AI-Based Software Defect Predictors: Applications and Benefits in a Case Study

Ayse Tosun[1], Ayse Bener[2], Resat Kale[3]

[1,2]Software Research Laboratory (SoftLab), Department of Computer Engineering
Bogazici University, Bebek, 34342, Istanbul, Turkey
[3]Turkcell Technology, Gebze, Istanbul Turkey
[1]ayse.tosun@boun.edu.tr, [2]bener@boun.edu.tr, [3]resat.kale@turkcellteknoloji.com.tr

Abstract

Software defect prediction aims to reduce software testing efforts by guiding testers through the defect-prone sections of software systems. Defect predictors are widely used in organizations to predict defects in order to save time and effort as an alternative to other techniques such as manual code reviews. The application of a defect prediction model in a real-life setting is difficult because it requires software metrics and defect data from past projects to predict the defect-proneness of new projects. It is, on the other hand, very practical because it is easy to apply, can detect defects using less time and reduces the testing effort. We have built a learning-based defect prediction model for a telecommunication company during a period of one year. In this study, we have briefly explained our model, presented its pay-off and described how we have implemented the model in the company. Furthermore, we have compared the performance of our model with that of another testing strategy applied in a pilot project that implemented a new process called Team Software Process (TSP). Our results show that defect predictors can be used as supportive tools during a new process implementation, predict 75% of code defects, and decrease the testing time compared with 25% of the code defects detected through more labor-intensive strategies such as code reviews and formal checklists.

Introduction

Software defects are more costly if discovered and fixed in the later stages of the development life cycle or during production (Brooks 1995). Therefore, testing is one of the most critical and time consuming phases of the software development life cycle and accounts for 50% of the total cost of development (Brooks 1995).

The testing phase should be planned carefully in order to save time and effort while detecting as many defects as possible. Different verification, validation and testing strategies have been proposed so far to optimize the time and effort utilized during the testing phase: code reviews (Adrian, Branstad and Cherniavsky 1982; Shull et al. 2002), inspections (Wohlin et al. 2002; Fagan 1976) and automated tools (Menzies et al. 2007; Nagappan, Ball, Murphy 2006; Ostrand, Weyuker, Bell 2005). Defect predictors improve the efficiency of the testing phase in addition to helping developers assess the quality and defect-proneness of their software product (Fenton and Neil 1999). They also help managers in allocating resources. Most defect prediction models combine well-known methodologies and algorithms such as statistical techniques (Nagappan, Ball, Murphy 2006; Ostrand, Weyuker, Bell 2005; Zimmermann et al. 2004) and machine learning (Munson and Khoshgoftaar 1992; Fenton and Neil 1999; Lessmann et al. 2008; Moser, Pedrycz, Succi 2008). They require historical data in terms of software metrics and actual defect rates, and combine these metrics and defect information as training data to learn which modules seem to be defect-prone. Based on the knowledge from training data and software metrics acquired from a recently completed project, such tools can estimate defect-prone modules of that project.

Recent research on software defect prediction shows that AI-based defect predictors can detect 70% of all defects in a software system on average (Menzies et al. 2007), while manual code reviews can detect between 35 to 60% of defects (Shull et al. 2002), and inspections can detect 30% of defects at the most (Fagan 1976). Furthermore, code reviews are labor-intensive since depending on the review procedure, they require 8 to 20 LOC/minutes for each person in the software team to inspect the source code (Menzies et al. 2007). Therefore, AI-based models are popularly used by various organizations (Menzies et al. 2007; NASA MDP 2007; Nagappan, Ball, Murphy 2006; Nagappan, Murphy, Basili 2008; Ostrand, Weyuker, Bell 2005). They resemble the working principle of a human brain that collects previous knowledge on a given topic,

analyzes that information and comes up with a claim or prediction on the subject. AI-based predictors learn specific patterns concerning defect-proneness from past projects and use this information to predict the defect-proneness of new projects. As more projects are observed throughout the development life cycle, more data is collected, and predictions are more accurate.

