Received: 29 July 2018 | Revised: 15 January 2019 | Accepted: 14 February 2019

DOI: 10.1002/smr.2163

SPECIAL ISSUE PAPER

WILEY Journal of Software: Evolution and Process

Maintenance process modeling and dynamic estimations based on Bayesian networks and association rules

Angelos Chatzimparmpas | Stamatia Bibi

Department of Informatics and Telecommunications Engineering, University of Western Macedonia, Kozani, Greece

Correspondence
Stamatia Bibi, Department of Informatics and Telecommunications Engineering, University of Western Macedonia, Kozani, Greece.
Email: sbibi@uowm.gr

Abstract

Managing the maintenance process and estimating accurately the effort and duration required for a new release is considered to be a crucial task as it affects successful software project survival and progress over time. In this study, we propose the combination of two well-known machine learning (ML) techniques, *Bayesian networks (BNs)*, and *association rules (ARs)* for modeling the maintenance process by identifying the relationships among the internal and external quality metrics related to a particular project release to both the maintainability of the project and the maintenance process indicators (ie, effort and duration). We also exploit *Bayesian inference*, to test the effect of certain changes in internal and external project factors to the maintainability of a project. We evaluate our approach through a case study on 957 releases of five open source JavaScript applications. The results show that the maintainability of a release, the changes observed between subsequent releases, and the time required between two releases can be accurately predicted from size, complexity, and activity metrics. The proposed combined approach achieves higher accuracy when evaluated against the BN model accuracy.

KEYWORDS

developers' activity, JavaScript, maintainability, maintenance, software quality, source code quality

1 | INTRODUCTION

Software quality involves many different activities throughout the operational lifecycle of an application all targeted to the delivery of software that meets the specified requirements and satisfies the evolving needs of end-users. Quality concerns are often intertwined with software maintenance that is closely related to technical debt.[1] Software maintenance is one of the most demanding activities during the software lifecycle, consuming up to 75% of the total project resources,[2] mainly due to the many constraints related to the interdependencies of the source code artifacts of the software under maintenance. According to the IEEE 1219[3] software standards document, *software maintenance* is defined as the "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." The *maintainability* is "the ease of performing maintenance activities," while *maintenance effort* is subsequently defined as the "effort required to reduce or eliminate maintenance problems." Despite the importance of software maintenance and its crucial effect on the quality of a software application over time, its management and assessment is still an immature discipline in software engineering research and practice. The main problem is that software maintainability (as a single synthesized factor) and maintenance process indicators (ie, effort and number of modifications) are most of the times addressed separately without being able to test the effect of the low level internal and external metrics to both maintainability and maintenance indicators at the same time.

This gap can be alleviated by modeling the relationship among the process and product metrics related to both the maintainability of an application and the maintenance process. Nowadays, mature software organizations collect a wealth of data regarding software development and maintenance, an analysis of which can help towards extracting knowledge for effectively monitoring the whole maintenance process. Using

well-established machine learning (ML) techniques, practitioners and researchers can explore the potential of this data by identifying relations among project and process characteristics and the various maintainability indicators. Ideally, both exploratory (ie, models that extract frequently appearing patterns among the variables) and predictive models (ie, models that provide estimations of certain variables) could be of practical use in the context of software maintenance. *Exploratory* models can describe specific relationships among particular combinations of internal and external quality characteristics and their influence to the maintainability of an application and subsequently to the related maintenance process indicators (ie, effort and time required for a maintenance cycle). *Predictive models* could be of practical use by providing estimations on the values of the maintainability and the maintenance process indicators.

In this study, we propose the combination of two well-known ML techniques, *Bayesian networks (BNs)*[4] and *association rules (ARs)*,[5] for monitoring the maintainability of applications over time by supporting the whole maintenance process. We define a three-step process for software maintenance assessment and prediction based on BN and AR models that includes (1) the data collection and preparation step where the data are transformed in a suitable form so as to serve as an input to the models, (2) the creation and evaluation of the BN and AR models where we demonstrate the means to evaluate and identify defective aspects of the BN models that can be confronted with the application of ARs, and (3) the evidence-based estimation and inference step where the software manager is able to insert updated knowledge in the model and test its effect on the outcomes.

The suggested approach attempts to take advantage of the unique attributes of the two methods. In particular:

- The formalism of BN enables the development of a probabilistic model for software maintenance. BNs are effective in dealing with uncertainty and enable us to measure through Bayesian inference the effect in mathematical terms of certain changes in internal and external project factors to the maintainability of a project.
- BNs as a predictive modeling method can provide a complete estimation framework that classifies an application release to a certain maintainability interval, based on data from several historical releases. On the other hand, a BN model can be easily supplemented with expert judgment in the cases where project managers have scarce statistical data of present or past projects and wish to produce a semi-automated BN model that can be easily controlled by humans.[6]
- AR pattern recognition supports the discovery of frequently appearing cause-effect relationships among the application attributes and the maintainability indicators. AR is a method for descriptive modeling that identifies patterns often ignored by predictive models that tend to minimize the set of variables used for prediction purposes. Therefore, AR can identify particular combinations of attribute values that have a common effect on the maintainability indicators that, otherwise, would have been excluded.
- The representation form of AR is transparent, easy to understand, and reproduced in order to offer a more precise explanation of how the prediction has been made. This representation is important to a problem domain such as software maintenance, where the manager must trust the output; otherwise, the model may be ignored.[7]

In order to evaluate the proposed approach and demonstrate its efficiency, we perform an empirical study on 957 releases of five popular applications which are developed in the JavaScript (JS) scripting language. The motivation behind the need to analyze JS applications regarding maintenance is twofold: (1) The fact that JS is considered as a weakly typed programming language, meaning that it has looser type rules, that may generate unpredictable results. In this context, we want to see whether this fact may cause problems to the maintenance of projects and focus on identifying the aspects that may create them. (2) The fact that many programmers rely upon popular JS frameworks (JQuery is one of them) for building their web applications reveals the need to further explore the potentials of JS frameworks in terms of maintenance and adjustment to user demands. For the purpose of this study, we analyzed the source code of the subsequent releases of five popular JS applications and additionally record a set of project activity metrics and release metrics such as the changes performed between releases and the time required for them. The results of applying the proposed approach for monitoring the maintenance activities of five JS applications show that the maintainability metric can be accurately predicted from size metrics along with activity metrics like the *number of commits* (NoCom). The changes observed between subsequent releases, as an indicator of the maintenance effort, and the time required between two releases are dependent on the maintainability of the previous release. In general, we observed variances between the estimation models derived for the five applications a fact that appoints the need for customized maintenance models based on the background and the specific characteristics of each application.

The rest of the paper is organized as follows: Section 2 presents related work; Section 3 provides the details of the two modeling techniques (ie, BN and AR), and Section 4 outlines the proposed approach. In Section 5, we present the case study design details. In Section 6, we present the results, while in Section 7, we discuss the results and provide implications to researchers and practitioners. Finally, Section 8 concludes the paper by summarizing our findings and referring to possible future work.

2 | RELATED WORK

During the past decades, several ML methods have been proposed to predict software quality. *Regression* models were one of the first fundamental ML algorithms used by Niessink et al[8] and Dagpinar et al.[9] The former evaluated the predicted maintenance effort against analogy-

based, function points, and expert estimations. The latter predicted maintainability with object-oriented metrics. There are different types of regression models like linear[10] and multiple linear regression.[11] Misra[10] performed an experimental statistical analysis in order to assess maintainability by using different types of metrics (eg, complexity and size). Additionally, Fioravanti et al[11] estimated the maintenance effort by dividing metrics to (1) functional-based and (2) counting members. Other ML techniques such as *fuzzy models*[12] and *neural networks*[13] have been explored for assessing the maintainability and maintenance effort correspondingly based on a set of object-oriented complexity and size metrics. Zhou et al[14] proposed the use of *multiple adaptive regression splines* (MARS) as an effective way to predict maintainability from size and object-oriented complexity metrics and evaluated the applicability of the method on two data sets. Some studies experiment with compilations of different ML methods like Kaur et al,[15] who compared *back propagation artificial neural network*, *generalized regression neural network* (GRNN), *fuzzy inference systems* (FIS), and *adaptive neuro-fuzzy inference systems* (ANFIS) for the assessment of maintenance effort. *Support vector machine* (SVM) and *TreeNet* were used by Jin et al[16] and Elish et al,[17] respectively. In the SVM approach, the maintainability was predicted based on object-oriented metrics. First, the authors used *fuzzy C-means clustering* to find the distance from the clustering center, and by doing that they were left with five metrics. Then, SVM predicts their relation with the software maintainability referred as maintenance effort which is equal to the lines changed per class. The TreeNet approach[17] was compared against other techniques like multivariate adaptive *regression splines*, *multivariate linear regression* (MLR), *support vector regression* (SVR), *artificial neural network* (ANN), and *regression tree* (RT) and was found competitive in estimating the maintenance effort. Fontana et al[18] present an overview of ML approaches in the context of identifying code smells. The main problem with the majority of the proposed techniques is that the estimations they derive work as a "black-box," ie, they do not present the rationale of the estimate. It is not easy for a software engineer to understand and intuitively confirm the estimation models derived from regression models.[19]

On the other hand, *BN* modeling adopts a representation formalism that can be easily accepted by practitioners. Several researchers have proposed BNs within the context of software quality estimation. Okutan et al[20] suggested the use of BNs for software defect predictions and also introduced two new metrics the *number of developers* and the *lines of code quality*. Fenton et al[21] also used BN modeling for predicting software defects, by separating projects into development lifecycles and modeling them to *dynamic Bayesian networks* (DBNs). DBNs are different phase BNs which are constructed by one of the three activity classes: (1) specification and documentation; (2) design and development; and (3) test and rework. Wagner[22] proposed a BN approach for modeling the *activity-based quality model* (ABQM) based solely on expert judgment for constructing the networks and the associated probabilities. Wagner[22] modeled the activities during the software maintenance process with their associated outcomes (comment lines, size, and cyclomatic complexity) and estimated the average maintenance effort. Bibi et al[23] employed BNs to model source code quality characteristics of 20 open source Java applications with maintenance process indicators like duration, effort, and production.

In this study, we go beyond existing literature by

- Modeling the relationship among the process and product metrics related to *both the maintainability of an application and the maintenance process indicators* like duration and effort (measured as the changes occurring from one release to another).
- Employing a plethora of metrics related to the maintainability of applications like *internal source code quality metrics* (size, duplications, and complexity) and *external and internal activity metrics* (number of developers, user acceptance, number of code interventions, and bugs reporting). Developer-related factors, as the number of developers and the NoCom, have also been found as important factors by Catolino et al[24] for estimating the proportion of changes performed during maintenance.
- Proposing *evidence-based inference* for testing the effect of certain changes on internal quality factors (ie, *lines of code*) and activity factors (ie, *NoCom*) to maintainability.

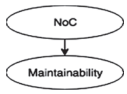
3 | BACKGROUND INFORMATION

In this section, we present the background theory of the two ML methods employed in this study to model the maintenance process. In Section 3.1, we provide an overview of BN models and exemplify with a short example. Next, in Section 3.2, we present the basic concepts of AR mining.

