Noname manuscript No.
(will be inserted by the editor)

Software Defect Prediction Using Bayesian Networks

Ahmet Okutan and Olcay Taner Yıldız

the date of receipt and acceptance should be inserted later

Abstract There are lots of different software metrics discovered and used for defect prediction in the literature. Instead of dealing with so many metrics, it would be practical and easy if we could determine the set of metrics that are most important and focus on them more to predict defectiveness. We use Bayesian networks to determine the probabilistic influential relationships among software metrics and defect proneness. In addition to the metrics used in Promise data repository, we define two more metrics, i.e. NOD for the number of developers and LOCQ for the source code quality. We extract these metrics by inspecting the source code repositories of the selected Promise data repository data sets. At the end of our modeling, we learn the marginal defect proneness probability of the whole software system, the set of most effective metrics, and the influential relationships among metrics and defectiveness. Our experiments on nine open source Promise data repository data sets show that response for class (RFC), lines of code (LOC), and lack of coding quality (LOCQ) are the most effective metrics whereas coupling between objects (CBO), weighted method per class (WMC), and lack of cohesion of methods (LCOM) are less effective metrics on defect proneness. Furthermore, number of children (NOC) and depth of inheritance tree (DIT) have very limited effect and are untrustworthy. On the other hand, based on the experiments on Poi, Tomcat, and Xalan data sets, we observe that there is a positive correlation between the number of developers (NOD) and the level of defectiveness. However, further investigation involving a greater number of projects is needed to confirm our findings.

Keywords Defect prediction, Bayesian networks

1 Introduction

Developing a defect free software system is very difficult and most of the time there are some unknown bugs or unforeseen deficiencies even in software projects where the principles of the software development methodologies were applied carefully. Due to some defective software modules, the maintenance phase of software

Işık University, Meşrutiyet Koyu Universite Sokak, Dış Kapı No:2 Şile/Istanbul, Turkey

projects could become really painful for the users and costly for the enterprises. That is why, predicting the defective modules or files in a software system prior to project deployment is a very crucial activity, since it leads to a decrease in the total cost of the project and an increase in overall project success rate.

Defect prediction will give one more chance to the development team to retest the modules or files for which the defectiveness probability is high. By spending more time on the defective modules and no time on the non-defective ones, the resources of the project would be utilized better and as a result, the maintenance phase of the project will be easier for both the customers and the project owners.

While making a critique of the software defect prediction studies, Fenton and Neil [15] argue that although there are many studies in the literature, defect prediction problem is far from solution. There are some wrong assumptions about how defects are defined or observed and this causes misleading results. Their claim can be understood better when we notice that some define defects as observed deficiencies while some others define them as residual ones.

When we look at the publications about defect prediction we see that in early studies static code features were used more. But afterwards, it was understood that beside the effect of static code metrics on defect prediction, other measures like process metrics are also effective and should be investigated. For example, Fenton and Neil [15] argue that static code measures alone are not able to predict software defects accurately. To support this idea we argue that, if a software is defective this might be related to one of the following:

– The specification of the project may be wrong either due to contradictory requirements or missing features. It may be too complex to realize or not very well documented.
– The design might be poor, it may not consider all requirements or it may reflect some requirements wrongly.
– Developers are not qualified enough for the project.
– There might be a project management problem and the software life cycle methodologies might not be followed very well.
– The software may not be tested enough, so some defects might not be fixed during the test period.

None of the above factors are related to code metrics and all of them may very well affect defect proneness. So, the question is which factors or metrics are effective on defectiveness and how can we measure their effect?

In defect prediction literature, there are many defect prediction algorithms studied like regression [43] [10] [40], rule induction [40], decision tree approaches like C4.5 [42], case-based reasoning (CBR) [23] [22] [40], artificial neural networks [24] [44] [21] [40], linear discriminant analysis [31], k-nearest neighbour [6], k-star [25], Bayesian networks [12] [35] [46] and support vector machine based classifiers [26] [19] [20] [41]. According to the no free lunch theorem [45], there is no algorithm which is better than other algorithms on all data sets. That is why, most of the time it is difficult to generalize the performance of one algorithm and say that it is the best technique for defect prediction. According to Myrtveit et al. [32], "we need to develop more reliable research procedures before we can have confidence in the conclusion of comparative studies of software prediction models". Furthermore, Shepperd et al. [40] argues that the accuracy of a specific defect prediction method is very much dependent on the attributes of the data set like its size, number of attributes and distribution. That is why, it is better to ask which method is the best in a specified context rather than asking which one is the best in general.

Bayesian network is a graphical representation that shows the probabilistic causal or influential relationships among a set of variables that we are interested in. There are a couple of practical factors for using Bayesian networks. First, Bayesian networks are able to model probabilistic influence of a set of variables on another variable in the network. Given the probability of parents, the probability of their children can be calculated. Second, Bayesian networks can cope with the missing data problem. This aspect of Bayesian networks is very important for defect prediction since some metrics might be missing for some modules in defect prediction data sets.

Looking at the defect prediction problem from the perspective that all or an effective subset of software or process metrics must be considered together besides static code measures, Bayesian network model is a very good candidate for taking into consideration several process or product metrics at the same time and measuring their effect. In this paper, we build a Bayesian network among metrics and defectiveness, to measure which metrics are more important in terms of their effect on defectiveness and to explore the influential relationships among them. As a result of learning such a network, we find the defectiveness probability of the whole software system, the order of metrics in terms of their contribution to accurate prediction of defectiveness, and the probabilistic influential relationships among metrics and defectiveness.

Menzies and Shepperd explain the possible reasons behind the conclusion instability problem [29]. In their analysis, they state that there are two main sources of conclusion instability, (i) bias showing the distance between the predicted and actual values and (ii) variance measuring the distance between different prediction methods. The bias can be decreased by using separate training and validation data sets and the variance can be decreased by repeating the validation many times. We use different stratified training and test sets in each experiment to avoid conclusion instability.

In another research, Menzies et al. show what appears to be useful in a global context is often irrelevant for particular local contexts in effort estimation or defect prediction studies. They suggest to test if the global conclusions found are valid for the subsets of the data sets used [27]. We repeat our experiments 20 times with 2/3 rd subsets of each data set to check if our results suffer from conclusion instability.

Posnett et al. explains the ecological inference risk which arises when one builds a statistical model at an aggregated level (e.g., packages), and infers that the results of the aggregated level are also valid for the disaggregated level (e.g., classes), without testing the model in the disaggregated level [39]. They show that modeling defect prediction in two different aggregation levels can lead to different conclusions. To be on the safe side in terms of ecological inference risk, we not only perform our experiments at class level rather than file, package or module levels, but we also test our proposed method with the subsets of the data sets we use, before making a generalization.

This paper is organized as follows: In Section 2, we give a background on Bayesian networks. In Section 3, we present a brief review of previous work on software defect prediction using Bayesian networks. We explain our proposed method in Section 4 and give the experiments and results in Section 5 before we conclude in Section 7.

2 Bayesian Networks

A Bayesian network is a directed acyclic graph (DAG), composed of E edges and V vertices which represent joint probability distribution of a set of variables. In this notation, each vertex represents a variable and each edge represents the causal or associational influence of one variable to its successor in the network.

Let $X = \{X_1, X_2, ...X_n\}$ be n variables taking continuous or discrete values. The probability distribution of X_i is shown as $P(X_i|a_{x_i})$ where a_{x_i}'s represent parents of X_i if any. When there are no parents of X_i, then it is a prior probability distribution and can be shown as $P(X_i)$.

The joint probability distribution of X can be calculated using chain rule:

$$\begin{aligned} P(X) &= P(X_1|X_2, X_3, ..., X_n)P(X_2, X_3, ..., X_n) \\ &= P(X_1|X_2, ..., X_n)P(X_2|X_3, ..., X_n)P(X_3, ..., X_n) \\ &= P(X_1|X_2, ..., X_n)P(X_2|X_3, ..., X_n)...P(X_{n-1}|X_n)P(X_n) \\ &= \prod_{i=1}^{n} P(X_i|X_{i+1}, ..., X_n) \end{aligned} \tag{1}$$

Given the parents of X_i, other variables are independent from X_i, so we can write the joint probability distribution as

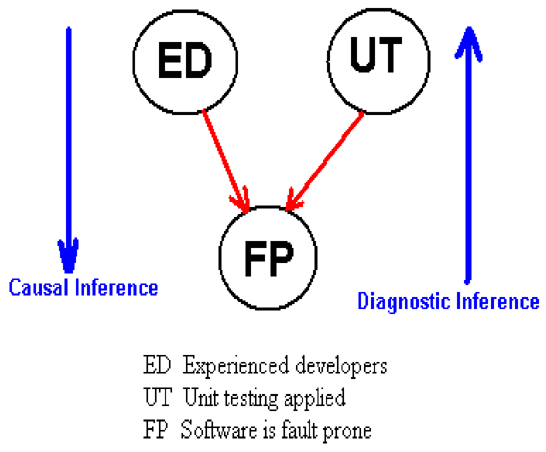
$$P(X) = \prod_{i=1}^{n} P(X_i|a_{x_i}) \tag{2}$$