We conducted a comprehensive metrics program and built a defect prediction model at a large telecommunication company in Turkey during a period of one year (Tosun, Turhan, Bener 2009). During this metrics program, we collected static code metrics and churn metrics from the company's 9 projects in 10 releases. We matched the pre-release defects (the defects detected during the testing phase) of the previous releases with the source codes at file level. Then we made predictions on the new releases of the projects. We calibrated our model based on its prediction accuracy and discovered that it is possible to detect on an average of 88% of defective files using a defect predictor (Tosun, Turhan, Bener 2009).

In this paper, we describe our model from a machine learning perspective with measurable benefits such as the defect detection capability and the cost-benefit analysis. We present the pay-off of the model used and show how the model has been implemented in the company. We also used our model for defect prediction and compared its performance with a pilot process change, which employed labor-intensive checklists and formal procedures for detecting defects before the testing phase. Our results show that our prediction model automatically finds 75% of the defects detected in unit testing, code reviews and inspections only in a few seconds. Therefore, we conclude that the impact of process changes is limited.

Brief Information about the Organization

The organization we collaborated within this study is the leading GSM operator in Turkey and the third biggest GSM operator in Europe in terms of number of subscribers. As of 31 December 2009, it is providing mobile communication services to 35.4 million subscribers, with additional 26.1 million subscribers in Azerbaijan, Kazakhstan, Georgia, Ukraine and Northern Cyprus. It was founded in 1994, and since 2006, it has an R&D centre with around 200 engineers. In this R&D centre, they develop software products and solutions for mobile operators all over the world. Some of these solutions are network solutions, value added services, SIM related solutions, terminal based solutions, billing and charging solutions, data mining, data warehouse, customer and channel management systems and applications. Their legacy system contains millions lines of code that are being maintained. The majority of their software is implemented with Java, Jsp, PL/SQL and other new technologies such as SOA.

As with any other company, time and budget constraints put constant pressure on R&D. As customers require new functionality or technology changes, the company has to respond faster and faster with new software releases. Therefore, its approach to development is incremental, with each new release adding new functionality or a software modification to previous releases. In such a limited time, the software team cannot apply any measurement process to assess the overall software quality. Therefore, there was an urgent need to implement a measurement and analysis program to monitor defects, reduce defect rates and testing effort, and to improve software quality. We have built a measurement repository, bug tracing/matching system and a defect prediction model for the company. In this study, we explain the implementation of the defect prediction model after it has been calibrated with local data to achieve the highest prediction accuracy.

Description of the Prediction Model

Our learning-based defect predictor is a typical machine learning application: It contains a training phase to learn from the data related to previous projects and a testing phase to predict the potential defect-free and defective modules of the new project. A module could be a package, class, file, or method inside the source code. Erroneous predictions of the defect-free modules in the form of defects (false alarms) force testers to inspect "safe" modules and waste their precious time. On the other hand, missing defective modules (false negatives) would cause more expensive and hard-to-fix failures on the final software product. Thus, false negatives need to be avoided.

Basic Terminology: Input and Output Variables

As a classification task, our input variables are a set of static code attributes such as lines of code (LOC), complexity, operand and operator counts extracted from the source code. Static code attributes are widely used and easily collected through automated tools (Menzies et al. 2007; Moser et al. 2008; Lessmann et al. 2008) and proposed by various researchers such as McCabe (1976) and Halstead (1977). The full list of attributes collected from the source code in this study is illustrated in Appendix A.

In literature, various researchers have also used other type of metrics such as object-oriented design metrics (Basili, Briand, Melo 1996; Chidamber, Kemerer 1994), in-process metrics (Nagappan, Ball, Murphy 2006), and organizational metrics (Nagappan, Murphy, Basili 2008) in order to predict defects. Although increasing the information content of input data by adding different types of metrics has positive effects on defect prediction capability, it is not easy to collect in-process and organizational metrics from an organization. Therefore, we have preferred to use the source code as a means of collecting metrics, i.e. input variables.

In addition to code attributes, there are class labels for each module such as 0 as defect-free and 1 as defective in the training set. If a module in the software system has

been associated with a bug (code defect) during the testing phase, it is labeled as 1; otherwise, it is labeled as 0. It is not necessary to count the number of defects a module is associated with since our aim in this study is not predicting the number of defects. More precisely, the training set is an *N-by-M* matrix where *N* is the number of modules taken from past projects and *M* is the number of code attributes *(M-1)* extracted from their source code as well as a class label to indicate whether a defect has been detected on that module during testing.

