Manuscript
Click here to view linked References

Practical Considerations in Deploying Statistical Methods for Defect Prediction: A Case Study within the Turkish Telecommunications Industry

Ayşe Tosun[1], Ayşe Bener[2], Burak Turhan[3], Tim Menzies[4]
[1,2]Software Research Laboratory (Softlab), Department of Computer Engineering, Boğaziçi University, Istanbul, Turkey, +90 212 359 7227
[3]Department of Information Processing Science, University of Oulu, Oulu, Finland, 358 8 553 3169
[4]Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, 1 304 376 2859
[1]ayse.tosun@boun.edu.tr, [2]bener@boun.edu.tr, [3]burak.turhan@oulu.fi, [4]tim@menzies.us

Abstract

Context: Building defect prediction models in large organizations has many challenges due to limited resources and tight schedules in the software development lifecycle. It is not easy to collect data, utilize any type of algorithm and build a permanent model at once. We have conducted a study in a large telecommunications company in Turkey to employ a software measurement program and to predict pre-release defects. Based on our prior publication, we have shared our experience in terms of the project steps (i.e. challenges and opportunities). We have further introduced new techniques that improve our earlier results.

Objective: In our previous work, we have built similar predictors using data representative for U.S. software development. Our task here was to check if those predictors were specific solely to U.S. organizations or to a broader class of software.

Method: We have presented our approach and results in the form of an experience report. Specifically, we have made use of different techniques for improving the information content of the software data and the performance of a Naïve Bayes classifier in the prediction model that is locally tuned for the company. We have increased the information content of the software data by using module dependency data and improved the performance by adjusting the hyper parameter (decision threshold) of the Naïve Bayes classifier. We have reported and discussed our results in terms of defect detection rates and false alarms. We also carried out a cost-benefit analysis to show that our approach can be efficiently put into practice.

Results: Our general result is that general defect predictors, which exist across a wide range of software (in both U.S. and Turkish organizations), are present. Our specific results indicate that concerning the organization subject to this study, the use of version history information along with code metrics decreased false alarms by 22%, the use of dependencies between modules further reduced false alarms by 8%, and the decision threshold optimization for the Naïve Bayes classifier using code metrics and version history information further improved false alarms by 30% in comparison to a prediction using only code metrics and a default decision threshold.

Conclusion: Implementing statistical techniques and machine learning on a real life scenario is a difficult yet possible task. Using simple statistical and algorithmic techniques produces an average detection rate of 88%. Although using dependency data improves our results, it is difficult to collect and analyze such data in general. Therefore, we would recommend optimizing the hyperparameter of the proposed technique, Naïve Bayes, to calibrate the defect prediction model rather than employing more complex classifiers. We also recommend that researchers who explore statistical and algorithmic methods for defect prediction should spend less time on their algorithms and more time on studying the pragmatic considerations of large organizations.

Keywords
Software defect prediction, experience report, Naïve Bayes, static code attributes.

1. INTRODUCTION

Telecommunications is a highly competitive and booming industry in Turkey and the neighboring countries. The leading telecommunications (GSM) operator in Turkey operates in Azerbaijan, Kazakhstan, Georgia, Northern Cyprus, and Ukraine with a customer base of 53.4 million. The company has grown very rapidly and successfully since its inception in 1994. It has an R&D center including 200 researchers and engineers with one to ten years of experience. The company's legacy software has millions of lines of code that need to be maintained.

The company is under constant pressure to launch new and creative campaigns in a limited amount of time and on tight budgets. As the technology changes and the customers require new functionalities, the company has to respond faster than ever by means of new software releases. Currently, they make releases every two weeks. They use incremental software development [21], where each release has additional or modified functionalities, compared to the previous releases. Therefore, the time to track and fix the problems in their software and to ensure the overall software quality is limited.

To ensure overall software quality, each stage of the development process must include verification, validation and testing (VV&T) activities [29]. Tight schedules, limited budgets and continuous changes in project requirements make this very hard to manage. The traditional software development lifecycle prioritizes the testing phase as the most critical phase for verifying the adequacy of the software and its consistency with the requirements [29]. Similar to traditional practices, testing is one of the most critical stages in the company's development phase [2, 12, 14]. Accordingly, the managers are interested in any opportunity to improve their software development practices.

In the literature, different VV&T strategies such as code inspections, reviews and defect prediction tools, are proposed to use the time and resources effectively while catching as many defects as possible in the software [29]. Among these, defect prediction models are helpful tools that guide the user, i.e. developer or tester, to specific parts of the software, which are more likely to fail. Defect predictors help developers reduce the testing effort and managers allocate resources more efficiently. A variety of software defect prediction models have been proposed over the years [1, 2, 6, 10, 17, 20]. Most of them have combined well-known methodologies and algorithms from various engineering domains. The most common methods used in these models are statistical techniques [2, 10, 17] and machine learning [1, 6, 20]. They require previous data from completed projects in terms of software metrics and actual defect rates. Software metrics can be quantitative measures such as size and complexity [30], whereas defect rates represent the location (i.e. file/ functional unit/ LOC where the defect occurred) and the number of defects detected in the software [31]. Prediction models combine these metrics and defect information as training data to find out which modules are more defect-prone. Based on the knowledge gathered from past projects and the software metrics extracted from a new project, the model can estimate defect-prone modules of the new project. In such studies, a module whose defect-proneness is estimated can be either a functional unit, i.e. software method, or a file in the source code.

Our study constructs a metrics program and a decision support system to predict defects and employ a more effective release management process. Previous studies have implemented extensive automated metrics and defect prediction programs using data mining such as the NASA Metrics Data Program [1]. In this paper, we describe our experience in applying those programs to a telecommunications company, while omitting all the technical details to build this metrics program and defect prediction model. While the company's development practices are much different from those of NASA, the automated statistical and machine learning methods ported with relatively little effort. However, we have spent much time learning the organization and working within their current practices. We, therefore, recommend that researchers who explore statistical and machine learning methods for defect prediction should spend less time on their algorithms, but work more on pragmatic considerations of large organizations. Note that in the following case study, we had to respond to user challenges and project demands numerous times. Because we were responsive, we were able to report that practitioners can benefit from a learning-based defect predictor to solve the problems of the company's software development processes such as the allocation of resources, defect tracing and measurement in a large software system. Our prior publication [28] summarizes the preliminary results obtained from working with this company: 88% detection rates with 28% false alarms using code metrics and version history. In this paper, we have extended those results in two ways: a) increasing the information content of data, b) adjusting the hyperparameter of a Naïve Bayes classifier based on the local software data. After deploying the initial model, we have increased the information content of the data by adding dependencies between modules. Results show that dependencies provide valuable information for defect predictors, reducing false alarms by 8%. However, conducting measurement programs to collect these data is extremely difficult in large organizations operating in a competitive industry with tight schedules. Therefore, we have also focused on improving the performance of a Naïve Bayes classifier by adjusting its hyper parameter, i.e. decision threshold. Threshold optimization further decreases false alarms by 30%, thereby reducing the estimated inspection costs of testers to detect defective modules.

This paper is organized as follows: In Section 2, we explicitly define our goals in this project aligned with the objectives of the company. In the subsections of Section 3, we explain our prior publication [28] in terms of plans, the challenges we came across during each phase and the methodologies to solve these problems in detail. In Section 4, we present our latest findings based on two new techniques (increasing the information content of data in Section 4.1 and adjusting the hyperparameter of Naïve Bayes in Section 4.2). In Section 5, we discuss the threats to the validity of our results. Finally, we share our best practices as well as mistakes during this project and conclude with suggestions to both academics and practitioners who would plan to build local defect predictors in large organizations.

2. GOALS OF THE PROJECT

Our goals in this project are defined as *building a code measurement repository, defect tracing/matching program* and *a defect prediction model.* We have jointly agreed on these goals with the R&D manager, project managers and the development team during the project kick-off meeting. The senior management strongly believed that they should adjust their development processes to increase software quality and to allocate resources effectively. We have decided on the roles and responsibilities and aligned the goals of our project with their business goals (Table 1). We have clearly explained to them the final deliverables.

In order to track the progress of the project, we held monthly meetings with the project team and quarterly meetings with the senior management to present the progress and discuss the following steps. Senior management meetings were quite important either to escalate the problems or to agree on critical decisions we had to make throughout the project. During the life of the project, researchers from our software research laboratory (SoftLab) were on-site on a weekly basis to work with the coders, testers and quality teams.

Previously, the company did not employ any measurement process due to tight schedules and heavy workloads. Therefore, we have planned our work in four main phases:

- In the first phase, we aimed to measure the static code attributes at function level, i.e. the level of functions that are defined as individual methods in the source code.
- In the second phase, we planned to match software methods, i.e., functional units, with pre-release defects.
- In the third and fourth phases, we planned to build and calibrate a defect prediction model, assuming that we would be able to collect enough data to train our model.

However, the outcomes of every phase led us to redefine and extend the original scope and objectives in the following stages.

3. PHASES OF THE PROJECT

Our prior publication [28] includes project details such as code measurement and analysis (Phase I), defect tracing and matching (Phase II), defect prediction modeling (Phase III), and extended phase in defect prediction (Phase IV). In this paper, we also explain these phases in order to keep track of the progress in the project. Furthermore, we add two new experiments that investigate the *data* and the *algorithm*. In Section 4, the information content of the data is increased using dependencies between software files. Then, the decision threshold of the algorithm is adjusted using ROC curves. These experiments improve the performance of the prediction model in terms of false alarms. We present these findings in Section 4.

3.1 Phase I: Code Measurement and Analysis

In the first three months of the project, we aimed to analyze the company's development practices and to conduct a literature survey of measurement and defect prediction in the telecommunications industry. At the end of this phase, we expected to agree on the list of static code attributes and to decide on an automatic tool to collect them. We also planned to collect the first set of static code attributes from the source code in order to make a raw code analysis. In order to do that, we planned to choose the sample projects through which we would collect data at function level.

We decided that static code attributes could be used to investigate specific trends or characteristics in the code and current development practices. Static code attributes are widely used to predict defect-proneness in software systems [1, 3, 6, 12, 15, 16, 17, 18]. Furthermore, they are easily collected through automated tools. Therefore, we defined the set of static code attributes from NASA MDP Repository [12], as the metrics to be collected from the software in the company. Basically, we collected complexity metrics proposed by McCabe [5], metrics related to the unique number of operators and operands, which are proposed by Halstead [4], size metrics to count executable and commented lines of code, and CK object-oriented metrics [19] from Java applications. The set of metrics collected in different granularity levels can be seen in Table 2. We also donated all data that came from nine applications of the software to Promise, i.e. online data repository [7]. Thus, the data are publicly available for replicated studies and new experiments.

