Software Qual J (2011) 19:515–536
DOI 10.1007/s11219-010-9128-1

An industrial case study of classifier ensembles for locating software defects

Ayşe Tosun Mısırlı · Ayşe Başar Bener · Burak Turhan

Published online: 13 January 2011

Abstract As the application layer in embedded systems dominates over the hardware, ensuring software quality becomes a real challenge. Software testing is the most time-consuming and costly project phase, specifically in the embedded software domain. Misclassifying a safe code as defective increases the cost of projects, and hence leads to low margins. In this research, we present a defect prediction model based on an ensemble of classifiers. We have collaborated with an industrial partner from the embedded systems domain. We use our generic defect prediction models with data coming from embedded projects. The embedded systems domain is similar to mission critical software so that the goal is to catch as many defects as possible. Therefore, the expectation from a predictor is to get very high probability of detection *(pd)*. On the other hand, most embedded systems in practice are commercial products, and companies would like to lower their costs to remain competitive in their market by keeping their false alarm *(pf)* rates as low as possible and improving their precision rates. In our experiments, we used data collected from our industry partners as well as publicly available data. Our results reveal that ensemble of classifiers significantly decreases pf down to 15% while increasing precision by 43% and hence, keeping balance rates at 74%. The cost-benefit analysis of the proposed model shows that it is enough to inspect 23% of the code on local datasets to detect around 70% of defects.

Keywords Defect prediction · Ensemble of classifiers · Static code attributes · Embedded software

A. T. Mısırlı (✉)
Department of Computer Engineering, Boğaziçi University, 34342 Bebek, Istanbul, Turkey
e-mail: ayse.tosun@boun.edu.tr

A. B. Bener
Ted Rogers School of Information Technology Management, Ryerson University,
M5B 2K3 Toronto, Canada
e-mail: ayse.bener@ryerson.ca

B. Turhan
Department of Information Processing Science, University of Oulu, Oulu, Finland
e-mail: burak.turhan@oulu.fi

1 Introduction

The quality of software is defined with different attributes such as reliability, timeliness and usability depending on the type of software system (Kan 2002). Any attempt to increase the quality of software should be carefully planned by considering the most important attributes. In this context, effective use of testing resources enhances product quality as well as time-to-market. Different verification, validation and testing (VV&T) activities, i.e. inspections (Wohlin et al. 2002), manual or automated models (Basili et al. 2002; Menzies et al. 2007a), are proposed so far to use testing time and resources effectively. Among these, defect predictors are useful tools for software organizations to manage their testing resources effectively through focusing on defect-prone software modules, and hence, to improve software quality. They perform significantly better in terms of defect detection performance, compared to other VV&T activities (Menzies et al. 2007a). Using such tools is easier and faster to run for highlighting defect-prone modules compared to inspections since they are able to provide static analysis on source codes with metrics extraction tools (Basili et al. 2002; Menzies et al. 2007a; Tosun et al. 2009; Kocaguneli et al. 2009).

Various prediction models are proposed in the literature to identify defect-prone parts of the software in terms of different application domains (i.e. telecommunications (Ohlsson and Wohlin 1998), white goods (Tosun et al. 2008), the purpose of the approach (i.e. classification (Menzies et al. 2007a; Tosun et al. 2008), ranking (Ohlsson and Wohlin 1998) and data used (i.e. source code (Menzies et al. 2007a), documents (Biffl et al. 2003). A previous study conducted by Runeson et al. (2001) showed that it is difficult for companies to build a defect predictor that provides the best performance. Each application domain has different characteristics that is reflected in the end product.

This study specifically includes projects from the embedded systems domain, since it is carried out with our industry partner which operates in the embedded systems domain. Embedded systems are used in many industries such as white goods, automotive, telecommunications and aerospace (Li and Yao 2003). They are deployed pervasively once they are developed and their software is generally a section of a larger system (Amasaki et al. 2005). Previously, the software in embedded systems was only used to control the hardware. However, the purpose of embedded systems has grown with the increase in demand (Amasaki et al. 2005). This increase in demand makes the software more sophisticated (Lee 2002), and hence, more important. Unlike general software systems, reliability standards always remain very high (Lee 2002). According to Lee (2002), embedded software systems are reactive in the sense that they have real-time constraints and are often safety-critical. Their failures can result in the loss of human life. So, the impact of residual defects in embedded software would be much higher than defects in other types of software.

Developers need to ensure reliability of their software in order to decrease the cost of fixing defects during later stages of development life cycle. Tight schedules and increasing competition in such industries, on the other hand, enforce limited testing which may prevent identifying severe defects in the software. This dramatically affects quality attributes, such as timeliness, reliability and dependability (Kan 2002). Therefore, developers in the embedded software domain need additional techniques to preserve the reliability of software. As early warning mechanisms, defect predictors would be very helpful for practitioners in order to improve product quality in embedded systems in a shorter time and with fewer resources, compared to other verification, validation and testing activities (Khoshgoftaar and Gao 1996; Oral and Bener 2007).

According to Brooks (1995), half of software development costs are spent on unit and system testing. In a highly competitive market segment with very tight profit margins, software companies that operate in the embedded domain need additional verification, validation and testing strategies to manage their test workbench. Some of these techniques are code reviews (Shull et al. 2002; Adrian et al. 1982), inspections (Wohlin et al. 2002) and intelligent oracles such as defect prediction models (Menzies et al. 2007a; Ostrand et al. 2005; Fenton et al. 2007). Code reviews are able to detect 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 8–20 LOC/min for each tester to inspect the source code (Menzies et al. 2007a).

Building defect prediction models is becoming easier with the help of automated metric extraction tools (Menzies et al. 2007a; Kocaguneli et al. 2009). Once a local repository is built using such tools as well as scripts for combining version control systems with bug repositories, it takes only minutes to train the model with historical data and test for current releases simultaneously (Tosun et al. 2009). Therefore, software companies need such oracles to detect as many defects as possible, and hence, to guide testers through problematic modules so that they can fix more defects in shorter time periods. Reducing the testing effort even by 1% helps software companies to efficiently use their scarce resources. Therefore, our objective is to guide our industry partner on the allocation of testing resources. To accomplish this, we build a learning-based defect predictor which would produce high detection rates as well as low false alarm rates.

We use a novel approach, an ensemble of classifiers, to solve the problem of optimizing testing resources in embedded software development in an industrial case study setting. Rather than relying on the outcome of a single classifier, the ensemble approach combines signals from multiple classifiers to produce an outcome, and it manages to capture the strengths of uncommon techniques as well as helping to overcome problems of noise in data. In our previous work, we also built an ensemble of classifiers using three machine learning techniques to predict the defective modules of the public datasets (Tosun et al. 2008). In this study, we conduct a broader research on the performance of classifier ensembles: (a) Comparison of three single classifiers with the previous ensemble (Ens_1) in terms of classification accuracy and cost-benefit analysis, (b) Proposing a new ensemble of classifiers (Ens_2) according to the performance of single classifiers, and (c) Comparison of Ens_1 with Ens_2 in terms of classification accuracy and cost-benefit analysis. Our results indicate that Ens_2 significantly decreases false alarms by 5%, thereby, lowering the testing efforts for the embedded data we used from 35% to 23%. Furthermore, it manages to keep high balance rates as in Ens_1.

In practice, we construct our proposed model based on the company's historical data. We show that cross validation experiments on all datasets produce valuable results: Cost-benefit analysis reveals that false alarms are often raised in small modules and Ens_2 reduces the test efforts by 67% on average. The voting scheme used in Ens_2 also helps to decrease false alarms and increases the precision by predicting fewer modules as defective. Furthermore, we discuss the performance of the proposed model when used prospectively to predict defects on an ongoing project, where we "really" do not know whether a module was defective or not. The initial findings for an ongoing project show that an ensemble of classifiers achieves a detection rate of 80% in predicting defect-prone modules of embedded software.