The test set, on the other hand, contains attributes extracted from the modules of a new project whose defect labels are unknown. Therefore, the output variable (*Y*) of the model would be the class labels of modules in the test set as defect-free or defect-prone.

The Use of AI Technology

We have used a Naïve Bayes classifier as the algorithm of our prediction model. The Bayes Theorem defines the posterior probability as proportional to the prior probability of the class $p(C_i)$, and the likelihood of attributes, $p(X|Y=C_i)$ (cf. Alpaydin 2004). In binary classification problems such as defect prediction, Naïve Bayes computes the posterior probability of a module being *defective,* or the probability of a module being *defect-free,* given its attributes. Then, it assigns a module to the *defective* class if its posterior probability is greater than a pre-defined threshold (0.5). Otherwise, the module is classified as *defect-free.*

We have used a Naïve Bayes classifier for several reasons. First of all, it is a widely used, simple and robust machine learning technique in various applications such as pattern recognition (Kuncheva 2006), medical diagnosis (Uyar et al. 2009) and defect prediction (Menzies et al. 2007; Moser et al. 2008; Tosun, Turhan, Bener 2009). It is also easy for field practitioners to understand and implement. Second, defect prediction models with a Naïve Bayes classifier deliver the best prediction accuracy on public datasets compared with models with other classifiers (Menzies et al. 2007). One of the reasons for the success of the Naive Bayes classifier over other methods is that it combines signals coming from multiple sources. It is not affected by the "brittleness" of data (minor changes in training sample do not give completely different results) by polling numerous Gaussian approximations to the numeric distributions (Menzies et al. 2007). Therefore, minor correlations between attributes or samples in the training set within the field of software defect prediction do not confuse Naive Bayes classifiers. Third, a recent study by Lessmann et al. (2008) presents that the importance of classification algorithms in defect prediction may be less than previously assumed, since no significant performance differences exist among the top 17 classifiers. This result is very important for our case study since it reduces the necessity of trying all classification techniques. Thus, instead of applying different algorithms, we have selected Naïve Bayes as the algorithm of our model and focused on calibration based on local data.

Performance Evaluation

We use Receiver Operator Characteristics (ROC) curves to assess the discriminative performance of a binary Naïve Bayes classifier (Heeger 1998). In a ROC curve, our objective is to reach the point *(1, 0)* in terms of *(y, x)*, where the y-axis represents the *true positive rate* and the x-axis represents the *false positive rate*. We have computed these performance measures to evaluate the accuracy of our model. However, similar to defect prediction research (Menzies et al. 2007; Lessmann et al. 2008), we name the true positive rate as the *probability of detection rate (pd)* and the false positive rate as the *probability of false alarm rate (pf)* in this study. The ideal classification, point *(1, 0)* in a ROC curve can be reached when we correctly classify all defective modules *(pd=1, i.e. 100%)* with no false alarms *(pf=0, i.e. 0%)*.

Finally, the prediction outcomes depending on the actual class labels of modules can be represented as a confusion matrix as shown in Table 1. The common classifier performance measures are derived from this confusion matrix (Menzies et al. 2007).

Table 1. Confusion matrix

Actual	Predicted	
	Defective	Defect-free
Defective	TP	FN
Defect-free	FP	TN

Probability of the detection rate (PD) is a measure of accuracy for correctly classifying defective modules. It corresponds to the *true positive rate* in machine learning and should be as close to 1 as possible:

$$(PD) = TP / (TP + FN) \quad (3)$$

Probability of the false alarm rate (PF) is a measure of accuracy to represent the false alarms when we misclassify defect-free modules. We must avoid high *PF* rates in software defect prediction models since they would increase the testing effort.

$$(PF) = FP / (FP + TN) \quad (4)$$

It is very rare to achieve the ideal case with 100% PD and 0% PF rates using a prediction model. When the model is triggered often to increase the PD rate, the PF rate would, in turn, increase. Therefore, our objective is to get as high PD rates as possible while keeping the PF rates at a minimum.

