Usage of Multiple Prediction Models Based On Defect Categories

Bora Caglayan[1], Ayse Tosun[2], Andriy Miranskyy[3], Ayse Bener[4], Nuzio Ruffolo[5]
Boğaziçi University[1,2], IBM Canada Ltd.[3,5], Ryerson University[4]
Department of Computer Engineering[1,2],
Ted Rogers School of Information Technology Management[4]
Istanbul, Turkey[1,2] Toronto, Canada[3,4,5]
{bora.caglayan[1], ayse.tosun[2]}@ boun.edu.tr, {andriy[3], ruffolo[5]}@ ca.ibm.com,
{ayse.bener[4]}@ ryerson.edu

ABSTRACT

Background: Most of the defect prediction models are built for two purposes: 1) to detect defective and defect-free modules (binary classification), and 2) to estimate the number of defects (regression analysis). It would also be useful to give more information on the nature of defects so that software managers can plan their testing resources more effectively.
Aims: In this paper, we propose a defect prediction model that is based on defect categories.
Method: We mined the version history of a large-scale enterprise software product to extract churn and static code metrics. and grouped them into three defect categories according to different testing phases. We built a learning-based model for each defect category. We compared the performance of our proposed model with a general one. We conducted statistical techniques to evaluate the relationship between defect categories and software metrics. We also tested our hypothesis by replicating the empirical work on Eclipse data.
Results: Our results show that building models that are sensitive to defect categories is cost-effective in the sense that it reveals more information and increases detection rates (pd) by 10% keeping the false alarms (pf) constant.
Conclusions: We conclude that slicing defect data and categorizing it for use in a defect prediction model would enable practitioners to take immediate actions. Our results on Eclipse replication showed that haphazard categorization of defects is not worth the effort.

Keywords

defect prediction, defect categories, software quality

1. INTRODUCTION

Building learning-based oracles that can predict the defect-prone modules in a software product has been one of the widely used approaches in both empirical software engineering research and industrial practice [12, 21, 16, 14]. In these models, defect-prone modules of new projects are predicted by learning from the historical defect data of the project, from other projects within the company, or from similar cross-company projects [22], using static code [12], churn, or social network metrics [14].

Most of the proposed prediction models treat the problem of defect prediction as a binary classification problem, in which a module is labelled as defective or not. Prediction outcome does not give any information to software project managers about the type and nature of the defect-prone modules. However, in practice, knowing defect categories may be useful to match defect types with relevant testing skills.

Many prediction models assume that, each defect category follows similar patterns, and hence a general model selects a certain set of metrics as indicators of defects based on this assumption. However, in reality, a functional requirement-related defect may differ from a security-related defect since metrics that indicate each type of defect could be different. Therefore, in this research, we propose to build a defect-category-sensitive prediction model to better guide practitioners in resource allocation. Following is our main research question:

- **How can we increase the information content of defect predictor outcomes?**

Previous approaches used pre-release and post-release defects separately as two different categories and built different models for them [14, 23]. There are various shortcomings of those approaches since they assume that software projects follow the same development cycle. For example, in open source projects, a substantial number of pre-release bugs may be reported by users because of the early availability of the releases. In projects with an iterative development model, pre-release and post-release phases may be mixed if the project is deployed partially into live systems.

In this research, we propose a defect-category-sensitive prediction model. We mined the version history of a large-scale enterprise software product to extract three defect categories. Initially, we used statistical techniques to compute correlations among various categories of defects and software metrics. We compared the performance of our proposed model with a general one in terms of defect detection capability. We also used an Eclipse dataset (releases 2.0, 2.1 and 3.0) to replicate our proposed approach.

Our results show that defect category information should be defined carefully to enrich existing information. Simple classification as pre- and post-release defects is not worth the effort. However, classifications that are based on the testing methodology (i.e., white or black box) are useful to achieve higher prediction performance.

The main contributions of this paper are therefore to:

1. Mine the data repository of a large-scale enterprise software product and extract churn metrics and defect types
2. Analyze relations between metrics and defect categories
3. Build a general defect prediction model
4. Build a category-based defect prediction model and combine category-based defect prediction models to compare with the general defect prediction model
5. Replicate the methodology with Eclipse data

The structure of the paper is as follows: In the related work section, we review relevant literature. In the methodology section, we give details of the dataset, our data extraction method, and the proposed defect prediction models. We also show the differences of correlation values of metrics with different categories of defects. In the results section, we give the results of our experiments. Then, we state threats to the validity of our research and conclude with a discussion on practical implications and future work.