3.1.1 Challenges during Phase I

As mentioned earlier, the company did not have a process or an automated tool to measure code attributes. Our suggestion was to purchase a commercially available automated tool to extract metrics information easily and quickly. However, due to budget constraints and concerns about the adequacy of functionalities of the existing tools, the senior management did not want to invest in such a tool. Moreover, their software systems contained source codes and scripts written in different languages such as Java, JSP and PL/SQL. Therefore, the management was not convinced about finding a single cost effective tool that would embrace all languages and easily extract similar attributes from all of them.

Their software system has the standard three-tier architecture with presentation, application and data layers. However, the content in these layers cannot be separated as distinct projects. Any enhancement to the existing software somehow touches all or some of the layers at the same time, making it difficult to identify code ownership as well as to define distinct software projects.

Another problem we came across was partially related to the process of metrics collection. When we observed the software development process with the coding practices of the team, we saw that collecting static code attributes in the same manner with NASA datasets, i.e. at function level, is not adequate in the company, since it is almost impossible, due to lack of automated mechanisms, to match those attributes with the defect data afterwards.

3.1.2 Our Methodology

Metric Extraction: We had developed an all-in-one metrics extraction and analysis tool, *Prest*, to extract code metrics [9, 24]. Compared to other commercial and open source tools, Prest embraces many distinctive features, and it is freely available. It extracts 22 to 27 static code attributes in different granularities, i.e. package, class, file, and method level. The number of attributes varies according to the granularity level, since certain attributes, such as object-oriented metrics, are only extracted at class level. Prest is able to parse programming languages such as C, C++, Java, JSP, PL/SQL, and to form a dependency, i.e. call graph, matrix, that keeps the interrelations between the modules of the software systems. Prest has two different views: Project and Prediction. The Project view has a simple user interface, which allows the user to import the source code and parse his/her project using one or more language parsers. It is able to parse different parts of the code, using parsers of different programming languages. Additional analysis on modules such as new threshold definition helps to identify critical modules whose attributes do not meet the coding standards of NASA MDP [12]. It is also possible to define new virtual metrics by combining two or more metrics, to apply logarithm filtering as a pre-processing step and to display histogram of the selected metrics in the Project view. A sample screenshot taken from Prest is present in Figure 1. In the Prediction view, a training set and a test set can be separately uploaded. Then a Naïve Bayes classifier or Decision Tree algorithm can be applied, from which we have used the former one as the algorithm of our predictor model. Using the Project view of Prest, we were able to collect static code attributes from the files and the methods of 24 critical applications of the company.

Project Selection: This was one of our first critical decisions in the project in order to define the scope and choose the projects and/or units of production that would be studied. The project managers initially wanted to focus on the presentation and the application layers because of the complexity of the architecture. However, we decided in unison to take twenty-four interdependent Java applications embedded in these layers due to the fact that layers contain more than one application; they are implemented by many development teams and, sometimes, more than one software process is used. Therefore, we selected these Java applications with clearly defined process teams, development process and packages affected in each layer. We refer to these applications as GSM projects during this study.

Level of granularity: As mentioned in the previous section, we were unable to use method-level metrics in the software system since we could not collect method-level defect data from the developers in such a limited period of time. Therefore, we aggregated our method-level attributes to the file-level by taking the minimum, maximum, average, and sum values of each file [17] in order to make them compatible with the defect data.

3.2 Phase II: Defect Tracing and Matching

The second phase of our study was originally planned to store defect data, i.e. the defects found during the testing and pre-release phases. If things ran their normal course in Phase I, we would collect metrics from the completed versions of 24 projects and also match the defects.

3.2.1 Challenges during Phase II

This phase took much longer than we had anticipated. First of all, there was no process for defect tracing. Defects were not often stored during development activities. Secondly, there was no process to match the defects with the files in order to keep track of the reasons for any change in the software system.

The project team realized that it would need too much time and effort to match each defect with its file manually for such a complex system. Additionally, the developers did not volunteer to participate in this process since keeping defect reports would increase their already heavy workload.

3.2.2 Our Methodology

To solve the above-mentioned issues, we called for an emergency meeting with the senior management and the heads of the development, testing and quality teams. Consequently, the company agreed to change their existing code development process. They forced a version control log to keep changes in the source code committed by the development team. Previously, the developers did not enter any comments into the log other than the project ID while checking into the system. Therefore, we were unable to identify which changes were intended for fixing a defect. Changes in their software could either be for fixing defects or new requirement requests, all of which were uniquely numbered in the system. Whenever a developer checked in the source code to the version control system, he/she had to provide additional information about the modified file, i.e. the ID of the test defect or requirement request. Then, we would be able to retrieve those defect logs from the history and match them with the files of the projects in the same version. After the process change, we assigned defect flags, indicating 1 for defective files and 0 for defect-free files, whose code attributes were extracted by Prest.

During the adaptation of the process change, we carried out an additional analysis on static code attributes to point out some of the problems related to the coding practices of the company's development team. Our intention was to investigate the development practices in the company, mention the critical aspects and convince the team and the managers for the required course of actions. We took the best practice coding standards of NASA MDP Repository [12] and compared them with our measurements at each function level. Table 4 presents the average values of 18 static code attributes in the function level for both layers (second column), the presentation (third column) and the application (fourth column) layers in the GSM company and their minimum and maximum values (fifth and sixth columns) based on NASA standards. Attributes shaded as gray are outside the standards, which are the number of operands, the number of operators, Halstead vocabulary and the ratio of commented LOC to code LOC. Based on our analysis, we saw that there are two fundamental issues in the coding practices: a) No comments at all, which makes the source code hard to read and understand by other developers, and b) limited usage of vocabulary, i.e. the number of operator and operands are below the standards. Their development practice has been based on decreasing the complexity of the software modules. However, this leads to an unnecessarily modular system, i.e. very small modules containing very few actions.

This code analysis supports the necessity to improve the software quality. As an alternative analysis, we conducted a rule-based code analysis process based on static code attributes. Our aim here was to estimate what amount of code had to be reviewed and how much testing effort was needed to inspect the defect-prone files in the company [14]. To this end, we simply defined rules for each attribute based on its recommended minimum and maximum values in Table 4. These rules are fired if the selected attribute of a module (i.e. a functional unit) is not in the specified interval. This also indicates that the method could be defect-prone; therefore, it should be manually inspected. The results of the rule-based model can be seen in Table 5, where there are 17 basic rules with the corresponding attributes and two additional rules derived from all of the attributes. Based on 19 rules, software modules that should be investigated are presented in Table 5. The columns in Table 5 show the number of modules, i.e. functional units, which should be investigated (*Modules*), their percentage over all modules (*Module %*) as well as *LOC* and its percentage over all source code (*LOC %*) for each rule. The shaded rows in Table 5 correspond to the attribute rules that cause inspecting greater than 50% LOC. Rule #18 is fired if any of the 17 rules are fired. This rule shows that we need to inspect 100% LOC to find the defect-prone methods of the overall system. Rule #19 is fired if all basic rules except the Halstead rules are fired. This reduces the firing frequency of the former rule such that 45% of the code (341655 LOC) should be reviewed to detect the potentially problematic files in the software.

We had seen that the rule-based code review process was impractical in the sense that we would need to inspect 45% of the code [14]. Thus, we obviously need more intelligent oracles to reduce the testing effort and the defect rates in the software system.

3.3 Phase III: Defect Prediction Modeling

The original plan in this phase was to start constructing our prediction model with the data collected from the projects. We planned to test the performance of our model with those referred to in the literature. We would try different experimental designs, sampling methods, and algorithms to build such a model. In this section, we briefly explain the methodologies applied and the local prediction model proposed in this study.

3.3.1 The Proposed Model

We planned to build a learning-based defect predictor for the company. A learning based predictor follows a typical machine learning application: the model is trained with previous data such as projects whose software metrics and actual defects are known at file level, and it is tested on a new project whose defects are unknown. The learner of the model was set as a Naïve Bayes classifier since a) it is simple and robust as a machine learning technique, b) it has performed the best prediction accuracy on 8 public datasets, compared to other machine learning methods [1], and c) a recent study by Lessmann et al. has also shown that most of the machine learning algorithms are not significantly better than each other in the defect prediction domain [18]. A Naïve Bayes classifier is derived from the Bayes Theorem such that the posterior probability of an instance x being in class C_i is proportional to the prior probability of C_i and the likelihood of p(x|Ci). Then, it is normalized using evidence:

$$p(C_i|x) = \frac{p(x|C_i)\,p(C_i)}{p(x)} \tag{1}$$

In binary classification problems such as defect prediction, a Naïve Bayes classifier computes the posterior probabilities of a module as "defective" or "defect-free" given its software metrics, i.e. static code attributes. It then assigns the module x to the first class, i.e. *defective class* if its posterior probability is greater than or equal to the default threshold which is 0.5. Otherwise, the instance is assigned to the second, i.e. *defect-free*, class. This classification rule can be summarized as follows:

$$\begin{cases} C_1, & if \;\; p(C_1|x) \geq 0.5 \\ C_2, & otherwise \end{cases} \tag{2}$$

In Equation 2, C_1 refers to the *defective* class, whereas C_2 refers to the *defect-free* class. This process is rather straightforward such that it simply uses attribute probabilities (likelihood) derived from historical data to make predictions. In order to compute the likelihoods, Gaussian probability distribution function is used, where the attributes (x) are assumed as independent with different means (μ) and a common covariance matrix ($\sum$), as follows:

$$f(x) = \frac{1}{\sqrt{2\pi\Sigma}} e^{\frac{-(x-\mu)^2}{2\Sigma}} \tag{3}$$

To assess the performance of our predictor model, three measures have been proposed: the probability of detection rate, *pd*; the probability of false alarm rate, *pf*, and, the *balance* rate [13]. *Pd* measures the percentage of defective modules that are correctly classified by the predictor. *Pf,* on the other hand, is a measure to calculate the ratio of defect-free modules that are wrongly classified as defective with our predictor. Receiver Operator Characteristics (ROC) curves are often used to evaluate the performance of an algorithm in terms of hit rate, *pd,* and false alarm rate, *pf*, with varying decision thresholds [13]. In the ROC curve (Figure 2), the lower left point (0,0) in terms of (*pf,pd*) represents assigning all instances of the *defect-free* class. The upper right corner (1,1) indicates assigning all instances to the *defective* class. The upper left point (0,1) represents ideal classification, when we detect 100% of the defective modules while keeping the false alarm rate as 0%. Finally, the *balance* indicates how close our estimate is to the ideal case by calculating the Euclidean distance between the performance of our model in terms of *pf, pd* and the point *0* (pf), *1*(pd). We have computed these measures using the common confusion matrix (Table 3) and the formulas below:

$$\text{pd} = \text{TP} / (\text{TP} + \text{FN}) \tag{4}$$
$$\text{pf} = \text{FP} / (\text{FP} + \text{TN}) \tag{5}$$
$$\text{bal} = 1 - \sqrt{(0-\text{pf})^2 + (1-\text{pd})^2} \Big/ \sqrt{2} \tag{6}$$

In order to interpret our results to business managers, we also agreed to construct a cost-benefit analysis (*CB*) based on Arisholm and Briand's work [23]. This approach compares the inspection effort suggested by a defect prediction with a random testing strategy. Based on that, the reduction in testing effort can be calculated with the following formula:

$$100\,x\left(\frac{MRT - MDF}{MRT}\right) \tag{7}$$

In Equation 7, MRT represents the number of Modules (files in our study) that must be inspected through a random testing strategy, whereas MDF represents the number of Modules that must be inspected with a defect predictor. For example, if a predictor detected 70% of the defective files (MRT) by looking at 50% of all files (MDF), then the reduction in the inspection effort to find 70% of the defective files would be around 30% using the predictor.