Utilization of the Model and Pay-off

We built our defect prediction model on the software system of a telecommunication company. Previously, a tool that collected metrics from the source code did not exist. Moreover, although the defects were logged in a version management system, they were not matched with

the source code at any granularity level, i.e. package, file, method, or LOC.

We started a metrics program to collect the required data for building our defect predictor. We developed an open-source metrics extraction and a defect prediction tool called *Prest* (2009), and collected code metrics from Java and Jsp files. Previously, there was no process in the company for bug tracking. Furthermore, there was no process to match defects with the files in order to keep track of the reasons for any change in the software system. Therefore, we implemented an organization-wide process change that is fully supported by the senior management (Tosun, Turhan, Bener 2009). This process change helped us to store defects as well as to match them at files level.

The static code attributes at file level and the defect labels matching the files (more precisely, Java and Jsp files) were collected from 9 different projects in 10 releases, and this dataset was donated to a public data repository, Promise (2007). Then, the project-based defect prediction was performed such that the defective files of a project at release *n* were predicted using the static code attributes and the defect labels of the same project at release *n-1*. Based on this training-testing strategy, we assessed the performance of our predictor and discovered that the deployed model with a Naïve Bayes classifier correctly classifies 90% of the defective files while producing 50% false alarms (Tosun, Turhan, Bener 2009). Since false alarms were very high, we included a new software metric in addition to static code attributes, such as version history flags indicating the latest activity date on files as inputs to the model. This flag shows whether a file has been edited at least once for six months. If it does not have any activity for a long time, then it is less likely that the file contains defects. Using version flags further improved the prediction performance by decreasing the false alarm rates on an average of 28%, from 50% to 22% (Tosun, Turhan, Bener 2009).

Table 2 shows the summary of the prediction performances in 9 releases. We have made predictions for an average of three projects in every release and took the mean and the standard deviation of the prediction performances in terms of *pd* and *pf* rates. As it is seen in Table 2, we have successfully achieved 87% detection rate in 9 releases with 26% false alarms. Our defect predictor helps detecting defective modules using less time and effort. Furthermore, it guides testers through specific files and reduces the inspection effort compared to code reviews and inspections. The process is less labor-intensive if local data is collected as required.

The practical benefits of using a defect predictor have been further computed using a cost-benefit analysis from Arisholm and Briand (2006). The authors compared the inspection effort suggested by a defect prediction with a random testing strategy. Based on that, the gain in the effort to (GE) can be calculated with the following formula:

$$100x\left(\frac{\text{MRT}-\text{MDF}}{\text{MRT}}\right) \quad (5)$$

In Equation 5, 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. We have conducted the cost-benefit analysis of our predictor to present the practical benefits for the company. If we would use a random testing strategy, we would have to inspect 87% of the files to be able to detect 87% of the defects. However, our model highlights only 25% of the files that contain 88% of defects. Therefore, the gain in the inspection effort is 72%. Table 4 shows that the implemented model reduces the inspection effort by 72.5% on average through highlighting the critical parts in the software system. Rather than looking at 87% of the files, we can inspect only 24% of the files and detect 87% of the defects in the system.

Table 2. Performance of the prediction model

Releases	PD	PF	GE
2	77	33	58
3	92	21	81
4	82	23	78
5	75	15	74
6	87	18	83
7	83	21	71
8	98	33	68
9	88	29	72
10	97	41	68
Average (Std. Dev.)	87 (8.1)	26 (8.5)	72.5 (7.6)

Deployment of the Model

The prediction results given above were so satisfactory that the quality assurance team at the company decided to integrate the model into their configuration management system. They planned to use the prediction model *prior to the testing phase* so that the defect-prone files would be investigated by a) the *developer* before he/she transfers the project to the test team or b) the *test*er so that his/her effort would be assigned to the critical parts only.

As mentioned above, we have implemented Prest (2009), an open source metrics extraction and defect prediction tool, during this study. This tool not only extracts code metrics from different granularity levels of projects written in Java, Jsp, C and C++, but also includes a defect prediction component in which a Naïve Bayes classifier can be executed on a new project given a training set.