2. RELATED WORK

Common approaches in software engineering research are to localize defects by identifying their causes and to predict the location and categories of defects. In terms of fault localization, researchers such as Dallmeier and Zimmermann [3] proposed semi-automatic mechanisms, whereas others such as Fatta et al. [4] enhanced the fault localization approach based on pattern-mining algorithms. These studies often investigate the failed and successful test executions and extract benchmarks for fault detection. Defect prediction models are another way to localize defects by highlighting defect-prone components, such as files and methods, in the software systems, and they are widely investigated in recent years by many researchers [12, 19, 23, 15].

Some studies have also identified the causes of the software defects that cause a software failure in terms of the size of software modules [9] and the dependencies between software modules [15, 8, 20]. Koru et al. [8] heavily investigated the relationship between the size (LOC) and defect-proneness of software modules. Their results showed that, given limited resources, software developers should focus on small modules, such as classes, since small modules are proportionally more problematic than large modules. Later, Koru and El Emam investigated the relationship between dependencies and the size of software modules in large-scale software systems [9]. Their results show that smaller modules are proportionally more dependent. As a supportive argument, Zimmermann and Nagappan also proposed that software dependencies are significantly correlated with post-release defects [15].

Although the above-mentioned studies investigate the location, causes, and probability of software defects, few researchers have conducted research on defect categories. Most of the studies analyzed pre-release and post-release defects separately in software systems [15, 12, 19]. However, few of them were able to get both pre-release and post-release defect data for software systems to conduct a comparative analysis [23, 17]. Stringfellow et al. [18] used a commercial medical record system whose defects are classified as post-release, i.e., field, and pre-release, i.e., system test. Their objective was to find defects during system testing, but not in the field, by building capture-recapture and curve-fitting models. During their study, Stringfellow et al. [18] considered only components that did not have defects during system tests, but had post-release defects, and predicted only post-release defects using their approach.

Table 1: Large-Scale Enterprise Product Dataset Information

Project Description	A Set of Architectural Functionality
Language	C and C++
Versioning System	IBM® Rational® ClearCase®
Number of Methods	7742
Snapshot Date	10 months before release

Leszak et al. [10] did a broader retrospective analysis on defect modification requests (MRs) in three major aspects: root-cause analysis, process metric study, and code complexity investigation. The authors investigated implementation, interface and external defects and estimated efforts to fix those defect MRs. According to the analysis, 75% of the effort is spent to fix implementation defects that are related to problems in the algorithm and functionality. The reason for these defects is stated as time pressure by 40% of developers. In terms of post-release defects, 68% are classified as functionality related. Furthermore, authors found that a majority of defects originate not in early phases but rather in later phases.

A recent study done by Ratzinger et al. [6] investigated the validity of several hypotheses: a) short time periods such as months are better to predict defects; b) defects with high severity are difficult to predict; c) models for defects discovered by internal staff are similar to models for defects reported by users. They mined version and issue tracking systems of five applications to extract 63 metrics. According to their proposed models, predictions prior to a release are more accurate than predictions after a release, since post-release stages are more difficult to assess than others. Furthermore, they found that post-release faults have various root causes, since each fault is reported by different type of users. On the other hand, defects discovered by the internal staff have more regularity, since testers often inspect the system based on pre-defined test cases and scenarios. Thus, depending on the defect type that is to be estimated, the cause of defect should be included in the prediction model.

Although each study mentions that there are different types and trends of defects, none of them built prediction models based on these differentiations. In our study, we aim to build a prediction model that takes defect categories into consideration in order to increase the information content of the outcome.

3. DATASET

We used a large-scale enterprise software product and Eclipse Project to conduct our empirical work. The enterprise software product of the company is one of the most popular commercial relational database solutions with a 20 years old code base. We used a 500 kLOC part of the product that constitutes a set of architectural functionalities as the dataset. Some technical details of the project are presented in Table 1. We used static code metrics and churn metrics of the project at method level. A discussion on categorization of defects can be found in Section 6. The defects that have been detected by customers are grouped in a different category.

Defect categorization and recording are less strictly done in open source development culture than in industry projects. Therefore, replication was a major challenge for us because of a lack of such data. Since we could not find any project with a similar type of

Table 2: Defects in Methods by Category

Methods	Count	Percentage
Total Methods	7742	100%
Methods with any Defects	2006	26%
Methods with FT Defects	337	4%
Methods with ST Defects	269	3%
Methods with Field Defects	445	5%

Table 3: Spearman rank correlations between defects

Defect Type	FT	ST	Field
ALL	0.604	0.466	0.617
FT	1	0.247	0.477
ST		1	0.466
Field			1

defect categorization with the product, we had to mine a different defect categorization from an open source project. Therefore, we used Eclipse Project to construct a defect prediction model based on pre- and post-release categories. Eclipse Project is a widely used multi-language software development platform written mainly in the Java language [1]. We used static code metrics and churn metrics of Version 2.1, 2.2, and 3.0 of Eclipse Project collected at file level.