3.1 | Bayesian networks

BNs are *directed acyclic graphs* (DAGs) that consist of a set of nodes and directed links between them.[4] Each node represents a random variable that can take mutually exclusive values according to a probability distribution, which can be different for each node. Each link expresses probabilistic cause-effect relations among the linked variables and is depicted by an arc starting from the influencing variable (parent node) and terminating on the influenced variable (child node). The directions of the links indicate the direction of the impact. Causal networks can be used in order to follow how a change of certainty in one variable may change the certainty for other variables. Therefore, to each variable *A* with influencing

(A)



(B)

NoC (Number of Classes)		Low (1-10 classes)	High (10-30 classes)
Maintainability	High (< 3)	0.7	0.45
	Low(≥3)	0.3	0.55

FIGURE 1 A, A BN for maintainability; B, conditional probability table for maintainability estimation

variables (parents) *B1*, ..., *Bn*, the conditional probability table (CPT) $Pr(A|B_1,.., B_n)$ is attached. Formally, the relationship between two nodes (A and B) can be expressed with the help of Bayes rule:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{1}$$

The construction of BN models is based on identifying the nodes that are conditionally independent based on the d-separation properties that are analytically presented in Nielsen and Jensen.[4]

A simple BN example is presented in Figure 1. The model consists of two nodes. The first node (NoC) represents the number of classes in a software application, and the second node (Maintainability) represents the maintainability of the application measured with the help of experts, ie, an expert assigns a maintainability value that ranges from 0, which is an excellent maintainability indicator to values higher than 3 that are considered poor maintainability indicators). We consider that the values of these two nodes fall into two discrete categories (low and high). For the node NoC, let us suppose that low values range between 1 class and 10 classes. For example, the first column of the *CPT* states that if the number of classes is low, then there is 70% probability that the maintainability will be high and 30% probability that the maintainability will be low. The tool used for constructing the networks and the classifier was Genie modeler* .The algorithm used for learning BNs is analytically presented in Druzdzel.[25]

3.2 | Association rules

Association rules (ARs) belong to descriptive modeling techniques and have as a target to describe the data and their underlying relationships with a set of rules that jointly define the variables of interest.[5] ARs are among the most popular representations of local pattern recognition as they find frequent combinations of attribute values that lay in data sets. An AR is a simple probabilistic statement about the co-occurrence of certain events. Each rule consists of two parts. The left part is the rule body and is the necessary condition in order to validate the right part, rule head. Each rule states that if the rule body is true, then the rule head is also true with probability p. ARs are Boolean propositions with true or false values. Given a set of observations over attributes A_1, A_2, ..., A_n in a data set *D*, a simple AR has the following form:

$$\begin{aligned} &\mathbf{A1 = X \text{ and } A2 = Y => A3 = Z} \\ &\text{Confidence} = P(A3 = Z|A1 = X, A2 = Y) \\ &\text{Support} = \text{freq}(X \cap Y \cap Z, D). \end{aligned}$$

This rule is interpreted as following: when the attribute A_1 has the value *X* and attribute A_2 has the value *Y*, then there is a probability p (confidence) that attribute A_3 has the value *Z*. For this rule, two major statistics are computed, confidence and support values. Confidence is the probability p defined as the percentage of the records containing *X*, *Y*, and *Z* with regard to the overall number of records containing *X* and *Y* only. Support is a measure that expresses the frequency of the rule and is the ratio between the number of records that present *X*, *Y*, and *Z* to the total number of records in the data set (*D*). In this study, Weka is used for extracting the ARs based on the algorithms that are analytically presented in Hall et al.[26]

4 | PROPOSED APPROACH

In this section, we present the proposed approach that employs BNs and ARs for performing probability inference and addressing causality in the context of software maintenance. The proposed approach contains three main phases: (1) data collection and preparation, (2) BN and ARs model extraction and evaluation, and (3) evidence-based estimation and inference.

4.1 | Data collection and preparation

During this phase, the software engineer is expected to prioritize the maintenance goals and activities and define the metrics that should be collected in order to measure and control the maintainability of the application, effectively. For example, based on the type and the goals of an application, he can set different maintenance goals and more specific metrics to measure them (eg, in the case of mobile applications user engagement

*https://www.bayesfusion.com/genie/

or acquisition metrics could be of high relevance with respect to maintenance). Hence, the first step of this phase is the *data collection (step 1.1)*. A set of candidate metrics that we suggest for the *maintainability* and the *maintenance process assessment* are *internal source code quality metrics*, *external quality metrics*, and *process metrics*.

Internal quality metrics can consist of

- *Source code size metrics* that involve the descriptive statistics of an application such as the lines of code, the number of functions, the number of files, the number of attributes, the number of statements.
- *Source code complexity metrics* that may include the cyclomatic complexity,[27] the cognitive complexity, or even metrics that count the average complexity per function.
- *Source code object-oriented* metrics applies only in applications that are developed with object-oriented languages and may include metrics as defined in the QMOOD model.[28]

External quality metrics can consist of

- *Operational metrics* that may include the number of operational bugs reported, the number of new functionalities requests, and the number of malfunctions.
- *End-user activity* that may include the number of end-users and the end-user quality rating.

Process metrics consist of

- *Development team metrics* that may include the number of developers, the experience of developers, and the number of interventions made by the development team in each release.
- *Changes metric* that is an indicator of the changes performed between two subsequent releases.
- *Duration metric* that is an indicator of the time required for performing maintenance activities between two subsequent releases.

The next step (*step 1.2*) is to *define a set of tools* that will ease and automate the metrics recording procedure for monitoring the maintenance process of an application. For the internal source code size and complexity assessment of applications, there is a set of open source freely available tools that can assist in that process (ie, SonarQube and language-specific tools, Jdeodorant, Percerons, and JSClassFinder[†]). We suggest the use of SonarQube that is language-agnostic tool that calculates a variety of different metrics. The end-user metrics can be derived with the help of online forums, user lists, phone-desks recording systems, or even bug-tracking/recording systems if these are made available to the end-users. The process metrics normally are reported by the maintenance/development team that should be able to track the time required and the type of changes performed between the releases of an application.

As a final step of this phase, we define the *data preparation (step 1.3)* that may include the creation of new variables (eg, calculate the values of derived variables such as the average size of functions in an application), outlier removal, and data modifications including the transformation of variables presenting continuous values to categorical ones. The transformation of continuous variables to categorical variables is a prerequisite for deriving the BN and ARs models and should be performed with great caution as the number of classes considered and the discretization method might affect the final estimation models' accuracy and representativeness. The number of classes considered can be selected empirically or by adopting one of the rules suggested by literature such as Sturge's rule.[29] The same stands for the selection of the discretization method that can be based solely on empirical observation or on automated methods such as equal width binning, equal frequency binning, and clustering.[30] According to Peng et al,[31] equal frequency binning is a method that provides: (1) clear insights of the distribution of the observations, (2) creates more stable classes compared with equal-width binning, and (3) can handle more effectively the outliers compared with equal-width binning. Equal-width binning on the other hand may result in a single class concentrating the majority of observations, while the rest of the classes remain with very few observations. Therefore, we suggest the classification of the values of the independent variables by employing equal-frequency binning. As mentioned in Peng et al,[31] discretization according to intuition is more appropriate for simple and practically meaningful real data sets, especially in the case of the dependent variables, and it is less vulnerable to outliers. Therefore, for the dependent variables, we suggest a selection of the "cutting-off" points of the classes based on the empirical distribution of the values of the dependent variables (the empirical selection of classes is discussed analytically in Section 6.1.1).

[†]SonarQube: https://www.sonarqube.org/
JDeodorant: https://github.com/tsantalis/JDeodorant
Percerons: http://www.percerons.com/
JSClassfinder: https://github.com/aserg-ufmg/JSClassFinder

4.2 | Bayesian network and association rules model extraction and evaluation

During this phase, the software engineer has to define the parameters of the two ML models, derive the initial models, and evaluate them. The first step of this phase is the *development of the Bayesian network (step 2.1)*. In this step, expert input is required to *set the learning parameters of the BN model*. Initially, the engineer needs to *set the order of the variables* that affects the causal relationships that the model will define. The metrics order should be selected by taking into account two considerations: (1) the maintenance phase that refers to the time when the values of the metrics will be available in the sense that we are first aware of certain bugs related to the particular release, while sometime afterward when the bugs are closed, we observe differences in the source code metrics, and (2) the relatedness of metrics. For example, *complexity* metrics like *cyclomatic complexity number* and *cognitive complexity* are closely related. We think that *complexity* precedes *cognitive complexity*, in the sense that the value of the last one "includes" the value of *cyclomatic complexity number*. In general, we suggest that the hierarchy of the general metrics considered in the context of maintenance complies with the following order: *external quality metrics* (operational metrics and user activity), *internal team activity metrics*, *source code quality metrics* (size, complexity, and object-oriented metrics), and *maintenance process metrics* (changes, duration). During this step, we also need to *set the number of parents* of each node (an ideal number is 3 so as not to increase the complexity of the CPTs). For automatically deriving the BN model, there are a number of freely available tools such as Genie and Weka.[‡] In this study, we selected Genie because it provides a variety of parameterization options, it allows the modeler to insert expert knowledge, and it also allows evidence-based updates of the models. After the extraction of the BN model, the *evaluation of the fitting accuracy of the model (step 2.2)* proceeds. This is necessary in order to identify cases, where the BN model is not able to accurately estimate the variables of interest and focus on the specific classes of values that are misclassified. For this purpose, it is suggested to study the receiver operating characteristic (ROC) curves for each class of the dependent variables, along with the confusion matrix, that will provide further information on the particular classes that the BN model presents week estimation accuracy. In this case, we isolate the classes that present poor classification results and proceed to the next step that involves the ARs mining so as to retrieve representative rules for the particular wrongly estimated classes. In some of the cases where the target variable is misclassified, the probabilities stated in the CPT are equally distributed among the values of the independent variables, and therefore these are among the cases that also need to be further explored.

The next step of this phase is the *Association rules mining (step 2.3)*. In this step, the software engineer identifies all frequent and pertinent set of attributes. Frequent set of attributes are considered those, whose associated support exceeds a certain support threshold. The support value threshold depends on the number of observations existent in the data set for a certain class of the dependent variable. If the observations are very few, then the support threshold should be set in very low values. Pertinent set of attributes are those whose associated confidence exceeds a certain confidence threshold. Finally, the rules are sorted according to their confidence values, and the set of rules related to the classes of interest of the dependent variable (those appointed in step 2.2) are selected so as to provide a more accurate and targeted estimate. In this study, we considered as confidence threshold 50% of the cases and as support threshold 5% of the cases. The confidence percentage of 50% is adopted so as to ensure that a pattern has at least 50% probability to be correctly identified in the observations. The support threshold is set to relatively low values so as to identify patterns that are not frequent, but they are still related to the estimation of the classes of dependent variables. The last step of this phase is the *combination of the BN and AR* models (step 2.4). In this step, the software engineer sorts all the estimations derived from the CPTs of the BN network based on the assigned probabilities for each class of the dependent variable. The probabilities are indicators of the confidence level of the estimation. Then, the estimator sorts the AR rules based on the confidence value of each rule. When estimating a new observation, the software engineer selects the model that provides the estimate that presents the highest confidence level, by comparing the confidence levels of the estimations provided by the two models (see the analytical estimation example of Section 6.1.2).

The steps of this phase are summarized in Table 1.

4.3 | Evidence-based estimation and inference

During this phase, the software engineer is able to test the impact of certain changes in the application functionality to the maintainability of the application and to perform an estimate on the expected changes to the internal source code and the time required to perform the relevant maintenance actions. For instance, the engineer can set initial evidence to the model, referring to the current status of the application, and by performing Bayesian inference, he can update the beliefs of the network based on the new information provided. Figure 2 provides a simple example of this process. Figure 2A presents the initial BN produced by the distribution of values of the metrics observed for all the releases of an application. If at the particular point, the software engineer plans to reduce the number of functions so as to fall within the lowest class (s1), and he also estimates that the changes required for the next release will belong to the highest class (s5), he can set the relevant evidence to the network (see Figure 2B, Number of functions set to s1, and Changes set to s5) and test how the rest of the metrics will be affected. We see that in this example the *maintainability* is expected to improve (lowest class means better maintainability assessment), the *complexity* is expected to be reduced (by

[‡]Genie: https://www.bayesfusion.com/
Weka: https://www.cs.waikato.ac.nz/ml/weka/

TABLE 1 The steps for Bayesian network and association rules model extraction and evaluation

1. Development of the Bayesian network (step 2.1).
Input1: <internal quality metrics, external quality metrics, process metrics>
- *define the order of variables*
- *set the maximum number of parents for each node*

Output1 <1> <the BN model, the CPTs>
2. Evaluate the fitting accuracy of the BN model (step 2.2).
Input2: <BN model, CPTs>
- *create the ROC curves and the confusion matrix so as to*
  1. Find the classes of the dependent variables that are assigned incorrectly to another class.
  2. Find the classes that "dominate" other classes. In this case, the majority of cases are estimated to the "dominant" classes, causing the "omission" of other classes during the estimation.
- *examine the CPTs of the dependent variables*
  3. Isolate the rows that cannot actually provide an estimate, these are the rows that present equal probabilities between the different classes of the dependent variable.
  4. For these rows record the combination of the classes of the independent variables values that provide equal probabilities for estimating the dependent variable.

