	0.058	0.368	0	0.377	0.226	0.484	0.256	0.216	4
OO	0.406	0.17	**0.619**	**0.618**	0.209	6	0.395	**0.297**	0.49	0.263	0.214	6
LOC	0.348	0.039	0.408	0.04	0.077	-3	0.38	0.222	0.475	0.25	0.172	2
Entropy of changes (Section IV-D)												
HCM	0.366	0.024	0.495	0.13	0.308	-2	**0.416**	-0.001	0.526	0.244	0.308	5
WHCM	0.373	0.038	0.34	0.165	0.49	-1	**0.401**	0.076	**0.533**	0.273	0.288	7
EDHCM	0.209	0.026	0.345	0.253	0.22	-4	0.371	0.07	0.495	0.258	0.306	3
LDHCM	0.161	0.011	0.463	0.267	0.216	-4	0.377	0.064	**0.581**	**0.28**	0.275	6
LGDHCM	0.054	0	0.508	0.209	0.141	-3	0.364	0.03	**0.562**	0.263	0.33	5
Churn of source code metrics (Section IV-E)												
CHU	0.445	0.169	**0.645**	**0.628**	0.456	8	0.371	0.226	0.51	0.251	0.292	5
WCHU	**0.512**	0.191	**0.645**	**0.608**	0.478	11	**0.419**	**0.279**	**0.56**	**0.278**	0.285	13
LDCHU	**0.557**	**0.214**	0.581	**0.616**	0.458	11	0.395	**0.275**	**0.563**	**0.307**	0.293	11
EDCHU	**0.509**	**0.227**	0.525	**0.598**	0.467	11	0.362	0.259	0.464	**0.294**	0.28	6
LGDCHU	0.473	0.095	**0.642**	0.486	0.493	5	**0.442**	0.188	**0.566**	0.189	0.29	7
Entropy of source code metrics (Section IV-F)												
HH	0.484	0.199	**0.667**	0.514	0.433	7	**0.405**	**0.277**	0.484	0.266	0.318	9
HWH	0.473	0.146	**0.621**	**0.641**	0.484	8	**0.425**	0.212	0.48	0.266	0.263	5
LDHH	**0.531**	**0.209**	0.596	0.522	0.343	8	**0.408**	**0.272**	**0.53**	**0.296**	0.333	13
EDHH	0.485	**0.226**	0.469	0.515	0.359	5	0.366	**0.273**	**0.586**	**0.304**	0.337	11
LGDHH	0.479	0.13	**0.66**	0.447	0.419	4	**0.421**	0.185	0.492	0.236	**0.347**	8
Combined approaches												
BF+CK+OO	0.492	**0.213**	**0.707**	**0.649**	**0.586**	13	**0.439**	**0.277**	**0.547**	**0.282**	**0.362**	15
BF+WCHU	**0.536**	0.193	**0.645**	**0.627**	**0.594**	13	**0.448**	0.265	**0.533**	**0.282**	0.31	11
BF+LDHH	**0.561**	**0.217**	**0.615**	**0.601**	**0.592**	15	**0.422**	0.221	**0.533**	**0.305**	**0.352**	12
BF+CK+OO+WCHU	**0.559**	**0.25**	**0.734**	**0.661**	**0.61**	15	**0.425**	**0.306**	0.524	**0.31**	0.298	11
BF+CK+OO+LDHH	**0.587**	**0.262**	**0.73**	**0.68**	**0.618**	15	**0.44**	**0.291**	**0.571**	**0.312**	**0.377**	15
BF+CK+OO+WCHU+LDHH	**0.62**	**0.277**	**0.754**	**0.691**	**0.65**	15	**0.408**	**0.326**	**0.592**	**0.289**	**0.341**	15

Figure 6: Explanative and predictive power for all the apporaches

effort prediction models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 60–69. IEEE Press, 2012.

[4] L. C. Briand, J. W. Daly, and J. K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

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

[7] 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, 2012.

[8] K. El Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001.

[9] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on software engineering*, 25(5):675–689, 1999.

[10] C. Fraley and A. E. Raftery. Mclust version 3: an r package for normal mixture modeling and model-based clustering. Technical report, DTIC Document, 2006.

[11] C. Fraley and A. E. Raftery. Bayesian regularization for normal mixture estimation and model-based clustering. *Journal of Classification*, 24(2):155–181, 2007.

[12] J. H. Friedman. Multivariate adaptive regression splines. *The annals of statistics*, pages 1–67, 1991.

[13] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.

[14] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56. ACM, 2010.

[15] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change

OVERVIEW OF THE STUDIED CLASSIFICATION TECHNIQUES.

Family	Technique	Abbreviation
Statistical Techniques	Naive Bayes	NB
	Simple Logistic	SL
Clustering Techniques	K-means	K-means
	Expectation Maximization	EM
Rule-Based Techniques	Repeated Incremental Pruning to Produce Error Reduction	Ripper
	Ripple Down Rules	Ridor
Neural Networks	Radial Basis Functions	RBFs
Nearest Neighbour	K-Nearest Neighbour	KNN
Support Vector Machines	Sequential Minimal Optimization	SMO
Decision Trees	J48	J48
	Logistic Model Tree using Logistic Regression	LMT
Ensemble Methods using LMT, NB, SL, SMO, and J48	Bagging	Bag+LMT, Bag+NB, Bag+SL, Bag+SMO and Bag+J48
	Adaboost	Ad+LMT, Ad+NB, Ad+SL, Ad+SMO and Ad+J48
	Rotation Forest	RF+LMT, RF+NB, RF+SL, RF+SMO, and RF+J48
	Random Subspace	Rsub+LMT, Rsub+NB, Rsub+SL, Rsub+SMO, and Rsub+J48

Figure 7: Class level source code metrics by et al.

history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[16] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

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

[18] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 263–272. IEEE, 2005.

[19] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*,

THE STUDIED TECHNIQUES RANKED ACCORDING TO THE DOUBLE SCOTT-KNOTT TEST ON THE PROMISE CORPUS.

Overall Rank	Classification Technique	Median Rank	Average Rank	Standard Deviation
1	Rsub+J48, SL, Rsub+SL, Bag+SL, LMT, RF+SL, RF+J48, Bag+LMT, Rsub+LMT, and RF+LMT	1.7	1.63	0.33
2	RBFs, Bag+J48, Ad+SL, KNN, RF+NB, Ad+LMT, NB, Rsub+NB, and Bag+NB	2.8	2.84	0.41
3	Ripper, EM, J48, Ad+NB, Bag+SMO, Ad+J48, Ad+SMO, and K-means	5.1	5.13	0.46
4	RF+SMO, Ridor, SMO,and Rsub+SMO	6.5	6.45	0.25

Figure 8: Class level source code metrics by et al.

pages 279–289. IEEE, 2013.

[20] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[21] A. G. Koru and H. Liu. Building effective defect-prediction models in practice. *IEEE software*, 22(6):23–29, 2005.

[22] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[23] 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. IEEE Computer Society, 2011.

[24] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190. IEEE, 2008.

[25] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. ACM, 2005.

[26] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292. IEEE, 2005.

[27] 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. ACM, 2006.

[28] J. Nam and S. Kim. Heterogeneous defect prediction. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 508–519. ACM, 2015.

[29] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller.

Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.

[30] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[31] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[32] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–371. IEEE Computer Society, 2011.

[33] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[34] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, 29(4):297–310, 2003.

[35] M. Sullivan and R. Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, pages 2–9, 1991.

[36] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM, 2016.

[37] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1. IEEE Computer Society, 2007.

[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. ACM, 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 the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

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