The rest of the paper is organized as follows: We briefly discuss related work on defect prediction and embedded systems. Then, we explain datasets and the ensemble of classifiers with the algorithms implemented. We show our experimental design in model construction and provide the results for each of the experiments consecutively. Finally, we present an analysis of the results from an ongoing project and discuss the threats to validity as well as future directions before concluding our work.

2 Related work

Although there are many studies in software defect prediction, few focus on embedded software systems. In this section, we briefly mention three important defect prediction models proposed specifically for embedded systems and argue how our work is different and/or compatible with them. Then, we state our reasons for using an ensemble of classifiers by referring to previous studies.

Specialized prediction models for embedded systems are investigated by Khoshgoftaar and Allen (1999), who built a classification and regression tree for predicting high risk software modules in telecommunications system software. They used different releases of the software and added new process metrics that kept not only changes in the code, but also requirements that affected the change. Although they showed that process metrics improved the capability of predicting defect-prone modules of a specialized dataset, process related metrics were not available in our dataset. Therefore, we did not have the opportunity to replicate this study.

Another study constructed a Bayesian Belief Network that merged both software and hardware development of an embedded system (Amasaki et al. 2005). They mainly reconstructed a Bayesian Network of Fenton et al. (2007) and modified it by using an embedded software development life cycle. The base data came from completed projects. The results of the embedded and general model indicated once more that it is necessary to use an embedded model for practical use. However, it is, in practice, difficult to construct a network using both software and hardware development stages. It requires too much time and effort for a company to build such a model specific to embedded systems. Instead we use static code attributes extracted from the source code since they have been widely used and easily collected through automated tools (Menzies et al. 2007a).

Lastly, Khoshgoftaar and Gao (1996) proposed a multi-strategy classifier for embedded software, which cascaded a rule-based approach with two case-based learning methods. This research showed the advantage of using combined techniques and encouraged researchers to try new approaches. We have also been motivated from the fact that combining techniques would present better detection capabilities. Therefore, we propose an ensemble of classifiers to benefit from the strengths of multiple techniques.

In this research we use static code attributes as predictor variables. A complete list of these attributes is available on line in the Promise repository (Boetticher et al. 2007). Static code attributes used in defect prediction have been accepted as valuable metrics by many researchers, for example (Menzies et al. 2007a; Munson and Khoshgoftaar 1992; Padberg et al. 2004). A recent study by Marchenko and Abrahamsson (2007) also supported the usefulness of static code attributes in embedded software.

Studies on defect prediction have used various machine learning techniques such as statistical analysis (Ohlsson and Wohlin 1998), regression (Basili et al. 1996), trees (Khoshgoftaar and Allen 1999; Khoshgoftaar and Gao 1996), neural networks (Khoshgoftaar and Szabo 1996) and Naïve Bayes (Menzies et al. 2007a; Munson and Khoshgoftaar 1992; Padberg et al. 2004; Turhan and Bener 2007). There have been discussions on finding the best classifier for defect predictors. Lessmann et al. (2008) argued that their 15 best performing classifiers were statistically indistinguishable from each other in terms of the area under the receiver operating characteristic (ROC) curve. The authors did not use any filtering or transformation techniques. Instead, they used the algorithms on the original data to measure their effectiveness on detecting defect-prone modules. Menzies et al. (2007a) showed that Naïve Bayes produced better results when static code attributes were log filtered and InfoGain was used as the attribute selection technique. In their later studies, several algorithms with several transformation techniques were investigated by Jiang et al. (2008) and it was seen that Naïve Bayes was arguably the simplest classification algorithm and that it was greatly improved with the discretization of data. One of the reasons for the success of Naïve Bayes over other methods is that it combines signals coming from multiple sources, i.e. attributes (Menzies et al. 2007a). In this paper we shift this focus from attribute level to classifier level. We investigate whether combining signals, i.e. predictions, from multiple predictors (classifiers) produced better results for locating defects. Therefore, we use an ensemble of classifiers to predict defects in embedded software. We collected data from the industry and donated it to the Promise repository (Boetticher et al. 2007). Therefore, all projects used in this study are available online. So, this work can easily be repeated, improved or refuted by other researchers (Boetticher et al. 2007).

3 Methodology

In previous sections, we have pointed out general characteristics of embedded systems and their problems during software development practices. Previous studies have pointed out that embedded systems should have a high level of reliability in their software such that any failure in the software may not be tolerated and repairing them would be very expensive (Khoshgoftaar and Allen 1999). On the other hand, due to tough competition and diminishing margins, managers look for ways to lower their costs without compromising the quality of end products (Tosun et al. 2008). This creates a dilemma such that to lower the cost, software development life cycle activities including testing has to be cost effective while to have a high quality product, test coverage should be increased. One solution to this problem may be to construct intelligent oracles to manage software testing activity.

3.1 Research questions

Recent research proposed by Rombach et al. (2008) points out that research and industry need to cooperate on current practices of software engineering in order to gain insight in how software development can be managed better. One of the reasons for current problems such as schedule and budget overrun, and system failures especially in safety-critical embedded applications is summarized as "*non-compliance with best practice principles of process design*" (Rombach et al. 2008). These typical challenges exist in practical software engineering due to lack of empirical evidence regarding the benefits and limitations of existing approaches in this context. For instance, reviews and inspections to detect failures earlier in development life cycle provide missing empirical facts to the industry. Results obtained from effective methods or tools, i.e. early detection mechanisms, would definitely guide managers to baseline their state of practice in a quantitative manner rather than relying on mythical facts (Rombach et al. 2008).

Related to this, we have examined current problems of software development in the embedded industry, which are briefly discussed in previous sections. Based on the fact that software engineering research should also complement new approaches with empirical facts from industrial case studies (Rombach et al. 2008), we define our research question that would address the challenges in embedded software development. From an industrial perspective, software managers aim to decrease their testing efforts while decreasing defect rates, thereby producing high quality embedded software. In our previous work, we proposed our research goal as "building a learning-based defect predictor for embedded systems" (Tosun et al. 2008). Using a combination of classifiers, we aimed to guide developers to defect-prone parts of the software and decrease testing efforts by predicting as many defects as possible, i.e. producing high detection rates (Tosun et al. 2008). Since developers eliminate most of the defects in the software with less testing effort, they would, in turn, improve their product quality. We observed that combining classifiers would detect 77% of the defective modules on average by inspecting 23% of the code in embedded systems (Tosun et al. 2008). However, increasing the detection rate also increased the false alarms when an ensemble of classifiers model was used. In mission-critical systems high detection rates at the cost of false alarms may be a desirable outcome. However, in commercial applications, companies need to employ cost effective oracles, since an increase in false alarms would waste inspection costs by guiding testers through actually safe modules. Therefore, in this paper, our objective is "building a learning-based defect predictor for embedded systems that would decrease false alarms while producing high detection rates".

3.2 Experimental design

We take an algorithmic approach where we aim to improve the algorithm by proposing a different ensemble model and a voting scheme. In our experimental design, we first introduce Ens_1 on NASA and local datasets. Then, we respectively apply our new ensemble approach, Ens_2 on these datasets to validate its classification accuracy:

1. Experiment #1 uses four NASA datasets, which represent embedded system characteristics to present the previous ensemble approach, Ens_1, and its results (Tosun et al. 2008). It also includes a comparison of this ensemble with a recent study based on Naïve Bayes classification (Menzies et al. 2007a).
2. Experiment #2 differs from Experiment #1 in terms of the training data used. Experiment #2 is also designed with Ens_1 to build a local model for the company, so it makes use of their local repository for training. Moreover, it validates the performance of Ens_1 on external datasets.
3. Experiment #3 checks the performance of three algorithms (Artificial Neural Network, Naïve Bayes and Voting Feature Intervals respectively) and their individual strengths on all datasets. It includes classification analyses on all datasets and cost-benefit analyses on local projects.
4. Experiment #4 proposes a new ensemble of classifiers, Ens_2, and uses both local projects and NASA datasets to validate its strength against Ens_1 as well as three algorithms.