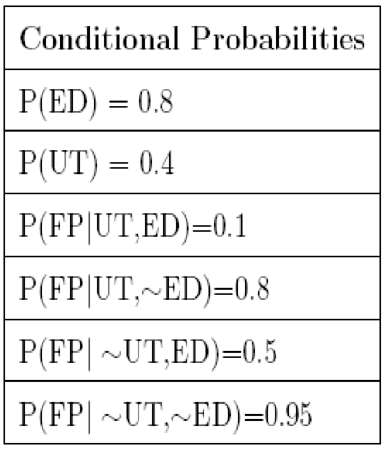
On the other hand, Bayes' rule is used to calculate the posterior probability of X_i in a Bayesian network based on the evidence information present. We can calculate probabilities either towards from causes to effects ($P(X_i|E)$) or from effects to causes ($P(E|X_i)$). Calculating probability of effects from causes is called causal inference whereas calculating probability of causes from effects is called diagnostic inference [2]. Figure 1 shows a sample Bayesian network and conditional probability tables. Assume that we would like to investigate the effect of using experienced developers (ED) and applying unit testing methodology (UT) on defectiveness (FP). Furthermore, each variable can take discrete values of on/off, that is developers are experienced or not, unit testing used or not used. Suppose we would like to make a causal inference and calculate the probability of having a fault prone software if we know that the developers working on the project are experienced. We shall calculate

$$P(FP|ED) = P(FP|ED, UT)P(UT|ED) + P(FP|ED, \sim UT)P(\sim UT|ED)$$

We can write $P(UT|ED) = P(UT)$ and $P(\sim UT|ED) = P(\sim UT)$ since the variables ED and UT are independent. Then we have,

$$P(FP|ED) = P(FP|ED, UT)P(UT) + P(FP|ED, \sim UT)P(\sim UT)$$





Conditional Probabilities
P(ED) = 0.8
P(UT) = 0.4
P(FP|UT,ED)=0.1
P(FP|UT,~ED)=0.8
P(FP| ~UT,ED)=0.5
P(FP| ~UT,~ED)=0.95

Fig. 1 A sample Bayesian network to illustrate Bayesian inference

Feeding up the values in the conditional probability table, $P(FP|ED)$ is calculated as 0.34. Assume that we are asked to calculate the probability of having experienced developers given the software is fault prone, i.e. $P(ED|FP)$.

Using Bayes' rule we write

$$P(ED|FP) = \frac{P(FP|ED)P(ED)}{P(FP)} \tag{3}$$

We can also write

$$\begin{aligned} P(FP) = {} & P(FP|UT, ED)P(UT)P(ED) \\ & + P(FP|UT, \sim ED)P(UT)P(\sim ED) \\ & + P(FP| \sim UT, ED)P(\sim UT)P(ED) \\ & + P(FP| \sim UT, \sim ED)P(\sim UT)P(\sim ED) \end{aligned} \tag{4}$$

Since $P(FP|UT, ED)$, $P(FP|UT, \sim ED)$, $P(FP| \sim UT, ED)$, and $P(FP| \sim UT, \sim ED)$ can be read from the conditional table, the diagnosis probability $P(ED|FP)$ can also be calculated. As it can be seen in these examples of causal and diagnostic inferences, it is possible to propagate the effect of states of variables (nodes) to calculate posterior probabilities. Propagating the effects of variables to the successors, or analyzing the probability of some predecessor variable based on the probability of its successor is very important in defect prediction since software metrics are related to each other and that is why the weight of a metric might be dependent on another metric based on this relationship.

2.1 K2 Algorithm

In Bayesian network structure learning, the search space is composed of all of the possible structures of directed acyclic graphs based on the given variables (nodes). Normally, it is very difficult to enumerate all of these possible directed acyclic graphs without a heuristic method. Because, when the number of nodes increases, the search space grows exponentially and it is almost impossible to search the whole space. Given a data set, the K2 algorithm proposed by Cooper and Herskovits, heuristically searches for the most probable Bayesian network structure [8]. Based on the ordering of the nodes, the algorithm looks for parents for each node whose addition increases the score of the Bayesian network. If addition of a certain node X_j to the set of parents of node X_i does not increase the score of the Bayesian network, K2 stops looking for parents of node X_i further. Since the ordering of the nodes in the Bayesian network is known, the search space is much more smaller compared to the entire space that needs to be searched without a heuristic method. Furthermore, a known ordering ensures that there will be no cycles in the Bayesian network, so there is no need to check for cycles too.

K2 algorithm takes a set of n nodes, an initial ordering of the n nodes, the maximum number of parents of any node denoted by u and a database D of m cases as input and outputs a list of parent nodes for every node in the network. The pseudo code of the K2 algorithm is given in Algorithm 1. For every node in the network, the algorithm finds the set of parents with the highest probability taking into consideration the upper bound u for the maximum number of parents a node can have. During each iteration, the function $Pred(x_i)$ is used to determine the set of nodes that precede x_i in the node ordering.

```
for $i \leftarrow 1$ to $n$ do
    $\pi_i := 0$;
    $P_{old} := f(i, \pi_i)$ (See Equation 5);
    $OKToProceed :=$ true;
    while $OKToProceed$ and $|\pi_i| < u$ do
        let $z$ be the node in $Pred(x_i) - \pi_i$ that maximizes $f(i, \pi_i \cup \{z\})$;
        $P_{new} := f(i, \pi_i \cup \{z\})$;
        if $P_{new} > P_{old}$ then
            $P_{old} := P_{new}$;
            $\pi_i := \pi_i \cup \{z\}$
        else
            $OKToProceed :=$ false
        end
    end
    write(Node: $x_i$, Parent of $x_i$ : $\pi_i$)
end
```

Algorithm 1: The K2 algorithm [8]

The algorithm calculates the probability that the parents of x_i are π_i using the following equation:

$$f(i, \pi_i) = \prod_{j=1}^{q_i} \frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \prod_{k=1}^{r_i} N_{ijk}! \tag{5}$$

Table 1 List of software metrics we used in our modeling

Metric	Metric full name
WMC	Weighted method per class [7]
DIT	Depth of inheritance tree [7]
NOC	Number of children [7]
CBO	Coupling between objects [7]
RFC	Response for class [7]
LCOM	Lack of cohesion of methods [7]
LCOM3	Lack of cohesion in methods [18]
LOC	Lines of code
LOCQ	Lack of coding quality

where π_i is the set of parents of x_i, $q_i = |\phi_i|$ where ϕ_i is the set of all possible instances of the parents of x_i for database D. Furthermore, $r_i = |V_i|$ where V_i is the set of all possible values of the x_i. On the other hand, N_{ijk} is the number of instances in database D for which x_i is instantiated with its k^{th} value, and the parents of x_i in the set π_i are instantiated with the j^{th} instantiation in the set ϕ_i. And lastly,

$$N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$$

gives the number of instances in D where the parents of x_i are instantiated with the j^{th} instantiation in ϕ_i.

3 Previous Work

Pai and Dugan [35] use Bayesian networks to analyze the effect of object oriented metrics [7] on the number of defects (fault content) and defect proneness using KC1 project from the Nasa metrics data repository. They build a Bayesian network where parent nodes are the object oriented metrics (also called C-K metrics) and child nodes are the random variables fault content and fault proneness. After the model is created, they make a Spearman correlation analysis to check whether the variables of the model (metrics) are independent or not. They have found that SLOC, CBO, WMC, and RFC are the most significant metrics to determine fault content and fault proneness. They discover that the correlation coefficients of these metrics (SLOC, CBO, WMC, and RFC) with fault content are 0.56, 0.52, 0.352, and 0.245 respectively. On the other hand, they also find that neither DIT nor NOC are significant and depending on the underlying model, LCOM seems to be significant for determining fault content.

According to Zhang [46], Bayesian networks provide a very suitable and useful method for software defect prediction. They suggest to build a Bayesian network that reflects all software development activities like specification, design, implementation, testing and consider Bayesian network generation in three steps: defining Bayesian network variables, defining the causal relationships among the network variables and generating the probability distribution of each variable in the network and calculating the joint probability distribution of the hypothesis variables.

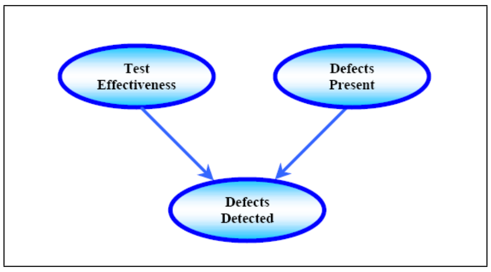


Fig. 2 Bayesian network suggested by Fenton et al. (2002)

Fenton et al. [12] suggest to use Bayesian networks for defect, quality, and risk prediction of software systems. They use the Bayesian network shown in Figure 2 to model the influential relationships among target variable "defects detected" (DD) and the information variables "test effectiveness" (TE) and "defects present" (DP). In this network, DP models the number of bugs/defects found during testing. TE gives the efficiency of testing activities and DD gives the number of defects delivered to the maintenance phase. For discretization, they assign two very simple states to each variable namely low and high. Using the Bayesian network model, Fenton et al. show how Bayesian networks provide accurate results for software quality and risk management in a range of real world projects. They conclude that Bayesian networks can be used to model the causal influences in a software development project and the network model can be used to ask "what if?" questions under circumstances when some process underperforms [12].

Furthermore, Gyimothy et al. use regression and machine learning methods (decision tree and neural networks) to see the importance of object oriented metrics for fault proneness prediction [16]. They formulate a hypothesis for each object oriented metric and test the correctness of these hypotheses using open source web and email tool Mozilla. For comparison they use precision, correctness and completeness. They find that CBO is the best predictor and LOC is the second. On the other hand, the prediction capability of WMC and RFC is less than CBO and LOC but much better than LCOM, DIT, and NOC. According to the results, DIT is untrustworthy and NOC can not be used for fault proneness prediction. Furthermore, the correctness of LCOM is good although it has a low completeness value.