Ouput2 <2.1> <classes of the dependent variables that present low classification accuracy>
<2.2> <classes of the independent variables that cannot estimate the dependent ones>
3. Association rules mining (step 2.3).
Input3: <Output1> <Output2>
- *find all frequent and pertinent rules*
  5. Find the frequent rules, those presenting *support* values above a certain threshold.
  6. Find the pertinent rules, those presenting *confidence* values above a certain threshold.
- *filter the rules based on output 2*
  7. Select the rules that present as a *head* the classes of the dependent variables of output <2.1>.
  8. Select the rules that present as a *body* the classes of the independent variables of output <2.2>.

Output3: <3> <a set of association rules that complete the BN model>
4. Combing the Bayesian network and the association rules models (step 2.4).
Input4: <Output1>, <Output3>
- sort the estimations of the CPTs of the BN model based on the confidence level (CPT probabilities) of the estimate
- sort the estimations of the AR model based on the confidence values.
- estimate a new observation based on the estimations provided by the model that presents higher confidence value

Final Ouput: <the combined estimation model>

64% probability will fall in the lowest interval), and *cognitive complexity* presents increased probabilities in classes s3, s4, and s5. As for the time required for such changes, it is estimated within the two lowest classes (s1 interval is estimated by 25% probability and s2 interval by 19% probability). Therefore, this phase is a two-step procedure where the engineer *inserts evidence to the model (step 3.1)* and subsequently tests the impact of the new information be updating the model and *performing Bayesian inference (step 3.2)*.

5 | CASE STUDY DESIGN

In this section, we present the design of the case study performed to examine the effectiveness of the proposed approach in modeling the relationships among the various maintainability indicators (internal/external quality metrics and process metrics), as presented in Section 4, on (1) the actual maintainability rating of applications as assessed by SonarQube tool and (2) on the amount of changes and the time required for the subsequent, following-up release of an application. The case study was performed following the guidelines of Runeson et al[32]. In the following subsections, we present the research questions, the data collection process, and the data analysis method employed.

5.1 | Research questions

The main goal of this study is to create and evaluate a decision support tool, employing BN and ARs that will aid software practitioners to identify the most important maintainability indicators in the context of the application environment of interest helping them to monitor and plan maintenance activities. The purpose of this tool is twofold: (1) to assess the maintainability of an application based on internal and external quality metrics; (2) to predict the time intervals between releases and the number of changes required from one release to another. According to these goals, two research questions are formulated:

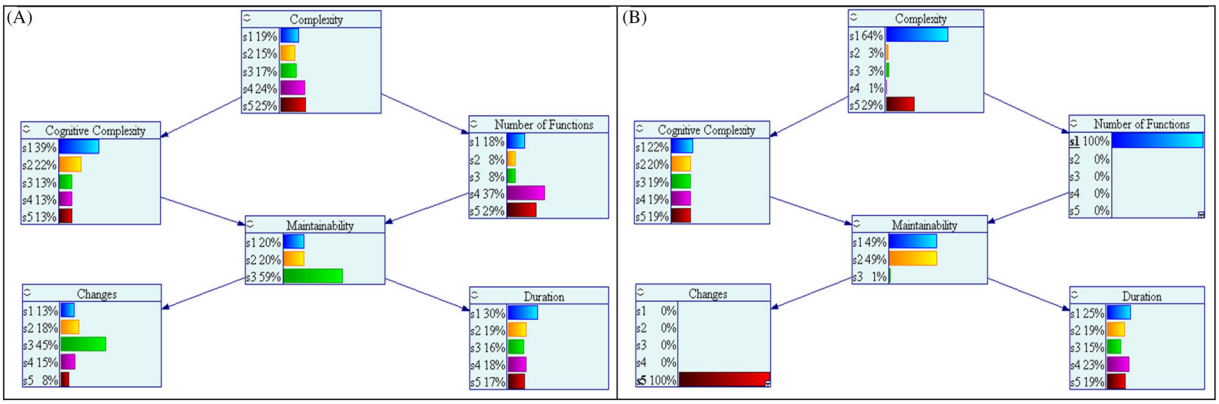


FIGURE 2 A, Inserting evidence to the model and B, performing Bayesian inference

RQ1: Is it possible to *assess the maintainability* of applications based on internal/external quality indicators and process metrics by adopting the *proposed approach*?

The aim of this question is to quantify the impact of internal and external quality indicators and process metrics on maintainability so as to be able to assess the maintainability rating of a release. Our target is to create a model that is able to estimate accurately the maintainability rating of a new release based on the quality and process metrics of the release. For this purpose, we identify and isolate the metrics that influence the most the maintainability of an application. Because maintainability is a synthesized attribute that depends on several internal source code factors, the activity of the developers, and on the end-user community activity, it is interesting to see which of the associated metrics may affect the most the maintainability of an application.

RQ2: Is it possible to *accurately estimate a) the time* required and *b) the level of maintenance changes* required for the next release, based on the internal/external quality indicators, the process metrics, and the maintainability of the current release of the application by adopting the *proposed approach*?

This question aims to identify the metrics that influence the most the level of changes performed between two successive releases and the time required for them. Estimating accurately the level of changes between releases based on various maintainability indicators is very important in scheduling time and effort intensive activities related to software maintenance.

5.2 | Case selection and data collection process

In order to evaluate the proposed approach, in terms of estimation accuracy, we analyzed release data coming from five popular JS applications hosted in Github[§] repository. The rationale behind the selection of these applications is summarized as follows:

- The applications are among the most popular ones (based on the number of forks) according to Github.
- More than 90% of the source code of the applications is written in JS (Many projects preceding in popularity were ignored due to the fact that their source code presented smaller percentages of JS code).
- All of the applications present more than one hundred releases. Therefore, each one of them can provide a maintenance data set large enough to be mined with the help of ML techniques.

Table 2 presents several descriptive statistics for the JS projects examined in this study.

5.3 | Evaluation process

To evaluate the proposed approach in terms of accuracy and representativeness, we split each of the five data sets, representing the five JS applications, into two sets the training set and the test set. The training set will be used to create the predictive models and the test set will be used to

[§]https://github.com/

TABLE 2 JavaScript projects examined in this study

			First Release		Last Release	
Program	**Progr. Languages**	**Releases**	**Date**	**Size (LoC)**	**Date**	**Size (LoC)**
JQuery	93.8% JS, 5.4% HTML, 0.8% other	146	07/2006	1278	03/2017	54 957
Ghost	91.5% JS, 4.7% HTML, 3.7% CSS, 0.1% other	116	06/2013	13 052	08/2017	55 922
Vue	97.7% JS, 2.3% other	207	08/2013	12 840	07/2017	89 095
Video.js	96.3% JS, 3.6% CSS, 0.1% HTML	327	10/2011	5791	09/2017	19 552
Material-UI	98.7% JS, 1.3% TypeScript	161	11/2014	2649	09/2017	34 480

evaluate the models. In all of the applications, we employed 70% of the early releases as a training set while the rest 30% of releases (the most recent ones) were used as a test set for evaluation purposes. To answer the first RQ, we will assess the value of *maintainability* metric, which will be the dependent variable. In the second RQ, the dependent variables are the *changes* and *duration*, that serve as indicators of the level of changes and the time required between successive releases, that will be the dependent variables. The evaluation process reporting of the empirical results will be performed, by comparing the estimations provided for the three variables (1) by the BN model and (2) the combined BN, AR model to the actual values of the variables. In an attempt to evaluate both regression and classification accuracy of the methods, we adopt two regression accuracy metrics and three classification accuracy metrics. For assessing the regression accuracy of the methods, we need to transform back into a "single point estimate" the value of the class estimated. For this purpose, we use the median point of the suggested interval that a class estimates that is considered as the "single point estimate."

The two metrics used to calculate the regression accuracy of the models are

- *Mean magnitude relative error (MMRE)* that shows the magnitude of the difference between the actual value and the estimate. The MMRE is defined as follows:

$$\mathrm{MMRE} = \frac{100}{n}\sum_{i=1}^{n}\frac{|P_i - E_i|}{P_i}$$

where N is the number of instances in the test set, P_i is the actual value of the dependent variable, and E_i is the 'single point estimate.'

- *Pred* (25) metric: The prediction within 25%, PRED (25), shows the percentage of cases that are estimated within 25% of the actual value.

In order to assess the classification accuracy of the proposed approach, we calculate the following metrics:

- *Accuracy* metric: Is the ratio between the correctly classified instances to the total number of instances used for evaluation purposes. It can be defined as follows:

$$\mathrm{Accuracy} = \frac{cc}{N}$$

where cc is the number of correctly classified instances and N is the number of instances in the test set.

- *Precision* metric: Is the ratio between the correct classifications performed for a particular class of the dependent variable to the total number of classifications performed pointing to this class. It can be defined based on the number of true positives and false positives as follows:

$$Precision = \frac{tp}{tp + fp}$$

where tp is the number of true positives and fp is the number of false positives for a particular class.

- *Recall* metric: Is the ratio between the correct classifications performed for a particular class of the dependent variable to the total number of observations belonging to this class. It can be defined based on the number of true positives and false positives as follows:

$$Recall = \frac{tp}{tp + fn}$$

where tp is the number of true positives and fn is the number of false negatives for a particular class.

6 | RESULTS

In this section, we present the results of the *case study* performed on five JS applications to evaluate the accuracy and efficiency of the proposed approach. At first, we exemplify the proposed approach, presented in Section 4, for assessing the maintainability of an application and monitoring the maintenance process (Section 6.1). In Section 6.2, we evaluate the maintenance models derived previously for the five applications and answer the two RQs.

6.1 | Creating the maintenance models

In the following subsections, we present an overview of the activities performed for modeling the maintainability and the maintenance indicators for JS applications by applying BNs inference and *rule* induction. In Section 6.1.1, we discuss the data collection process and the data preparation activities performed. In Section 6.1.2, we present the process of BN and AR model extraction and evaluation and the outputs of the two models.

6.1.1 | Data collection and preparation

During *the data collection (step 1.1)* phase, we downloaded the successive releases of the five applications from the date when the first release was launched and collected the metrics presented in Table 3. Table 3 presents the name of each metric, a short description of the metric, and the method/tool used to derive the metric.

We selected to employ these metrics for the following reasons:

a) Regarding the *internal source code metrics*, we selected the metrics referred to recent studies assessing software maintainability.[33,34] We employed metrics relevant to the *source code size* of an application, the *modularity* of the application, the redundancy (*duplications* in the code), and the *source code complexity* of the applications. Therefore, we selected the metrics referenced in these studies that belong to the aforementioned categories. Additionally, we considered two cumulative metrics, *code smells* and *internal bugs*, as high-level indicators of the weaknesses in design and the reliability of the application correspondingly. Metrics though related to the object-oriented nature of the language that are also studied in the context of software maintainability (like depth of inheritance tree, cohesion, etc.) were excluded. Because JS language until ECMAScript 6 supported an ad-hoc procedure for defining and handling objects, it was not possible to extract object-oriented metrics, for most of the versions analyzed. Though, we still managed to have in all the releases a measurement regarding the number of classes in each release which is included in the analysis.

b) Regarding the *external quality indicators*, the *activity metrics*, and the process metrics, we considered metrics that (1) take into account the open source software nature of the applications under study and (2) that can be accurately derived from Github repository from which the applications were retrieved. Therefore, we selected to include in the analysis the metrics like number of forks, number of developers, NoCom, etc, that were directly available from Github. We did not include the type of commit (bug fix, addition, deletion) because extracting such information would require the subjective judgment of the authors, especially when in most cases the commits are accompanied by single line comments. As a proxy of the maintenance activities, we considered the Number of Open/Closed Bugs reported in the Issue Tracker between releases can be considered. This metric can indirectly assess the number of fixes performed between successive releases.