In all experiments, we follow the procedures of *conceptual replication* studies, in which diverse sources and their effects on the results of the same research question are observed by proposing different experimental procedures (Shull et al. 2008). As indicated by Shull et al. (2008), replications help the software engineering researchers to address internal and external validity problems. These types of studies also lead the research community to build a solid knowledge about the influence of conditions on the experimental results and observations. Similarly in our case study, we observe a recent study on defect predictors

Fig. 1 The pseudo code for the experimental design

```
PROJECTS = {CM1, PC1, PC3, PC4, AR3, AR4, AR5}
$M = 10$
$N = 10$
PRE-PROCESSING = {Pca, none, InfoGain}
FILTERS = {none, log-filtering}
ALGORITHMS = {ANN, NB, VFI}
for all $P \in$ PROJECTS do
    for $i = 1$ to $M$ do
        $P'$ = Randomize the order of $P$
        $S$ = Generate $N$ small sets from $P'$
        for $j = 1$ to $N$ do
            $Test = S[j]$
            $Train = S - S[j]$
            for all $A \in$ ALGORITHMS do
                {Select a filtering method for $A$}
                Apply *filter* from FILTERS onto $A$
                Model = Apply $A$ on $Train$
                PredictionsA = Apply Model to $Test$
            end for
        end for
    end for
    {ensemble takes the majority vote for each input}
    Prediction = Ensemble(Pred_ANN, Pred_NB, Pred_VFI)
end for
```

and reproduce those via new techniques and datasets in order to find the best approach for embedded software.

We have overcome the sampling bias by using *M***N*-way cross validation where both *M* and *N* are selected as 10 (Hall and Holmes 2003). We create 10 stratified bins: 9 of these 10 bins are used as training sets while the last one is used as the test set. We randomize the dataset $M = 10$ times and create $N = 10$ sets in each iteration. We apply three algorithms on these different subsets and combine the results in an ensemble with two different voting schemes. The pseudo code of the model is shown in Fig. 1.

We use four different performance measures to evaluate the classification accuracy of our ensemble. These are probability of detection (*pd*), probability of false alarms (*pf*), balance (*bal*) and precision (*prec*). The first two measures can be analyzed on a typical ROC curve used for comparing different predictors (Heeger 1998). A ROC curve is represented in Fig. 2. In the ideal scenario (upper left corner in ROC curve), hit rate, *pd*, should be 1, i.e. the predictor should catch all defects. Moreover, false alarms, *pf*, should be 0, meaning that the predictor should never label a defect-free module as defective. In general, an increase in *pd* would also increase *pf* rates since the model triggers more often to achieve the ideal case (Menzies et al. 2007a). To see how close our estimates are to the ideal case, we use a balance metric, which is the Euclidean distance between the ideal point and where we are on the ROC curve in reality.

The last metric, *prec*, defines the ratio of correctly detected modules over all modules that are predicted as defective. It has a strong relation with *pd* and *pf*, such that when *pd* is fixed for a dataset, *pf* rate is controlled by precision and the class distribution of the data (Menzies et al. 2007b). To compute *pd*, *pf*, *bal* and *prec*, a confusion matrix in Table 1 is required (Heeger 1998). From the confusion matrix, performance measures are calculated using Eqs. 1, 2, 3 and 4.

$$pd = \mathrm{TP}/(\mathrm{TP} + \mathrm{FN}) \tag{1}$$

Fig. 2 A simple ROC curve

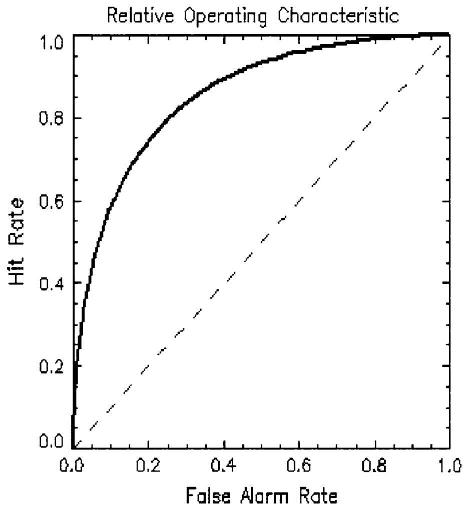


Table 1 Confusion matrix

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

$$pf = \mathrm{FP}/(\mathrm{FP} + \mathrm{TN}) \tag{2}$$

$$bal = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \tag{3}$$

$$prec = \mathrm{TP}/(\mathrm{FP} + \mathrm{TP}) \tag{4}$$

Finally, we conduct a cost-benefit analysis to evaluate the decrease in testing effort, which is explained in Sect. 4.2.

3.3 Data sources

We use data from two different organizations which are publicly available in the Promise Data Repository (Boetticher et al. 2007). The first four projects are from NASA and they display characteristics of embedded systems (Oral and Bener 2007). The other three projects are from our industry partner. In our previous study (Tosun et al. 2008), reasons to use organizationally and operationally different software were explained as follows:

– To make a generalization by expanding the test-bed of embedded datasets that include projects from various sources.
– To prove the external validity of our results with different resources.
– To validate the performance of the ensemble model on the NASA dataset, which has been used several times in other studies (Basili et al. 2002; Menzies et al. 2007a).

In order to determine which NASA datasets to use, we derived the characteristics from available requirements and application areas. After examining the domain requirements, we include CM1, PC1, PC3 and PC4 in our test-bed (Oral and Bener 2007; Tosun et al. 2008). CM1 is a spacecraft instrument and has the characteristics of embedded systems such as timeliness, liveness (meaning that the system must not terminate or run into deadlock), heterogeneity and reactivity. Other projects are from a flight software system which is one area that embedded software is necessarily implemented.

There have been discussions on the quality of NASA datasets, such as in the work of (Khoshgoftaar and Seliya 2004; Khoshgoftaar et al. 2005). Authors argued that the accuracy of predictive models is highly influenced by a) the quality of data and b) the appropriateness of selected classifier. The quality of data and noise are always issues in software datasets, and it is possible to overcome this problem either by eliminating noisy instances via noise detection algorithms or by using noise-tolerant algorithms. According to the authors, noise detection techniques such as polishing are appealing, however they should be carefully applied in order not to remove the correct instances or insert further noise. Thus, they proposed ensemble of classifiers to filter noise (Khoshgoftaar et al. 2005). The results of their study showed that an ensemble of classifiers provides more confident predictions, because the more classifiers are used, the most conservative the predictions are.

The noise in data is a very well known problem in machine learning and many researchers in software engineering (Twala and Cartwright 2010), bioinformatics (Libralon et al. 2009) and pattern recognition (Verbaeten and Assche 2003; Xu et al. 2009) have investigated the problem. They reported that both ensemble methods and data filtering techniques are effective ways to detect and remove noise in the data. Our objective in using ensemble methods in this research has been to improve the performance of defect prediction methods to detect more defects and to produce as low false alarms as possible. Considering that software data may have noise affecting the prediction outcome, the ensemble method not only improves the prediction outcome but also is effective in noise removal as indicated in the literature.