On the other hand, Zhou and Leung use logistic regression and machine learning methods (naive Bayes network, random forest, and nearest neighbor with generalization) to determine the importance of object oriented metrics for determining fault severity [47]. They state a hypothesis for each object oriented metric and test them on open source Nasa data set KC1 [1]. For ungraded severity, they observe that SLOC, CBO, WMC, and RFC are significant in fault proneness prediction. Furthermore, LCOM is also significant but when tested with machine learning methods the usefulness of NOC is poor and the result is similar for DIT.

Bibi et al. use iterative Bayesian networks for effort estimation by modeling the sequence of the software development processes and their interactions. They state that Bayesian networks provide a highly visual interface to explain the relationships of the software processes and provide a probabilistic model to represent the uncertainty in their nature. They conclude that Bayesian networks could be used for software effort estimation effectively [4]. Furthermore Minana and Gras use Bayesian networks to predict the level of fault injection during the phases of a software development process. They show that Bayesian networks provide a successful fault prediction model [37].

Amasaki et al. propose to use Bayesian networks to predict the quality of a software system. To generate a Bayesian network they use certain metrics collected during the software development phase like product size, effort, detected faults, test items, and residual faults. They conclude that the proposed model can estimate the residual faults that the software reliability growth model can not handle [3].

Fenton et al. review different techniques for software defect prediction and conclude that traditional statistical approaches like regression alone is not enough. Instead they believe that causal models are needed for more accurate predictions. They describe a Bayesian network, to model the relationship of different software life cycles and conclude that there is a good fit between predicted and actual defect counts [14]. In another study, Fenton et al. propose to use Bayesian networks to predict software defects and software reliability and conclude that using dynamic discretization algorithms while generating Bayesian networks leads to significantly improved accuracy for defects and reliability prediction [13].

In another research, Dejaeger et al. compare 15 different Bayesian network classifiers with famous defect prediction methods on 11 Data sets in terms of the AUC and H-measure. They observe that simple and comprehensible Bayesian networks can be constructed other than the simple Naive Bayes model and recommend to use augmented Bayesian network classifiers when the cost of not detecting a defective or non defective module is not higher than the additional testing effort [9]. Furthermore, as future work, they propose to discover the effects of different information sources with Bayesian networks which is something we consider by defining two extra metrics i.e. LOCQ and NOD and measuring their relationship with other metrics and defectiveness.

Regarding the effects of the number of developers on defect proneness there are contradictory findings in the literature. For example Norick et al. use eleven open source software projects, to determine if the number of committing developers affects the quality of a software system. As a result, they could not find significant evidence to claim that the number of committing developers affects the quality of software [34]. Furthermore, Pendharkar and Rodger investigate the impact of team size on the software development effort using over 200 software projects and conclude that when the size of the team increases, no significant software effort improvements are seen [36]. On the other hand, Nagappan et al. define a metric scheme that includes metrics like number of engineers, number of ex-engineers, edit frequency, depth of master ownership, percentage of organization contributing to development, level of organizational code ownership, overall organization ownership, and organization intersection factor to quantify organizational complexity. They use data from Windows Vista operating system and conclude that the organizational metrics predict failure-proneness with significant precision, recall, and sensitivity. Furthermore, they also show that organizational metrics are better predictors of failure-proneness than the traditional metrics used so far like code churn, code complexity, code coverage, code dependencies, and pre-release defect measures [33]. Furthermore Mockus et al. use two open source projects, the Apache web server and the Mozilla browser to define several hypotheses that are related to the developer count and the team size. They test and refine some of these based on an the analysis of Mozilla data set. They believe that when several people work on the same code, there are many potential dependencies among their work items. So they suggest that regarding to the team size, around an upper limit of 10-15 people, coordination of the work for the team becomes inadequate [30].

4 Proposed Approach

4.1 Bayesian network of Metrics and Defect Proneness

It is very important to model the associational relationships among the metrics and defect proneness. We first generate a Bayesian network among software metrics and defect proneness and then using this network, we calculate an overall marginal defectiveness probability of the software system. This network provides us two very important results:

– The dependencies among the metrics we choose. Which metrics are affected by other metrics and which ones are the most effective on defect proneness.
– The defect proneness probability of the software system itself. By learning from the data set, the Bayesian network tells us the marginal defectiveness probability of the whole system and one can interpret this as the probability of having at least one or more defects in a software module that is selected randomly.

In defect prediction studies, using static code metrics alone may ignore some very crucial causes of defects like poor requirement analysis or design, lack of quality of design or coding, unexperienced developers, bad documentation, managerial or financial problems. Although all of these factors could lead to an increase in defect proneness, static code metrics do not consider them effectively. Using Bayesian networks might be much more meaningful when additional data on causal, explanatory variables are available and included in the model. Unfortunately, it is not too easy to measure these causal and explanatory variables when there is no information regarding the software development processes. By inspecting project repositories of some data sets, we add two metrics i.e. LOCQ and NOD to our Bayesian model, in order to measure the effect of lack of coding quality and the number of developers.

We introduce a new metric we call lack of coding quality (LOCQ) that measures the quality of the source code. We run PMD source code analyzer plugin in Netbeans, to generate the LOCQ values for each class of the open source Apache projects listed in Table 3. PMD inspects the given source code and looks for potential problems like possible bugs, dead code, suboptimal code, overcomplicated expressions, and duplicate code. It counts the number of detected problems for each class and package in the software system. We believe that this measurement gives an idea about the quality of the source code and has a relationship with defectiveness. That is why, we include the LOCQ metric in our experiments and try to understand how it is related with the defect proneness and other well known static code metrics in the literature.

We ask if for a specific class or file, the number of developers is positively related with the extent of defectiveness or not? Receiving inspiration from famous idiom “too many cooks spoil the broth” we wonder if a higher number of developers for a certain class or file, leads to a more defective or messed up source code? For some of the data sets listed in Table 3 that have developer information in the source code files, we generate the number of developers (NOD) metric, which shows the number of distinct developers per each class in the software system. Then we learn a Bayesian network from each of these data sets and extract the relationship of the NOD metric with defectiveness.

One problem to reach a clear conclusion on this issue is the conclusion instability problem. We think that conclusion instability comes from an inherent property of software engineering data sets; i.e. real world data is noisy. To remove noise from the data sets and draw more accurate conclusions, we cross check our results on 10 subsets of each data set. While generating the subsets, we stratify the data with different seeds and include 67 percent of the data each time. For the remaining data sets like Ant, Lucene, and Synapse, there was no developer information in the source code repositories, so we could not generate the NOD metric for them. Since the developer information is not available for all data sets but just for a subset of them, we present the experiments carried out with the NOD metric separately in Section 5.6.

While we model the influential relationships among different product and process metrics, we learn the Bayesian network from the data set. Figure 3 shows the general form of our model. In this Bayesian network, we see the interactions among different product, process or developer metrics. We may see that a metric is not affected by any other metric whereas some metrics may be affected by one or more product metrics (like $Metric_5$). According to this Bayesian network $Metric_5$, $Metric_6$, and $Metric_7$ are the most important metrics since they affect defectiveness directly. On the other hand, $Metric_1$, $Metric_2$, $Metric_3$, and $Metric_4$ are less important since they are indirectly related with defectiveness.

As a summary, the Bayesian network we propose is a graph G of E edges and V vertices where each V_i represents a metric and each E_j represents the dependency between two metrics or between a metric and defectiveness. If an edge E is present from metric m_2 towards metric m_1, then this would mean metric m_1 is effective on metric m_2. Similarly, if there is an edge from defectiveness to metric m_1, then it would mean that metric m_1 is effective on defectiveness. This way, we determine the metrics that affect defectiveness directly or indirectly.

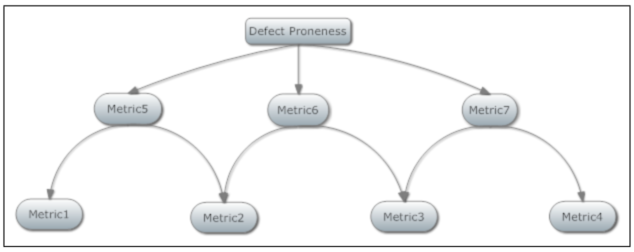


Fig. 3 Proposed Bayesian network to model the relationships among metrics

Table 2 The order of software metrics used during Bayesian network construction

Metric Groups	**Order (left to right)**		
Group$_1$	LOC	CBO	LOCQ
Group$_2$	WMC	RFC	
Group$_3$	LCOM - LCOM3	DIT	NOC

4.2 Ordering Metrics for Bayesian Network Construction

In order to learn a Bayesian network with K2 algorithm, it is necessary to specify the order of the nodes. That is why, we decide to order the software metrics considering their effect on defectiveness, prior to the generation of Bayesian networks.

We believe that as the size of a software system gets larger, the probability of having fault prone classes increases, since more effort would be needed to ensure a defect free software. We also believe that besides size, complexity of the software is also very important because as the design gets more complex, it would be more difficult for developers to ensure non-defectiveness. That is why, for the initial ordering of the metrics, we decide to give LOC and CBO as the first metrics since LOC is the best indicator of software size and CBO shows how much complex a software system is by counting the number of couples for a certain class where coupling means using methods or instance variables of other classes. As one would easily accept, as coupling increases, the complexity of the software system would also increase. Furthermore, as everybody can accept, when the quality of the source code increase, the probability of having a defective software decreases. So, we introduce the LOCQ metric as the third metric in the first group after LOC and CBO.

Although, RFC may explain complexity to some extent, it may not be the case if a class is using internal methods or instance variables only. That is why, RFC together with WMC are entitled as the second group of metrics. On the other hand, NOC indicates the number of children of a class and is not a good indicator for both size and complexity, since the parent-child relationship does not contribute to the complexity if there is no caller-callee relationship between them which is the case for most of the time. Due to similar reasons DIT also does not explain size or complexity alone. So, we decide to give DIT and NOC as the last metrics in the initial ordering.