3.3.2 Challenges during Phase III

Although we started collecting data from the completed versions of the software system, we realized that constructing such a dataset for training the predictor model would take a long time. We saw that building an immature version history is an inconsistent process. Developers could not allocate extra time to write all the defects they fixed during the testing phase because of their workload and other business priorities. In addition, matching those defects with the corresponding files of software could not be automatically handled. We could not form an effective training set for a long time. Therefore, we were not able to train our defect prediction model with the previous data from the company.

3.3.3 Our Methodology

Instead of waiting for a complete dataset, we used an alternative technique to move ahead. In our previous research, we had suggested that companies such as this GSM operator should train defect predictors with the data from other companies, i.e. cross-company data [6]. Cross-company data can be used effectively in the absence of a local data repository, especially when additional filtering techniques are used:

- Selecting similar projects from cross-company data using nearest neighbor sampling [8].
- Increasing the information content of data using dependency data between modules [22].

We selected NASA projects as the cross-company data, which are publicly available [7]. These projects are collected over a period of five years from numerous NASA contractors. They represent various systems from flight simulators to ground control systems and video guidance systems to spacecraft instruments. As Basili et al. [39] argued, conclusions from NASA data are relevant to US software development industry. Several authors argue that these projects may not be representative for the test sets in terms of the software domain where they are implemented [32]. However, in a recent study by Zimmermann et al. [32], authors investigated cross project predictions, which employed projects only from similar software domains or software processes as the training data, and found that cross-project data did not lead to accurate predictions. Therefore, we have used NASA projects without filtering them by software domains or software processes.

NASA projects contain more than 20.000 modules, of which we randomly used 90% as the training data to predict the defective modules in our projects [6]. This random split is carried out 20 times to change the projects selected for training and to avoid sampling bias during the training process. From this subset, we selected a subset of projects that were similar to those in our test data in terms of the Euclidean distance in the 17 dimensional metric spaces [8]. The nearest neighbors in this random subset were used to train the predictor. The test data consisted of local (GSM) projects from the software system of the company whose static code attributes were extracted while defects were unknown. So our analysis using cross-company data would give the estimated defect rates for the local projects.

We added one more analysis using cross-company data to increase the information content by adding dependency data between the modules of the projects. The previous research proposed by Turhan et al. [22] showed that false alarms can be reduced from 30% to 20% by using a call graph based ranking framework in a public embedded software data [7]. A call graph matrix is an *M-by-M* matrix where M is the number of files in the software system. Each entry m_{ij} of the matrix takes 1 if file i calls file j; otherwise, it is 0. The call graph based ranking framework (CGBR) approach is applicable to any static code attribute based prediction model, where static code attributes measure the intra-module complexities and call graphs models inter-module interactions in software systems. This approach is inspired by Google's web page ranking method [38], and it computes ranks for each module in the software system using call graph matrices. Modules that are popularly called (columns in the call graph matrix with the highest number of 1s) by others have the highest ranks, and these ranks are then used as weights (multipliers) for the static code attributes of all modules. We also included caller-callee relations between the modules of NASA and the GSM projects to adjust code metrics with this framework. We repeated the random split 20 times, applied call graph based ranking with new sets of NASA projects each time and raised a flag for modules that were estimated as defective in at least 10 trials [8].

Table 6 shows the results from this analysis on 24 projects in terms of estimated defect rate (%), estimated defective LOC, total LOC, and the percentage (%) of estimated defective LOC over the whole code. We were not able to analyze projects 7, 23 and 25 for our limited computational resources at the time. Indeed, we had the call graphs but were not able to analyze them because for large projects, processing NxN matrices is required, and the process becomes computationally resource-intensive, as N-number of modules gets larger. Analyses were carried out on an average laptop computer.

Results present the estimated defect rate as 8% in the software system of the company. There is a major difference with the rule-based approach in terms of its practical implications. According to the rule-based model, LOC required to inspect corresponds to 45% of the whole code, while the method level defect rate is estimated as 14%. On the other hand, according to the learning-based model, LOC required to inspect corresponds to only 2% of the code, where the method level defect rate is estimated as 8%. Therefore, we could once more see the benefits of a learning-based model to decrease testing efforts by guiding testers through defective parts of the software.

The difference between the two models occurs, since a rule-based model makes decisions based on individual metrics, and it has a bias towards more complex and larger modules. On the other hand, a learning based model combines all 'signals' from each metric and estimates defects located in smaller modules [1, 16]. It is important to mention that this analysis was completed in the absence of local data.

Therefore, we used the results to show the tangible benefits of building a defect predictor to the managers and the development team in the company.

3.4 Phase IV: Defect Prediction Extended

This phase did not exist in our original plan since we underestimated the time and effort that was necessary to build a local data repository. In this phase, we were able to collect within-company data, i.e. static code attributes and local defect data, from ten previous versions of the software. The properties of the projects in terms of the number of files and defect rates are illustrated in Table 7. These data are now publicly available for other researchers to reproduce, refute and improve our results [7]. We observed discontinuities in projects between different releases perhaps because some projects were rarely deployed or withdrawn at a certain release. Starting from this phase, we trained the prediction model using the project data of the company from previous versions and tested it on the next version. We decided on the experimental design of the local prediction model in Phase IV. First, we investigated the amount of data, i.e. the ratio between defective and defect-free modules, which was needed to establish predictions. We chose the "micro-sampling" approach based on the study done by Turhan et al. [3]. Micro-sampling approach is a special case of under-sampling, where the number of the defect-free instances in the training set is reduced to the number of defective instances. The results of the study [3] have shown that more than 50 samples of which 25 are defective and 25 are defect-free, do not improve the performance of the algorithms. Therefore, we formed all the training sets with size M=2N, with equal N number of defective and defect-free instances. Secondly, we investigated the content of training data in the model to predict the defective files of the projects in the next release. We conducted two experiments on GSM3 and GSM4 projects to decide on the best strategy:

- We applied micro-sampling on the latest previous release of each project, i.e. the latest release in the scope of which a project was deployed and its metrics as well as defect data were collected in order to form the training set.
- We applied micro-sampling to the latest previous release of all projects as the training set [14].

GSM3 and GSM4 projects were selected, since these two projects were the ones whose defect data were available in the majority of the 10 releases to form the training set. The test data are the current release of that project such that its development phase has just been completed. Table 8 shows a sample of two versions and two projects. The third column (version-level) presents the prediction performance of the first experiment, when we choose our training set from all eight projects of the previous version to predict the defective modules of projects GSM3 and GSM4. We conducted 100 iterations to randomly select N defect-free files from all the projects in the previous release for the training set. We took the average performance of 100 iterations. The last column (project-level), on the other hand, shows the performance of the second experiment, when we used only the previous version of GSM3 or GSM4 to predict the defective modules of the selected project in the current version. It is observed that both of the approaches produce high *pd* rates in the range from 78% to 100%. Results show that the project level defect predictor would be better (bold cells in Table 8) although we had high false alarm rates. Therefore, we used project-level defect prediction for the following experiments.

3.4.1 Challenges during Phase IV

The results of this analysis presented in Table 8 show that we still produced high false alarms by selecting the training data from the previous versions of the specific project only, compared to the results when training data were selected from the previous versions of all projects included in the software. False alarms are critical for such predictors because they cause developers or testers to inspect more modules than necessary. This, in fact, contradicts with one of the initial aims of constructing a defect predictor: decreasing the testing effort. Since false alarms burden the test team with additional costs, it is hard to adopt our predictor to their actual development practices. Therefore, we had to find a strategy to detect as much defective modules as possible while decreasing false alarms to a reasonable cost level.

3.4.2 Our Methodology

We discussed the causes of high false alarms in our monthly meetings with the project team and found that we needed to remove files that had not been changed for the last six months from the version history, i.e. January 2008 for the available dataset. The simple intuition behind this is as follows: most of the files in the software system are not edited much since they are either library files or they constitute the core assets of the system. Improvements or new functionalities of the projects in every release often take place on a small portion of the files, thereby leaving most of the files unchanged. Including all files of the previous release to build the training set disturbs the learning strategy of the model by damaging the information content of data. We built a simple assumption on defect-proneness of a module in order to prevent this:

- *"It is highly probable that a module is defect-free if it has not been changed since January 2008."*

Then, we added a flag to each file of the projects to indicate whether the file had been actively changed or remained passive since January. The model controls each of its predictions by looking at the history flag of these files. If the model predicts a file as defective although it has not been used since January, then it is re-classified as defect-free. This approach was incredibly simple and successful in comparison to using churn as an additional metric to our model. Furthermore, matching churn and static code attributes was not feasible for most of the files due to low change rates. We decided upon a period of *six months* by trying out three possible time periods for extracting version flags:

3 months, 6 months and a year. Among these time periods, using a six months period to extract version flags provided the best prediction performance for all projects.
The results of our experiments using only code metrics (Model I) and using code metrics along with history flags (Model II) are summarized in Table 9 for all public datasets. We can clearly observe that using version history improves the predictions significantly in terms of *pf* rates. Our model succeeded in decreasing false alarms from 50% to 28% on average using version history. The change in *pf* rates varies in terms of projects in the range of {0%, 63%} due to discontinuities in the changed projects throughout the version history. In addition, we managed to have stable high *pd* rates on an average of 88% while reducing *pf* rates successfully. Furthermore, we spent less effort to detect 88% of these defective modules: the cost-benefit analysis (CB column in Table 9) shows that we managed to decrease the inspection effort to detect defective modules by 72%, decreasing from 88% to 25%. As a result, using a defect prediction model enables developers to allocate their limited amount of time and effort to only defect-prone parts of a system. Managers can also see the practical implications of such decision-making tools, which reduce the testing effort and cost.