We acknowledge though that the maintainability indicators should be customized when the proposed methodology is applied in the context of proprietary software maintenance activities.

The final set of metrics was collected with the following process:

- As a first step, we mined the webpage of Github projects to get general project information like *releases dates* and several *activity* metrics like *NoCom*. We also extracted the synthesized variable *duration* that was derived by calculating the number of days between the release dates of two sequential releases.
- Each release of the project was initially analyzed by SonarQube to get several source code *complexity metrics* and the maintainability assessment. The *maintainability* metric assessment is performed by SonarQube based on the technical debt ratio, that is the ratio between the effort to develop the application and the effort to fix all maintainability issues presented in the application. SonarQube has been employed in other studies as well within the context of software maintainability[1,35] and technical debt.[36] More details about the calculation of maintainability can be found in SonarQube's documentation.[¶] Then, each release of the project was analyzed by JSClassFinder[37] to get several source code *size and modularity metrics*.

[¶]https://docs.sonarqube.org/display/SONAR/Metric+Definitions#MetricDefinitions-Reliability
[4]https://git-scm.com/

TABLE 3 Maintainability factors and the associated metrics

Factor	Metric Name	Calculation Method/Values for Each Release	Tool
External quality indicators	#* of forks (forks)	A fork is a copy of the repository. In Github, it is a very common practice as it allows external developers to propose modifications to the master release of a project or use a project as for their new ideas.	D**
	# of bugs (NoB)	The number of bugs from Github's issues list (issue tracking).	D
Activity	#of developers (NoD)	The number of developers/contributors involved in a specific release.	D
	# of commits (NoCom)	The number of commits for every release.	D
	Comments rate	Density of comment lines in each release = *Comment lines/ (LoC + comment lines) * 100*	SQ
	Total lines of code	Total lines = LoC + comment lines	D
Source code size and modularity	Lines of code (LoC)	The number of physical lines that contain a character in each release (excluding a whitespace or a tabulation or part of a comment line).	SQ
	# of attributes (NoA)	The number of attributes for every release.	JS
	# of classes (NoC)	The number of classes in each release (including nested classes, interfaces, enums, and annotations).	SQ
	# of subclasses (NoS)	The number of subclasses of each release.	JS
	# of functions (NoF)	The number of functions in every release.	SQ
	# of files (NoFil)	The number of files of each release.	SQ
	# of statements (NoStat)	The number of statements in every release (ie, if, else, while, for)	SQ
Source code complexity and duplications	Cyclomatic complexity number (CCN)	The number of independent paths through the code for each release. This metric is an indicator of the testability of source code.	SQ
	Complexity	Complexity = CCN/LoC Complexity for each line of code in every release.	D
	Complexity/class	Complexity/class = CCN/NoC Average complexity by class in each release.	D
	Complexity/file	Complexity/file = CCN/number of files Average complexity by file for every release.	D
	Cognitive complexity	How hard it is to understand the code's control flow measured.	SQ
	Duplicated lines	The number of lines involved in duplications of each release.	SQ
	Duplicated blocks	The number of duplicated blocks of lines for every release. Ten successive and duplicated statements determine a duplicated block.	SQ
Maintainability	Source code bugs (SCB)	The number of bug issues found in the source code. The bugs can refer to handling of exceptions, security threats, or even unreachable code.	SQ
	Code smells	Total count of code smell issues.	SQ
	Maintainability	Is a rating that relates to the value of the technical debt ratio. Where technical debt ratio is measured as: Remediation cost/ (cost to develop 1 line of code* LoC)	SQ
Maintainability process indicators	Changes	The number of functions that have been added, removed, and modified in total for every release.	D
	Duration	For every release date, we subtract the previous release date, starting from the most recent release to the oldest. Consequently, the first release has zero duration.	D

*#: Number of.

**SQ: SonarQube, JS: JavaScriptClassFinder, D: Derived by the authors.

- As a last step, we calculated the value of the *changes* variable that represents the number of functions that have been added, changed, or removed in total between two successive releases. This value has been extracted by using the "git diff" command of git with a similar approach to Mens et al.[38]

During the *data preparation (step 1.2)*, we checked for the existence of outliers, by visually inspecting the box plots of the three dependent variables (*maintainability*, *changes*, and *duration*). We observe in Figure 3 that the *maintainability* attribute in four of the projects is well-balanced, without presenting significant outliers. On the other hand, *duration* and *changes* attributes present larger deviations in the values of projects Ghost and Material-UI projects. We further checked manually a sample of the outlier values and concluded that these values were "legitimate" and not caused by the data collection tools inaccuracy. Therefore, we decided that these values should remain in the analysis and transformed into another form (in our case the classes) as suggested by Osborne and Overbay.[39] Keeping the releases that present values that exceed the expected ones is very important as the target of the proposed approach is to estimate also the situations where "extreme" maintenance measures should be taken.

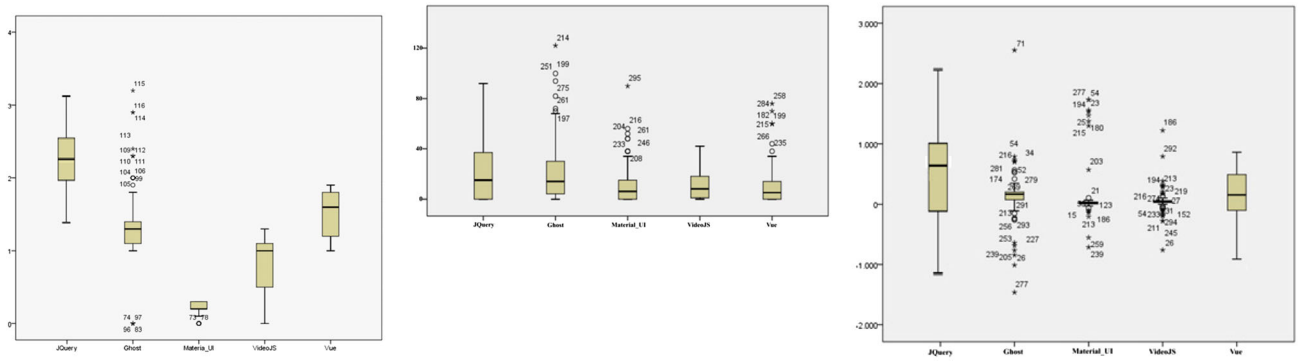


FIGURE 3 Boxplots for *maintainability*, *changes*, and *duration* for the five JS projects

We then proceeded with the discretization of the variables that presented continuous values. The discretization process was performed in our case as follows:

1) In the case of the independent variables with continuous values, we selected to perform equal frequency binning. The number of intervals was defined to be five approximating *very low values*, *low values*, *average values*, *high values*, and *very high values*. According to Peng et al,[31] equal frequency binning is a method that provides (1) clear insights of the distribution of the observations, (2) creates more stable classes compared with equal-width binning, and (3) can handle more effectively the outliers compared with equal-width binning. Equal-width binning in our case resulted in a single class concentrating the majority of observations, while the rest of the classes presented very few observations. Such a distribution of classes would provide no important information in the estimation models.
2) In the case of the dependent variables, we differentiated the discretization approach. As mentioned in Peng et al,[31] discretization according to intuition is more appropriate for simple and practically meaningful real data sets, especially in the case of the dependent variables, and it is less vulnerable to outliers. Therefore, for the dependent variables, we selected the interval classes as following: (1) Because the *maintainability* variable presented in most cases values concentrated within a certain range (see the boxplots of Figure 3), we selected three intervals as proxies of *high*, *average*, and *low* maintainability. The cutting-off points were decided based on the empirical distributions of the values considering the 25% and 75% percentile values. (2) For the *duration* and *changes* variables, which present larger deviations in their values, we selected five intervals so as to handle more effectively the outlier values. The cutting-off points for the *effort* variable were decided based on the empirical distributions of the values considering the 25% and 75% percentile values and the upper and lower bounds defined in the whiskers of the boxplots. The cutting-off points for the *duration* variable were decided based on the empirical distributions of the values considering the 25%, 50%, and 75% percentile values and the upper bounds defined in the whiskers of the boxplots because the lower whisker is almost the same as the lowest values of the observation belonging to the 1st percentile.

Table 4 presents the classes assigned to each variable and the correspondence of each class to continuous values for JS application.

6.1.2 | Bayesian network and association rules model extraction and evaluation

In this phase, as a first step of the analysis, we *derived the Bayesian networks (step 2.1)* based on the maintenance data collected for the five applications. The order of the variables was set according to the proposed approach (see Section 4.2) considering that for each new release, we are aware, in time order, of the following: The number of bugs identified in the previous release, the activity so far by the developers, and the NoCom, size metrics, complexity metrics, maintainability, and lastly maintainability indicators. The resulting networks for all five applications are presented in Figure 4. Further on in this section, we will exemplify the proposed approach on JQuery application.

In JQuery application, we see that among the most important maintenance drivers, we find *LoC* metric as a representative of the size metrics, influencing maintainability by 36%, and we also see the *NoCom* as a representative of the development team, influencing maintainability by 37%, and the *code smells* influencing maintainability by 15%. Additionally, we can observe that *maintainability* influences the *changes* by 29% and the *duration* by 29% which is also influenced by the *LoC* metric by 27%. Tables A1–A3 of the Appendix present indicatively the CPTs for the three dependent variables for JQuery application.

As a next step, we *evaluate the BN models (step 2.2)* in terms of fitting accuracy that is the ability of the model to accurately estimate the observations from which it has been trained. Table 5 presents values of the precision, the recall, and the accuracy metrics for the BN model of Figure 4.