Fenton and Neil (1999) also investigated the quality of software engineering data used in defect prediction studies. Authors argue that statistical techniques and the quality of data often undermine model validity. They focused on issues such as multicolinearity and data points removal in order to filter highly correlated input variables, rank the most significant ones or filter noisy instances. The authors suggested principal components analysis to solve the problem of multicolinearity. However, removing data points that seem noisy is a more difficult issue, since you really do not know which data points are actually noisy before the analysis, and hence, it is difficult to tell your reasoning beyond removing those points.

We used principal components to reduce the number of input variables and rank variables that are significantly correlated with defects. However, we do not include data filtering and polishing within the scope of this paper. We also believe that our proposed ensemble would address the potential noise in NASA datasets and reduce the effect of noisy instances on a single classifier.

The second data source is from our industry partner located in Turkey. Since they implement software for white-goods, their functional requirements are different than safety- and mission-critical NASA projects. However, this also brought diversity into our study for generalization. The general properties of all projects used in this study can be seen in Table 2. Projects from our industry partner are referred as AR3, AR4 and AR5. Their defect rates show that class distribution is highly imbalanced.

To assess the performance of our proposed model prospectively on ongoing projects, we use a new local dataset, namely AR6. We donated this dataset as well as other local projects to Promise (Boetticher et al. 2007) by providing static code attributes and defect

Table 2 Properties of datasets

Project	# Attributes	# Modules	Defect rate
CM1	19	498	0.09
PC1	19	1109	0.06
PC3	19	1563	0.10
PC4	19	1458	0.12
AR3	29	63	0.12
AR4	29	107	0.18
AR5	29	36	0.20
AR6	29	101	0.15

labels after the project was completed. Properties of AR6 used in Sect. 4.3 (Results on Ongoing Projects) are also presented in Table 2.

3.4 The ensemble of classifiers

In defect prediction studies, selection of a suitable classification algorithm that is always the most accurate is almost impossible, as it is the case in all domains and for all problems (Kuncheva 2004). Every algorithm has different assumptions on the distribution of data and produces different error rates. A combination of classifiers in an ensemble may improve the performance when the classifiers are carefully selected. In our previous study, we proposed to combine three different classifiers, two of which, i.e. Neural Networks and Naïve Bayes, were used in other defect prediction studies (Khoshgoftaar and Szabo 1996; Menzies et al. 2007a; Turhan et al. 2009; Turhan and Bener 2007, 2008). The third method was Voting Feature Intervals (Demiroz and Guvenir 1997), which, to the best of our knowledge, has not been used in any defect prediction study.

There are several techniques to calculate the output and accuracy of the ensemble such as voting, weighted voting or other fixed rules (Kittler et al. 1998). To make a classification, we previously employed a simple voting methodology (Tosun et al. 2008), that is, we classified a module as defective/defect-free when majority of the votes in ensemble was defective/defect-free. In this paper, we propose to combine two classifiers discarding the Neural Network and change the voting scheme. We have seen that a suitably chosen set of classifiers and voting scheme would lead to better performance. The reason for choosing a different voting strategy is that if the number of the classifiers in the ensemble is even, it is possible to find a case where there are equal numbers of defective and defect-free decisions for a module. We have applied a different voting approach to overcome such situations. It is similar to *AND* logical operator: If all algorithms classify a module as defective, then it is labeled as defective. Otherwise, it is labeled as defect-free.

As mentioned before, we use three classifiers throughout this study. The first algorithm of the proposed model, the Naïve Bayes method, is implemented as in (Menzies et al. 2007a). It is derived from Bayes' Theorem, where the prior probability and evidence explain the posterior probability of an event.

$$P(H|E) = \frac{P(H)}{P(E)} \prod_{i} P(E_i|H) \tag{5}$$

In Eq. 5, we denote the class label of a module, i.e. defective or defect-free, as H. Evidences, E_i, are the attributes collected from the projects (i.e. static code attributes).

In order to find the posterior probability of a new module (given its attributes) we need to use the prior probabilities of two classes and the evidence from historical data. For numeric attributes with pairs of mean and standard deviation (μ, σ), probability of each attribute given H is calculated with a Gaussian density function (Alpaydin 2004):

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

First, data was log filtered, i.e., attribute values were replaced with their logarithms since the distribution of static code attributes is highly skewed and heavy tailed. Second, the InfoGain attribute selection method was used to rank the attributes in datasets. This method selects the attributes that provide the largest Information Gain. We selected the top eleven attributes from the resulting list, based on the analysis of our prior research (Oral and Bener 2007).

The second algorithm of our model, Artificial Neural Network, is used as in the work of Khoshgoftaar and Szabo (1996) with project attributes as inputs and class labels as output. We implemented dynamic node creation to determine the number of hidden nodes in the middle layer of the network (Alpaydin 2004). This procedure starts with one hidden layer and one node in this layer. Then it iteratively adds new nodes to the hidden layer or increases the number of hidden layers until there is no more improvement in terms of prediction performance. Since the number of hidden units in a layer is preferred to be less than the number of inputs (i.e. project attributes), we used PCA to reduce the dimensionality and complexity of the model (Alpaydin 2004). We transformed data to a new coordinate system where we chose eigenvectors which cover 95% of the variance in the original data in order to form principal components (i.e. weighted project attributes). Before calculating eigenvectors, we normalized the original data, i.e. project attributes, in the form of [0,1].

The last algorithm of the ensemble is Voting Feature Intervals (VFI)(Demiroz and Guvenir 1997). We compared this algorithm with Naïve Bayes on real-world datasets and concluded that it is more robust, comparable and even better than Naïve Bayes in terms of accuracy. However, accuracy is not a good performance measure for imbalanced datasets (Menzies et al. 2007a). We show that VFI is comparable with Naïve Bayes in terms of other performance metrics as well. VFI places minimum and maximum values of each attribute for each module. The difference between these values forms an interval for each attribute. When a new input is taken, the number of samples of each class within this interval are counted and summed to form votes. The prediction is based on selecting the class with the highest vote. Since VFI uses all attributes, it is not recommended to use any pre-processing algorithm prior to VFI (Demiroz and Guvenir 1997). Since one of the objectives is to improve the defect prediction rate, we decide to use VFI in software defect prediction.

As explained in our previous study (Tosun et al. 2008), the first ensemble, Ens_1, applied a simple voting strategy such that if at least 2 out of 3 algorithms predicts a module as defective, then the module is classified as defective. On the other hand, in this study, the proposed ensemble (Ens_2) takes the predictions from each algorithm and predicts a module as defective if all of the algorithms predicted it as defective. The ensemble combines different strategies to learn the datasets better since each algorithm forms a model based on their findings from the data. Therefore, the model does not solely learn on a single algorithm. Rather, we observe different predictions and combined the strengths of multiple techniques to determine the outcome. After combining predictions with a voting strategy, the ensemble computes its prediction performance. It does not apply any weighting mechanism for the voting. Thus, each algorithm has the same impact on the final prediction. Ens_2 has a lower tendency of predicting a module as defective than Ens_1, since it does not take into account the predictions of ANN and looks for a common decision from both NB and VFI. In Sect. 4 we present the results of each algorithm as well as both ensembles.

4 Results

4.1 Classification analysis

In Experiments #1 and #2, the objectives are (a) building an ensemble of classifiers to predict defect-prone modules in embedded software systems, and (b) achieving high *pd* rates with the cost of high *pf* and *prec* rates using Ens_1. In Experiment #3, we investigate separate performances of the algorithms used in Ens_1. According to the findings in Experiment #3, we calibrate the proposed ensemble as Ens_2 to improve the prediction performance in terms of *pf* and *prec* rates (in Experiment #4).