4. CALIBRATION OF THE FINAL MODEL

We successfully built our defect predictor for the company using local data and presented our results to the project team. The results of the project show that the company's business goal of decreasing the testing effort without compromising on the level of product quality can be achieved through intelligent oracles. We used two different methods to calibrate the final model for the company in order to improve the prediction performance. However, in our prior publication [28], we had faced challenges during the adaptation of these methods. First of all, the file-level call graph based ranking (CGBR) method did not work due to the transition to Service Oriented Architecture (SOA). SOA does not allow us to capture caller-callee relations through simple and static file interactions. Secondly, we only used static code attributes from Java files to build our model. However, there are many JSP files and PL/ SQL scripts that contain very critical information on the interactions between the application and data layers. Thus, a simple call-graph based ranking at file-level could not capture the overall picture and consequently failed to increase the information content in our study.
In this paper, we have extended our prior publication [28] by pointing out these two important issues. Firstly, we increased the information content of the software data by producing SOA-based dependency data. Secondly, we examined the algorithm of our predictor Naïve Bayes. We had recommended spending less time on algorithms when working with large organizations. Thus, we have slightly modified a Naïve Bayes classifier to improve its prediction performance rather than trying more complex algorithms. In this section, we describe these extensions in detail.

4.1 Improving the Information Content of Software Data

We considered two different approaches to improve the quality of defect prediction studies. The first one is increasing the information content of the training data [3]. The second one is using more appropriate algorithms in order to improve the prediction performance [15]. We selected the first approach in our prior experiments so far since the information content of static code attributes is so limited that more complex algorithms would not detect new information [3]. Using history flags, we successfully achieved better prediction rates in terms of *pd* and *pf* when compared to defect prediction solely through static code attributes [28]. However, we did not include all data available for the projects, such as the static code attributes and the defect information from Jsp files and SOA-based dependency data. Therefore, two analyses on increasing the information content of data were performed: a) information retrieval from Jsp files, and b) call graph generation from SOA. Since Prest is capable of extracting static code attributes from Jsp files, we have also collected defect data for Jsp files of the two sample projects, GSM2 and GSM3 during releases 11 and 12. Our experiments show that using both Java and Jsp files increase the information content as well as the defect data for the training set, thereby improving the defect prediction performance of our model [24].

4.1.1 Challenges

During Phase IV, we had difficulties in capturing caller-callee relations between SOA modules and/or the services of the GSM company. Prest has the ability to extract call graphs from C, C++, Java, and PL/SQL source code files. However, these call graphs are built only by checking the static source codes. Service Oriented Architectures provide a new way of programming allowing a call inside the source code to initiate a service outside the system or to trigger a file read/write operation [33]. Although it seems that we are able to extract call graphs from static source code files using Prest, it is not valid for such technologies, where we have to reveal all components at run time. In other words, the communication between two or more services coordinating a certain activity cannot be seen in the generated call graphs. These services may run on different servers that are not accessible. They may be implemented in a programming language that is not supported by Prest, or calls may not be extracted from the source code. Therefore, if we use the call graphs extracted from the source code in service oriented architectures, we cannot improve the performance of defect predictors.
Several dynamic call graph generation tools [34, 35, 36], which can track components at run time, exist. However these tools have certain constraints: a) they support a limited number of web services, b) they need a source code and running configurations, c) some of them are platform and instruction set dependent, and d) most of them have JVM dependency. It is necessary to adopt Prest to the new method of programming the company adopted, i.e. SOA, where the communication between two or more services could not be captured using static file interactions. Therefore, we were not able to extract call graphs from the software system as easily as expected.

4.1.2 Our Methodology

In order to overcome these problems, we took advantage of a Configuration Management Database (CMDB) in the GSM company, which is a comprehensive application tool that has been implemented for the last six months. We implemented an add-on to Prest that could extract call graphs from Configuration Management Databases. The Configuration Management Database (CMDB) is a repository where all the entities and their relationships to a system are stored. CMDB allows us to understand and track the components, such as services, their release time and churn data about any file in the software system easily and effectively. This database includes attributes of each component in the system such as service name, release number and storage time. In addition, every component is connected to the others by useful information such as the parent level and the application depth. Checking the status of running applications and tracking the defects from the lowest level subroutine to the source are the basic properties of CMDB. For SOA, we can generate a comprehensive call graph that covers every interaction between the components by using CMDB. Contrary to call graph extraction from static source codes, we can build a call graph with all actions executed at run time such as database triggers and remote method invocations.

We extracted call graphs of the GSM4 project using information from CMDB. We implemented a new add-on to Prest that takes the parent-child relations from CMDB and converts them to call graph matrices in order to feed them into our prediction model. Employing a call graph based ranking framework, we predicted the defective files of the GSM4 project for the release 15. The final results of the call graph experiments can be seen in Figure 3. The first chart in Figure 3 presents the prediction performance in terms of *pd*, *pf*, and *balance* using static code attributes from Java and Jsp files. The second chart shows that the performance does not improve when a file-level call graph based framework is used. The final chart represents a decrease of 8% in the *pf* rates when the training data is enriched with SOA call graphs. This approach has a limitation, and that is the use of CMDB. We extracted call graphs by using the relations from CMDB. Therefore, it is only applicable to companies that store all entities and their relationships via CMDB. However, the approach is not limited to the GSM company. It can be easily applied to the data of other companies.

4.2 Adjusting the Decision Threshold of a Naïve Bayes Classifier

During this case study, we plan to focus on software data rather than the algorithm. However, the recent results obtained with Jsp files and SOA call graphs have led us to define new research directions. We have decided to examine the algorithm of our defect prediction model in order to adjust its hyper-parameters.

4.2.1 Challenges

We see that adding new data would definitely improve the performance of our defect predictor. However, collecting new data was very difficult and time-consuming. Furthermore, we were still working on software projects whose defect rates were less than 1%, since defect-file matching has not been fully accepted by the development team[1]. Therefore, the software data has an imbalanced nature with very few defective modules compared to defect-free ones. We tried the micro-sampling technique to overcome the problem of imbalanced data. However, there are ongoing discussions in the machine learning community on whether sampling techniques would solve the problem of imbalanced datasets. According to Provost [26], using standard machine learning algorithms without adjusting their decision threshold may be a critical mistake. On the other hand, Maloof examined the decision thresholds of simple AI techniques and different sampling ratios on datasets with imbalanced class distributions [25]. The results show that varying either of these is equivalent, but the precise conclusion is more complex and domain-specific.

4.2.2 Our Methodology

We also studied decision threshold optimization using ROC curves with 13 public datasets [27]. We adjusted the decision threshold of a Naïve Bayes classifier for each dataset using cross validation on training data and plotted ROC curves to illustrate the optimum threshold. Our results reveal that threshold optimization significantly decreases the *pf* rates by 11% on average, while keeping *pd* rates stable.

In this study, we examined GSM data by adjusting the decision threshold of a Naïve Bayes classifier in our defect prediction model, and we compared the results with over- and under-sampling strategies. We conducted five different experiments to analyze the software data collected so far: a) using the original training set, b) using under-sampling strategy, c) using oversampling strategy, d) using decision-threshold optimization, and e) using version history with an optimum decision threshold. First of all, the original training set was used in the model without modifying class distributions. Secondly, under-sampling was applied on the training set by selecting the number of defect-free instances given a certain ratio, i.e. {10, 15, 20 … 100} %, in order to obtain the best defective vs. defect-free ratio. Since micro-sampling is a special case of under-sampling, which decreases the number of defect-free modules to a level equal to the number of defective modules, we had to also assess whether micro-sampling is the best sampling strategy for GSM data. In the third experiment, we conversely increased the number of defective instances in the training set given a certain ratio, i.e. {10, 15, 20 … 100} %. To determine the best defective vs. defect-free ratio, we randomly added a number of defective instances (according to the specified ratio) to the training set. In both sampling approaches, we gradually decreased or increased the defective vs. defect-free ratio so that we could find the best result with the highest prediction rates (pd, pf, bal).

In addition to sampling experiments, we applied decision threshold optimization on GSM data and adjusted the performance of this experiment using history information. To this end, we focused on the algorithm of our defect predictor – a Naïve Bayes classifier. When

[1] The new process can be overridden by entering any text into the log message. A text mining is out of the scope to determine whether it is an existing defect ID or not. However, managers make periodical checks for the correctness of these log messages entered by developers.

using a Naïve Bayes classifier, we calculated the *pd* and *pf* rates using a default threshold, i.e. 0.5, which mapped to a single point on the ROC curve. Since software data have an imbalanced nature, we sought to assess the performance of our defect predictor for varying thresholds in the range 0-1. We changed the classification rule of the Naïve Bayes classifier as follows:

$$\begin{cases} C_1, & \text{if} \quad p(C_1|x) \geq t \\ C_2, & \text{otherwise} \end{cases} \tag{8}$$

Different *pd* and *pf* values are calculated for threshold *t*, and the resulting set is plotted as a 2D ROC curve in terms of *pf, pd*. Our aim is to find the best threshold that is the closest to the ideal point on the ROC curve. We computed the distance between each *pf, pd* pair and *0,1* using Equation 9. In order to find the optimum decision threshold, we have selected the *pf, pd* pair that gives the minimum distance.

$$\text{distance(pd,pf)} = \sqrt{(1-pd)^2 + (0-pf)^2} \tag{9}$$

For GSM datasets, we initially used the default threshold of the Naïve Bayes classifier to predict the defect-prone files of a project's current release and computed the performance measures. Then, we iteratively changed the decision threshold within a range between 0.01 and 0.99 and computed the performance measures for each threshold value. We chose the best threshold that would give the highest prediction accuracy for each project. As the last step, we adjusted our prediction by using both the optimum decision threshold for the Naïve Bayes classifier and the version history. In other words, we found the best threshold with the highest accuracy for a project and adjusted the predictions of the model using history flags to further reduce the false alarms. We have computed *pd, pf* rates for all five experiments consecutively using GSM data. All results obtained from these five experiments are illustrated in Table 10.
For each dataset, the third and fourth columns in Table 10, i.e. *pd* and *pf* respectively, are the prediction performances of our defect predictor using the original training set with the default threshold of the Naïve Bayes classifier. *Pd, pf* pairs of under-sampling results with their optimal sampling ratio for each dataset, r_{under}, can be seen between the fifth and the seventh columns of Table 10. Similarly, we presented the performance of the oversampling technique in terms of *pd, pf* pairs and the optimum sampling ratio, r_{over}, between the eighth and the tenth columns of Table 10.
Decision threshold optimization was applied as the forth experiment. We used the optimum decision threshold of the classifier t_{opt} for each dataset and presented the resulting *pd, pf* pairs in the eleventh and the twelfth columns of Table 10. Finally, we trained a Naïve Bayes algorithm with the optimum decision threshold using code metrics and the version history, and we presented the results in the last two columns of Table 10.
As seen in Table 10, different thresholds and sampling ratios have been selected for each project based on its current release. That is, the optimum threshold of a project for release *n+1* totally depends on the project data used in release *n*. In other words, the optimum threshold of a project for release *n+1* may change when the model is trained using the data from release *n-1*. We also plotted the change in *pd* rates (Figure 4) and the change in *pf* rates (Figure 5) in order to visualize the effects of each experiment. Figures 4 and 5 clearly present the improved prediction performance based on using decision threshold optimization. Although it seems that detection rates have not statistically changed in any of these experiments, we conducted pair-wise t-tests with 95% confidence between five different false alarm rates. Based on these tests, the decision threshold optimization significantly reduces the number of false alarms in comparison to the original training data over- and under-sampling techniques. However, the approach based on using the optimum decision threshold and the version history outperforms all in terms of false alarms, i.e. reduction by 12% from the fourth to the fifth experiment. Thus, the decision threshold optimization on code metrics using version history information produces the best performance ever, reducing the number of false alarms by 30% in comparison to a Naïve Bayes classifier with default threshold on code metrics.