Following our reasoning, we generate three groups of metrics where LOC, CBO, and LOCQ are in *Group*$_1$, WMC and RFC are in *Group*$_2$ and LCOM, DIT, and NOC are in *Group*$_3$. *Group*$_1$ metrics are more important than *Group*$_2$ metrics and *Group*$_2$ metrics are more important than *Group*$_3$ metrics in terms of their effect on defectiveness (See Table 2).

5 Experiments and Results

5.1 Experiment Setup

In our experiments, we use Bayesian networks to determine the influential or associational relationships among the software metrics and defectiveness and identify the most effective metrics by giving them scores considering their effect on defect proneness. While choosing the data sets from Promise data repository, first we look at the data sets that are large enough to perform cross validation. So, we eliminate some of the data sets in the repository that are small in terms of size. Second, to extract additional metrics LOCQ and NOD, we need the source repositories of the data sets. That is why, we prefer the data sets in Promise data repository whose source code is available in the open source project repositories. For instance the Log4j data set has defect data for versions 1.0, 1.1, and 1.2 in the Promise data repository, but the sources corresponding to these exact versions are not present in the Apache repository. We eliminate some data sets whose sources are not available or are skewed which could affect the learning performance of the Bayes net classifier. Based on these criteria we select public data sets Ant, Tomcat, Jedit, Velocity, Synapse, Poi, Lucene, Xalan, and Ivy from Promise data repository [5] (See Table 3 for the details of the datasets).

We use Weka [17] for Bayesian network structure learning where we learn the network structure from the data sets and use SimpleEstimator while constructing Bayesian networks for defect proneness. Furthermore, we select K2 as the search algorithm and use predefined ordering of nodes of LOC, CBO, LOCQ, WMC, RFC, LCOM, LCOM3, DIT, and NOC.

Table 3 Brief details of data sets used in the experiments

Data Set	**Version**	**No. of Instances**	**% Defective Instances**
Ant	1.7	745	22.28
Tomcat	6.0	858	8.97
Jedit	4.3	492	2.24
Velocity	1.6	229	34.06
Synapse	1.2	256	33.59
Poi	3	442	63.57
Lucene	2.4	340	59.71
Xalan	2.5	741	48.19
Ivy	2.0	352	11.36

It is a common way to look at the error rates of classifiers while making comparisons. But, this is not true in real life, because first, the proportions of defective and non defective classes are not equal. For instance, in defect prediction, most of the time the proportion of defective modules is quite different from the proportion of non defective ones. Furthermore, the cost of false positives (FP) and false negatives (FN) are not the same i.e. FN is more costly than FP. ROC analysis is used in the literature and considers TP rate (also called sensitivity) and FP rate (also called false alarm rate) together [11]. ROC curve is a two dimensional graphical representation where TP is the y-axis and FP is the x-axis. It is always desirable to have high sensitivity and small false alarm rate. So, as the area under the ROC curve (AUC) gets larger, the classifier gets better. That is why, we use AUC while comparing the performance of the Bayesian networks in our experiments.

5.2 Results

Table 4 shows the AUC values for each Bayesian network. The results show that Ivy dataset has the highest AUC value whereas Xalan data set has the smallest AUC value. A high AUC value implies that the data set used is able to define the structure of the Bayesian network better. The AUC values found in our experiments are relatively high and we believe that this makes our results more important and reliable.

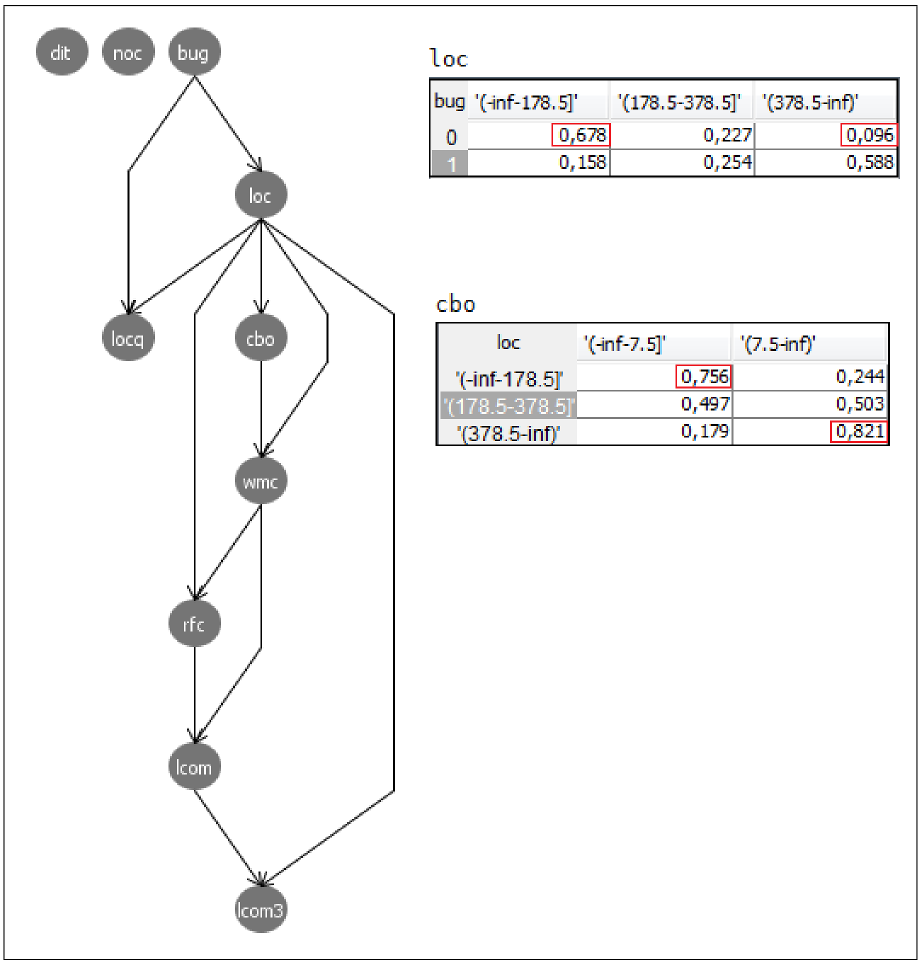
Table 4 The AUC values of the Bayesian networks in our experiments

Data Sets	**AUC**
Ant	0.820
Tomcat	0.766
Poi	0.845
Jedit	0.658
Velocity	0.678
Synapse	0.660
Lucene	0.633
Xalan	0.624
Ivy	0.846

We obtain the Bayesian network shown in Figure 4 for Ant data set. The metrics LOC and LOCQ are the most effective whereas CBO, WMC, RFC, LCOM, and LCOM3 are indirectly and less effective on defectiveness. However, DIT and NOC are not effective at all on the bug attribute. When we look at the conditional probability table of LOC, we observe that there is a positive correlation between LOC and defect proneness. For instance the non-defectiveness probability is 0.678 for small LOC, whereas it is 0.096 for high LOC which means that as the LOC increase the probability of defectiveness increases too. We observe a similar relationship between LOCQ and defectiveness also where the defect proneness probability is high for high LOCQ values. Furthermore, we observe that when the LOC is high, the probability of having a high CBO is also high which means that there is a positive correlation between LOC and CBO too.

The Bayesian network for Tomcat data set shows that LOC and CBO are directly effective on defect proneness, whereas WMC, LOCQ, RFC, LCOM, and LCOM3 are indirectly effective. But DIT and NOC metrics are not effective in determining the defect prone classes (See Figure 5). Furthermore, when both LOC and CBO are high, defectiveness probability is higher (0.9) compared to the case when either one of them is small. Similarly, when both are low, the non defectiveness probability is higher (0.793) compared to cases when either one of them is high.

The Bayesian network obtained for Poi data set is shown in Figure 6. First of all, CBO, LOC, LOCQ, WMC, and LCOM are the most effective metrics since they are directly connected with defectiveness. RFC, DIT, and LCOM3 are indirectly effective on fault proneness. On the other hand, NOC is not effective at all. Similarly, from the conditional probability table of CBO, we see that there is also a positive correlation between CBO and defect proneness. Furthermore, we observe that when both LOC and CBO are high, the defect proneness probability is



loc

bug	'(-inf-178.5]'	'(178.5-378.5]'	'(378.5-inf)'
0	0,678	0,227	0,096
1	0,158	0,254	0,588