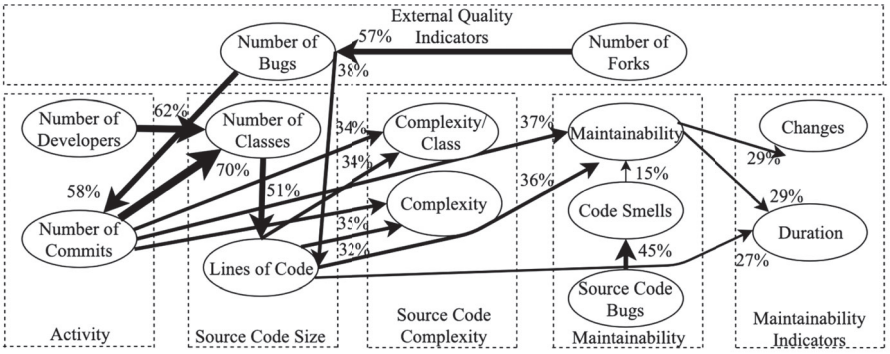
TABLE 4 Details about metrics' categorization for JQuery application

	S1	S2	S3	S4	S5
Number of forks	0>	>1258	>3524	>7640	10 124-13 250
Number of bugs	2>	>49	>67	>85	118-182
Number of developers	1>	>50	>100	>150	240-268
Number of commits	48>	>1098	>1679	>2860	4791-6156
Comments rate	7.8%>	>9.5%	>10.8%	>13.3%	14.4%-57.2%
Total lines	2037>	>29 430	>36 302	>43 934	64 983-78 846
Lines of code	1278>	>22 550	>27 440	>32 239	45 672-54 947
Number of attributes	141>	>169	>224	>265	331-358
Number of classes	44>	>56	>63	>68	72-81
Number of functions	177>	>2469	>3007	>3380	4595-5596
Number of files	9>	>69	>76	>83	159-203
Number of statements	882>	>13 937	>17 470	>20 125	25 959-30 483
Cyclomatic complexity	553>	>5972	>8601	>10 386	13 828-16 364
Complexity	0.259>	>0.291	>0.301	>0.317	0.346-0.405
Complexity/class	16.15>	>90.80	>131.21	>169.72	187.51-228.98
Complexity/file	34.6>	>74.7	>96.1	>111.7	140-177.5
Cognitive complexity	585>	>6953	>11 444	>13 661	21 509-26 615
Source code bugs	4>	>5	>7	>9	18-32
Duplicated lines	0>	>2070	>9780	>16 612	25 801-40 082
Duplicated blocks	0>	>70	>113	>209	423-536
Code smells	285>	>994	>1702	>2411	3119-3828
Maintainability	1.5>	>2.4	>2.8	-	-
Changes	-1280>	>0	>100	>200	400-2176
Duration	0>	>3	>10	>22	40-93

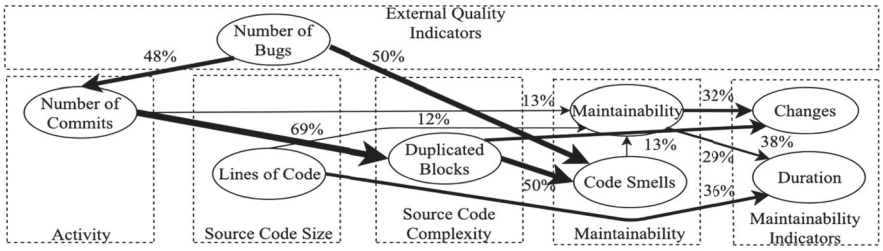
We observe that the fitting accuracy of the JQuery model for the *maintainability* indicator is very high as the estimation accuracy of the model is 93.1%. The *changes* indicator is estimated correctly in 39.2% of the cases, and the *duration* indicator presents accuracy 47%. Due to the low estimation percentages of the *duration* nodes and the *changes* nodes, we further investigate the confusion matrix of the two nodes along with the ROC curves. Figure 5 presents the confusion matrix for *duration* node and Figure 5 the ROC curve for the same indicator for the class s5 where many misclassifications are observed. The ROC curve originates from information theory and provides a way of expressing the quality of an estimation model. The x-axis of Figure 5A (specificity) refers to the false positive estimations, while the y-axis (sensitivity) refers to the true positive estimations.

Originally, the ROC curve is used for binary classifications, but in our case the ROC curve is built for each class of the target variable, considering the rest of the classes as one that represents the negative value. The dim diagonal line shows a baseline ROC curve of a hypothetical classifier that has no value as we expect that a strong classifier will present a ROC curve above this diagonal line. Above the curve, we see the area under the ROC curve (AUC) displayed. The ROC curve also can provide us information regarding the probability threshold that is used for classifying each point, the sensitivity of the estimation, and its specificity. For example, in the plot of Figure 5A, we highlighted one point in the curve in which the model provides a probability threshold value $P = 0.1$, sensitivity (true positive rate) = 0.4, a false positive rate 0.125, and a specificity (true negative rate) $1 - 0.125 = 0.875$. From the plot of Figure 5, we reach the conclusion that there is a high probability to misclassify a project to the fifth class of the *duration* variable as many points are concentrated to the false-positives high values. By inspecting the misclassification matrix, we focus on the classes of the dependent variable that are highly misclassified and try to identify rules related to these classes in the next step (step 2.3).

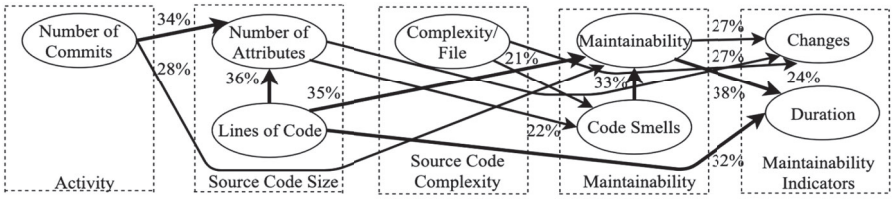
As a next step, we proceed with the *Association rules mining (step 2.3)*, so as to extract representative patterns for estimating the three maintenance indicators, focusing on the classes where the estimation accuracy of BNs is low. Therefore, we apply the a priori rule mining algorithm to identify relevant rules that will be able to identify patterns between *maintainability*, *changes*, and *duration* as target classes and the rest of the maintainability metrics. The derived rules for the three dependent variables for JQuery application are presented in Table 6. The derived rules for the rest of the four applications are presented in Table A4 of the Appendix.



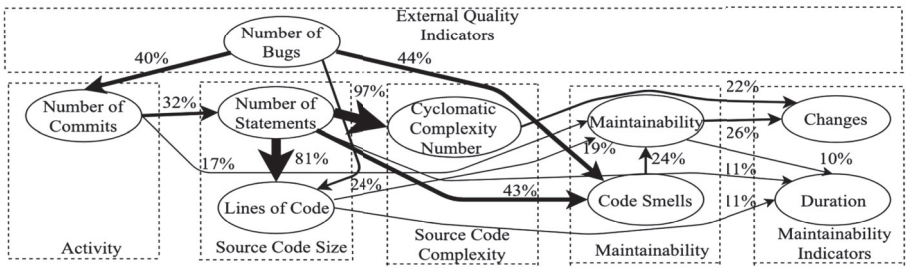
(A) The Bayes Network obtained for JQuery.



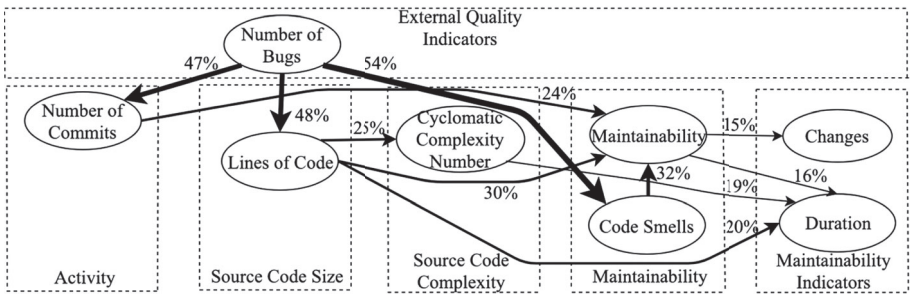
(B) The Bayes Network obtained for Ghost.



(C) The Bayes Network obtained for Vue.



(D) The Bayes Network obtained for Video.js.



(E) The Bayes Network obtained for Material-UI.

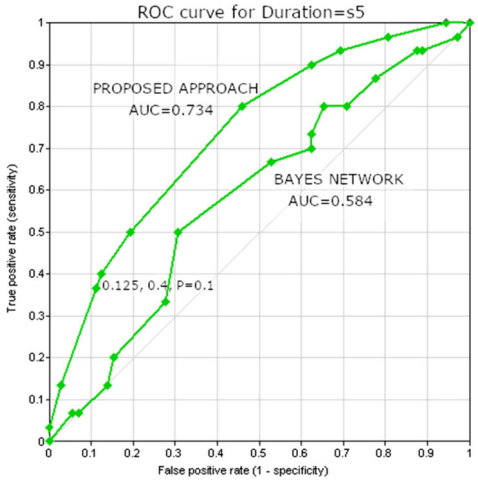
FIGURE 4 The Bayes networks obtained for the five JS applications. A, The Bayes network obtained for JQuery. B, The Bayes network obtained for ghost. C, The Bayes network obtained for Vue. D, The Bayes network obtained for Video.js. E, The Bayes network obtained for Material-UI

For example, the first rule classifies the maintainability of a project to the lowest interval when the values of *NoF* variable are at the second interval with a support value at 20.5% and a confidence value at 85.7%.

In Figure 5, we can see the improved ROC curve for the class s5 of variable duration and the misclassification matrix for the proposed approach that combines BN and AR models.

TABLE 5 *Precision*, *recall*, and *accuracy* of Bayes network for the training set of JQuery

	Precision	Recall	Accuracy
Maintainability	93.1	93.1	93.1
Changes	22.2	39.2	39.2
Duration	45.5	47	47



(A) ROC curve for *duration* metric and value s5 for the BN model and the proposed approach.

Bayesian Network					
	s1	s2	s3	s4	s5
s1	**3**	7	0	0	0
s2	1	**18**	1	1	0
s3	2	6	**2**	3	1
s4	1	7	4	**6**	9
s5	3	10	2	11	**4**
Proposed Approach					
	s1	s2	s3	s4	s5
s1	**6**	4	0	0	0
s2	2	**17**	0	2	0
s3	3	4	**1**	6	0
s4	2	4	0	**13**	8
s5	2	10	1	6	**11**

(B) Confusion matrix for variable *duration* for the BN model and the proposed approach.

FIGURE 5 ROC curves for duration metric (class s5) and misclassification matrix for JQuery application. A, ROC curve for *duration* metric and value s5 for the BN model and the proposed approach. B, Confusion matrix for variable *duration* for the BN model and the proposed approach

Estimation example

Let us assume that the current release presents the data of Table 7.

The rationale now behind making a new estimation for the *maintainability* metric is summarized as following:

1. Advice the CPT for making an initial estimation. According to the CPT (see the Appendix, Table A1), we need the values of the *LoC* variable, *NoCom* and *Code Smells*. In our case, LoC has the value s3 and *NoCom* the value s1 and *Code Smells* the value s3. According to the CPTs, all the classes of maintainability present equal probability, and therefore we cannot make a secure estimation in that case.
2. We continue by inspecting the rules of Table 6. The first rule that applies in our case is rule number 2, as the NoC variable has the value s4. We see that the rule also presents relatively high support and confidence values and therefore can be considered a representative rule for performing the estimate.

6.1.3 | Evidence-based estimation and inference

After creating the BN model of Figure 4, we are interested to test which actions can be performed during the maintenance process of an application that can help towards improving its maintainability. Therefore, in the BN model of Figure 4A, we set initial evidence relevant to the node of *maintainability* and assign its value to category s1 (which is a representative of increased maintainability). This assignment is instantiated as evidence in the initial BN model that is then updated through the inference mechanism.[4] Figure 6 presents the updated BN model that presents each node as a bar chart that shows the probability of the node to receive a particular value.