We have customized the defect prediction component of Prest for the company. A graduate student from our research laboratory and an engineer from the company completed the implementation of this tool on the company's configuration management system. A Java program was implemented to perform the following steps:

- A shell script was written to call Prest and extract code metrics from a specified project.
- Shell scripts were written a) to retrieve the defects detected for a specified project from quality center and b) to match those defects with files that already included code metrics.
- The training set was prepared from the previous release of a specified project.
- The test set was prepared from the current release of a specified project.
- Prest was called once again a) to activate the prediction component, b) to load the training and test sets, and c) to run the algorithm and make the prediction.
- The program returned defect-prone files of a specified project for its current release.

This model has been applied on two major components of the company's software system in the last six months. It lists defect-prone files of these components at the beginning of the testing phase so that testers' efforts would be assigned to critical parts. Every 2 weeks, a new release with 10 to 15 work packages and more than 400 graphical user interfaces of these components is being published. Since the release period is short, each release package contains at most one or two new functionalities and the rest are modifications/upgrades for the current system. These work packages are tested using 1000 to 1500 test cases by a total of 20 testers. Due to time constraints, the testing phase is limited to 5 days on average. Thus, each tester needs to run 10 to 15 automated test cases per day, i.e. 8 hours, in order to inspect 80% of the functionality in total. It is also necessary for each tester to conduct manual inspections to ensure 100% test coverage. However, in reality, the development phase is delayed with frequent requirements changes due to revisions in government regulations. Thus, testers often have only 3 days to complete the verification of a release. During this period of time, a tester can execute 30 to 45 test cases. All test cases executed by 20 testers in 3 days can cover only 48% of the overall functionality. Therefore, the company applied our defect prediction model to prioritize critical parts of the code and assign the company's few resources to those parts immediately. The model inspects 24% of files corresponding to 35 (23%) different functionalities and it detects 87% of defects. As a result, each tester is required to run 9 to 13 automated test cases per day to inspect a 71% of the functionality in total. In other words, the company has managed to decrease the effort in person-hours from 1.25 to 1.1 (decrease by 11.2%) with the help of our defect prediction model. The quality assurance team also counted the number of post-release defects for the last 5 releases and found that, since the model successfully catches most of these defects during the testing phase, post-release failures due to a code defect have been decreased from 59% to 32% (decrease by 44%).

We have done an additional analysis to compare the effects of our model with a new process implementation on improving the quality of a software product, i.e. decreasing the defect rates and reducing the testing effort. Recently, a pilot project has been conducted for implementing a new process the so called, Team Software Process (TSP), in the company. A software team of four people developed a new project by applying the fundamental principles of TSP and reported all tasks they accomplished and planned as well as the actual times required for completing these tasks, the defects detected and removed during unit testing and the independent testing phases.

Comparison with a Process Change: Team Software Process

What is Team Software Process (TSP)? Along with Personal Software Process (PSP), TSP (SEI 2010) helps engineers:

- ensure the quality of software products,
- create secure software products, and
- improve process management in an organization.

Engineering groups often use TSP to apply integrated team concepts to the development of software-intensive systems (SEI 2010). A launch process leads the teams and managers to establish their goals, define the roles within the team, assess risks, and produce a team plan. This process first directs the goals and the plans of engineers in the company individually. Then it helps create self-directed teams who take ownership of their plans and processes and direct their tasks accordingly. Using PSP, engineers do not only improve the process of planning and estimating the size and effort related to their tasks, but they also understand the means of managing quality and reducing defects. According to case studies carried out in various organizations such as Motorola, engineers have achieved less than 0.1 defects per KLOC on nearly 18 projects (SEI 2010). Although the objectives and claims of applying such a process are very strong, it is clear that TSP, along with PSP, obliges engineers individually to ensure that they adopt good practices in term of engineering disciplines.

Analysis on the Pilot TSP Project. The management selected a pilot group of four engineers to complete a new project using the TSP principles. Their objectives were a) to observe the applicability of TSP in their organization, and b) to evaluate the benefits of the process change in terms of productivity, estimation accuracy, the defects detected in unit testing, and defect density in the testing and production phases.

The pilot project would provide a modification for one of the four major software components in the large software system. When compared with the large system, for which we have implemented a predictor model, the size of the pilot project can be viewed as 1/5 of the entire system. At the end of development life cycle, it contains 107 Java packages with around 105.925 executable LOC. The pilot project team executed two launches during the

project life cycle. Every launch started by defining the tasks required for completing the project, assigning each task to an engineer (or a group of engineers), estimating the time and size of the tasks and the number of defects injected and removed for each phase.