cbo

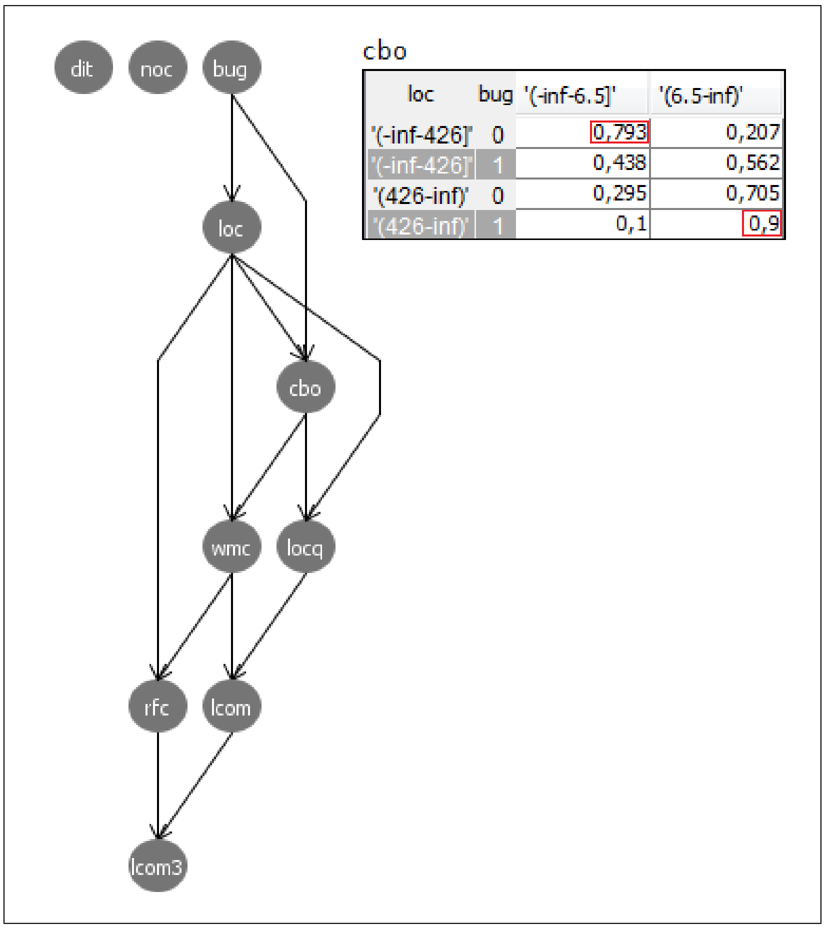
loc	'(-inf-7.5]'	'(7.5-inf)'
'(-inf-178.5]'	0,756	0,244
'(178.5-378.5]'	0,497	0,503
'(378.5-inf)'	0,179	0,821

Fig. 4 Bayesian network showing the relationships among software metrics in Ant version 1.7

the highest (0.904). This shows that the size and the complexity metrics together affects defectiveness more compared to the effect of either size or complexity alone.

For the remaining 6 data sets, Bayesian networks generated are shown in Figure 7. For Synapse data set, DIT, NOC, and LCOM3 metrics are not effective on defect proneness. On the other hand, LOC is the most effective metric and LOCQ, CBO, WMC, RFC, and LCOM are indirectly effective on defect proneness. The probability of having a defect free software is 0.839 for small LOC, whereas it is 0.161 for higher LOC values. We also observe that there is a positive correlation between LOC and LOCQ metrics. For instance, when the LOC is small, LOCQ is also small with a probability of 0.942. Similarly, the probability of having both LOC and LOCQ high is 0.877.

We observe a similar result for Lucene data set also where LCOM3, DIT, and LOC are not effective on defect proneness whereas CBO and LOC are the most effective metrics. For higher CBO values, the defect proneness probability is 0.743



cbo

loc	bug	'(-inf-6.5]'	'(6.5-inf)'
'(-inf-426]'	0	0,793	0,207
'(-inf-426]'	1	0,438	0,562
'(426-inf)'	0	0,295	0,705
'(426-inf)'	1	0,1	0,9

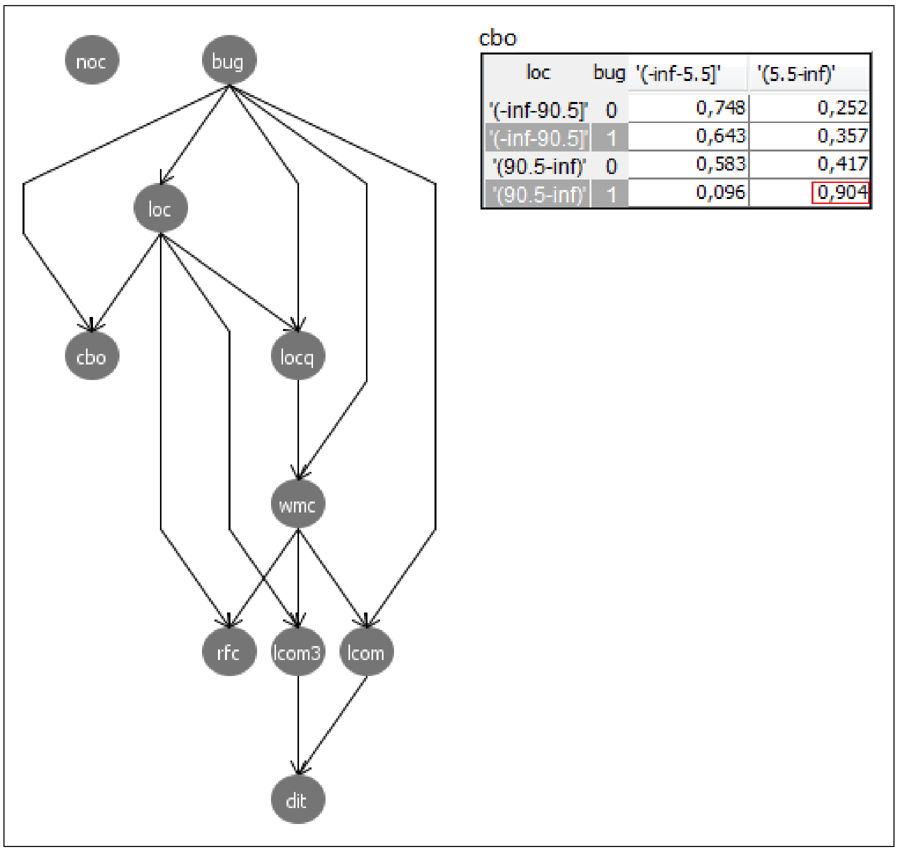
Fig. 5 Bayesian network showing the relationships among software metrics in Tomcat version 6.0

whereas it is only 0.257 for smaller CBO. That means as the coupling between objects increase, the probability of defectiveness increases also. On the other hand, LOCQ, WMC, RFC, and LCOM are indirectly effective on defect proneness.

Similar to the previous findings, for Velocity data set, DIT, NOC, WMC, and LCOM found to be independent of defect proneness where LOC is directly effective on defectiveness. On the other hand, CBO, LOCQ, RFC, and LCOM3 are indirectly and less effective compared to LOC. When we look at the conditional probability table for LOC, we observe that the defectiveness probability is 0.64 for higher LOC whereas it is 0.36 for smaller LOC values.

For JEdit data set, metrics DIT and NOC are independent from defect proneness whereas LOC, CBO, WMC, and LCOM3 are effective. On the other hand, LOCQ, RFC, and LCOM are indirectly and less effective compared to LOC, CBO, WMC, and LCOM3 metrics.

We observe that LOC and LCOM3 metrics are directly effective on defect proneness whereas DIT, NOC, CBO, and LCOM are not effective for Xalan data



cbo

loc	bug	'(-inf-5.5]'	'(5.5-inf)'
'(-inf-90.5]'	0	0,748	0,252
'(-inf-90.5]'	1	0,643	0,357
'(90.5-inf)'	0	0,583	0,417
'(90.5-inf)'	1	0,096	0,904

Fig. 6 Bayesian network showing the relationships among software metrics in Poi version 3.0

set. On the other hand, LOCQ, WMC, and RFC are indirectly effective on defectiveness. Furthermore, for a lower LOC, the probability of having a lower LOCQ is 0.967 whereas the probability of a higher LOCQ is 0.033. Similarly, for a higher LOC, the probability of having a higher LOCQ is 0.568, whereas the probability of a lower LOCQ is 0.432.

For Ivy data set, LOC, CBO, and LOCQ are the most important metrics, whereas DIT, NOC, and LCOM3 are not effective on fault proneness. Furthermore, WMC, RFC, and LCOM are less effective on defectiveness compared to LOC, CBO, and LOCQ. Similar to the previous findings, when both LOC and CBO are high, the defectiveness probability is the highest (0.983).

To measure the effect of metrics quantitatively, we give scores to the metrics in each experiment. If a metric is affecting defectiveness (directly or indirectly) we assign it a score of 1, if it has no relationship with defectiveness it is assigned a zero score. Table 5 shows the scores of metrics assigned in each experiment. According to the average scores, LOC, CBO, LOCQ, WMC, and RFC are the most effective metrics, whereas DIT and NOC are the least effective ones. Furthermore, DIT and NOC are untrustworthy since their effectiveness is not consistent in all experiments. For instance, DIT is effective in Poi whereas it has no importance