5. THREATS TO VALIDITY

Like all other empirical studies, the data, methodologies and the algorithm must have influenced our study. First of all, the reasons for using a Naïve Bayes classifier are extensively discussed in Section 3.3.1. We used Naïve Bayes because it is easy to implement, robust and identical in terms of performance measures compared to other machine learning algorithms.
In this study, we have built a prediction model that is calibrated according to the problems of local data. Therefore, we intend to generalize neither the results nor the model. We have presented our experiences in terms of problems and proposed methodologies to provide insight to other researchers who will work on building such defect predictors in other companies. Furthermore, we did not propose new methodologies, but we applied already published techniques such as using cross-company data and sampling. Therefore, we do not claim that our model is a generic in terms of defect prediction research.
Applied methodologies are already discussed in the literature in terms of their validity. We applied cross company data when the company did not have local data to train the model. This is very practical since we did not waste time during the evolution of this project. In our experiments, the training and test sets contained very few defected files, and this imbalanced ratio between defective and defect-free modules in software data is a major threat. We utilized sampling strategies to avoid this threat. Sampling strategies are well known techniques that deal with imbalanced or limited data problems. From the practitioners' perspective, instead of waiting for collecting a more balanced local data, various sampling strategies can be applied to get immediate prediction results. Sampling strategies introduce some bias on data since we controlled the class distributions of the training data in order to increase the prediction performance on the test data. However, we made several repetitions during sampling experiments to avoid sampling bias.

We used probability of detection (PD) and probability of false alarm rates (PF) to assess the performance of our proposed model. Some of the researchers in defect prediction studies have been using *precision* instead of *pf* [10, 11]. Measuring *pf* helps researchers to evaluate the additional costs our predictor would cause. In defect prediction research, *precision* is also used to assess the performance of such models instead of the *pf* rate [2, 10, 11]. Since both *precision* and *pd* assess the number of the defective modules that are actually detected by the predictor, we preferred to use *pd* only. However, based on the signal theory, when such predictor models are triggered often to increase *pd, pf* rates would increase in turn [1]. High *pf* rates, on the other hand, indicate that we unnecessarily highlight the safe modules in the software system and waste additional amount of time for testing them. Therefore, our objectives in building such predictors are increasing *pd* as much as possible while avoiding high *pf* rates.

6. LESSONS LEARNED

During this project we had many challenges to overcome, and we constantly redefined our processes and planned for new sets of actions. In this section, we would like to discuss what can be used as best practices and what needs to be avoided in the future. We hope that this study and our self-evaluation will shed some light for other researchers and practitioners.

6.1 Best Practices

Managerial Support: From the beginning till the end of this work, we had the full support of the senior management as well as the mid-level management. They were available and ready to help whenever we needed them. We believe that without such a support, a project like this would not have been concluded successfully.

Project planning and monitoring: One of the critical success factors was having a detailed project plan, and rigorously following and monitoring that plan. This enabled us to identify problems early on and to take the necessary precautions on time. Although we faced many challenges, we were able to finish the project on time, achieving and extending its intended goals. Weekly and monthly meetings also brought up new and creative research ideas. As the research team, we mapped the project plan and its deliverables to new research topics and academic studies. Moreover, the company has also gained valuable outcomes, which are described in the next best practice, i.e. "multiplier effect".

The Multiplier Effect: One of the benefits of doing research in an industrial setting such as this GSM company is that researchers can work on-site, access massive amounts of data, conduct many experiments, and produce a lot of results. This attitude toward research provided benefit to the GSM company as well, enabling them to obtain more output, which influenced five other process areas in addition to those originally planned. It is definitely a win-win situation. Although our initially stated goal was to conduct a measurement and defect prediction study focusing only the testing stage, this project has had an influence on all other stages of SDLC: 1) *the design phase* by using dependencies between the modules of the software system, 2) *the coding phase* by adding static code measurement, raw code analysis and a rule-based model, 3) *the coding phase* by employing a sample test-driven development to measure the effectiveness of TDD on reducing the defect rates, 4) *the testing phase* by building a defect predictor to decrease the testing effort, and finally, 5) *the maintenance phase* by pointing out the complex modules that need to be refactored in the following release.

Existence of Well Defined Project Life Cycles and Roles/Responsibilities: The development lifecycle in the company is arranged in a way that all stages, i.e. requirements, design, coding, testing and maintenance are separately assigned to different groups in the team. Therefore, the segregation of duties is successfully operated in the company. We benefited from this organizational structure while working on our project. Contacting the test team to obtain the defect data and the development team to take measurements from the source code was easy since the teams and their responsibilities were well defined.

Mature Relationship with the Company: Because we have collaborated with the GSM company for more than two years, it was easier to communicate with the software teams including managers, developers, testers, and to access various kinds of data that were necessary for our further analysis. We started to comprehend the cause-effect relationships among various factors and evaluated our findings based on this type of company-related data. We did additional analysis on the development phase by collecting churn data, extracting churn metrics and analyzing statistical correlations between version control commits and the complexity of software projects. Furthermore, we focused on the people factor throughout the development lifecycle and started measuring cognitive bias. Cognitive bias can be defined as a person's tendency to make errors in judgment. Recent empirical software engineering research investigates the effect of personality on inserting defects into the software system by looking at various psychological factors [37]. We are applying several similar tests to measure the bias of the developers in the company.

Learners of the Model: We selected a simple and robust technique, namely Naïve Bayes, for our defect prediction model since we planned to keep the focus of this study on challenges in large organizations rather than the performance of our predictor's algorithm. It is possible to achieve a detection rate of 88% on average with a false alarm rate of 28% using the Naïve Bayes classifier. Within the scope of our research, we have also adjusted the decision threshold of the Naïve Bayes classifier. According to a study by Lessmann et al. [18], most of the machine learning algorithms do not differ significantly from one another in terms of defect prediction. Following this view, one may conclude that the selection of the classifier is less important than generally assumed and practitioners are free to choose from a broad set of candidates when building defect predictors. We have also found that simple algorithms such as Naïve Bayes are practical for building local predictors in the industry. They are easy for practitioners to understand; they do not require parameter optimization such as the number of hidden layers in multi-layer perceptrons; and, they are robust in the face of changes such as decision threshold optimization, which decreases the false alarms from 26% to 14% on average. Therefore, we also recommend applying a simple and robust technique such as a Naïve Bayes classifier during the construction of a defect prediction program in large companies.

6.2 Things to Avoid in the Future

Lack of Tool Support: Automated tool support for measurement and analysis is fundamental for this type of projects. In this project, we have developed a metrics extraction tool to collect code metrics easily. However, we were unable to match the defects with the corresponding files. Therefore, it took too much time to construct a local defect prediction model. Our next plan would definitely be initiating an automated defect tracing/matching mechanism with the company. Thus, we highly recommend that before a similar project starts, an automated tool support for defect collection and matching be employed.

Lack of Documentation and Architectural Complexity: Large and complex systems have distinguishing characteristics. Therefore, proper documentation is paramount to understand the complexities especially when critical milestones are defined at every stage of such a project. Lack of documentation has led to many challenges as we moved along, forcing us to change our plans several times.

Lack of On-Site Presence: During the beginning of this collaboration, we thought that weekly or bi-weekly meetings would be enough to receive and analyze all data in detail. Unfortunately, it did not happen as initially planned because the required data could not be collected accurately and on time. Thus, it took much time to collect the necessary data without wasting the limited time of the software team. We solved this problem by working on-site twice a week with a permanent desk and PC allocated to us and having read-only accesses to version control systems and databases.

7. CONCLUSION

Decision making is a critical business problem that managers in various industries have to deal with. Our research was an empirical study where we collected data, designed experiments, and then presented and evaluated the results of these experiments. Contrary to classical machine learning applications, we have focused on better understanding the data at hand. This case study provides a live laboratory environment that was necessary to achieve this goal. As always, both sides of the equation (academia and practice) must be ambitious to reach the desired goals. Our empirical results show that a metrics program can be built in a period of less than a year: as few as 100 data points are good enough to train the model [3]. In the meantime, the company can use cross-company data to predict defects by using simple filtering techniques. Once a local repository is built and version history information is used, we will be able to compare our prediction with real defect data and show that it detects 88% of the defective modules with 28% of false alarms. Since false alarm rates are very high compared to the average performances seen on NASA datasets (75%, 21%) in terms of *pd, pf*, in this study, we focused on the algorithm and adjusted the hyper-parameter of the Naïve Bayes classifier to set the optimum decision threshold for GSM projects. With decision threshold optimization, we produced a detection rate of 86% with 14% false alarms. Therefore, we recommend that other researchers, who will collaborate with large organizations on defect prediction modeling, select a simple classifier like Naïve Bayes and adjust its hyper-parameter before choosing a more complex classifier or an ensemble of classifiers.

The main discussion related to our recent findings is to answer the question, "How to apply threshold optimization in practical defect prediction?" Since we adjusted the decision threshold of the Naïve Bayes classifier depending on the training set in each release, the optimal decision threshold for a single project or a single release has not been discovered yet. We have alternatives for selecting the best decision threshold for a given project: cross validation on the training set, taking the average of the optimum thresholds of the latest n releases, or a dynamic decision threshold prediction. Even though the last alternative gives the best result in practice, we have not found a way of achieving it yet. Therefore, it is one of our future research directions in this study to define the optimum decision threshold in practice. Moreover, we plan to add PL/SQL defect data and code metrics for further increasing the information content of the training data.

Currently, we are working on calibrating the local defect predictor in the GSM company in order to conduct real-time predictions based on real data, i.e., when the implementation of the software has just been completed and the testing phase has begun. We are trying to integrate this predictor to their testing practices so that they benefit from the predictions, which would expect to detect 86% of actual defective modules on average and allocate their time to those critical parts. Prest has been slightly modified to serve for preprocessing techniques such as SOA call graph generation, log filtering, decision threshold adjustment, and automated defect matching. Finally, a member of the software team in the company has been assigned for integrating Prest into their development lifecycle. Based on our initial observations, predictions with Prest guide the testers toward actual defective modules. We have selected the GSM4 project for an initial prediction study and found that we can correctly classify defective files (90% of them are found) by eliminating files that have not been changed for longer than six months. Currently, the company is conducting process improvement activities in the course of which they will check the estimated defect rate of a project before it moves on to the testing phase as well as other factors related to the analysis and design phases. If the defect rate is higher than a predefined threshold, then a thorough code review will be done for this project before it is sent for testing.