Engineers in the pilot group purposefully applied the TSP principles for almost six months. They prepared reports with statistics on estimation accuracy, productivity and defect rates. Based on these reports, we have formed a chart that represents the actual time spent in software phases (rates out of 1.0) aligned with the estimated time periods in Figure 1.

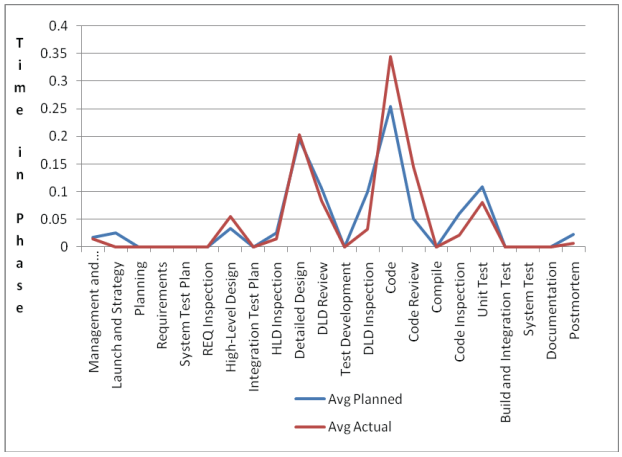

Figure 1. Time in hours spent for each phase.

By looking at this figure, we can see that the team actually applied TSP and made accurate estimations in terms of the time spent in each phase. They spent 95% of their total effort on high-level design (HLD), detailed design (DLD), DLD review, implementation, code review, and unit tests. Specifically, coding and code review took slightly more than expected. Code reviews and inspections constitute 25% of this effort. When we observe the defects detected and removed in each phase (Table 3), we see an interesting pattern: 25% of all defects were detected and fixed during code reviews and inspections in comparison to 28% of defects detected during unit testing.

A panel on IEEE Metrics reports that code reviews and manual inspections can detect 60% of defects on average (Shull et al. 2002). However, although 17% of total time was spent for code reviews and inspections in the course of this project, the percentage of the defects detected during independent testing activities was still 29 (i.e. testing phase). Normally, TSP should increase the number of defects detected in code reviews and unit testing since it provides a guide for increasing the quality of the work of engineers (software developers). Furthermore, TSP argues that it should decrease the defect density in the testing phase, and increase the software quality.

We argue that if a learning-based defect predictor is used as complementary to code reviews and inspections, it will help reduce both the time and the effort spent during coding.

Table 3. Number of defects detected and fixed in phases

Phase Removed	Count
Unit Testing	66
Code Inspection & Reviews	58
DLD Review	38
Design	6
Independent Testing	68
Total	**236**

Table 4. Predictions on pilot project using our prediction model

#Attributes	# Packages	Defectives (%)	PD	PF	GE
20	107	15%	75%	26%	10%

Defect Prediction on the TSP Project. To evaluate and compare the benefits of using a defect predictor with a process change, we have made predictions on the TSP project to detect the defects observed during "*coding*" (including code reviews, inspection and unit testing). In TSP, defect logs were kept in detail and when possible, they were matched with a Java package in the project. We used Prest to extract static code attributes from the Java packages of the pilot project. Then we matched the defects detected during "*coding*" with code metrics of the packages. We have assigned 1 as defective and 0 as defect-free to every package in the source code if there was a minimum of one defect. Finally, we have made package-level predictions on the pilot project. Table 4 summarizes the defect ratio in the package level and the prediction performance of our predictor. It shows that using a smart and automated tool, we managed to detect 75% of all defects without spending too much effort on inspecting the entire code through the use of labor-intensive checklists. We can decrease the inspection effort by 10% compared to a random testing strategy. This gain in the inspection effort is lower than what we proposed in Table 2 due to the granularity level we used in TSP analysis. Matching the files with defects rather than packages would prove to be more beneficial for reducing the inspection efforts. Thus, we see that a process change itself is limited to reduce the inspection effort or to improve the quality of coding in terms of the number of defects detected during unit testing. If we used such a tool during the TSP implementation, it would enable us to save time and find the defects that were missed during the "*coding*" phase but detected during the testing phase. The rate of false alarms seems to be high – a fact that would waste the limited testing effort on actual defect-free modules. However, in this analysis, we have only predicted the defects detected during the "*coding*" phase, but we have not matched the defects detected during the testing phase with the software packages. Therefore, the packages that are misclassified as defect-prone (false

alarms) may also contain a defect detected during the testing phase.