In Experiment #1, we compare Ens_1 with the model proposed on NASA datasets (Menzies et al. 2007a), CM1, PC1, PC3 and PC4 to validate its performance. We conjecture that if our results are at least comparable with their study, we can verify that the ensemble approach is worth using in the context of embedded software (Tosun et al. 2008). In Table 3, the prediction performances of Ens_1 and the model of Menzies et al. (2007a) are presented. From the results, we could argue that Ens_1 achieves good results in all NASA datasets. We outperform the results of Menzies et al. (2007a) in terms of *pd* on average from 74% to 82%. Although the ensemble increases false alarms by 8%, and hence, decreases precision by 3%, Mann–Whitney *U* tests show that the balance rates of two models are not significantly different. Thus, in the first ensemble, Ens_1, we manage to achieve high detection rates on public datasets.

The results of Experiment #2, which presents the performance of Ens_1 on local datasets, can be seen in Table 4. For local datasets, AR3, AR4 and AR5, we correctly predict 76% of defective modules on average. Moreover, the average pf rate for local projects is 22%, which is not as high as in the case of NASA datasets. We also increase precision rates up to 71% (45% on average). This shows us in our case, defect prediction using an ensemble model would definitely be helpful for detecting as many defects as possible and, hence, reducing testing effort. It would correctly classify defective modules and guide developers

Table 3 Experiment #1: Comparison of Ens_1 with (Menzies et al. 2007a) (Bench) in public datasets (Tosun et al. 2008)

	pd		*pf*		*prec*		*bal*	
	Ens_1	Bench	Ens_1	Bench	Ens_1	Bench	Ens_1	Bench
CM1	0.81	0.71	0.41	0.27	0.18	0.22	0.68	0.72
PC1	0.75	0.48	0.28	0.17	0.17	0.17	0.74	0.61
PC3	0.81	0.80	0.36	0.35	0.20	0.21	0.71	0.71
PC4	0.90	0.98	0.35	0.29	0.26	0.32	0.74	0.79
Avg.	0.82	0.74	0.35	0.27	0.20	0.23	0.72	0.74

Table 4 Experiment #2: Performance of Ens_1 model on the company's local data

	AR3	AR4	AR5	Avg.
pd	0.62	0.80	0.85	0.76
pf	0.20	0.36	0.10	0.22
prec	0.31	0.34	0.71	0.45
bal	0.69	0.70	0.81	0.73

Table 5 Experiment #3: Performance of ANN

	CM1	PC1	PC3	PC4	$Avg._{Nasa}$	AR3	AR4	AR5	$Avg._{local}$
pd	0.74	0.67	0.74	0.89	0.76	0.12	0.45	0.50	0.36
pf	0.40	0.26	0.35	0.29	0.32	0.31	0.43	0.43	0.39
prec	0.17	0.16	0.19	0.30	0.21	0.05	0.19	0.25	0.17
bal	0.66	0.71	0.69	0.78	0.71	0.34	0.50	0.53	0.46

Table 6 Experiment #3: Performance of NB

	CM1	PC1	PC3	PC4	$Avg._{Nasa}$	AR3	AR4	AR5	$Avg._{local}$
pd	0.57	0.54	0.69	0.70	0.63	0.62	0.75	0.75	0.71
pf	0.28	0.21	0.24	0.26	0.25	0.32	0.32	0.10	0.25
prec	0.18	0.16	0.25	0.27	0.22	0.22	0.35	0.68	0.42
bal	0.64	0.64	0.72	0.72	0.68	0.64	0.71	0.80	0.72

Table 7 Experiment #3: Performance of VFI

	CM1	PC1	PC3	PC4	$Avg._{Nasa}$	AR3	AR4	AR5	$Avg._{local}$
pd	0.83	0.95	0.90	0.96	0.91	0.62	0.80	0.87	0.76
pf	0.58	0.45	0.60	0.63	0.57	0.14	0.43	0.21	0.26
prec	0.14	0.14	0.15	0.17	0.15	0.39	0.30	0.54	0.41
bal	0.57	0.68	0.57	0.55	0.59	0.71	0.66	0.82	0.73

to fewer modules to inspect rather random reviews, which is the case in our industry partner.

When we analyze Ens_1 for all ($NASA$ + $local$) projects, the average performance is (79, 29%) in terms of (*pd*, *pf*). This means that the ensemble can predict 4 out of 5 defective modules, while making false alarms in 3 out of 10 defect-free modules. Therefore we can conclude that our previous model is useful at least for the embedded systems we used. As mentioned, Ens_1 consists of ANN, NB and VFI algorithms and their strengths are combined to achieve better predictions. On the other hand, it is a good practice to check individual performances of the algorithms. Thus, we evaluate each algorithm in Experiment #3, whose results are illustrated in Tables 5, 6 and 7.

From these results, it is clear that overall performance of an ensemble is not directly affected by any of these algorithms. It does not compute average performances for each dataset. In contrast, the majority of predicted class for each module is selected as the final predicted class value. Thus, *Ens*$_1$ computes *pd* as high as possible.

When three algorithms are examined independently, ANN performs the worst, i.e. 36% pd and 17% prec, on local projects. The reason for this behaviour may be due to the size of local datasets, which contain very few instances (less than 200) compared to NASA projects. Since ANN has too many parameters, such as the number of hidden units and hidden layers that need to be optimized, it requires work on larger datasets (like in NASA). Therefore, ANN is not a suitable algorithm for small datasets. VFI and NB compete with the ensemble in terms of pd and pf respectively. According to Mann–Whitney *U* tests, VFI computes higher *pd* rates, whereas it also produces higher false alarms which make the balance rate insignificant from the others. On the other hand, NB contributes to the model by decreasing false alarms significantly (validated by Mann–Whitney *U* tests). However, it does not compute as high *pd* rates as in *Ens*$_1$.

Finally, when *prec* rates are investigated, there is a clear conclusion according to Mann–Whitney tests: NB is insignificant with *Ens*$_1$ on NASA datasets, whereas it is outperformed by *Ens*$_1$ on local projects. On the other hand, VFI is outperformed by *Ens*$_1$ on all projects except AR3. *Ens*$_1$ produces 31% *prec* rate on average, compared to 26% produced by VFI.

To lower *pf* rate, as in the case of NB, and achieve high *pd* rates, as in the case of VFI, we build a new ensemble in Experiment #4. We extract ANN from the previous ensemble and perform the classification analysis again for both local projects and NASA datasets. The new ensemble, *Ens*$_2$, aims to lower false alarms and improve precision rates of the previous ensemble specifically for this case study. Furthermore, we aim to see whether this new ensemble is better than the model proposed by Menzies et al. (2007a). This time, *Ens*$_2$ uses AND approach for voting: It predicts a module as defective if both algorithms predict it as defective. Otherwise, the module is classified as defect-free. Table 8 shows changes in the prediction performance, i.e., *pd*, *pf* and *bal*.

Results on local projects (AR3, AR4, AR6) show that false alarms are decreased by 5%, i.e. from 22% in *Ens*$_1$ to 17% in *Ens*$_2$, and achieve potential savings in exchange for 7% changes in pd rates using *Ens*$_2$. We applied Mann–Whitney *U* tests and observed that the decrease in false alarms is statistically significant with 95% confidence. Although *Ens*$_1$ is significantly better than *Ens*$_2$ in terms of pd, they are insignificant in terms of *bal*. The reason for that is also related to the significant increase in precision rates (from 45% in *Ens*$_1$ to 48% in *Ens*$_2$). Our objective in Experiment #4 is to achieve high balance and precision rates by lowering false alarms. According to Table 8, *Ens*$_2$ is not successful in increasing balance rates, but it manages to improve precision of the predictions and lower false alarms while keeping detection rates around 70%.