3.1 Data Collection

We extracted both static code and churn metrics from the large-scale enterprise project 10 months before release date at method level. File-level prediction was not feasible because of the large file sizes (5 kLOC on average). Because of large file sizes, most of the files had a defect associated, and therefore a model based on file level granularity did not reveal much information. Static code metrics of every snapshot of the source code were extracted remotely by Prest tool developed by Softlab [7]. Churn metrics were extracted from change data provided by the company at method level. A snapshot of the source code 10 months before release date was taken as the base point that reflects the beginning of the test phase of the product. A visual summary of our data extraction method can be seen in Figure 1. Every module that had a defect after the snapshot date was labeled as defective. Afterwards, defect types were labelled by their category. Three categories were extracted from the bug descriptions entered during the project life cyle as following:

- FT Defects: Defects that are associated with the bugs found during function test [13].
- ST Defects: Defects that are associated with the bugs found during system test [13].
- Field Defects: Defects that are associated with the bugs found in the field (by the customers).

In Table 1, counts of methods with defects by categories are given. In our empirical work, we only focused on defects that could be categorized with confidence by the company.

We compared our results with pre- and post-release based categories extracted from the Eclipse dataset. Eclipse defect data has been collected previously by Schroter et. al. [17] and categorized as pre-release if they are reported within six months prior to the release and as post-release if they are reported within six months after a release. We mined the churn metrics from the source code repository of Eclipse Project[2].

4. METHODOLOGY

We initially performed statistical analysis on large-scale enterprise software data in order to identify a) correlations between software modules with FT, ST, and Field defect types (Section 4.1), b) unique characteristics of software modules with FT, ST, and Field defects in terms of software metrics (Section 4.2), and c) the relationships between software metrics (Section 4.3). Based on these statistical experiments, we constructed our proposed prediction models by applying different feature selection techniques and algorithms as well as different defect types (Sections 4.4 and 4.6). In Section 4.5, we present the performance measures we used to compare different prediction models that we constructed.

4.1 Relationship between Defect Types

Schroter et al. [17] observed Eclipse data, i.e., the correlations between pre-release and post-release defects as well as correlations between defects and software metrics. Their results show that post-release defects are not significantly correlated with pre-release defects and they are difficult to predict using process metrics. We also investigated the relationship between defect types in a large-scale enterprise software dataset.

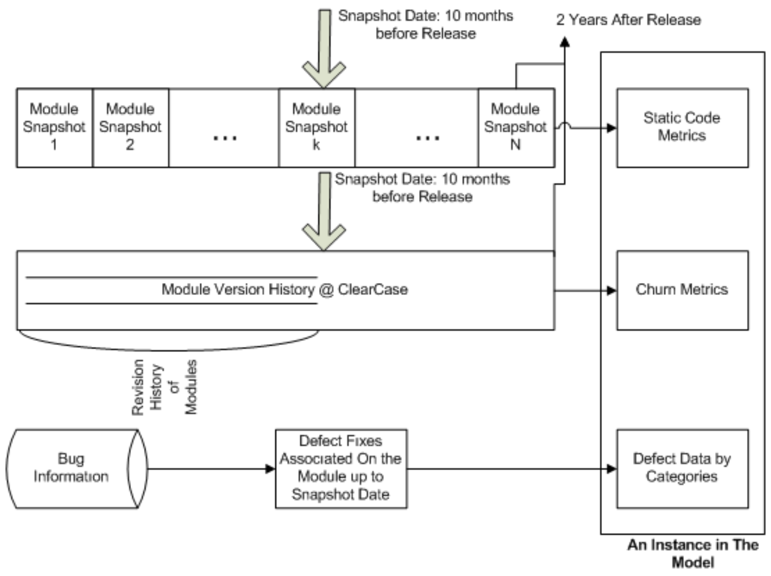
Of all software methods that have defects during functional verification testing, 20% also have defects during system verification testing, whereas 33% also have defects discovered in the field. On the other hand, among all software methods that have defects during system verification testing, 27% of them also have defects discovered in the field. Of the software methods that have both functional and system testing, 55% also have defects discovered in the field. This ratio is very high; however, these software methods constitute 8% of all field defects, i.e., 37 out of 445 field defects. Therefore, field defects are difficult to predict using defects found during functional and system verification tests.