To sum up, we have observed that process change and applying organizational procedures are beneficial to obtain high quality software products within the set time period and budget. However, they may not provide a solution to all problems such as decreasing the defects, increasing quality, the accurate estimation of size and effort in the software development life cycle. AI-based tools such as defect predictors could be used in conjunction with new processes to save time and effort as well as to decrease as many defects as possible.

Maintenance

Similar to many AI-based models, our model also requires calibration. The company decided to train the model with new data in periods of three months and make predictions on new releases. Since the model has been successfully integrated with the company's software system, one person from the quality assurance team is selected to form a new training set every three-months and update the model with new parameters. The company also motivates the teams for making such tools part of their routine during the development and testing stages. This way, it will be easier to apply the model in collaboration with the development and test teams in order to analyze the code quality and to predict the critical parts of the software. Furthermore, we plan to track the prediction performance and the usage of the model in the company through one year and will calibrate the algorithm if necessary.

Lessons Learned

There are certain challenges during the development and implementation of predictive models. During the development process, we easily collected software metrics using our open source metric extraction tool Prest. However, matching each defect with its corresponding file in order to form training set for the model was a challenging task. To do this, companies have to store certain data in their systems. First, it is necessary to keep track of any bug/defect recorded during the testing phase through a bug tracking system. Second, the changes applied on the source code due to a defect should be kept in a version history. Then, we can mine the version history to match every defect with all the files changed to fix the corresponding defect. After forming the training set, it is easy to apply any algorithm, and not necessarily only Naïve Bayes, on the training set to learn the parameters. Software metrics required to form the testing set can be quickly collected with Prest.

During the implementation process, we must ensure at the beginning that the model yields the optimal prediction accuracy for the local data collected from the organization. Then, it is important to decide how and when a defect predictor would be used within the development life cycle. We suggest that such predictors should be used prior to the testing phase in order to guide the testers through defect-prone modules in the software system. We have integrated our model into the company's configuration management database system (CMDB), which displays certain properties about the source code such as the difference between two releases, the complexity of the latest change and added/deleted LOC during a given a time period (i.e. release). Using our prediction model, this system also presents the defect-proneness of any software module selected from the system by assigning 1 as defect-prone and 0 as defect-free. Thus, developers, as well as testers, can track the defect-proneness of their code in every release.

The application has been in use at the company for eight months now. The company plans to improve the predictions by adding new metrics from version management systems. Furthermore, they plan to use these predictions to compute defect rates for every release and compare these defect rates with a pre-defined reliability threshold. If estimated defect rates of a release is higher than the company's reliability threshold, then quality assurance team would decide to delay or cancel the release.

Acknowledgment

This research is supported in part by the Turkish Scientific Research Council (TUBITAK) under the grant number EEEAG108E014, and Turkcell Inc.

Appendix A

Table A.1. List of static code attributes (NASA 2007)

Attribute	Description	Attribute	Description
McCabe metrics			
Cyclomatic density, vd(G)	the ratio of module's cyclomatic complexity to its length	*Essential complexity, ev(G)*	the degree to which a module contains unstructured constructs
Design density,dd(G)	condition/ decision	*Cyclomatic complexity, v(G)*	# linearly independent paths
Essential density,ed(G)	(ev(G)-1)/(v(G)-1)	*Maintenance severity*	ev(G)/v(G)
Halstead metrics			
Difficulty (D)	1/L	*Length (N)*	N1 + N2
Level (L)	(2/n1)*(n2/N2)	*Programming effort (E)*	D*V
Volume (V)	N*log(n)	*Programming time (T)*	E/18
Lines of code metrics			
Unique operands	n1	*Executable LOC*	Source lines of code that contain only code and

			white space
Branch count	# branches	*Total operators*	N1
Decision count	# decision points	*Total operands*	N2
Condition count	# conditionals	*Unique operators*	n2

References

Alpaydin, E. eds 2004. *Introduction to machine learning*. The MIT Press.