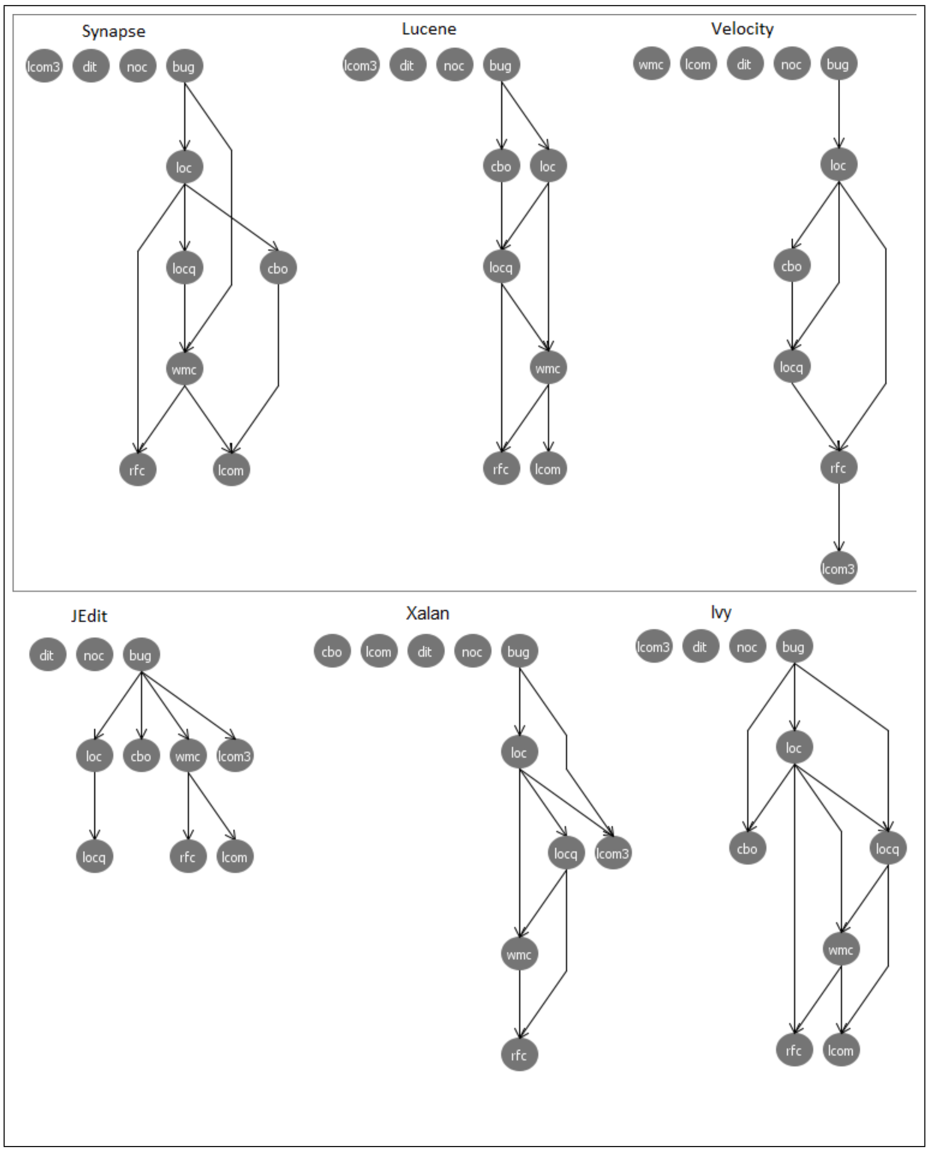


Fig. 7 Bayesian networks showing the relationships among software metrics and defect proneness (bug) in different data sets

in other data sets. Similarly, NOC is independent from defect proneness in all experiments. Moreover, we observe that LCOM and LCOM3 are more effective compared to DIT and NOC and less effective compared to others.

Table 5 The scores of metrics obtained

Data Sets	LOC	CBO	LOCQ	WMC	RFC	LCOM	LCOM3	DIT	NOC
Ant	1	1	1	1	1	1	1	0	0
Tomcat	1	1	1	1	1	1	1	0	0
Poi	1	1	1	1	1	1	1	1	0
Jedit	1	1	1	1	1	1	1	0	0
Velocity	1	1	1	0	1	0	1	0	0
Synapse	1	1	1	1	1	1	0	0	0
Lucene	1	1	1	1	1	1	0	0	0
Xalan	1	0	1	1	1	0	1	0	0
Ivy	1	1	1	1	1	1	0	0	0
Average	**1**	**0.89**	**1**	**0.89**	**1**	**0.78**	**0.67**	**0.11**	**0**

5.3 Conclusion Instability Test

Some times the results found for a data set, might not be valid for its subsets due to some uncommon local attributes [27]. So, we check if the results shown in Table 5 are valid for the subsets of the data sets too. Therefore, we make 20 experiments with different 2/3rd subsets of each data set and calculate the average score for each metric based on these 20 experiments. While generating the subsets, we stratify the data and use a different seed to ensure each subset is different from the previously generated ones. For each data set, the average scores we find at the end of 20 experiments are listed in Table 6. When we look at the average scores of the metrics on all data sets, we observe that although the results are slightly different from the results presented in Table 5 in terms of ordering, there are very strong similarities. For instance, still LOC, CBO, LOCQ, WMC, and RFC are the most important metrics. Furthermore DIT and NOC are the least effective and untrustworthy metrics. Similar to the results shown in Table 5, LCOM and LCOM3 are more effective compared to DIT and NOC and less effective compared to other metrics.

Table 6 The average scores of the metrics obtained for 20 different subsets of Ant, Tomcat, Poi, Jedit, Velocity, Synapse, Lucene, Xalan, and Ivy.

Data Sets	LOC	CBO	LOCQ	WMC	RFC	LCOM	LCOM3	DIT	NOC
Ant	1.00	1.00	1.00	1.00	1.00	0.90	0.70	0.10	0.00
Tomcat	1.00	1.00	0.90	0.90	1.00	1.00	0.50	0.10	0.10
Poi	1.00	0.70	1.00	0.90	1.00	0.80	0.80	0.40	0.00
Jedit	0.60	0.50	0.30	0.60	0.50	0.40	0.00	0.00	0.00
Velocity	0.90	0.50	0.50	0.40	0.90	0.10	0.30	0.00	0.00
Synapse	0.70	0,80	0.60	0.60	0.70	0.60	0.00	0.00	0.00
Lucene	0.30	0.70	0.60	0.80	0.90	0.60	0.20	0.00	0.00
Xalan	1.00	0.10	0.95	0.55	0.95	0.65	0.50	0.05	0.10
Ivy	1.00	0.90	1.00	1.00	0.95	0.70	0.50	0.00	0.00
Average	**0.83**	**0.69**	**0.76**	**0.75**	**0.88**	**0.64**	**0.39**	**0.07**	**0.02**

5.4 Effectiveness of Metric Pairs

We look at the 180 Bayesian networks (generated for 20 subsets of 9 data sets), in terms of which metric pairs are the most effective on defectiveness. For a specific Bayesian network, if both of the metrics in the pair have a relationship with defectiveness we assign a score of 1 to the metric pair. If either or neither of the metrics is related with defect proneness we assign a zero score for the metric pair. The sum of the scores of the metric pairs calculated for all subsets of the data sets are shown in Table 6 (Only the most effective ten metric pairs are included in the list). We observe that metric pairs LOC-RFC, RFC-LOCQ, RFC-WMC are the most effective pairs and their scores are 145, 136, and 133 respectively. We see that the metric pairs that have the highest scores are composed of metrics that got the highest score in the previous evaluation where each metric is considered alone (See Table 5). For instance the metric pair LOC-RFC got the highest score and we see that metrics LOC and RFC alone are among the metrics that got the highest scores in Table 5.

Table 7 The scores of metric pairs obtained for the 20 subsets of Ant, Tomcat, Poi, Jedit, Velocity, Synapse, Lucene, Xalan, and Ivy data sets.

Metric Pairs	**Total Score**
LOC-RFC	145
RFC-LOCQ	136
RFC-WMC	133
LOC-LOCQ	131
LOC-WMC	121
RFC-CBO	120
LOCQ-WMC	119
WMC-CBO	109
LOC-CBO	108
LOCQ-CBO	106

5.5 Feature Selection Tests

Using Bayesian network model, it is possible to make probabilistic causal or diagnostic inferences about the effectiveness of a metric on another metric or on the defectiveness. At the end of learning a Bayesian network, we not only determine the set of most important metrics but also find the relationship among them and the probability of their effect on defect proneness. Therefore, with Bayesian networks we are able to model the uncertainties better, compared to other machine learning methods. Although they do not give the extent of influential relationships among metrics and defectiveness, using Feature selection methods, we can determine the most important metrics and make a cross check with the results of Bayesian network model.

At the end of our experiments with Bayesian networks, we observe that considering all data sets we use in our experiments, LOC, CBO, LOCQ, WMC, and RFC are the most effective metrics and DIT and NOC are the least effective ones

(See Tables 5 and 6). We run two feature selection algorithms CFS (CfsSubsetEval attribute evaluator with BestFirst search method) and Relief (ReliefFAttributeEval with Ranker search method) to see which metrics are selected as the most important attributes and whether the results of the feature selection experiments are different from the results shown in Table 5. For CFS tests, for each data set, each metric is assigned a score of 1 if it is among the selected metrics and it is assigned 0 otherwise. Table 8 shows the results of feature selection tests with CFS. When we look at the average scores of the metrics, we observe that LOC, CBO, LOCQ, and RFC are among the most important features and DIT and NOC are the least important ones. So, although the ordering found at the end of feature selection is slightly different from the ordering shown in Table 5, except WMC, we can say that there is a coherence between the feature selection test and our experiments in terms of the most and least effective attributes.

Table 8 The results of feature selection tests with CfsSubsetEval and BestFirst search method where a metric is assigned a score of 1 if it is selected and is assigned zero score otherwise.

Data Sets	LOC	CBO	LOCQ	WMC	RFC	LCOM	LCOM3	DIT	NOC
Ant	1	1	1	0	1	1	0	0	0
Tomcat	1	1	1	0	1	0	0	0	0
Poi	0	1	1	0	1	1	1	0	0
Jedit	0	0	0	0	1	0	0	0	0
Velocity	1	1	0	0	1	0	1	0	0
Synapse	1	1	0	1	1	0	0	0	0
Lucene	0	1	1	0	1	1	0	0	1
Xalan	1	0	1	0	1	0	1	0	0
Ivy	1	1	1	0	1	0	1	0	0
Average	**0.67**	**0.78**	**0.67**	**0.11**	**1**	**0.33**	**0.44**	**0**	**0.11**

We repeat the feature selection test with Relief where Ranker is used as the search method. Relief gives the rankings of the attributes for each data set. When we take the average of the rankings for each metric on all data sets, we observe that except for DIT metric, the results of the feature selection tests with Relief are coherent with our results shown in Table 5. For instance, still LCOM, LCOM3, and NOC are less effective compared to other metrics. Similar to the results shown in Table 5, metrics RFC and CBO are the among most effective metrics. Although the average rankings found for LOC, LOCQ, and WMC are not so good, they are still more important compared to LCOM, LCOM3, and NOC (See Table 9).