We conducted Spearman rank correlation tests to check whether defect types are correlated with each other. Table 3 presents correlation coefficients between the number of all defects regardless of their type (ALL), the number of defects found during function test (FT), the number of defects found during system test (ST), and the number of defects found in the field (Field). It is seen that FT and Field defects are correlated with ALL defects with correlation coefficients around 60%, since their percentages among all defects are the highest. However, FT, ST, and Field defect types are not correlated with each other. Hence, they should be coming from different distributions. Mann-Whitney U-tests also show that the three defect types have significantly different medians ($p < 0.05$).

4.2 Distribution of Software Metrics According to Defect Types

To find unique characteristics of software modules with FT, ST, and Field defects, we plotted the distributions of software metrics. For each defect type (FT/ ST/ Field), we computed frequencies of each metric for three groups: 1) software methods with FT or ST or Field defects, 2) methods with ALL defects and 3) methods with NO defects.

First, when we observe the distributions of all metrics, we have seen that certain metrics show very similar trends for the three groups. Second, although certain metrics show very similar trends for the three groups, methods with NO defects are easily distinguishable with median shifts. Third, for certain metrics, all three groups have different distributions, i.e., different medians. Figure 2 illustrates the first case for the FT defect type in terms of cyclomatic complexity. It is seen that all methods have almost the same dis-



Snapshot Date: 10 months before Release
2 Years After Release
Module Snapshot 1
Module Snapshot 2
...
Module Snapshot k
...
Module Snapshot N
Static Code Metrics
Snapshot Date: 10 months before Release
Module Version History @ ClearCase
Churn Metrics
Revision History of Modules
Bug Information
Defect Fixes Associated On the Module up to Snapshot Date
Defect Data by Categories
An Instance in The Model

Figure 1: Defect matching of a module in enterprise software project data extraction

tribution, therefore, methods with FT defects cannot be predicted using cyclomatic complexity. Another metric for the FT defect type is executable LOC, which shows almost identical exponential distributions for all methods.

Figure 3 presents the distributions of commented LOC for methods with FT defects, methods with ALL defects, and methods with NO defects. As it is seen in Figure 3, methods with NO defects have a longer tailed normal distribution with a mean around 5, whereas other methods have short-tailed distributions with a mean around 2. Therefore, defect-free methods can be predicted using this metric. However, defective methods cannot be separated from each other. Halstead's vocabulary also represents the same trend for FT defect type. To represent the third case, Figure 4 shows the distributions of total number of operands for methods with FT defects, methods with ALL defects, and methods with NO defects. It is shown that the distribution of methods with FT defects are shifted to the right (in terms of the mean of a normal distribution) compared to the distribution of methods with ALL defects, whereas methods with NO defects show a totally different distribution. Therefore, we can identify defect-free methods using the distribution type, and we can identify methods with FT defects using the medians of their normal distributions.

Similar to the analysis presented for the FT defect type, we also observed unique characteristics for methods with ST defects and Field defects by forming three method groups. Results show that certain metrics are unique for one defect type, such as FT, and they are not significant to identify other defect types, such as ST or Field. For instance, design density and number of edits are significant metrics for identifying methods with ST defects from others. However, they are not significant for identifying methods with FT defects. On the contrary, the cyclomatic complexity metric is found as an insignificant indicator for the FT defect type as well as ST and Field defect types. Thus, this metric cannot be used to predict any of these defect types in company data sets.

In summary, we have seen that methods with ALL defects can be easily identified by certain software metrics. However, methods with FT, ST, and Field defects are hard to distinguish from other methods. Therefore, we should find different metric sets that would identify each defect type easily.

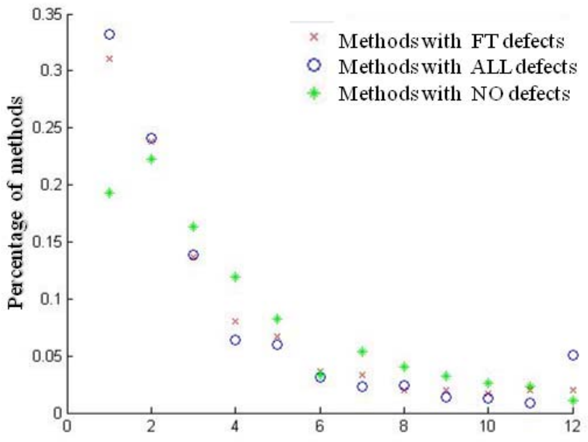


Figure 2: Distributions of "cyclomatic complexity" for methods with FT defects, methods with ALL defects, and methods with NO defects