Adrian, R.W., Branstad, A. M., Cherniavsky, C. J. 1982. *Validation, Verification and Testing of Computer Software*, ACM Computing Surveys (14), 22: 159-192.

Arisholm, E., Briand. L.C. 2006. *Predicting fault-prone components in a java legacy system.* In Proceedings of the 2006 ACM/IEEE International symposium on Empirical software engineering, 8-17, ACM, New York, NY, USA.

Basili, V., Briand, A., Melo, W.L. 1996. *Validation of object oriented design metrics as indicators of quality indicators*, IEEE Transactions on Software Engineering (22), 751-761.

Boetticher, G. Menzies, T., Ostrand, J.T. 2007. *The PROMISE Repository of Empirical Software Engineering Data.*http://promisedata.org/repository.

Brooks, A. eds. 1995. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley.

Chidamber, S.R., Kemerer, C.F. 1994. *A metrics suite for OO design,* IEEE Transactions on Software Engineering (20), 476-493.

Fagan, M. 1976. *Design and Code Inspections to Reduce Errors in Program Development*. IBM Systems Journal (15), 3.

Fenton, N., Neil, M. 1999. *A Critique of Software Defect Prediction Models.* IEEE Transactions on Software Engineering (25), 675-689.

Halstead, H.M. eds 1977. *Elements of Software Science*. New York, Elsevier.

Heeger, D. 1998. *Signal Detection Theory.* Available at http://white.stanford.edu/ heeger/sdt/sdt.html.

Kuncheva, L.I. 2006. *On the optimality of Naïve Bayes with dependent binary features*. Pattern Recognition Letters (27), 7: 830-837.

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 (34), 4: 1-12.

McCabe, T. 1976. *A Complexity Measure*. IEEE Transactions on Software Engineering (2), 4: 308-320.

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

Moser, R., Pedrycz, W., Succi, G. 2008. *A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,* In Proceedings of the 30th International Conference on Software Engineering, 181-190.

Munson, J. C., Khoshgoftaar, T. M. 1992. *The detection of fault-prone programs*. IEEE Transactions on Software Engineering (18), 5: 423-433.

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.

Nagappan, N., Murphy, B., Basili, V. 2008. *The Influence of Organizational Structure on Software Quality: An Empirical Case Study.* In Proceedings of 30th International Conference on Software Engineering, 521-530.

Nasa/Wvu IV&V Facility, Metrics Data Program, available from http://mdp.ivv.nasa.gov, accessed 2007.

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 (31), 4: 340-355.

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

Shull, F., Boehm, V.B., Brown, A., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., and Zelkowitz, M. 2002. *What We Have Learned About Fighting Defects.* In Proceedings of the Eighth International Software Metrics Symposium, 249-258, 2002.

Software Engineering Institute (SEI). 2010. *Team Software Process*. Carnegie Mellon University, http://www.sei.cmu.edu/tsp/.

Tosun, A., Bener, A. and Turhan, B. 2009. *Practical Considerations in Deploying AI for Defect Prediction: A Case Study within the Telecommunication Industry*. In Proceedings of the 1st International Conference on Predictor Models (PROMISE), Best Paper Award.

Uyar, A., Bener, A., Ciray, H.N., Bahceci, M. 2009. *ROC Based Evaluation and Comparison of Classifiers for IVF Implantation Prediction*. In Proceedings of Second International ICST Conference on Electronic Healthcare for 21st century, Istanbul.

Wohlin, C., Aurum, A., Petersson, H., Shull, F., Ciolkowski, M. 2002. *Software inspection benchmarking - a qualitative and quantitative comparative opportunity*. In Proceedings of the 8th International Symposium on Software Metrics, 118-127, 2002 IEEE Computer Society, Washington, DC, USA.

Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A. 2004. *Mining version histories to guide software changes*. . In Proceedings of the 26th International Conference on Software Engineering, 563-572, IEEE Computer Society, DC, USA.