The assumptions we reach by observing the distribution of values of the nodes associated with the *maintainability* node are:

- Increased maintainability is associated with very low or low *code smells*, low or average *lines of code* (LoC), average or high *number of classes* (NoC) and very low or low *complexity*. This fact shows that caution should be taken for keeping code smells and source code complexity low. Also, developers should be careful when inserting new lines of code. A possible increase in *LoC* can be caused by the addition of new

TABLE 6 Association rules mined for JQuery application

A/A	Rule	Support	Confidence
Maintainability			
1	(NoF = s2) = > maintainability = s1	20.5%	85.7%
2	(NoC = s4) = > maintainability = s1	9.8%	70%
3	(NoC = s3) and (Total lines = s3) = > maintainability = s1	2.9%	100%
4	(NoCom = s5) = > maintainability = s3	29.4%	100%
5	(NoStat = s2) and (comments rate = s2) = > maintainability = s3	4.9%	80%
6	(code smells = s2) = > maintainability = s1	11.0%	100%
7	(code smells = s1) = > maintainability = s1	10.0%	100%
8	(code smells = s4) = > maintainability = s2	3.0%	66.6%
Duration			
1	(complexity/file = s5) and (cognitive complexity = s2) = > duration = s1	6.8%	71.4%
2	(NoFil = s2) and (duplicated blocks = s3) = > duration = s3	11.7%	58.3%
3	(NoF = s2) and (duplicated lines = s4) = > duration = s2	5.8%	83.3%
4	(NoA = s4) = > duration = s4	13.7%	64.2%
Changes			
1	(NoF = s5) = > changes = s4	12.7%	69.2%
2	(CCN = s2) = > changes = s5	21.%	59%
3	(cognitive complexity = s4) = > changes = s1	13.7%	71.4%
4	(cognitive complexity = s2) and (NoA = s1) = > changes = s1	3.9%	100%
5	(NoA = s4) = > changes = s1	7.8%	62.5%

TABLE 7 Characteristics of the project under estimation

Variable	Value	Variable	Value	Variable	Value
NoCom	s1	LoC	s3	Complexity	s1
NoB	s2	Comments rate	s4	CCN	s5
Maintainability	s1	Changes	s3	Complexity function	s3
NoF	s5	Duration	s2	Complexity/file	s2
NoA	s5	Duplicated files	s5	Complexity/class	s5
NoC	s4	Duplicated lines rate	s4	Cognitive complexity	s5
NoFil	s5	Duplicated blocks	s4	Function size	s4
NoS	s4	Duplicated lines	s5	Source code bugs	s2
Code smells	s3				

functionality; if that is the case, no precautions can be taken to avoid *LoC* increase. Although in the case of JQuery application, we observe that the increase in *LoC* was also caused by leaving lines of code that are no longer in use, in new releases. We noticed that there are large fluctuations in terms of lines of code between releases with certain portions of code being removed between them without adding new code. Moreover, we see that the modularization of code in classes, even in languages such as JS that is loosely object-oriented, helps towards creating applications that are easily maintainable. Also, we observe that low or average *LoC* is associated with low or very low *complexity*.

- Additionally, regarding the activity variables, we see that the number of *bugs* reported in the case of increased maintainability belong to the high interval (category s4, 41%), a fact that shows that releases with increased maintainability usually are preceded by extensive bug reporting, which in the case of JQuery is performed by the end-user community that are IT, professionals. Additionally, we see that in that case the *NoCom*, that is a representative metric of the number of interventions performed reside on the average category (category s3, 43%) which shows that it is not necessary to perform high *NoCom* to resolve issues.

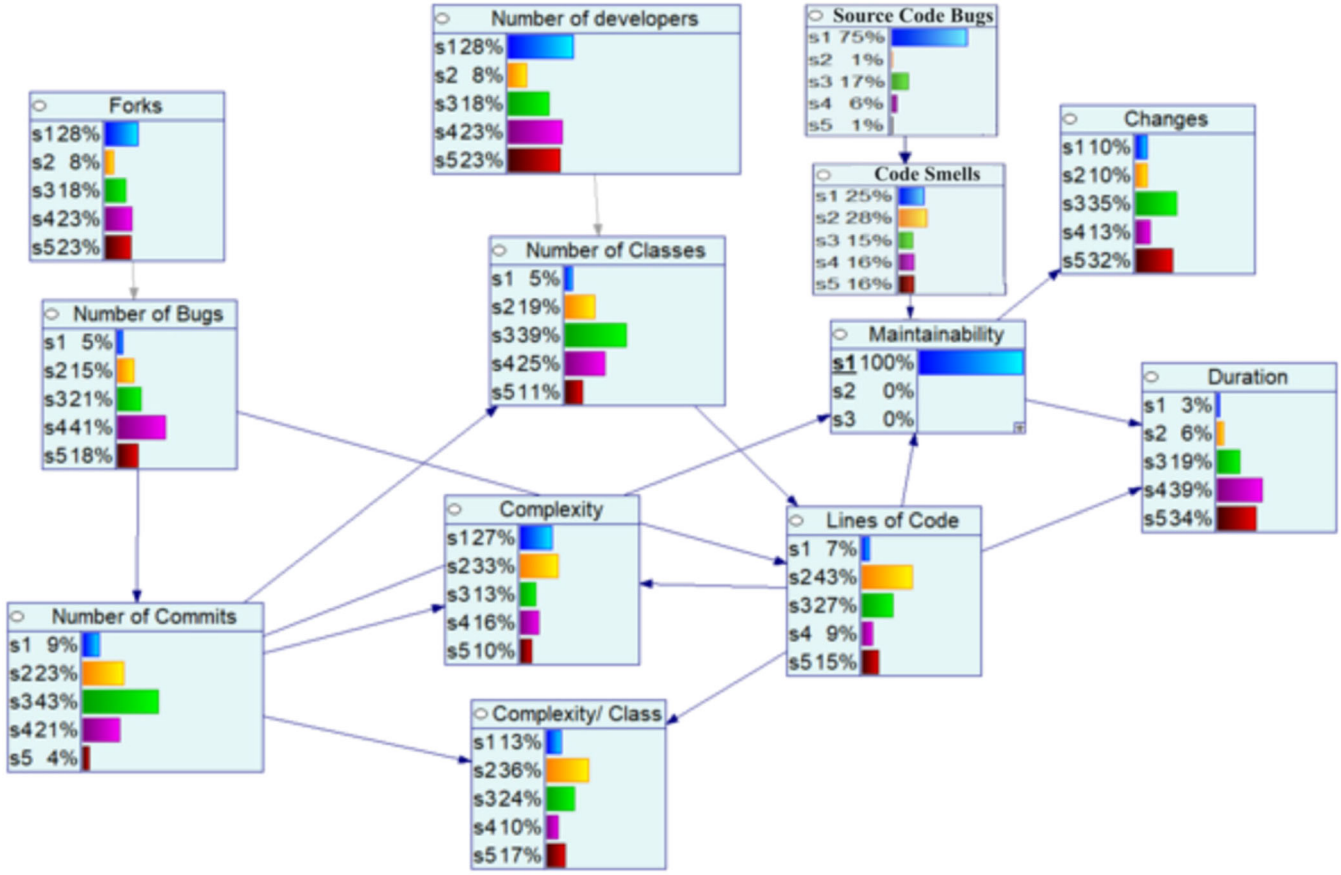


FIGURE 6 Updated BN model based on evidence inserted for the node *maintainability*

- Lastly, regarding the maintenance process indicators, we observe that high maintainability is associated with average (category s3, 35%) and very high (category s5, 32%) *changes* and long (s4, 39%) or very long (s5, 34%) *duration* between releases. The first inference relevant to the *changes* can be explained considering the fact that if a release has already increased *maintainability* (is already in category s1), then average changes (category s3) need to be performed; this is observed in successive releases that present increased maintainability. On the other hand, there are also many cases where a release presents low maintainability, and it seems that effort has been made to improve the maintainability of the subsequent release. In those cases, the *changes* performed belong to the highest category.

6.2 | Evaluation results

In this subsection, we evaluate the maintenance models derived previously and answer the two research questions presented in Section 5.1 by assessing the predictive accuracy of the proposed combined approach and compare it against the accuracy of BN alone in the test sets of the five applications participating in this case study. As mentioned, the test sets consist of 30% of the releases of each application, the most recent ones. Table 8 presents the predictive accuracy of BN and compares it against the proposed approach. Moreover, in Table 8, we can observe the MMRE and PRED(.25) values of BN which are compared with the proposed approach.

We can see that the proposed approach achieves an improvement in accuracy for all the three dependent variables. Maintainability value presents a limited improvement due to the already high estimation accuracy of BN. The high estimation accuracy for this metric is related to the fact that maintainability values remain, in all five projects, within certain intervals through the successive releases without presenting extreme values in the majority of cases. Also, it seems that the metrics selected for assessing maintainability are good indicators of its value. The maintainability value for all the five applications considered in this case study depends highly on the *code smells* presented in the source code, the *NoCom*, and the *LoC* of each release. On the other hand, the value of code smells is affected by different metrics based on the nature of the project. Other important metrics identified by ARs are the *NoF* (number of functions), *NoC* (number of classes), *comments rate*, and the *NoStat (number of statements)*. Hence, regarding *RQ1*, we can say that the maintainability of an application can be estimated with high confidence based on the proposed approach.

TABLE 8 *Classification and regression accuracy metrics* of the proposed approach compared against BN models

		Bayes Networks				Proposed Approach					
Program		Prec.	Rec.	Acc.	MMRE	Pred (25)	Prec.	Rec.	Acc.	MMRE	Pred (25)
	Maintainability	95.8	95.4	95.4	21.23	100	97.8	97.7	97.2	7.23	100
JQuery	Changes	54.7	54.5	54.5	76.08	36.36	67.6	63.6	63.6	37.16	88.63
	Duration	51.7	65.3	65.3	41.36	22.72	58.7	70.5	70.4	20.90	52.72
	Maintainability	80.0	80.0	80.0	18.52	100	97.7	97.5	97.5	6.95	100
Ghost	Changes	54.5	63.6	63.6	153.2	52	66.3	64.2	64.2	74.8	63.5
	Duration	36.3	45.7	45.7	156.5	48.57	64.6	63.0	63.0	80.8	61.42
	Maintainability	96.6	96.6	96.5	10.1	98.38	98.4	98.3	98.3	2.64	98.38
Vue	Changes	72.5	67.7	67.7	27.47	43.54	82.4	82.3	82.2	14.03	79.90
	Duration	63.4	59.6	59.6	79.48	50	73.6	72.6	72.5	39.76	75.80
	Maintainability	93.7	93.8	93.8	10.40	100	98.1	98.0	97.9	7.40	100
Video.js	Changes	39.9	42.8	42.8	85.11	29.59	59.7	53.1	53.0	20.62	68.36
	Duration	45.5	54.0	54.0	40.58	58.16	60.7	61.2	61.2	13.37	81.63
	Maintainability	97	97	97.2	9.41	100	100	100	100	3.41	100
MaterialUI	Changes	26.2	41.6	41.6	105.51	29.16	65.2	60.4	60.4	51.64	72.08
	Duration	56.7	64.5	64.5	57.57	60.41	68.2	66.7	66.7	39.91	66.66

The *changes* variable can be accurately estimated in most of the cases, with the proposed approach achieving an improvement of 21.6% in total for the five applications in the *accuracy* metric compared with the BN model. Apart from the *maintainability* value that affects the *changes*, other important metrics appointed by AR model are *NoA* (number of attributes), *NoF* (number of functions), and the *complexity*.

In the case of *duration*, the proposed approach achieves a smaller improvement of 16.8% compared with the BN model accuracy. Additionally, we can say that estimating the duration required for successive releases of open source JS applications is a challenging task that in some cases like the Ghost application is not easy to achieve high estimation accuracy. Probably, this can be explained by the fact that duration in OSS is not an actual measure of the time required for the changes but just a calendar time between successive releases that does not exactly reflect the reality. Apart from the *maintainability* and the *LoC* metrics that affect the *duration*, other important metrics appointed by the AR model are the *duplicated blocks*, *NoFil* (Number of Files), *NoA* (Number of Attributes), *NoF* (Number of Functions), *cognitive complexity*, and *complexity/file*. Hence, regarding *RQ2*, we can say that the maintenance process indicators of an application, like *changes* and *duration*, can be estimated with high confidence based on the proposed approach.

We should mention that the regression metrics presented in Table 8 are also encouraging showing the ability of the proposed approach to provide both class-estimated and "single point estimates."

The findings of this study were further discussed with eight OSS contributors, which at the moment are postgraduate, MSc, or PhD students, possessing more than 5-year experience in participating in OSS projects. In terms of monitoring the maintainability, it seems that contributors are mostly interested in avoiding "breaking points," which means they want to estimate the phase during which (1) the maintainability of an application exceeds normal values and the source code needs immediate refactoring so as to allow further changes; (2) a software application requires large amount of changes so as to meet user needs. The majority of the participants stated that three classes for *maintainability* assessment are enough and provide highly relevant information on whether the maintainability is at low, normal, or high levels. Some contributors stated that even a rough estimation of "pass" and "fail" in terms of maintainability would be appropriate for projects that present small deviations in the maintainability rating. Regarding the *duration* metric, they intuitively confirmed the results of Figure 4 that appoints a relationship between NoCom - > maintainability - > duration. They all agreed that the duration metric is highly dependent on the activity of the community because an active community will directly correct a reported bug a fact that leads to short release cycles. Regarding the *changes* metric, the contributors stated that their main interest is to be able to foresee the "extreme" situations that require large amount of changes. Therefore, they thought that reserving in the estimation model a class with "extreme values" of changes is highly appropriate.