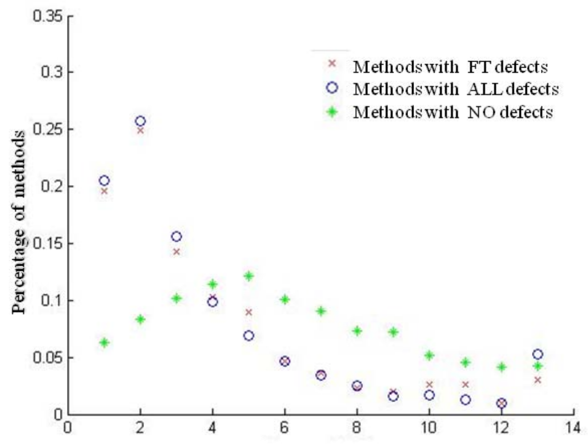


Figure 3: Distributions of "commented LOC" for methods with FT defects, methods with ALL defects, and methods with NO defects

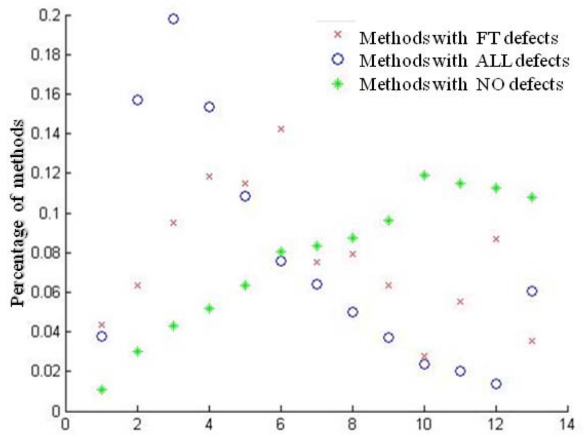


Figure 4: Distributions of "number of total operands" for methods with FT defects, methods with ALL defects, and methods with NO defects

4.3 Relationship between Software Metrics

In Sections 4.1 and 4.2, we have seen that each defect type has different characteristics in terms of software metrics, for which a metric set should be defined separately when building prediction models. To reduce the linearity between software metrics and identify the ones that are highly correlated with each other, we also computed Spearman rank correlations between software metrics and presented their results in this section. We highlighted metric pairs whose correlation coefficients are greater than 80% (p-value = 0.05). Because of the large space required to print a 29-by-29 matrix, we could not present the results of this analysis in the paper. Among all combinations of 29 metrics, 22 of them are significantly correlated with at least one metric. Design complexity metric has significant correlations with 22 other metrics. On the other hand, churn metrics are significantly correlated with each other although they are weakly correlated with size and complexity metrics. Therefore, we applied feature selection techniques such as PCA and InfoGain to reduce the linearity between dimensions and to select the most significant metrics.

4.4 Construction of the Prediction Model

Correlation analysis of different types of defects of the project shows that various types of defects follow different patterns in terms of static and code change metrics. For this reason, we constructed different predictors for different types of defects and compared their combinations with a general defect prediction model.

In Algorithm 4.1, feature selection and preprocessing steps are given. In total, these approaches gave $4 * 4 * 2 = 32$ possible combinations. For each type of defect, the best combination was found by comparing the results of all possible combinations.

As a result, we formed 4 different prediction models for FT, ST, Field defects, and for all defects in general, respectively. We implemented the algorithms and preprocessing steps defined in Algorithm 4.1 in Weka [5] and Matlab.

4.5 Performance Measures

In order to assess the performance of our proposed defect predictor on various metric sets, we used well-known performance measures: probability of detection (pd), and probability of false alarms (pf) rates [12]. Pd, which is also defined as recall, measures how good a predictor is in finding actual defective modules. Pf, on the other hand, measures the false alarms of the predictor, when it classifies defect-free modules as defective. In the ideal case, we expect a predictor to catch all defective modules (pd = 1). Moreover, it should not give any false alarms by misclassifying actual defect-free modules as defective (pf = 0). The ideal case is very rare, since the predictor is activated more often in order to get higher probability of detection rates [12]. This, in turn, leads to higher false alarm rates. Thus, we need to achieve a prediction performance that is as near to (1,0) in terms of (pd,pf) rates as possible. These parameters are found from the confusion matrix shown in Table 4. TP in confusion matrix is the number of true defect estimations, FP is the number of false defect estimations, TN is the number of true safe module estimations, and FN is the number of false safe module estimations. The optimum pf, pd rate combinations can vary from project to project. In a safety-critical project, pd rate can be more important, while on a budget-constrained project, low pf rate can be more important for resource allocation.