We validated the performance of *Ens*$_2$ on NASA datasets to compare with both *Ens*$_1$ and the model of Menzies et al. (2007a). In Table 8, it is easily seen that the new ensemble significantly improve false alarms, and hence, precision and balance rates on NASA datasets. As seen in Table 3, *Ens*$_1$ produces (82, 35, 20, 72%) in terms of (*pd*, *pf*, *prec*, *bal*)

Table 8 Experiment #4: Performance of the new ensemble, *Ens*$_2$

	AR3	AR4	AR5	$Avg._{local}$	CM1	PC1	PC3	PC4	$Avg._{Nasa}$
pd	0.62	0.70	0.75	0.69	0.82	0.86	0.86	0.83	0.84
pf	0.13	0.29	0.10	0.17	0.01	0.00	0.40	0.19	0.15
prec	0.41	0.36	0.68	0.48	0.93	1.00	0.20	0.38	0.63
bal	0.72	0.70	0.81	0.74	0.81	0.86	0.58	0.75	0.75

on NASA datasets, whereas Ens_2 significantly reduces false alarms by 20% and increases the precision by 43%. Therefore, although pd rates are the same in both ensembles (78%), Ens_2 is much better for embedded software systems in terms of the confidence (precision) of predictions.

Finally, these results are also strengthened by the cost-benefit analysis to measure the effects of defect predictors on decreasing the testing effort.

4.2 Cost-benefit analysis

We decided to perform a cost-benefit analysis on local projects to see the effects of low false alarm rates on the inspection effort. Moreover, it is important for managers and developers of the company to see the benefit of using a prediction model. In the company that we cooperated with during this study, testers and developers often started testing from a random module and they used their experience as well as prior knowledge to determine which modules were defect-prone. However, this approach is human-dependent, hence software managers wanted to avoid this random strategy. Therefore, managers also supported this analysis to see the practical benefits of using a prediction model.

It may be more effective to compare cost-benefit analysis with a testing approach that looks at modules according to their size (LOC). However, our industry partner did not use such a strategy during testing. Thus, a comparison with the LOC strategy would not present the real impact on the company in terms of inspection effort.

In a study by Arisholm and Briand (2006), a simple methodology was proposed to assess the cost effectiveness of a prediction model on the verification effort. Their basic assumption was that the number of lines of code was proportional to the verification effort of predicted defective classes. That is, a random selection of classes during a system testing would require the verification of X% of the code to find at most X% of defects. Arisholm and Briand observed that only 50% of the code should be inspected to detect 70% of defects. This also reduces the verification effort by 29% (Arisholm and Briand 2006). We formulate the gain in the verification/inspection effort as follows:

$$\text{gainedEffort} = 100\times(\text{LRT} - \text{LDF})/\text{LRT} \tag{6}$$

In Eq. 6, LRT is the number of lines of code (LOC) that must be inspected using a Random Testing strategy and it is computed as *totalLOC* * *pd*, whereas LDF is the number of LOC that must be inspected using a Defect Predictor. Using the same formula, we compute the gain in verification effort (%) for local projects.

For instance, if we choose a random selection of classes when testing AR3, Ens_1 needs to check 62% of the code, 3487 LOC (LRT in Eq. 6), to detect a maximum of 62% of defective modules, which is *pd* rate on AR3. On the other hand, the model tells us to check only 16 modules, which is in total 1986 LOC (LDF) out of 5624 LOC. Therefore, we decrease the inspection effort, i.e. LOC, down to 35% and reduce the verification effort by 43% (gainedEffort).

Tables 9, 10 and 11 show the cost-benefit analysis of using such a prediction model for the company's software team. For each dataset, we present the analysis in separate tables. Rows represent total number of LOC in the project (*totalLOC*), probability of detection rate (*pd*), number of defective LOC suggested by the prediction model (*LDF*), number of LOC required by a random inspection strategy (*LRT*), percentage of LOC traced to find defective modules (*changeLOC*) and decrease in the verification effort to predict defective modules using a defect prediction model rather than a random strategy (*gainedEffort*). The tables show the analysis for Ens_1 and Ens_2 models as well as three classifiers.

Table 9 Experiment #3: Cost-benefit analysis of VFI, ANN, NB, Ens_1 and Ens_2 on AR3

	VFI	ANN	NB	Ens_1	Ens_2
pd	0.62	0.12	0.62	0.62	0.62
pf	0.14	0.31	0.32	0.20	0.13
totalLOC	5624	5624	5624	5624	5624
LDF	1308	3198	2391	1986	1194
LRT	3487	675	3487	3487	3487
changeLOC (%)	23	57	43	35	21
gainedEffort (%)	63	−355	32	43	66

Table 10 Experiment #3: Cost-benefit analysis of VFI, ANN, NB, Ens_1 and Ens_2 on AR4

	VFI	ANN	NB	Ens_1	Ens_2
pd	0.80	0.45	0.75	0.80	0.70
pf	0.43	0.43	0.32	0.36	0.29
totalLOC	9196	9196	9196	9196	9196
LDF	3495	6900	2991	3624	2591
LRT	7357	4138	6897	7357	6437
changeLOC (%)	38	75	33	39	28
gainedEffort (%)	52	−67	57	51	60

Table 11 Experiment #3: Cost-benefit analysis of VFI, ANN, NB, Ens_1 and Ens_2 on AR5

	VFI	ANN	NB	Ens_1	Ens_2
pd	0.87	0.50	0.75	0.85	0.75
pf	0.21	0.43	0.10	0.10	0.10
totalLOC	2732	2732	2732	2732	2732
LDF	709	2001	502	567	502
LRT	2390	1366	2049	2390	2049
changeLOC (%)	26	73	18	20	18
gainedEffort (%)	70	−46	75	76	75

From the analysis, we can see that ANN has a negative impact on Ens_1 model. ANN significantly increases the verification effort so that testing effort is also adversely affected. However, Ens_1 yields a similar performance to that of VFI and NB, and each of them is superior to the other in only one of the projects. Ens_1 detects 76% of defective modules by inspecting 32% of the code. Moreover, it reduces the verification effort of predicting defective modules by 56% as opposed to the efforts of VFI and NB, which are 62% and 55%.

With Ens_2, we reduce the inspection effort down to 23%, which was 32% in Ens_1, to detect 69% of defective modules. Moreover, we lower the verification effort by 67%, which was 56%, and obtain an ensemble that works better than VFI and NB in terms of verification effort for predicting defective modules.

4.3 Results on ongoing projects

Our results on past projects show the industry partner that this research is relevant for their embedded software especially in allocating resources for their testing effort. In a highly competitive market with a tight profit margin, they believe that defect predictors would help managers to efficiently allocate their scarce resources before the test begins. That is why we decided to run our model prospectively on an ongoing project, AR6. We made predictions for AR6 before the testing phase actually took place. In AR6, there were 101 modules, 17 of which were predicted as defective. After we made the predictions for AR6, we waited until all revisions based on the testing results were made. Based on the final test results, our classifications are 80% accurate (12 out of 15 defective modules are correctly predicted) with 33% false alarms.

5 Threats to validity

Due to many factors affecting the software process and products, it is hard to claim generality of results in the software engineering domain. We have included different project data from different companies to overcome this threat. The external validity of generalizing from NASA projects has been discussed by Basili et al. (2002) and Menzies et al. (2007a). In summary, NASA uses contractors who are contractually obliged to demonstrate their understanding and usage of current industrial best practices. Nevertheless, in order to test claims of external validity for our experiments, we have also used additional data that we collected from our industry partners. In order to construct our ensemble, we have used two different voting schemes since voting schemes may affect the performance of an ensemble. We also observe that the selection of algorithms in the ensemble is quite important for the overall performance since removing ANN from the ensemble improves the performance (pf, prec, balance) of the defect predictor as well as the gain in inspection effort. We have applied cross-validation in all experiments to avoid sampling bias.