5.6 Effectiveness of the Number of Developers (NOD)

Among the data sets listed in Table 3, we use Poi, Tomcat, and Xalan to extract the number of developers since developer names could be retrieved from their source code repositories. We count the number of distinct developers (NOD) for each class of each data set. We use the NOD metric together with the metrics listed in Table 1 and learn a Bayesian network for each data set, to extract its relationship with other metrics and the extent of defect proneness. Furthermore, we select K2 as the search algorithm and use predefined ordering of nodes of LOC, NOD, CBO, LOCQ, WMC, RFC, LCOM, LCOM3, DIT, and NOC. To see the

Table 9 The results of feature selection tests with ReliefFAttributeEval and Ranker search method where the rankings of the metrics are shown for each data set (If the average ranking of a metric is smaller, then it means the metric is more important).

Data Sets	LOC	CBO	LOCQ	WMC	RFC	LCOM	LCOM3	DIT	NOC
Ant	5	6	3	4	2	9	8	1	7
Tomcat	5	2	4	6	3	9	8	1	7
Poi	6	5	3	1	2	7	9	4	8
Jedit	5	3	6	4	2	7	8	1	9
Velocity	5	2	6	4	1	7	9	3	8
Synapse	2	3	6	5	1	7	8	4	9
Lucene	7	2	5	4	3	8	1	9	6
Xalan	1	3	5	2	4	7	6	9	8
Ivy	5	2	4	6	3	9	8	1	7
Average	**4.56**	**3.11**	**4.67**	**4.00**	**2.33**	**7.78**	**7.22**	**3.67**	**7.67**

relationship of NOD and the level of defectiveness better, we define three states for defect proneness. All class instances where bug is zero are accepted as defect free classes. The classes that have 1 or 2 bugs, are marked as less defective, and the classes that have more than 2 bugs are accepted as more defective. As a result, we simply define three defect proneness states which are, defect free, less defective, and more defective. There is nothing special for the threshold values we use to define the level of defectiveness, someone else might use different thresholds or define more levels for defect proneness. The Bayesian networks we obtain at the end of our experiments are shown in Figure 8.

For all Bayesian networks learned, NOD is directly effective on defect proneness and we observe a positive correlation between NOD and the level of defectiveness. For instance for Poi data set, we see that as the number of developers increases, the defectiveness increases too. If the number of developers are less than 3, the non defectiveness probability is 0.997, but it is 0.003 if there are more than 3 developers per class. For Tomcat data set, we observe that if the number of developers is more than 1, the probability of a defect free class is 0.167. But the probabilities of having a less or more defective class are 0.514 and 0.812 respectively. For Xalan, we observe a similar relationship between NOD and the level of defectiveness, where if the number of developers is less than 2 then the non defectiveness probability is 0.705. If NOD is 2 or 3 then the non defectiveness probability is 0.276 and if NOD is greater than 3 then it is only 0.019. Apparently, as the number of developers increases, the non defectiveness probability decreases or the level of defectiveness increases.

To be sure that our results do not suffer from conclusion instability, and our observations are valid for the subsets of the data sets too, we repeat our experiments with the 10 subsets of each data set. Each data set is stratified and 67 percent of its data is included in the subsets. Furthermore, for each stratification a different seed is used. For the Bayesian network obtained in each experiment, a metric is assigned 1, if it has a relationship with defectiveness and assigned zero otherwise. The average scores of the metrics for Poi, Tomcat, and Xalan data sets are shown in Table 10. In 7 experiments for Poi, in all experiments for Tomcat, and in 7 experiments for Xalan, NOD is directly effective on defectiveness and there is a positive correlation with the developer count and defectiveness. Furthermore, supporting the results observed in the previous experiments (Tables 5 and

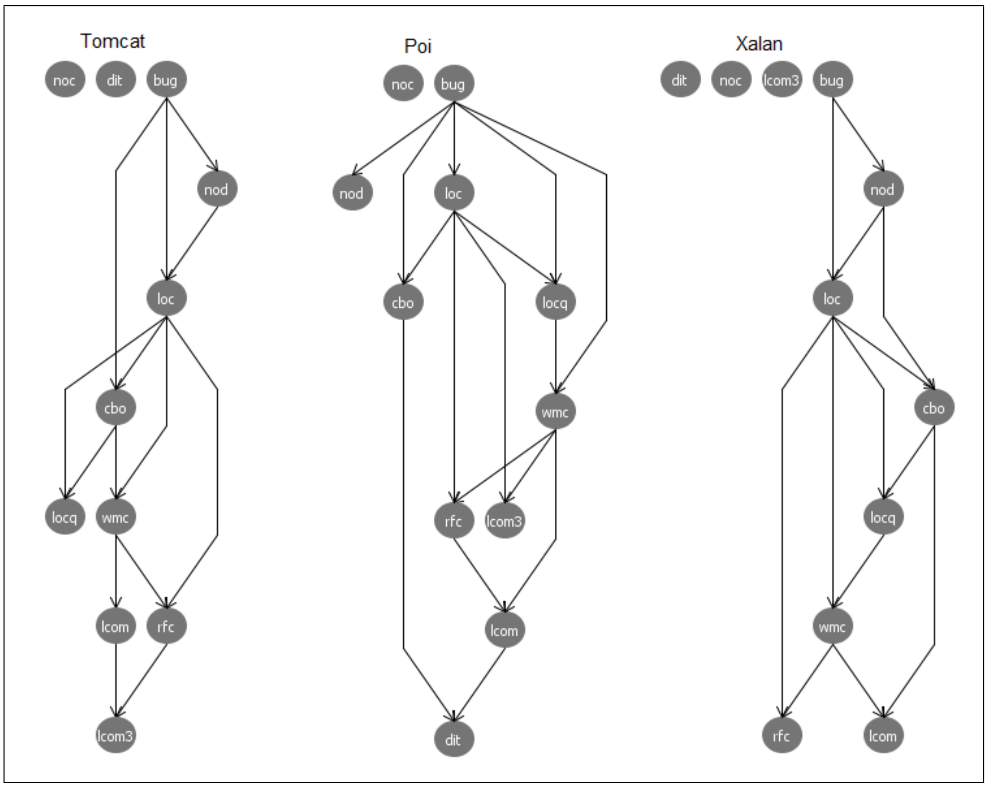


Fig. 8 Bayesian networks showing the relationship of the number of developers with the level of defectiveness in Poi, Tomcat, and Xalan data sets.

Table 10 The average scores of metrics obtained at the end of runs on 10 subsets of Poi, Tomcat, and Xalan data sets (If a metric affects defectiveness we assign it a score of 1, if it has no relationship with defectiveness it is assigned a zero score.)

	Data sets			
Metrics	**Poi**	**Tomcat**	**Xalan**	**Average**
NOD	0.7	1	0.7	**0.80**
LOC	1	1	1	**1.00**
CBO	1	1	0.9	**0.97**
LOCQ	1	1	0.9	**0.97**
WMC	1	0.9	0.9	**0.93**
RFC	1	1	1	**1.00**
LCOM	1	0.9	1	**0.97**
LCOM3	1	0.6	0.6	**0.73**
DIT	0.7	0	0	**0.23**
NOC	0	0	0	**0.00**

6), LOC and RFC are the most effective metrics whereas DIT and NOC are the least effective ones.

We compare the non defectiveness probabilities of two cases i.e. when the number of developers is 1 ($NOD = 1$) and when it is greater than 1 ($NOD > 1$). Figure

9 shows the non defectiveness probabilities of these two cases, for 30 experiments carried out on 10 stratified subsets of Poi, Tomcat, and Xalan (3 experiments for Poi and 3 experiments for Xalan data sets where NOD is not related with defectiveness are not included). To check for the statistical significance of the results, we apply a t-test (in 95 % confidence interval) to the non defectiveness probabilities of the two cases and show that the non defectiveness probability when NOD = 1 is better with a p value of zero.

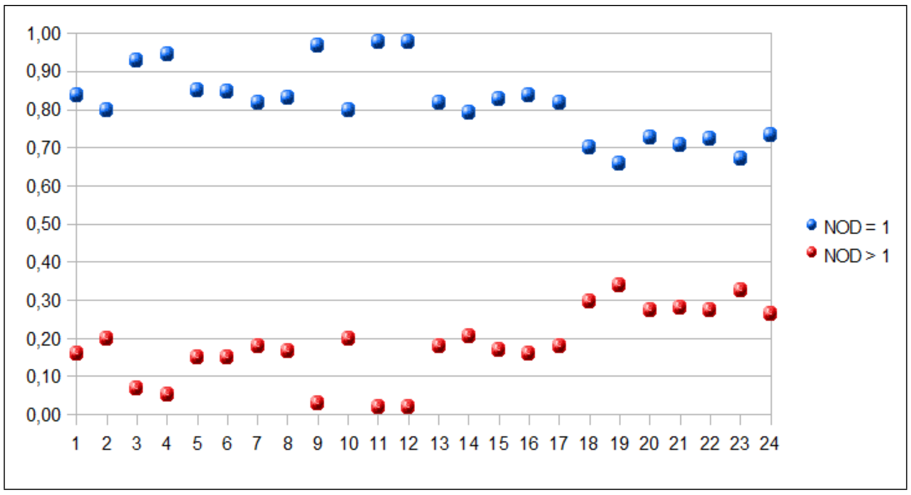


Fig. 9 The non defectiveness probabilities of two cases i.e. when the number of developers is 1 ($NOD = 1$) and when it is greater than 1 ($NOD > 1$)

We conclude that as the number of developers increases for a specific class, the class tends to be more defective and show that the common idiom "too many cookers spoil the broth" is valid for Software Engineering. At the end of our experiments, we observe that too many developers make a class more defect prone. This is due to the fact that when too many developers work on the same piece of code, the number of potential dependencies among their work items increases. We recommend project managers to explore the cost benefit curve of NOD versus defectiveness level where the value of adding more developers should be controlled with respect to the introduced number of defects. We must emphasize that this conclusion is based on the experiments on the data sets used and other researchers should make more experiments on more data sets to justify our findings.