We know that such metrics programs are well conducted in many companies such as Motorola [20], Microsoft [10, 11] and AT&T [2]. In such studies, learning-based approaches are often employed with process metrics that add more value on the personal aspects of the development team, churn metrics related with the version history and the process maturity of the development practices. Therefore, one of our research directions is to broaden this study with new metrics and new techniques that improve the performance of the algorithm, to conduct surveys, and to compare what we have done so far by adopting these approaches.

8. ACKNOWLEDGMENT

This research is supported in part by Turkish Scientific Research Council, TUBITAK, under grant number EEEAG108E014, and Turkcell A.Ş.

9. REFERENCES

[1] Menzies, T., Greenwald, J., Frank, A. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. IEEE Transactions on Software Engineering, vol.33, no.1 (January 2007), 2-13.

[2] Ostrand, T.J., Weyuker E.J., Bell, R.M. 2005. Predicting the Location and Number of Faults in Large Software Systems. IEEE Transactions on Software Engineering, vol.31, no.4 (April 2005), 340-355.

[3] Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., and Jiang, Y. 2008. Implications of Ceiling Effects in Defect Predictors. in the Proceedings of PROMISE 2008 Workshop, Germany.

[4] Halstead, H.M. 1977. Elements of Software Science. Elsevier, New York.

[5] McCabe, T. 1976. A Complexity Measure. IEEE Transactions on Software Engineering. vol.2, no.4, 308-320.

[6] Turhan, B., Menzies, T., Bener, A., Distefano, J. 2009. On the Relative Value of Cross-company and Within-Company Data for Defect Prediction. Empirical Software Engineering Journal (January 2009), DOI: 10.1007/s10664-008-9103-7.

[7] Boetticher, G., Menzies, T., Ostrand, T. 2007. PROMISE Repository of empirical software engineering data. http://promisedata.org/repository. West Virginia University, Department of Computer Science, 2007.

[8] Turhan, B., Bener, A., Menzies, T. 2008. Nearest Neighbor Sampling for Cross Company Defect Prediction. In Proceedings of the 1st International Workshop on Defects in Large Software Systems, DEFECTS 2008, 26.

[9] Prest. 2009. Department of Computer Engineering, Bogazici University, http://code.google.com/p/prest/

[10] Nagappan, N., Ball, T., Murphy, B. 2006. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. In Proceedings of the International Symposium on Software Reliability Engineering, NC, November 2006.

[11] Nagappan, N., Murphy, B., Basili, V. 2008. The Influence of Organizational Structure on Software Quality. in Proceedings of the International Conference on Software Engineering, Germany, May 2008.

[12] NASA WVU IV & V Facility, Metrics Program. 2004. http://mdp.ivv.nasa.gov

[13] Heeger, D. 1998. Signal Detection Theory. http://white.stanford.edu//~heeger/sdt/sdt.html

[14] Tosun, A., Turhan, B., Bener, A. 2008. Direct and Indirect Effects of Software Defect Predictors on Development Lifecycle: An Industrial Case Study. in Proceedings of the 19th International Symposium on Software Reliability Engineering, Seattle, USA, November 2008.

[15] Tosun, A., Turhan, B., Bener, A. 2008. Ensemble of Software Defect Predictors: A Case Study. In Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement, Germany, October 2008, 318-320.

[16] Turhan, B., Bener, A. 2009. Analysis of Naïve Bayes' Assumptions on Software Fault Data: An Empirical Study. Data and Knowledge Engineering Journal, vol.68, no.2, 278-290.

[17] Koru, G., Liu, H. 2007. Building effective defect prediction models in practice. IEEE Software, 23-29.

[18] Lessmann, S., Baesens, B., Mues, C., Pietsch, S. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. IEEE Transactions on Software Engineering, vol.34, no.4, July/August 2008, 1-12.

[19] Chidamber, S.R., Kemerer, C.F. 1994. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, vol.20, no.6, 476-493.

[20] Fenton, N.E., Neil, M., Marsh, W., Hearty, P., Radlinski, L., and Krause, P. 2008. On the effectiveness of early life cycle defect prediction with Bayesian Nets. Empirical Software Engineering, vol.13, 2008, 499-537.

[21] Ngo-The, A., Ruhe, G. 2009. Optimized Resource Allocation for Software Release Planning. IEEE Transactions on Software Engineering, vol.35, no.1, Jan/Feb 2009, 109-123.

[22] Turhan, B., Kocak, G., Bener, A. 2008. Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework. In Proceedings of the 34th EUROMICRO Software Engineering and Advanced Applications (EUROMICRO-SEAA), 2008, pp.191-198.

[23] Arisholm, E., Briand, C.L. 2006. Predicting fault prone components in a Java legacy system. In Proceedings of ISESE'06, 1-22.

[24] Kocaguneli, E., Tosun, A., Bener, A., Turhan, B., Caglayan, B. 2009. Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool. in Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE), Boston, USA, 2009.

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

[26] Provost, F. 2000. Machine learning from imbalanced data sets 101. In Proceeding of Working Notes AAAI00 Workshop Learning from Imbalanced Data Sets, 2000, 1-3.

[27] Tosun, A., Bener, A. 2009. Reducing False Alarms in Software Defect Prediction by Decision Threshold Optimization. In Proceedings of the 20th International Symposium on Empirical Software Engineering and Measurement, Orlando, Florida, USA, October 2009.

[28] Tosun, A., Bener, A., Turhan, B. 2009. Practical Considerations in Deploying AI for Defect Prediction: A Case Study Within the Turkish Telecommunication Industry. In Proceedings of the 5th International Conference on Predictor Models in Software Engineering, Vancouver, Canada, May 2009.

[29] Adrian, R.W., Branstad, A. M., Cherniavsky, C. J., 1982. Validation, Verification and Testing of Computer Software, ACM Computing Surveys, vol.14, no.22, pp.159-192.

[30] IEEE Standards Association, 2007. IEEE Standards Glossary of Software Enginering Terminology, http://standards.ieee.org/reading/ieee/std/se/610.12-1990.pdf.

[31] Fenton, N., Neil, M., 1999. A Critique of Software Defect Prediction Models, IEEE Transactions on Software Engineering, vol. 25, no.5, pp.675-689.

[32] Zimmermann, T., Nagappan, N., Gall, H., Giger, E., Murphy, B., 2009. Cross-Project Defect Prediction, In proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM/SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), p. 91-100, August 2009, Amsterdam, Netherlands.

[33] Dart, S., 1991. Concepts in configuration management systems, In proceedings of the 3rd international workshop on Software configuration management, p.1-18, June 12-14, 1991, Trondheim, Norway.

[34] Eclipse Test and Performance Tools. http://www.eclipse.org/tptp/home/.

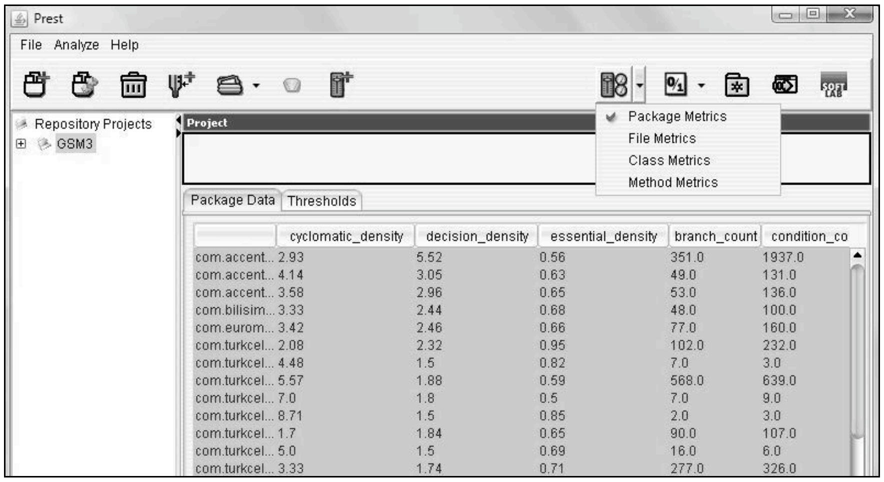
[35] Java Path Finder. http://javapathfinder.sourceforge.net/.

[36] Jinsight. http://www.research.ibm.com/jinsight/docs/.

[37] Salleh, N., Mendes, E., Grundy, J., Burch, G., 2009. An Empirical Study on the Effects of Personality in Pair Programming using Five-Factor Model, In proceedings of the 20th International Symposium on Empirical Software Engineering and Measurement, p.214-225, 15-16 October 2009, Orlando, USA.

[38] Brin, S., Page, L. 1998. The anatomy of a large-scale hypertextual Web search engine, Computer Networks and ISDN Systems, vol. 33, pp. 107-17. Also available online at http://infolab.stanford.edu/pub/papers/google.pdf

[39] Basili V., McGarry F., Pajerski R., Zelkowitz M. 2002. Lessons learned from 25 years of process improvement: the rise and fall of the NASA software engineering laboratory, in Proceedings of the 24th international conference on software engineering (ICSE) 2002, Orlando, Florida.



	cyclomatic_density	decision_density	essential_density	branch_count	condition_co
com.accent...	2.93	5.52	0.56	351.0	1937.0
com.accent...	4.14	3.05	0.63	49.0	131.0
com.accent...	3.58	2.96	0.65	53.0	136.0
com.bilisim...	3.33	2.44	0.68	48.0	100.0
com.eurom...	3.42	2.46	0.66	77.0	160.0
com.turkcel...	2.08	2.32	0.95	102.0	232.0
com.turkcel...	4.48	1.5	0.82	7.0	3.0
com.turkcel...	5.57	1.88	0.59	568.0	639.0
com.turkcel...	7.0	1.8	0.5	7.0	9.0
com.turkcel...	8.71	1.5	0.85	2.0	3.0
com.turkcel...	1.7	1.84	0.65	90.0	107.0
com.turkcel...	5.0	1.5	0.69	16.0	6.0
com.turkcel...	3.33	1.74	0.71	277.0	326.0

Figure 1. Screenshot from Prest: Displaying results at package level.