During this study, our objective was to increase the defect detection capability of a predictor model while decreasing false alarms and lowering inspection costs. Although this does not ensure that the predictor is a generic one for all software systems, it is intended to optimize embedded software development in terms of high quality using less testing effort. Thus, we have not only evaluated the classification analysis of our proposed model using *pd*, *pf*, *prec* and *bal*, but also included a cost-benefit analysis to investigate the testing effort that the model suggests for finding defect-prone modules. Therefore, in the embedded software context, our model is successful in finding defective modules of local projects, 69% of the time on average, while inspecting only 23% of the code. Furthermore, it produces significant results for NASA datasets by lowering false alarms by 20% and hence, improving precision rates by 43%.

During cost-benefit analysis, we compare the performance of the proposed model with a random testing strategy. Although there may exist other, more sophisticated, strategies, such as a LOC based ranking, used by large-scale software organizations, there was no formal testing strategy in our industry partner. Hence, we decided that using the simplest strategy would provide the gain in inspection effort more clearly. Finally, to ensure the statistical validity of our results, we have conducted the Mann–Whitney *U* test to evaluate whether the changes in false alarms can be significantly validated. Statistical tests prove the validity of our results in terms of false alarms and inspection efforts.

6 Conclusions and contributions

Many defect predictors have been proposed in previous studies. However only a few of them target embedded systems. To extend these studies, we collected local project data from embedded software. Since the data used in similar experimental settings is not publicly available, we donated our data to the Promise repository to encourage other researchers to repeat, improve or refute this study. Moreover, we use NASA projects that show embedded system characteristics to promote the usage of public datasets. Using various datasets, we propose a model based on an ensemble of classifiers, which includes Naïve Bayes and Voting Feature Intervals. We validate that an ensemble of classifiers takes the advantages of the different methods such that it combines multiple signals. Hence, it is a valuable methodology for defect prediction. Furthermore, the proposed ensemble Ens_2 reduces *pf* and increases *prec* when the combination of classifiers and voting scheme are suitably chosen.

For mission-critical systems, increasing the prediction rates is more important than lowering the false alarm rates. However, in a commercial setting where margins are tight, companies would like to equally focus on lowering false rates as well as increasing the prediction rate in a learning-based defect prediction model. This is the case in our industry partner, a white goods manufacturer. Therefore, we propose Ens_2. On *all* projects, Ens_2 detects 78% defective modules while producing 16% false alarms. Although it does not improve the prediction rates of the first ensemble (79%), Ens_2 significantly improves the precision from 31 to 56%. Furthermore, it manages to improve the balance rates from 60% to 78% on average (*all* projects). After the model is built, software managers have decided to integrate this model into their development practices such that testers run the model before they start testing any software product and they prioritize software modules according to the predictions of the model. Our future plan is to conduct a replication study of Koru et al. (2009) that investigates the relationship between the size and defect-proneness of software modules in the embedded systems domain.

We present the cost-benefit analysis of local projects to see how a prediction model would reduce the verification effort. Our results show that our proposed model could bring significant cost savings to the company. It is enough to trace nearly 23% of the code to catch around 70% of defective modules with the proposed ensemble, which is consistent with the Pareto distribution. So, instead of spending 70% of the verification effort for detecting 70% of the defective modules, the company can allocate fewer resources to detect the same level of defects.

Another contribution is that we collect data from embedded software projects and share them with the research community. Using an ongoing project in the company, we present our practical experiences and show that defect predictors contribute much to the decision-making process in the test phase.

As a future direction, the ensemble can be combined with multiple algorithms other than NB, ANN and VFI. The voting strategy of the ensemble can be changed with a weighted voting mechanism, but this technique has the disadvantage of using part of the data to calculate the weights and using less data to train the single classifiers. Moreover, predictions of the model can be analyzed with new datasets if and when they become available. We also plan to collect requirement metrics and construct a defect prediction model for both implementation and requirement defects. We have observed that misinterpreting the requirements causes most of the revisions reported in defect logs. However, our research partner does not use any automated tool that examines requirement documents and collects related metrics. If metrics for requirement documents can be collected as easily as code metrics, it will be valuable to generate a model with additional attributes from the requirement phase.

Acknowledgments This research is supported in part by Turkish State Planning Organization (DPT) under project number 2007K120610.

References

Adrian, R. W., Branstad, A. M., & Cherniavsky, C. J. (1982). Validation, verification and testing of computer software. *ACM Computing Surveys, 14*(22), 159–192.

Alpaydın, E. (2004). *Introduction to machine learning*. Cambridge: The MIT Press.

Amasaki, S., Takagi, Y., Mizuno, O., & Kikuno, T. (2005). Constructing a Bayesian belief network to predict final quality in embedded system development. *IEICE Transactions on Information and Systems, 134*, 1134–1141.

Arisholm, E., & Briand, L. C. (2006). Predicting fault-prone components in a java legacy system. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (pp. 8–17). ACM.

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

Basili, V. R., McGarry, F. E., Pajerski, R., & Zelkowitz, M. V. (2002). Lessons learned from 25 years of process improvement: The rise and fall of the NASA software engineering laboratory. In *ICSE '02: proceedings of the 24th international conference on software engineering* (pp. 69–79). ACM.

Biffl, S., Halling, M., & Kszegi, S. (2003). Investigating the accuracy of defect estimation models for individuals and teams based on inspection data. In *ISESE '03: Proceedings of the 2003 international symposium on empirical software engineering* (p. 232). IEEE Computer Society.

Boetticher, G., Menzies, T., & Ostrand, T. J. (2007). The PROMISE repository of empirical software engineering data West Virginia University, Lane Department of Computer Science and Electrical Engineering.

Brooks, F. P. (1995). *The mythical man-month: Essays on software engineering*. Reading: Anniversary Edition Addison-Wesley

Demiroz, G., & Guvenir, H. A. (1997). Classification by voting feature intervals. In *ECML '97: Proceedings of the 9th European conference on machine learning* (pp. 85–92). Springer.

Fagan, M. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal, 15*, 182–211.

Fenton, N., Neil, M., Marsh, W., Hearty, P., Marquez, D., Krause, P. & Mishra, R. (2007). Predicting software defects in varying development lifecycles using Bayesian nets Information and Software Technology. *Butterworth-Heinemann, 49*, 32–43.

Fenton, N., & Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering, 25*(5), 675–689

Hall, M. A., & Holmes, G. (2003). Benchmarking attribute selection techniques for discrete class data mining IEEE transactions on knowledge and data engineering. *IEEE Educational Activities Department, 15*, 1437–1447.

Heeger, D. (1998). *Signal detection theory*.

IEEE Glossary of Software Engineering Terminology. (1990). ANSI/IEEE Standard 610.12 IEEE, New York.

Jiang, Y., Cukic, B., & Menzies, T. (2008). Can data transformation help in the detection of fault-prone modules? In *DEFECTS '08: Proceedings of the 2008 workshop on defects in large software systems* (pp. 16–20). ACM, New York.

Kocaguneli, E., Tosun, A., Bener, A., Turhan, B., & Caglayan, B. (2009). Prest: An intelligent software metrics extraction. In *Analysis and Defect Prediction Tool, SEKE'09: Proceedings of the 21st international conference on software engineering & knowledge engineering (SEKE'2009)* (pp. 526–529). Boston, MA, USA, July 1–3.

Lee, E. A. (2002). *Embedded software, advances in computers 56*. London: Academic Press.

Kan, S. H. (2002). *Metrics and models in software quality engineering*. Reading: Addison-Wesley.