6 Threats to Validity

According to Perry et al. there are three types of validity threats that should be considered in research studies. We briefly explain the methodology we follow to alleviate these threats [38].

An internal validity threat might arise if a cause effect relationship could not be established between the independent variables and the results. We address this issue by cross checking our results on different subsets of the data sets. During our experiments, not only we use 10 fold cross validation, but we also replicate all of the experiments on 20 different subsets of all the data sets.

Construct validity threats might be observed when there are errors in the measurements. To mitigate this threat, first we automatize the metric extraction process and minimize the manual interventions, second we cross check the extracted metrics and try to find if any abnormal values exist.

External validity threats might arise if the results observed for one data set are not valid for other data sets. To mitigate external validity, we test our proposed method on several data sets and replicate the experiments on their subsets. Although our results are promising since metric effectiveness is investigated on more than one data set, further research with more data sets and more search algorithms is needed to justify our findings.

7 Conclusion

In this paper, we propose a novel method using Bayesian networks to explore the relationships among software metrics and defect proneness. We use nine data sets from Promise data repository and show that RFC, LOC, and LOCQ are more effective on defect proneness. On the other hand, the effect of NOC and DIT on defectiveness is limited and untrustworthy.

The main contributions of this research are:

- This paper uses Bayesian networks to model the relationships among metrics and defect proneness on multiple data sets. For instance Gyimothy et al. [16] used Mozilla data set whereas Zhou et al. and Pai and Dugan used KC1 data set from Nasa repository [47,35]. The results obtained using one data set might be misleading since a metric might perform well on one data set but poor on another one. As Menzies et al. suggest, it is not adequate to assess defect learning methods using only one data set and only one learner, since the merits of the proposed techniques shall be evaluated via extensive experimentation [28]. Our work is a good contribution to the literature, since we determine the probabilistic causal or influential relationships among metrics and defect proneness, considering 9 data sets at the same time.
- We introduce a new metric we call Lack of Coding Quality (LOCQ) that can be used to predict defectiveness and is as effective as the famous object oriented metrics like CBO and WMC.
- We extract the Number of Developers (NOD) metric for data sets whose source code include developer information and show that there is a positive correlation between the number of developers and the extent of defect proneness. So, we suggest project managers to be careful while assigning more than one developer to one class or file.
- It was found that in most experiments NOC and DIT are not effective on defectiveness.
- Furthermore, since LOC achieves one of the best scores in the experiments, we believe that it could be used for a quick defect prediction since it can be measured more easily compared to other metrics.
- LCOM3 and LCOM are less effective on defect proneness compared to LOC, CBO, RFC, LOCQ, and WMC.

As a future direction, we plan to refine our research to include other software and process metrics in our model to reveal the relationships among them and to determine the most useful ones in defect prediction. We believe that rather than dealing with a large set of software metrics, focusing on the most effective ones will improve the success rate in defect prediction studies.

References

1. Nasa/wvu iv and v facility, metrics data program available from http://mdp.ivv.nasa.gov/; internet accessed 2010.
2. E. Alpaydın. *Introduction to Machine Learning*. The MIT Press, October 2004.
3. S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno. A bayesian belief network for assessing the likelihood of fault content. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, Washington, DC, USA, 2003. IEEE Computer Society.
4. S. Bibi and I. Stamelos. Software process modeling with bayesian belief networks. *Proceedings of 10th International Software Metrics Symposium*, 2004.
5. G. Boetticher, T. Menzies, and T. Ostrand. Promise repository of empirical software engineering data http://promisedata.org/ repository, west virginia university, department of computer science, 2007.
6. G. D. Boetticher. Nearest neighbor sampling for better defect prediction. In *Proceedings of the 2005 workshop on Predictor models in software engineering*, PROMISE '05, 2005.
7. S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. *SIGPLAN*, 26(11):197–211, 1991.
8. G. F. Cooper and E. Herskovits. A bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347, 1992.
9. K. Dejaeger, T. Verbraken, and B. Baesens. Towards comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2012.
10. J. Ekanayake, J. Tappolet, H. Gall, and A. Bernstein. Tracking concept drift of software projects using defect prediction quality. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 51 –60, 2009.
11. K. E. Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56:63–75, 2001.
12. N. Fenton, P. Krause, and M. Neil. Software measurement: Uncertainty and causal modeling. *IEEE Software*, 19:116–122, 2002.
13. N. Fenton, M. Neil, and D. Marquez. Using bayesian networks to predict software defects and reliability. *Proceedings of the Institution of Mechanical Engineers, Part O, Journal of Risk and Reliability*, 2008.
14. N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra. Predicting software defects in varying development lifecycles using bayesian nets. *Inf. Softw. Technol.*, 49(1):32–43, Jan. 2007.
15. N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25:675–689, 1999.
16. T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.
17. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.
18. B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.
19. Y. Hu, X. Zhang, X. Sun, M. Liu, and J. Du. An intelligent model for software project risk prediction. In *International Conference on Information Management, Innovation Management and Industrial Engineering, 2009*, volume 1, pages 629 –632, 2009.
20. C. Jin and J.-A. Liu. Applications of support vector machine and unsupervised learning for predicting maintainability using object-oriented metrics. In *2010 Second International Conference on Multimedia and Information Technology (MMIT)*, volume 1, pages 24 –27, 2010.
21. A. Kaur, P. Sandhu, and A. Bra. Early software fault prediction using real time defect data. In *Machine Vision, 2009. ICMV '09. Second International Conference on*, pages 242 –245, dec. 2009.
22. T. Khoshgoftaar, E. Allen, and J. Busboom. Modeling software quality: the software measurement analysis and reliability toolkit. In *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on*, pages 54 –61, 2000.
23. T. Khoshgoftaar, K. Ganesan, E. Allen, F. Ross, R. Munikoti, N. Goel, and A. Nandi. Predicting fault-prone modules with case-based reasoning. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 27 –35, 2-5 1997.

24. T. M. Khoshgoftaar, A. S. Pandya, and D. L. Lanning. Application of neural networks for predicting program faults. *Ann. Software Eng.*, 1:141–154, 1995.
25. A. G. Koru and H. Liu. An investigation of the effect of module size on defect prediction using static measures. *SIGSOFT Software Enginering Notes*, 30, May 2005.
26. S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34:485–496, 2008.
27. T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok. Local vs global models for effort estimation and defect prediction. In *Proceedings of the 26st IEEE/ACM International Conference on Automated Software Engineering*, Lawrence, Kansas, USA, November 2011.
28. T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33:2–13, 2007.
29. T. Menzies and M. Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Software Engineering*, 17(1-2):1–17, 2012.
30. A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, pages 309–346, 2002.
31. J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. Softw. Eng.*, 18:423–433, May 1992.
32. I. Myrtveit, E. Stensrud, and M. Shepperd. Reliability and validity in comparative studies of software prediction models. *IEEE Transactions on Software Engineering*, 31:380–391, May 2005.
33. N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.
34. B. Norick, J. Krohn, E. Howard, B. Welna, and C. Izurieta. Effects of the number of developers on code quality in open source software: a case study. In G. Succi, M. Morisio, and N. Nagappan, editors, *ESEM*. ACM, 2010.
35. G. Pai and J. Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on Software Engineering*, 33(10):675–686, 2007.
36. P. C. Pendharkar and J. A. Rodger. An empirical study of the impact of team size on software development effort. *Inf. Technol. and Management*, pages 253–262, 2007.
37. E. Pérez-Miñana and J.-J. Gras. Improving fault prediction using bayesian networks for the development of embedded software applications: Research articles. *Software Testing, Verification and Reliability*, 16(3):157–174, Sept. 2006.
38. D. E. Perry, A. A. Porter, and L. G. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 345–355, New York, NY, USA, 2000. ACM.
39. D. Posnett, V. Filkov, and P. T. Devanbu. Ecological inference in empirical software engineering. pages 362–371. IEEE, 2011.
40. M. Shepperd and G. Kadoda. Comparing software prediction techniques using simulation. *IEEE Trans. Softw. Eng.*, 27, November 2001.
41. S. Shivaji, E. Whitehead, R. Akella, and S. Kim. Reducing features to improve bug prediction. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 600 –604, 2009.
42. Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *Software Engineering, IEEE Transactions on*, 32(2):69 – 82, feb. 2006.
43. M. Suffian and M. Abdullah. Establishing a defect prediction model using a combination of product metrics as predictors via six sigma methodology. In *Information Technology (ITSim), 2010 International Symposium in*, pages 1087 –1092, 2010.
44. M. M. T. Thwin and T.-S. Quah. Application of neural network for predicting software development faults using object-oriented design metrics. In *Neural Information Processing, 2002. ICONIP '02. Proceedings of the 9th International Conference on*, volume 5, pages 2312 – 2316 vol.5, nov. 2002.
45. D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.
46. D. Zhang. Applying machine learning algorithms in software development. In *Proceedings of the 2000 Monterey Workshop on Modeling Software System Structures in a Fastly Moving Scenario*, pages 275–291, 2000.

47. Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32:771–789, October 2006.