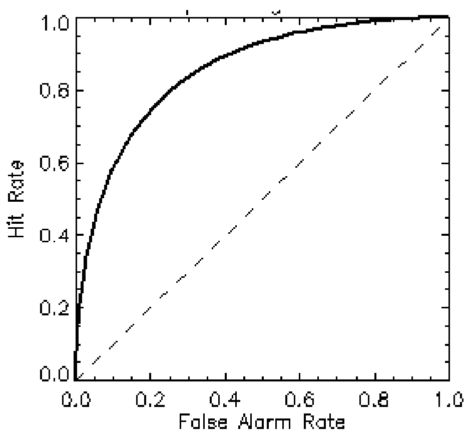


Figure 2. An example of a ROC curve

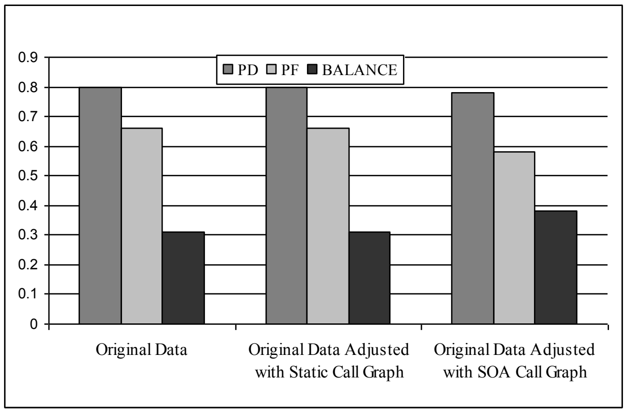
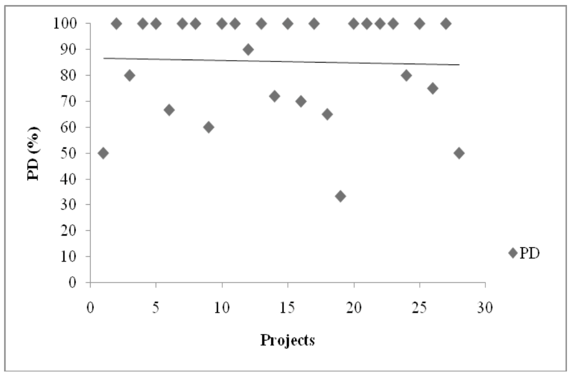
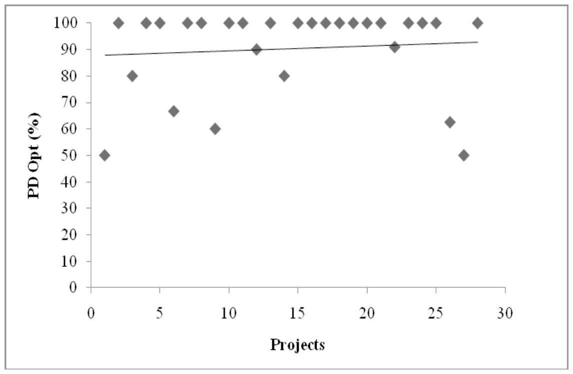


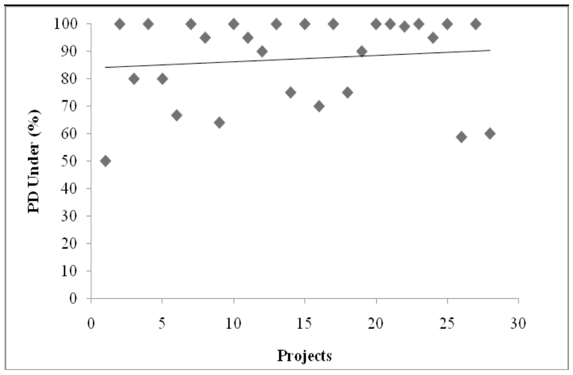
Figure 3. SOA Call graph analysis



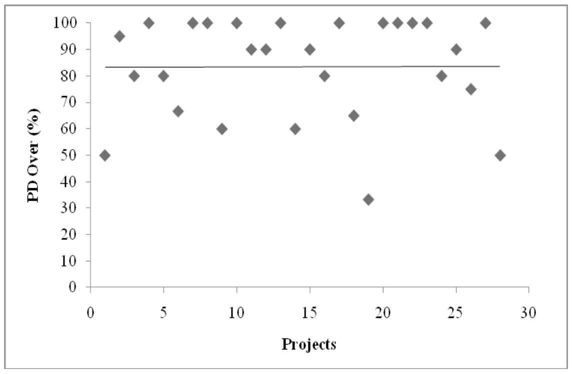
a) Using the original training set (PD)



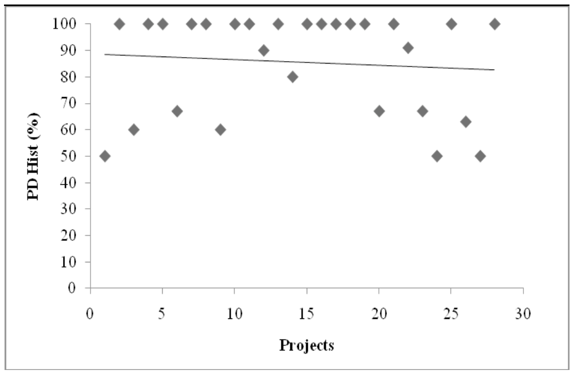
b) Using decision threshold optimization (PD Opt)



c) Using under-sampling (PD Under)

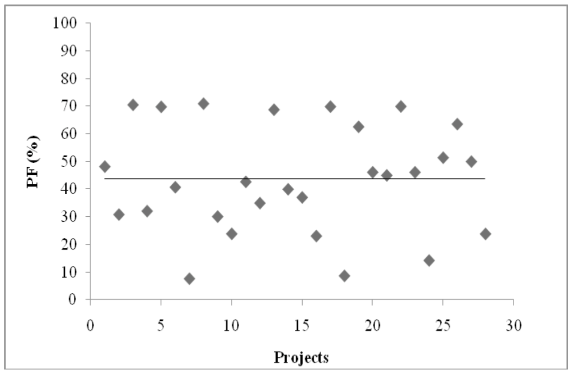


d) Using over-sampling (PD Over)

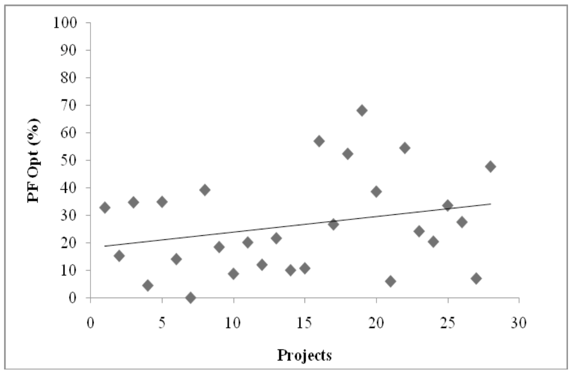


e) Using version history with optimum decision threshold (PD Hist)

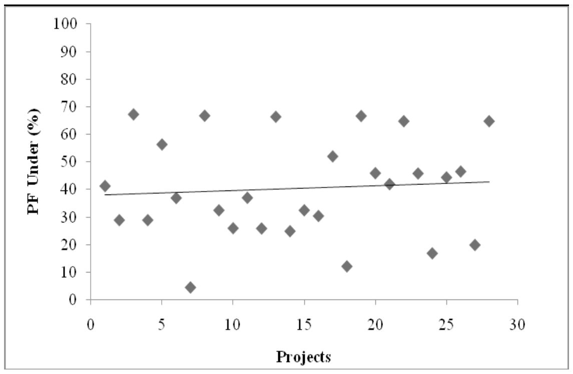
Figure 4. Probability of detection rates for five experiments.



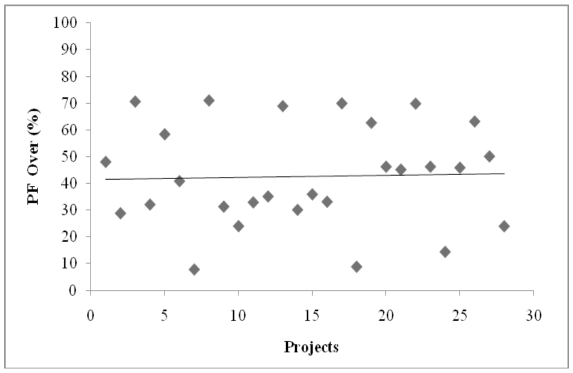
a) Using the original training set (PF)



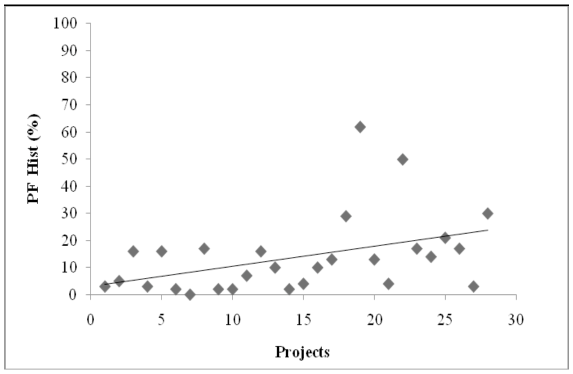
b) Using decision threshold optimization (PD Opt)



c) Using under-sampling (PF Under)



d) Using over-sampling (PF Over)



e) Using version history with optimum decision threshold (PF Hist)

Figure 5. Probability of false alarms for five experiments.

Table 1. Goals in line with business objectives

Goals of the project	Management objectives
Code measurement and analysis of the software system.	-Improve code quality
Storing a version history and defect data.	-Measure/ control the time to repair the defects
Construction of a defect prediction model to predict defect prone modules before testing phase.	-Decrease lifecycle costs such as testing effort. -Decrease defect rates

Table 2. Set of static code attributes used in this study. For the explanations of these metrics, see [1].

Metric name	Granularity (class: C, package: P, method: M, file: F)	Metric name	Granularity (class: C, package: P, method: M, file: F)
Cyclomatic density	C, P, M, F	Halstead length	C, P, M, F
Decision density	C, P, M, F	Halstead level	C, P, M, F
Essential density	C, P, M, F	Halstead programming effort	C, P, M, F
Branch count	C, P, M, F	Halstead programming time	C, P, M, F
Condition count	C, P, M, F	Halstead volume	C, P, M, F
Cyclomatic complexity	C, P, M, F	Maintenance severity	C, P, M, F
Decision count	C, P, M, F	Formal parameters	M
Essential complexity	C, P, M, F	Call pair length	M
LOC	C, P, M, F	Coupling between objects	C
Total operands	C, P, M, F	Fan in	C
Total operators	C, P, M, F	Number of children	C
Unique operands	C, P, M, F	Response for class	C
Unique operators	C, P, M, F	Weighted methods per class	C
Halstead difficulty	C, P, M, F		

Table 3. Confusion matrix

Predicted	Actual	
	defective	defect free
defective	TP	FP
defect free	FN	TN

Table 4. Raw code analysis.