Khoshgoftaar, T. M., & Szabo, R. M. (1996). Using neural networks to predict software faults during testing. *IEEE Transactions on Reliability, 45*, 456–462.

Khoshgoftaar, T. M., & Allen, E. B. (1999). Predicting fault-prone software modules in embedded systems with classification trees. In *HASE '99: The 4th IEEE international symposium on high-assurance systems engineering* (p. 105). IEEE Computer Society.

Khoshgoftaar, T. M., & Seliya, N. (2003). Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Software Engineering, 8* 255–283.

Khoshgoftaar, T., & Seliya, N. (2004). The necessity of assuring quality in software measurement data. In *METRICS '04: Proceedings of the software metrics, 10th international symposium* (pp. 119–130). IEEE Computer Society, Washington, DC, USA.

Khoshgoftaar, T., Zhong, S., & Joshi, V. (2005). Enhancing software quality estimation using ensemble-classifier based noise filtering. *Intelligent Data Analysis, 9*, 3–27

Khoshgoftaar, T. M., & Gao, K. (2006). Assessment of a multi-strategy classifier for an embedded software system. In *ICTAI '06: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence* (pp. 651–658). IEEE Computer Society.

Kittler, J., Hatef, M., Duin, R. P. W., & Matas, J. (1998). On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 20*(3), 226–239.

Kocaguneli, E., Tosun, A., Bener, A., Turhan, B., Caglayan, B. (2009). Prest: An intelligent software metrics extraction. In *Analysis and Defect Prediction Tool, SEKE 2009: Proceedings of the 21st international conference on software engineering and knowledge engineering* (pp. 637–642).

Koru, A. G., Zhang, D., El Emam, K., & Liu, H. (2009). An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering, 35*(2), 293–304

Kuncheva, L. I. (2004). *Combining pattern classifiers: Methods and algorithms*. Hoboken: Wiley-Interscience.

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.

Libralon, G., Carvalho, A., & Lorena, A. (2009). Ensembles of pre-processing techniques for noise detection in gene expression data. In *Proceedings of the 15th international conference on advances in neuro-information processing* (pp. 486–493). New Zealand.

Li, Q., & Yao, C. (2003). *Real-time concepts for embedded systems*. San Francisco: CMP Books.

Marchenko, A., & Abrahamsson, P. (2007). Predicting software defect density: A case study on automated static code analysis. In *XP '07: Proceedings of the International Conference on Agile Processes in Software Engineering and Extreme Programming* (pp. 137–140). Springer.

Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, IEEE Computer Society, 32*(11), 2–13

Menzies, T., Dekhtyar, A., Distefano, J., & Greenwald, J. (2007). Problems with Precision: A response to comments on data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33*(9), 637–640.

Munson, J. C., & Khoshgoftaar, T. M. (1992). The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering, IEEE Press, 18*, 423–433.

Ohlsson, N., & Wohlin, C. (1998). Experiences of fault data in a large software system. *Failure and Lessons Learned in Information Technology Management, 2*, 163–171.

Oral, A. D., & Bener, A. (2007). Defect Prediction for Embedded Software. *ISCIS '07: Proceedings of the 22nd international symposium on computer and information sciences* (pp. 1–6).

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.

Padberg, F., Ragg, T., & Schoknecht, R. (2004). Using machine learning for estimating the defect content after an inspection. *IEEE Transactions on Software Engineering, IEEE Press, 30*, 17–28.

Rombach, D., & Seelisch, F. (2008). Formalisms in software engineering: Myths Vs. empirical facts, CEE-SET'07. *Lecture Notes in Computer Science (LNCS), 5082*, 18–25.

Runeson, P., Ohlsson, M. C., & Wohlin, C. (2001). A classification scheme for studies on fault-prone components. In *PROFES '01: Proceedings of the third international conference on product focused software process improvement* (pp. 341–355). Springer, Berlin.

Shull, F., Boehm, V. B., Brown, A., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., & Zelkowitz, M. (2002). What we have learned about fighting defects. In *Proceedings of the eighth international software metrics symposium* (pp. 249–258).

Shull, F. J., Carver, J. C., Vegas, S., & Juristo, N. (2008). The role of replications in empirical software engineering. *Empirical Software Engineering Journal, 13*, 211–218.

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* (pp. 318–320).

Tosun, A., Turhan, B., & Bener, A. (2009). Practical Considerations in Deploying AI for defect prediction: A case study within the Turkish telecommunication industry. In *PROMISE'09: Proceedings of the first international conference on predictor models in software engineering*. Vancouver, Canada.

Twala, B., & Cartwright, M. (2010). Ensemble missing data techniques for software effort prediction. *Intelligent Data Analysis, 14*(3), 299–331.

Twala, B., Cartwright, M., & Shepperd, M. (2006). Ensemble of missing data techniques to improve software prediction accuracy. *Proceedings of International Conference on Software Engineering* (pp. 909–912).

Turhan, B., & Bener, A. (2008). Analysis of naive Bayes' assumptions on software fault data: An empirical study. *Data and Knowledge Engineering Journal, 68*, 278–290.

Turhan, B., & Bener, A. (2007). Software defect prediction: Heuristics for weighted naive bayes. In *Proceedings of the 2nd international conference on software and data technologies (ICSOFT'07)* (pp. 244–249).

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, 14*(5), 540–578.

Verbaeten, S., & Assche, A. V. (2003). Ensemble methods for noise elimination in classification problems. In *Proceedings of the 4th international conference on multiple classifier systems* (pp. 317–325). UK.

Wohlin, C., Aurum, A., Petersson, H., Shull, F., & Ciolkowski, M. (2002). Software inspection benchmarking—A qualitative and quantitative comparative opportunity. In *METRICS '02: Proceedings of the 8th international symposium on software metrics* (pp. 118–127). IEEE Computer Society.

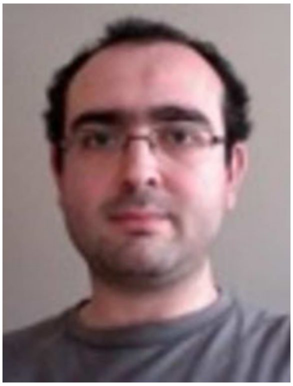
Xu, W., Qin, Z., Ji, L., & Chang, Y. (2009). A feature weighted ensemble classifier on stream data. In *Proceedings of international conference on computational intelligence and software engineering* (pp. 1–5). China.

Zhang, H., & Zhang, X. (2007). Comments on data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33*(9), 635–637.

Author Biographies

Ayşe Tosun Mısırlı is a research assistant and a PhD student in the Department of Computer Engineering at Boğaziçi University. She received her MS degree from the same department in 2008. She has graduated from Computer Science and Engineering at Sabancı University, Istanbul, Turkey in 2001 to 2006. Her research interests are software defect prediction, effort estimation, Bayesian modeling and software process modeling. She is a student member of IEEE Computer Society and ACM SIGSOFT.

Ayşe Başar Bener is an associate professor in the Ted Rogers School of Information Technology Management at Ryerson University. Prior to joining Ryerson, Dr. Bener was a faculty member and Vice Chair in the Department of Computer Engineering at Bogazici University. Her research interests are software defect prediction, process improvement, software quality and software economics. Bener has a PhD in information systems from the London School of Economic. She is a member of the IEEE, IEEE Computer Society and the ACM.



Burak Turhan received his PhD in computer engineering from Bogazici University. After his postdoctoral studies at the National Research Council of Canada, he joined the Department of Information Processing Science at the University of Oulu. His research interests include empirical studies on software quality, test driven development and defect prediction models. He is a member of IEEE, IEEE Computer Society and ACM SIGSOFT.