7 | DISCUSSION

In this section, we summarize some interesting implications for researchers and practitioners and discuss the threats to validity identified for this study.

7.1 | Implications to researchers and practitioners

The major findings of this study show that the proposed approach that combines two popular ML techniques, BN as a predictive model and AR as a descriptive model, can be very effective for modeling the maintenance process and needs to be further explored.

We believe that ***researchers*** need to concentrate on developing tools that will automate the learning process of the combined BN and AR models. These tools should be able to easily incorporate (1) expert knowledge and (2) knowledge coming from new releases without the need to rebuild the whole models. The existence of flexible decision support tools that focus on modeling the maintenance process of software applications will help towards improving the efficiency of maintenance activities. Additionally, researchers are encouraged to evaluate the proposed approach on maintenance activities performed within company software. From such a study, it would be interesting to observe differences in activity variables (development team size, team expertise, realistic time and effort recorded, etc.) that participate in the calculation of maintainability and also see the level to which expert knowledge regarding the assessment of the maintainability factor (instead of the assessment performed by third-party tools) can affect the estimation of the maintenance changes and duration variables. In such case, the first phase of the proposed approach can be further updated with new metrics and procedures to calculate them.

Regarding *practitioners*, we encourage them to adopt the proposed approach, in monitoring the maintenance process by (1) estimating the *maintainability* of the application along with the *changes* and the *duration* required when performing maintenance activities and (2) performing evidence-based inference to test the impact of changes on certain maintainability indicators. Additionally, we encourage practitioners to trust the tools employed in this study for the metrics calculation and the creation of the BN and AR models. The fact that (1) the majority of the metrics that are used for quantifying the maintenance variables can be automatically calculated from freely available tools and (2) the BN and AR models can be easily derived by existing ML suites is expected to boost the adoption of the proposed approach, and its practical benefits. Finally, we believe that the fact that the proposed approach is based on classification models (ie, models that predict a particular class, not a finite value) and can provide a more accurate, coarse-grained approach within the context of software maintenance prediction, can be valuable to software engineers that wish to adopt a safer estimation process. Still, the class estimates can be transformed to "single point estimates" in the case where a particular value is required.

7.2 | Threats to validity

In this section, we discuss the threats to validity that we have identified for this study. Regarding *conclusion validity* that refers to how reasonable are the findings of the analysis, we mention that regarding the statistical power of the results, we calculated a variety of regression and classification accuracy metrics (five in total) to validate the proposed approach, showing an improvement in the accuracy of the proposed approach compared with the Bayesian analysis. Also, regarding the error rate problem, we selected a common, predefined validation procedure for the five data sets that utilized 70% of the observations as a training set and the last, most recent ones, 30% of the observations as a test set so as to avoid bias on the sets selected. In addition, regarding the heterogeneity of data, we used project-specific data sets to ensure the relativity of the data included in the analysis. The findings of this study were also discussed with eight OSS contributors that intuitively confirmed the output of the analysis (Section 6.2). Regarding *construct validity*, we should mention that the set of metrics used to assess the maintainability of applications may affect the findings and the accuracy of the proposed approach. Our rationale behind selecting these metrics was based on the findings of current literature[33] and the special characteristics of JS programming language. We emphasized on the size and complexity source code metrics as performed by Kyriakakis et al[34] who identified maintenance patterns in PHP applications, omitting though several object-oriented metrics, as explained in Section 6.1. Therefore, we acknowledge the fact that important object-oriented metrics were excluded, but we plan to experiment in other more recent JS applications that fully adopt object-oriented attributes of the language as future work. Because the scope of the study is to demonstrate the proposed explorative approach in terms of efficiency and predictive accuracy, we believe that the exclusion of several object-oriented metrics did not affect the overall performance of the method. Additionally, we acknowledge that information regarding the type of the maintenance activities was not included in the analysis, because such information on OSS application development would demand at a great point the subjective assessment of the value of this metric by the authors. We believe that such information that may be available in the case of in-house development would help towards the improvement of the accuracy of the proposed approach and should be included in such an analysis in the future. Regarding *internal validity*, an attempt by this study to associate internal and external quality factors and process characteristics to software maintainability over time is presented. The causal relationships identified in this study and presented in Figure 4 are indicators of possible cause-effect relationships among the participating variables without though excluding other relationships between variables that may affect the maintainability of applications that did not participate in this study (ie, development team experience). With respect to *reliability*, we believe that the replication of our research is safe and the overall reliability is ensured. The process that has been followed in this study has been thoroughly documented in Section 6.1.1, so as to be easily reproduced by any interested researcher. The structural metrics calculation and the overall extraction of the defined data set were performed with the use of two widely used research tool (SonarQube and JSClassFinder). Concerning *the external validity* and in particular the generalizability supposition, changes in the findings might

occur if the applications for which the sample releases are analyzed is altered. A future replication of this study, on maintenance data from other projects, would be valuable to verify these findings.

8 | CONCLUSION

Software maintenance is one of the most demanding activities during the software lifecycle as it depends on a variety of software artifacts that are interrelated and the environment in which the application operates. In this study, we presented and validated an approach to model activity, process and product metrics related to both the maintainability of an application and the maintenance process with the help of two ML methods; BN and ARs. We exploited the predictive power of BNs and their ability to perform inference so as to estimate the maintainability of an application and the changes and the duration required for performing maintenance activities. Association rules were adopted in order to identify frequently appearing patterns in the cases where BNs were not able to perform precise estimates. To investigate the validity of the proposed approach, we performed a case study on five JS applications analyzing in total 957 releases of JS applications, testing the estimation accuracy of the derived models on 287 releases. The results from the case study suggested that the proposed approach is capable of providing accurate maintainability and maintenance process assessments. The combined approach outperforms the BN method alone in terms of estimation accuracy. Based on these results, implications for researchers and practitioners have been provided.

ORCID

Angelos Chatzimparmpas https://orcid.org/0000-0002-9079-2376
Stamatia Bibi https://orcid.org/0000-0003-4248-3752

REFERENCES

1. Amanatidis T, Chatzigeorgiou A, Ampatzoglou A. The relation between technical debt and corrective maintenance in PHP web applications. *Inf Softw Technol*. 2017;90:70-74. https://doi.org/10.1016/j.infsof.2017.05.004
2. Van Vliet H. *Software Engineering: Principles and Practice*. Hoboken, NJ: John Wiley & Sons; 2008 ISBN: 0470031468.
3. IEEE Standard for Software Maintenance. in IEEE Std 1219-1998, vol., no., pp.i-, 1998. https://doi.org/10.1109/IEEESTD.1998.88278
4. Nielsen TD, Jensen FV. *Bayesian Networks and Decision Graphs*. New York: Springer Science & Business Media; 2009.
5. Hand DJ, Mannila H, Smyth P. *Principles of Data Mining (Adaptive Computation and Machine Learning)*. Cambridge, MA: MIT Press; 2001 ISBN: 0-262-08290-X 9780262082907.
6. Settas D, Bibi S, Sfetsos P, Stamelos I, Gerogiannis V. Using Bayesian belief networks to model software project management antipatterns. in Software Engineering Research, Management and Applications, 2006. Fourth International Conference on, pp. 117-124. IEEE, 2006. https://doi.org/10.1109/SERA.2006.68
7. Bibi S, Stamelos I, Angelis L. Combining probabilistic models for explanatory productivity estimation. *Inf Softw Technol*. 2008;50(7-8):656-669. https://doi.org/10.1016/j.infsof.2007.06.004
8. Niessink F, Van Vliet H. Predicting maintenance effort with function points. In Software Maintenance, 1997. Proceedings., International Conference on, pp. 32-39. IEEE, 1997. https://doi.org/10.1109/ICSM.1997.624228
9. Dagpinar M, Jahnke JH. Predicting maintainability with object-oriented metrics—an empirical comparison. in null, p. 155. IEEE, 2003. https://doi.org/10.1109/WCRE.2003.1287246
10. Misra SC. Modeling design/coding factors that drive maintainability of software systems. *Softw Qual J*. 2005;13(3):297-320. https://doi.org/10.1007/s11219-005-1754-7
11. Fioravanti F, Nesi P. Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Trans Softw Eng*. 2001;27(12):1062-1084. https://doi.org/10.1109/32.988708
12. Aggarwal KK, Singh Y, Chandra P, Puri M. Measurement of software maintainability using a fuzzy model. *J Comput Sci*. 2005;1(4):538-542.
13. Aggarwal KK, Singh Y, Kaur A, Malhotra R. Application of artificial neural network for predicting maintainability using object-oriented metrics. *Trans Eng Comput Technol*. 2006;15:285-289.
14. Zhou Y, Leung H. Predicting object-oriented software maintainability using multivariate adaptive regression splines. *J Syst Softw*. 2007;80(8):1349-1361. https://doi.org/10.1016/j.jss.2006.10.049
15. Kaur A, Kaur K, Malhotra R. Soft computing approaches for prediction of software maintenance effort. *Int J Comput Appl*. 2010;1(16):80-86. https://doi.org/10.5120/339-515
16. Jin C, Liu J-A. Applications of support vector mathine and unsupervised learning for predicting maintainability using object-oriented metrics. in Multimedia and Information Technology (MMIT), 2010 Second International Conference on, vol. 1, pp. 24-27. IEEE, 2010. https://doi.org/10.1109/MMIT.2010.10
17. Elish MO, Elish KO. Application of treenet in predicting object-oriented software maintainability: a comparative study. in Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on, pp. 69-78. IEEE, 2009. https://doi.org/10.1109/CSMR.2009.57
18. Fontana FA, Mäntylä MV, Zanoni M, Marino A. Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng*. 2016;21(3):1143-1191.