Metric	Average	Application Layer	Presentation Layer	NASA Standard Min	NASA Standard Max
Intelligent Content	36.83	30.67	38.77		50
Maximum Nesting Depth	0.8	0.68	0.84		3
Volume	266.57	183.65	292.75	30	1000
Total Operators	27.61	20.48	29.86	50	125
Time	232.64	137.35	262.73		5000
Difficulty	3.65	2.96	3.87		35
Vocabulary	21.42	16.88	22.85	25	75
Effort	4187.46	2472.21	4729.11		1000000
Unique Operands	14.02	10.8	15.03	10	40
Unique Operators	7.4	6.08	7.82	15	40
Total Operands	18.11	13.13	19.68	25	70
Architectural Complexity	11.73	8.94	12.62		60
Level	0.52	0.57	0.51	0	1
Ratio of Comment to Code	0.02	0.03	0.02	0.15	
Length	45.72	33.61	49.54		300
Cyclomatic Complexity	3.42	2.54	3.7		10
Structural Complexity	1.12	0.97	1.17		5
Total Lines of Code	23.48	19.18	24.84		

Table 5. Rule based analysis

Rule No	Metric	Module	%	LOC	%
Rule 1	Intelligent Content	8245	17	507344	66
Rule 2	Maximum Nesting Depth	1307	3	155696	20
Rule 3	Volume	31260	65	345399	45
Rule 4	Total Operators	44117	92	530882	70
Rule 5	Time	143	0	53368	7
Rule 6	Difficulty	83	0	29545	4
Rule 7	Vocabulary	40442	84	444212	58
Rule 8	Effort	1626	3	234039	31
Rule 9	Unique Operands	41699	87	528542	69
Rule 10	Unique Operators	44086	92	464262	61
Rule 11	Total Operands	42774	89	507471	67
Rule 12	Architectural Complexity	1217	3	196641	26
Rule 13	Level	3270	7	28678	4
Rule 14	Ratio of Comment to Code	47062	98	729896	96
Rule 15	Length	525	1	122541	16
Rule 16	Cyclomatic Complexity	1735	4	223773	29
Rule 17	Structural Complexity	1036	2	112470	15
Rule 18	Any	47995	100	763025	100
Rule 19	Any - Halstead	6488	14	341655	45

Table 6. Defect prediction using cross company data

Project	Estimated defect rate	Estimated defective LOC	Total LOC	%LOC for inspection
GSM2	0.1	2458	80941	0.03
GSM3	0.03	1035	45323	0.02
GSM4	0.05	1130	53690	0.02
GSM5	0.06	2133	79114	0.03
GSM6	0.08	303	9767	0.03
GSM8	0.01	389	51273	0.01
GSM9	0.07	137	6258	0.02
GSM10	0.03	82	3507	0.02
GSM11	0.05	746	36280	0.02
GSM13	0.02	99	6206	0.02
GSM14	0.08	163	5803	0.03
GSM15	0.13	138	5423	0.03
GSM16	0.18	505	10221	0.05
GSM17	0.09	1509	61602	0.02
GSM18	0.09	44	2485	0.02
GSM19	0.08	119	5425	0.02
GSM20	0.06	65	2965	0.02
GSM21	0.18	1476	42431	0.03
GSM22	0.04	140	6933	0.02
GSM23	0.1	246	10601	0.02
GSM24	0.03	28	1971	0.01
GSM25	0.19	369	10135	0.04
GSM26	0.07	168	4880	0.03
GSM27	0.06	85	4526	0.02
TOTAL		13567	547760	
AVG	0.08			0.02

Table 7. General properties of GSM projects

Project		Release ID									
		1	2	3	4	5	6	7	8	9	10
GSM1	Total Files	-	-	-	218	220	-	280	-	-	-
	Defectives				2	2		1			
GSM2	Total Files	262	262	-	-	-	-	264	-	264	-
	Defectives	2	2	-	-	-	-	1	-	1	-
GSM3	Total Files	262	264	266	-	-	281	-	300	310	310
	Defectives	2	2	1	-	-	1	-	2	1	1
GSM4	Total Files	434	440	442	442	-	472	488	488	488	488
	Defectives	3	5	2	4	-	1	2	3	11	9
GSM5	Total Files	565	-	570	569	-	569	-	571	571	544
	Defectives	1	-	3	5	-	1	-	3	3	2
GSM6	Total Files	48	-	-	48	-	-	-	-	-	-
	Defectives	1	-	-	1	-	-	-	-	-	-
GSM9	Total Files	-	28	28	-	-	-	-	-	-	-
	Defectives	-	1	1	-	-	-	-	-	-	-
GSM10	Total Files	-	-	-	-	-	91	-	-	-	105
	Defectives	-	-	-	-	-	1	-	-	-	1
GSM11	Total Files	-	-	-	204	-	-	-	-	233	-
	Defectives	-	-	-	1	-	-	-	-	2	-
AVG	Total Files	314	248.5	326.5	296.2	220	353.2	344	453	373.2	361.7
	Defectives	1.8	2.5	1.7	2.6	2	1	1.3	2.6	3.4	3.2
	Defect Rate	%0.6	%1	%0.5	%0.8	%0.9	%0.3	%0.4	%0.6	%0.9	%0.9

Table 8. Results for version- versus project-level prediction. Shaded cells show the significance of project-level prediction over version-level prediction in terms of performance measures.

Release number	*Appl. Name*	*1st experiment with 8 appl.*			*2nd experiment with GSM 3 or 4*		
		pd	*pf*	*bal*	*pd*	*pf*	*bal*
2	GSM3	100	67	53	85	34 ▨	68 ▨
	GSM4	78	75	44	80 ▨	66 ▨	51 ▨
3	GSM3	92	51	60	100 ▨	36 ▨	75 ▨
	GSM4	81	63	45	90 ▨	71	44

Table 9. Results of local defect prediction model.

Release	*Name*	*Model I*				*Model II*			
		pd	*pf*	*bal*	*CB*	*pd*	*pf*	*bal*	*CB*
2	GSM2	50	49	50	0	50	19	62	60
	GSM 3	100	31	76	66	100	18	86	81
	GSM 4	80	75	45	17	80	62	54	32
3	GSM 3	100	22	84	73	100	15	89	84
	GSM 4	100	69	42	49	100	61	53	61
	GSM 5	67	63	49	17	67	9	76	87
	GSM 9	100	8	95	71	100	0	100	92
4	GSM 4	75	75	42	34	75	53	55	53
	GSM 5	70	41	65	62	70	10	75	88
	GSM 6	100	51	46	52	100	6	94	93
5	GSM 1	75	35	63	51	75	15	68	74
6	GSM 3	90	25	81	68	90	18	85	81
	GSM 4	100	79	44	35	100	29	80	77
	GSM 5	72	35	68	65	72	8	79	92
7	GSM 1	100	34	59	43	100	27	76	65
	GSM2	50	55	33	0	50	8	61	84
	GSM4	100	31	78	59	100	29	80	64
8	GSM3	95	23	82	70	95	16	87	85
	GSM4	100	81	36	46	100	63	52	50
	GSM5	100	64	54	37	100	21	85	68
9	GSM2	100	29	80	54	100	17	88	79
	GSM4	100	74	44	41	100	62	55	53
	GSM5	100	52	58	53	100	18	89	84
	GSM1 1	100	28	81	79	50	17	63	74
10	GSM3	100	57	60	41	100	35	76	65
	GSM4	88	69	49	36	88	60	54	44
	GSM5	100	40	72	60	100	7	95	93
	GSM1 0	100	95	29	33	100	61	56	59
	AVG	90	50	59	47	88	28	74	72

PD: Probability of detection rates (%)
PF: Probability of false alarms (%)
BAL: Balance rate (%)
CB: Cost-benefit analysis (gained verification effort %)
Model I: Predictor using only static code attributes
Model II: Predictor using static code attributes and version history

Table 10. New results of local defect prediction model

Release	*Name*	*PD*	*PF*	*PD Under*	*PF Under*	r_{under}	*PD Over*	*PF Over*	r_{over}	*PD Opt*	*PF Opt*	t_{opt}	*PD Hist*	*PF Hist*
2	GSM2	50	48	50	41	0.12	50	48	1	50	33	0.81	50	3
	GSM 3	100	31	100	29	0.3	95	29	2	100	15	0.69	100	5
	GSM 4	80	70	80	67	0.04	80	70	2	80	35	0.88	60	16
3	GSM 3	100	32	100	29	0.23	100	32	1	100	4	0.9	100	3
	GSM 4	100	70	80	56	0.11	80	58	2	100	35	0.72	100	16
	GSM 5	67	41	67	37	0.03	67	41	1	67	14	0.84	67	2
	GSM 9	100	8	100	5	0.15	100	8	1	100	0	0.7	100	0
4	GSM 4	100	71	95	67	0.03	100	71	1	100	39	0.86	100	17
	GSM 5	60	30	64	33	0.13	60	31	19	60	18	0.57	60	2
	GSM 6	100	24	100	26	1	100	24	1	100	9	0.95	100	2
5	GSM 1	100	43	95	37	0.04	90	33	1	100	20	0.63	100	7
6	GSM 3	90	35	90	26	0.25	90	35	1	90	12	0.7	90	16
	GSM 4	100	69	100	66	0.04	100	69	1	100	22	0.92	100	10
	GSM 5	72	40	75	25	0.11	60	30	3	80	10	0.62	80	2
7	GSM 1	100	37	100	33	0.6	90	36	3	100	11	0.66	100	4
	GSM2	70	23	70	31	0.26	80	33	96	100	57	0.46	100	10
	GSM4	100	70	100	52	0.01	100	70	1	100	27	0.86	100	13
8	GSM3	65	9	75	12	0.02	65	9	1	100	52	0.05	100	29
	GSM4	33	63	90	67	0.14	33	63	1	100	68	0.46	100	62
	GSM5	100	46	100	46	0.24	100	46	1	100	39	0.86	67	13
9	GSM2	100	45	100	42	0.02	100	45	1	100	6	0.96	100	4
	GSM4	100	70	99	65	0.05	100	70	1	91	55	0.69	91	50
	GSM5	100	46	100	46	0.19	100	46	1	100	24	0.93	67	17
	GSM11	80	14	95	17	0.01	80	14	1	100	20	0.31	50	14
10	GSM3	100	51	100	45	0.07	90	46	2	100	34	0.56	100	21
	GSM4	75	64	59	47	0.09	75	63	1	63	28	0.55	63	17
	GSM5	100	50	100	20	0.04	100	50	1	50	7	0.42	50	3
	GSM10	50	24	60	65	0.02	50	24	1	100	48	0.39	100	30
	AVG	85	44	87	40		83	43		90	26	0.68	86	14

PD: probability of detection rate (%) with default threshold. *PF*: probability of false alarm rate (%) with default threshold.
PDopt, PFopt: PD and PF rates using optimum decision threshold, *topt*
PDunder, PFunder: PD and PF rates, when under-sampling has been applied using the optimum sampling ratio, *runder*.
PDover, PFover: PD and PF rates, when over-sampling has been applied using the optimum sampling ratio, *rover*.
PDhist, PFhist: PD and PF rates when version history and decision threshold optimization has been used.