$$pd = \frac{TP}{TP + FN} \quad (1)$$

$$pf = \frac{FP}{FP + TN} \quad (2)$$

Algorithm 4.1: PSEUDO CODE FOR MODEL CONSTRUCTION(
)

```
$M = 10$
$N = 10$
$Datasets = \{ALLDefects, FTDefects, STDefects, FieldDefects\}$
$Algorithms = \{BayesNet, NaiveBayes, LogisticRegression\}$
$FeatureSelection = \{PCA, InfoGain, Allmetrics\}$
$PreProcessing = \{Normalize, Logfilter, Oversample, Undersample\}$
for each $data \in Datasets$
  do
for each $feature \in FeatureSelection$
  do
for each $filter \in PreProcessing$
  do
for each $algorithm \in Algorithms$
  do
$repeatMtimes$
$data' \leftarrow randomizetheorderofdata$
$data'' \leftarrow generateNbinsfromdata'$
for $i \leftarrow 1to10$
  do
$testset = data''[i]$
$trainset = data'' - data''[i]$
$predictions[i, , ] \leftarrow applyfeature, filterandalgorithmtotrainingandtestsets$
```

Table 4: Confusion Matrix

actual / **predicted**	defective	defect free
defective	TP	FP
defect free	FN	TN

4.6 Comparison of General Defect Prediction with Category-Based Defect Prediction

For comparison reasons, we selected the best general defect predictor and compared its performance with the combined performance of category-based predictors. We compared the performance of general defect prediction model with category-based defect prediction model using a simple method. Our proposed category-sensitive defect prediction model flags a module as defective if it is predicted as defective in any of the categories. For our dataset defect categories, our proposed approach labelled modules based on the following formula:

$$module_{st_{def}} \vee module_{ft_{def}} \vee module_{field_{def}} \Rightarrow module_{def} \quad (3)$$

This is a conservative estimate that labels a method as defective if it is predicted to be defective in any of the categories. The effect of this conservative estimate should result in higher overall pd rates without significantly changing pf.

5. RESULTS

We used logistic regression, Naive Bayes, and Bayes Net algorithms in our experiments. Bayes Net gave the best results both in the general model and in the category-sensitive model. The general defect prediction model does not require any sampling strategy and it provides pd and pf values of 0.64 and 0.27, respectively (Table 5).

FT Defects

To predict FT defects, we constructed a model with an oversampling using Weka's smote algorithm with pd and pf values of 0.69 and 0.17, respectively (Table 5).

ST Defects

To predict ST defects, we constructed a model with an oversampling using Weka's smote algorithm with resulting pd and pf values of 0.66 and 0.25, respectively (Table 5).

Field Defects

To predict field defects, we constructed a model with undersampling using Weka's resample algorithm with resulting pd and pf values of 0.67 and 0.25, respectively (Table 5).

In order to be able to compare these results with the general defect prediction model, we aggregated the category-sensitive defect prediction models by using the method that is explained in Section 4.4. Comparison of results is also presented in Table 5. The comparison shows that category-based classification gives statistically significant higher pd rates (by Mann-Whitney U Test with $p<0.05$).

Bayes Net algorithm gave significantly better prediction performance. This may be due to high correlation rates among software metrics. Bayes Net takes into account the cause-effect relationships of metrics dynamically. Moreover, in defect-category sensitive prediction models, different sampling strategies considerably improved the prediction performance of the algorithm because of the the imbalanced (uneven distribution of different labels) nature of the dataset[11].

5.1 Replication with Eclipse Dataset

Our results on large-scale enterprise software data show that the building separate models by defect category improves the pd rates provided that categorization includes relevant information. To validate this hypothesis, we investigated an open source software development, Eclipse, where defects are categorized as pre and post-release defects. It has three consecutive versions, 2.0, 2.1 and 3.0.

We initially conducted statistical analysis on Eclipse data as explained in Sections 4.1, 4.2, and 4.3 and found similar results with large-scale enterprise software data. Both pre- and post-release defects have different characteristics in terms of software metrics. Furthermore, Mann-Whitney U Test shows that defect types are significantly different in terms of medians and they are not correlated with each other. Therefore, we built separate prediction

Table 5: Results of Different Types of Defects

	Algorithm: Bayes Net, 10-Fold Cross Validation			
	No Resampling		With Resampling	
Prediction Type	**pd**	**pf**	**pd**	**pf**
General Defect Prediction	*0.64*	*0.27*	0.62	0.25
FT Defect Prediction	0.64	0.17	0.69	0.17
ST Defect Prediction	0.54	0.21	0.66	0.25
Field Defect Prediction	0.43	0.06	0.67	0.25
Combination of Defect Category Based Models	*0.71*	*0.28*		

models for each release by taking into account a) only pre-release defects, b) only post-release defects, and c) all defects. Using the same Algorithm 4.1, we have decided on the pre-processing technique and the algorithm for all defect types. Since there are three successive releases, we trained our proposed model with release 2.0 and 2.1, and tested on 2.1 and 3.0, respectively. Results are summarized in Tables 6 and 7. It is seen that:

- Bayes Net using all metrics with no sampling gives statistically the best results for all defect categories.
- Combination of two separate defect-category sensitive models with our proposed approach (OR) gives statistically insignificant results compared to a general prediction model.