19. Idri A, Khoshgoftaar TM, Abran A. Can neural networks be easily interpreted in software cost estimation?" in Fuzzy Systems, 2002. FUZZ-IEEE'02. Proceedings of the 2002 IEEE International Conference on, vol. 2, pp. 1162-1167. IEEE, 2002. https://doi.org/10.1109/FUZZ.2002.1006668
20. Okutan A, Yıldız OT. Software defect prediction using Bayesian networks. *Empir Softw Eng*. 2014;19(1):154-181. https://doi.org/10.1007/s10664-012-9218-8
21. Fenton N, Neil M, Marsh W, et al. Predicting software defects in varying development lifecycles using Bayesian nets. *Inf Softw Technol*. 2007;49(1):32-43. https://doi.org/10.1016/j.infsof.2006.09.001
22. Wagner S. A Bayesian network approach to assess and predict software quality using activity-based quality models. *Inf Softw Technol*. 2010;52(11):1230-1241. https://doi.org/10.1016/j.infsof.2010.03.016
23. Bibi S, Ampatzoglou A, Stamelos I. A Bayesian belief network for modeling open source software maintenance productivity. In IFIP International Conference on Open Source Systems, pp. 32-44. Springer, Cham, 2016. https://doi.org/10.1007/978-3-319-39225-7_3
24. Catolino G, Palomba F, De Lucia A, Ferrucci F, Zaidman A. Enhancing change prediction models using developer-related factors. *J Syst Softw*. 2018;143:14-28.
25. Druzdzel MJ. SMILE: Structural Modeling, Inference, and Learning Engine and GeNIe: a development environment for graphical decision-theoretic models. In Aaai/Iaai, pp. 902-903.1999. ISBN: 0-262-51106-1.
26. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The WEKA data mining software: an update. *ACM SIGKDD Explor Newslett*. 2009;11(1):10-18.
27. McCabe TJ. A complexity measure. *IEEE Trans Softw Eng*. 1976;SE-2(4):308-320. https://doi.org/10.1109/TSE.1976.233837
28. Bansiya J, Davis CG. A hierarchical model for object-oriented design quality assessment. *IEEE Trans Softw Eng*. 2002;28(1):4-17. https://doi.org/10.1109/32.979986
29. Sturge H. The choice of class interval. *J Am Stat Assoc*. 1926;21(153):65-66.
30. Torgo L, Gama J. Regression using classification algorithms. *Intell Data Anal*. 1997;1(4):275-292.
31. Peng L, Qing W, Yujia G. . Study on comparison of discretization methods. In Artificial Intelligence and Computational Intelligence, 2009. AICI'09. International Conference on (Vol. 4, pp. 380-384). IEEE.
32. Runeson P, Host M, Rainer A, Regnell B. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons; 2012 https://doi.org/10.1002/9781118181034
33. Baggen R, Correia JP, Schill K, Visser J. Standardized code quality benchmarking for improving software maintainability. *Softw Qual J*. 2012;20(2):287-307.
34. Kyriakakis P, Chatzigeorgiou A. Maintenance patterns of large-scale PHP web applications. in Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, pp. 381-390. IEEE, 2014. https://doi.org/10.1109/ICSME.2014.60
35. Wolski M, Walter B, Kupiński S, Chojnacki J. Software quality model for a research-driven organization—an experience report. *J Softw Evol Process*. 2018;30(5).
36. Digkas G, Lungu M, Avgeriou P, Chatzigeorgiou A, Ampatzoglou A. How do developers fix issues and pay back technical debt in the Apache ecosystem? in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 153-163. IEEE, 2018. https://doi.org/10.1109/SANER.2018.8330205
37. Silva LH, Hovadick D, Valente MT, Bergel A, Anquetil N, Etien A. JSClassFinder: a tool to detect class-like structures in JavaScript." arXiv preprint arXiv:1602.05891 2016.
38. Mens T, Fernández-Ramil J, Sylvain D. The evolution of eclipse. In Software Maintenance, 2008. ICSM 2008. IEEE International Conference on, pp. 386-395. IEEE, 2008. https://doi.org/10.1109/ICSM.2008.4658087
39. Osborne JW, Overbay A. The power of outliers (and why researchers should always check for them). *Pract Assess Res Eval*. 2004;9(6):1-12.

How to cite this article: Chatzimparmpas A, Bibi S. Maintenance process modeling and dynamic estimations based on Bayesian networks and association rules. *J Softw Evol Proc*. 2019;31:e2163. https://doi.org/10.1002/smr.2163

APPENDIX A

TABLE A1 CPT for *maintainability* metric

			Maintainability		
Number of Commits	Code Smells	LoC	s1	s2	s3
s1	s1	s3	0.9	0.05	0.05
	s2	s4	0.93	0.04	0.04
	s3	s4	0.21	0.75	0.04
		s5	0.26	0.69	0.05
	s4	s4	0.15	0.7	0.15
		s5	0.49	0.49	0.02
s2	s1	s2	0.94	0.03	0.03
		s3	0.87	0.06	0.06
	s2	s3	0.81	0.09	0.09
	s3	s5	0.12	0.86	0.02
	s4	s4	0.15	0.7	0.15
		s5	0.02	0.95	0.02
s3	s1	s2	0.87	0.06	0.06
		s3	0.7	0.15	0.15
	s2	s2	0.94	0.03	0.03
		s3	0.97	0.01	0.01
	s3	s4	0.02	0.96	0.02
s4	s2	s2	0.33	0.6	0.06
		s3	0.09	0.81	0.09
	s5	s1	0.7	0.15	0.15
		s2	0.26	0.26	0.49
		s3	0.02	0.95	0.02
		s4	0.26	0.72	0.02
s5	s1	s1	0.01	0.01	0.98
	s3	s1	0.09	0.09	0.81
	s4	s1	0.02	0.02	0.97
	s5	s1	0.05	0.05	0.9
		s2	0.09	0.09	0.81

TABLE A2 CPT for *duration* metric

		Duration				
Maintainability	LoC	s1	s2	s3	s4	s5
s1	s1	0.10	0.10	0.10	0.10	0.60
	s2	0.01	0.08	0.33	0.39	0.20
	s3	0.02	0.02	0.02	0.58	0.36
	s4	0.07	0.07	0.40	0.07	0.40
	s5	0.03	0.03	0.03	0.37	0.53

(Continues)

TABLE A2 (Continued)

		Duration				
Maintainability	LoC	s1	s2	s3	s4	s5
s2	s1	0.20	0.20	0.20	0.20	0.20
	s2	0.04	0.24	0.24	0.44	0.04
	s3	0.32	0.22	0.22	0.12	0.12
	s4	0.18	0.52	0.18	0.02	0.10
	s5	0.01	0.01	0.01	0.44	0.51
s3	s1	0.08	0.39	0.08	0.14	0.32
	s2	0.46	0.03	0.17	0.17	0.17

TABLE A3 CPT for *changes* metric

	Changes				
Maintainability	s1	s2	s3	s4	s5
s1	0.10	0.10	0.35	0.13	0.32
s2	0.37	0.08	0.19	0.24	0.11
s3	0.23	0.29	0.43	0.03	0.01

TABLE A4 Association rules for Ghost, Vue, Video.js, and Material-UI applications

Ghost			
Maintainability			
1	(SCB = s5) and (duplicated blocks = s2) = > maintainability = s1	22.0%	100%
2	(code smells = s5) = > maintainability = s3	24.0%	100%
3	(code smells = s4) = > maintainability = s2	21.0%	100%
4	(code smells = s3) = > maintainability = s3	8.0%	100%
Duration			
1	(comments rate = s3) = > duration = s3	5.0%	60%
2	(comments rate = s2) and (NoStat = s4) and NoC = > duration = s2	5.0%	50%
3	(comments rate = s1) and (NoStat = s5) = > duration = s2	5.0%	100%
4	(comments rate = s1) and (NoCom = s2) and (NoFil = s4) = > duration = s4	5.0%	66.6%
5	(comments rate = s1) = > duration = s3	5.0%	75%
6	(NoA = s2) and (complexity/class = s4) = > duration = s5	5.0%	60%
7	(NoA = s2) = > duration = s5	10.0%	50%
8	(code smells = s5) = > duration = s1	5.0%	50%
9	(NoStat = s1) = > duration = s3	5.0%	75%
Changes			
1	(NoCom = s3) = > changes = s1	7.0%	71.5%
2	(duplicated blocks = s4) and (code smells = s4) and (complexity/class = s1) = > changes = s5	6.0%	83.4%
3	(NoFil = s4) = > changes = s2	2.0%	100%
4	(NoCom = s5) and (maintainability = s3) and (NoA = s2) = > changes = s1	5.0%	100%
5	(NoCom = s5) and (maintainability = s1) = > changes = s5	5.0%	66.6%
6	(NoStat = s1) = > changes = s3	7.0%	71.5%

(Continues)

TABLE A4 (Continued)

Ghost			
7	(NoStat = s5) and (code smells = s5) = > changes = s3	5.0%	50%
8	(NoFil = s5) = > changes = s3	5.0%	50%
Vue			
Maintainability			
1	(complexity = s1) = > maintainability = s1	31.0%	100%
2	(complexity = s5) = > maintainability = s3	28.0%	96.4%
3	(complexity = s2) = > maintainability = s1	20.0%	100%
4	(duplicated lines = s1) and (NoStat = s1) and (duplicated blocks = s1) = > maintainability = s2	30.0%	93.3%
5	(SCB = s4) = > maintainability = s1	16.0%	100%
6	(SCB = s4) and (code smells = s4) = > maintainability = s2	15.0%	100%
Duration			
1	(code smells = s4) and (duplicated blocks = s4) = > duration = s4	10.0%	50%
2	(code smells = s5) and (complexity = s4) = > duration = s5	11.0%	45.4%
3	(code smells = s4) = > duration = s5	7.0%	71.4%
4	(complexity/class = s2) = > duration = s4	5.0%	100%
Changes			
1	(comments rate = s3) = > changes = s1	14.0%	100%
2	(NoCom = s4) = > changes = s5	14.0%	100%
3	(code smells = s5) and (complexity = s5) = > changes = s2	5.0%	100%
4	(maintainability = s2) and (NoFil = s4) and (code smells = s3) = > changes = s4	9.0%	66.6%
5	(NoC = s5) and (maintainability = s3) = > changes = s1	5.0%	75.0%
6	(NoFil = s5) and (maintainability = s2) and (NoF = s4) = > changes = s3	7.0%	57.1%
7	(code smells = s4) = > changes = s5	6.0%	66.6%
Video.js			
Maintainability			
1	(code smells = s2) and (SCB = s3) = > maintainability = s2	43.0%	100%
2	(NoCom = s3) and (code smells = s3) = > maintainability = s3	23.0%	100%
3	(code smells = s2) = > maintainability = s1	13.0%	84.6%
4	(code smells = s1) = > maintainability = s1	10.0%	100%
Duration			
1	(SCB = s5) = > duration = s4	5.0%	100%
2	(NoC = s3) and (complexity/class = s3) = > duration = s2	11.0%	90.9%
3	(NoC = s3) and (NoStat = s4) = > duration = s2	21.0%	57.1%
4	(NoStat = s3) and (Total lines = s3) = > duration = s2	12.0%	83.3%
5	(NoC = s3) = > duration = s3	14.0%	64.2%
6	(SCB = s3) = > duration = s2	5.0%	60%
Changes			
1	(SCB = s5) = > changes = s1	9.0%	66.6%
2	(code smells = s3) and (NoCom = s4) and (NoFil = s4) = > changes = s1	7.0%	50%
3	(code smells = s3) and (duplicated blocks = s4) and (total lines = s5) = > changes = s4	13.0%	38.4%
4	(code smells = s3) and (NoFil = s4) = > changes = s4	9.0%	66.6%
5	(code smells = s1) = > changes = s5	7.0%	85.7%
6	(NoFil = s4) and (complexity/class = s5) = > changes = s3	5.0%	60%
7	(SCB = s3) and (duplicated blocks = s2) = > changes = s4	5.0%	50%

(Continues)

TABLE A4 (Continued)

Ghost			
8	(SCB = s1) = > changes = s5	5.0%	80%
9	(complexity/class = s3) and (comments rate = s3) = > changes = s4	5.0%	100%
10	(complexity/class = s4) = > changes = s2	22.0%	50%
11	(complexity/class = s2) = > changes = s4	9.0%	44.4%
Material-UI			
Maintainability			
1	(complexity = s5) = > maintainability = s1	20.0%	100%
2	(NoFil = s4) = > maintainability = s2	13.0%	100%
3	(NoFil = s3) = > maintainability = s3	5.0%	100%
4	(duplicated blocks = s5) = > maintainability = s3	11.0%	100%
Duration			
1	(duplicated lines = s3) = > duration = s5	9.0%	88.8%
2	(duplicated lines = s2) and (CCN = s3) = > duration = s4	5.0%	66.6%
3	(duplicated blocks = s1) = > duration = s2	19.0%	63.1%
4	(duplicated blocks = s5) = > duration = s4	11.0%	54.5%
5	(comments rate = s4) = > duration = s3	5.0%	50%
Changes			
1	(complexity = s2) = > changes = s1	13.0%	53.8%
2	(NoCom = s2) and (comments rate = s5) = > changes = s3	6.0%	66.6%
3	(NoCom = s4) and (cognitive complexity = s4) = > changes = s5	8.0%	53.8%
4	(NoCom = s5) = > changes = s5	5.0%	80%
5	(CNN = s2) and (NoCom = s3) = > changes = s3	5.0%	100%
6	(CCN = s2) = > changes = s4	4.0%	75%
7	(NoF = s3) = > changes = s2	5.0%	66.6%