Although statistical analysis provides similar conclusions on defect types, during model construction, we found that building category-sensitive prediction models does not improve the prediction performance on Eclipse data since pre- and post-release categorization does not really reveal much information. In [17], authors explained that pre-release defects are collected within six months prior to each release, whereas post-release defects are collected within a six months period after the release. Since open source projects are accessible before their official releases in the form of stable or beta versions, test groups detecting the bugs (such as system and functional testing groups in the company) can differ from commercial projects. Furthermore, pre-release defects may also be reported by users who are willing to use beta versions and report bugs during the development. These bugs collected as pre-release may not be actual pre-release bugs.

Table 6: Results Pre- and Post-Release Sensitive Defect Prediction For Eclipse Dataset (Train with V2.0 Test on V2.1)

Defect Category	**pd**	**pf**
All Defects	0.75	0.38
Pre-Release Defects	0.81	0.46
Post-Release Defects	0.67	0.32
Combination of Defect Categories	0.76	*0.38*

Table 7: Results Pre- and Post-Release Sensitive Defect Prediction For Eclipse Dataset (Train with V2.1 Test on V3.0)

Defect Category	**pd**	**pf**
All Defects	0.65	0.27
Pre-Release Defects	0.65	0.26
Post-Release Defects	0.65	0.27
Combination of Defect Categories	0.65	0.26

6. THREATS TO VALIDITY

We consider three major threats to the validity of our experiments: construct, internal, external. To avoid the construct validity threats in terms of measurement artefacts, we used two popular performance measures in software defect prediction research, which are probability of detection and probability of false alarm rates. We decided on the snapshot date as 10 months before release date to extract metrics data from the large-scale enterprise software development tree and matched all defects with software methods after that date to make our model consistent with the development life cycle in the company. While labelling defect categories, we used the descriptions of testing phases and double-checked these labels to overcome any problems. According to Myers, development process can have a one-to-one correspondence with testing process so that we can avoid unproductive redundant testing [13]. Based on the model proposed by Myers [13], we defined FT defects as bugs found during functional testing, which checks external specifications of the software system. Furthermore, we defined ST defects as bugs found during system testing, which checks inconsistencies with the original objectives. Therefore, although it seems that we categorized defects based on when they are caught, in this study, defects are categorized according to their main causes. Field defects that have been corrected before our snapshot date are associated with the bugs found by a small set of companies who tested on early trial versions. The category of the defects found by these companies may differ from the field defects after the release date. As a result, this may have caused decreased performance in the field defect category compared to other categories. To avoid sampling bias in our experiments, we used 10-fold cross validation. For statistical validity, we considered the linearity assumptions in statistical tests and used Spearman and Mann-Whitney U tests, both of which compute rank correlations among variables. To externally validate our findings, we also used an open source project, Eclipse, and replicated our experiments conceptually. Although defect categorization is different in Eclipse data, our methodology was the same so that we can draw comparable results.

7. CONCLUSIONS AND FUTURE WORK

In this research, we challenged the **assumption of similar patterns in defective modules**. We used a defect-category-based estimation model in order to show that increase in prediction performance can be achieved by using category information. The results of our empirical work answer our research question:

- **How can we increase the information content of defect predictor outcomes?**
  Proposing prediction models for different defect categories gives more information content about the defect in a module. This helps us build models with higher benefit to managers. If we can give information about the nature of the

defect in the defect-prone modules managers can match the relevant skills with the defects. For example, based on performance analysis on different defect categories, certain employees may be better suited to a category of defects. This is important for more efficient resource allocation as well as higher end-product quality. By aggregating category-based models, we were able to increase the pd rate by 10%.

Theoretical Contributions of Our Work

Our analysis shows that before proposing a general defect prediction model, analyzing defect patterns in the data helps researchers. Our results show that different types of defects may follow different patterns. In these cases, category-based classification gives better results.

Practical Contributions of Our Work

Our work shows that effective categorization of defects would help companies localize their defects more effectively. By providing category-based estimations, we would be able to increase the information content of prediction output. That is, instead of saying yes or no to the defect prediction model, we can predict the category or possible categories of a defect. Our results on the Eclipse dataset showed that defect categorization per se does not mean much. Therefore, a systematic approach for categorization is necessary to collect reliable data and hence produce meaningful outcomes.

Future Work

In the future, we will search for more data that is recorded in a systematic method for defect categories.

Acknowledgements

This research is supported in part by Turkish Scientific Research Council (Tubitak) under grant number EEEAG 108E014 and Turkish State Planning Organization (DPT) under the project number 2007K120610. We would like to thank IBM Canada Lab – Toronto site for making their development data available for research and strategic help during all phases of this research.

Appendix: List of Software Metrics Used in This Study

Attribute	Description	Attribute	Description
McCabe metrics			
Cyclomatic density, vd(G)	the ratio of module's cyclomatic complexity to its length	*Essential complexity, ev(G)*	the degree to which a module contains unstructured constructs
Design density,dd(G)	condition/ decision	*Cyclomatic complexity, v(G)*	# linearly independent paths
Essential density,ed(G)	(ev(G)-1)/(v(G)-1)	*Maintenance severity*	ev(G)/v(G)
Halstead metrics			
Difficulty (D)	1/L	*Length (N)*	N1 + N2
Level (L)	(2/n1)*(n2/N2)	*Programming effort (E)*	D*V
Volume (V)	N*log(n)	*Programming time (T)*	E/18
Lines of code metrics			
Unique operands	n1	*Executable LOC*	Source lines of code that contain only code and white space
Branch count	# of branches	*Total operators*	N1
Decision count	# of decision points	*Total operands*	N2
Condition count	# of conditionals	*Unique operators*	n2
Churn metrics			
Number of edits	number of edits done on a file	*Lines Added*	total number of lines added
Number of unique committers	number of unique committers edited a file	*Lines Removed*	total number of lines removed

8. REFERENCES

[1] Eclipse project website. `http://www.eclipse.org`.

[2] B. Caglayan, A. Bener, and S. Koch. Merits of using repository metrics in defect prediction for open source projects. *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 31–36, May 2009.

[3] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, November 2007.

[4] G. Di Fatta, S. Leue, and E. Stegantova. Discriminative pattern mining in software fault detection. In *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*, pages 62–69, New York, NY, USA, 2006. ACM.

[5] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[6] M. P. Jacek Ratzinger and H. Gall. EQ-Mine: Predicting Short-Term Defects for Software Evolution. In *Proceedings of the Fundamental Approaches to Software Engineering at the European Joint Conference on Theory and Practice of Software*, pages 12–26. Springer Berlin, 2007.

[7] E. Kocaguneli, A. Tosun, A. B. Bener, B. Turhan, and B. Caglayan. Prest: An intelligent software metrics extraction, analysis and defect prediction tool. In *SEKE*, pages 637–642, 2009.

[8] A. G. Koru and K. E. Emam. The theory of relative dependency: Higher coupling concentration in smaller

modules. *IEEE Software*, 27:81–89, 2010.

[9] A. G. Koru, D. Zhang, K. El Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Trans. Softw. Eng.*, 35(2):293–304, 2009.

[10] M. Leszak, D. E. Perry, and D. Stoll. Classification and evaluation of defects in a project retrospective. *J. Syst. Softw.*, 61(3):173–187, 2002.

[11] M. A. Maloof. Learning when data sets are imbalanced and when costs are unequal and unknown. In *ICML-2003 Workshop on Learning from Imbalanced Data Sets II*, 2003.

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

[13] G. J. Myers, T. Badgett, T. Thomas, and C. Sandler. *The Art of Software Testing. 2nd ed.* John Wiley & Sons, 2004.

[14] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 364–373, Washington, DC, USA, 2007. IEEE Computer Society.

[15] N. Nagappan, L. Williams, M. Vouk, and J. Osborne. Using in-process testing metrics to estimate post-release field quality. In *ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability*, pages 209–214, Washington, DC, USA, 2007. IEEE Computer Society.

[16] 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.

[17] A. Schroter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk. In *Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters*, pages 18–20, 2006.

[18] C. Stringfellow, A. Andrews, C. Wohlin, and H. Petersson. Estimating the number of components with defects post-release that showed no defects in testing. *Software Testing, Verification and Reliability*, 12(2):93–122, 2002.

[19] A. Tosun, B. Turhan, and A. Bener. Practical considerations in deploying ai for defect prediction: a case study within the turkish telecommunication industry. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–9, New York, NY, USA, 2009. ACM.

[20] A. Tosun, B. Turhan, and A. Bener. Validation of network measures as indicators of defective modules in software systems. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–9, New York, NY, USA, 2009. ACM.

[21] B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. In *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pages 231–237, 2007.

[22] B. Turhan, T. Menzies, A. Bener, and J. Distefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering Journal*, 2009. in print. DOI 10.1007/s10664-008-9103-7.

[23] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 9, Washington, DC, USA, 2007. IEEE Computer Society.