SN Computer Science (2020) 1:4
https://doi.org/10.1007/s42979-019-0004-1

ORIGINAL RESEARCH

# An Analysis of Software Bug Reports Using Machine Learning Techniques

Ha Manh Tran[1] · Son Thanh Le[2] · Sinh Van Nguyen[2] · Phong Thanh Ho[1]

Received: 15 April 2019 / Accepted: 10 June 2019

## Abstract

Bug tracking systems manage bug reports for assuring the quality of software products. A bug report (alsoreferred as trouble, problem, ticket or defect) contains several features for problem management and resolution purposes. Severity and priority are two essential features of a bug report that define the effect level and fixing order of the bug. Determining these features is challenging and depends heavily on human being, e.g., software developers or system operators, especially for assessing a large number of error and warning events occurring on software products or network services. This study first proposes a comparison of machine learning techniques for assessing severity and priority for software bug reports and then chooses an approach of using optimal decision trees, or random forest, for further investigation. This approach aims at constructing multiple decision trees based on the subsets of the existing bug dataset and features, and then selecting the best decision trees to assess the severity and priority of new bugs. The approach can be applied for detecting and forecasting faults in large, complex communication networks and distributed systems today. We have presented the applicability of random forest for bug report analysis and performed several experiments on software bug datasets obtained from open source bug tracking systems. Random forest yields an average accuracy score of 0.75 that can be sufficient for assisting system operators in determining these features. We have provided some analysis of the experimental results.

**Keywords** Network fault detection · Fault management · Machine learning · Data analytics · Software bug report

## Introduction

Fault detection plays an important role in managing computer systems. The more complex computer systems are, the more difficult fault detection is. Several hindrances of managing large and modern computer systems and services focus on service availability, performance unpredictability and failure control [1] that are closely associated with fault detection. A normal fault detecting mechanism usually works with the involvement of system operators and the support of multiple monitoring tools. A running computer system requires monitoring tools running along with. These monitoring tools keep reporting the status of the system. System operators observe and analyze abnormal signs on the report and the system, then create and submit a bug report to a bug tracking system (BTS) for resolution. Research activities have dealt with automating some parts of the fault detecting mechanism. One of the recently advanced research activities aims at exploiting monitoring log data and historical bug data to early notify the critical status of a system, or even forecast the forthcoming fault of a system.

This article is part of the topical collection "Future Data and Security Engineering" guest edited by Tran Khanh Dang.

✉ Ha Manh Tran
hatm@hiu.vn

Son Thanh Le
ltson@hcmiu.edu.vn

Sinh Van Nguyen
nvsinh@hcmiu.edu.vn

Phong Thanh Ho
phonght@hiu.vn

[1] HongBang International University, 215 Dien Bien Phu, Ward 15, Binh Thanh District, Ho Chi Minh City, Vietnam

[2] International University, Vietnam National University, Block 6, Linh Trung Ward, Thu Duc District, Ho Chi Minh City, Vietnam

Published online: 29 June 2019

Bug tracking systems store bug report data to control the quality of software products. They are frequently used to organize the workflows that produce bug reports and forward to system operators for resolution. A bug report contains many features for problem management and resolution purposes. Two essential features, namely severity and priority, define the effect level and fixing order of the bug, respectively. Determining these features is to a large extent a human-driven process. Evaluating a large number of error and warning events occurring on real-time software products or network services autonomously is challenging. This study proposes an approach of using random forest for evaluating severity and priority for software bug reports autonomously. The contribution of this study is thus threefold:

1. Investigating bug features extracted from the unified bug schema [2] for evaluating severity and priority.
2. Comparing machine learning techniques for evaluating the priority and severity features of software bug reports and choosing an approach of using random forest for further investigation.
3. Providing the prototyping implementation and experiments for the fault analysis approach on a 100-workstation computing cluster.

The rest of the paper is structured as follows: the next section includes some background of fault data analysis techniques applied to software maintenance, system failure and reliability, some related work of random forest applied to failure detection and prediction. Section 3 epitomizes machine learning techniques and provides brief comparison of their characteristics and performance for bug data analysis capability. Section 4 describes the fundamentals of decision tree and random forest, the applicability of software bug data processing, and several processes of building random forest for bug report datasets. Some mathematical formulas and explanations are referred from the study of Breiman et al. [3]. Several experiments in Sect. 5 report the performance and efficiency of bug data analysis before the paper is concluded in Sect. 6.

## Related Work

Several studies have used fault case analysis for fault detection and resolution. Wang et al. [4] have proposed an automatic fault diagnosis method for web applications in cloud computing. The online incremental clustering method identifies access behavior patterns and models the correlation between workload and the metric of resource utilization. The method detects anomalies by discovering the abrupt change of correlation coefficients and locates suspicious metrics using the feature selection method. The study of Ferreira et al. [5] has proposed an approach of using machine learning techniques for automated fault detection on solar-powered wireless mesh networks. The approach applies knowledge discovery methodology and a pre-defined dictionary of faults and solutions for classifying new faults. The authors of the study [6] have presented an online failure prediction system for predicting failures on networks and systems. Several technical remarks of this system are: (1) using Random Forest technique to train prediction models; (2) exploiting Apache Spark to process network management events; (3) testing the system on the dataset of the Spanish bank data center. The study [7] has applied the random forest and C5.0 decision tree algorithms for improving the prediction of network faults. It uses the customer trouble ticket dataset associated with the Internet usage and signal measurement datasets to build predictive models. The study of Tran et al. [2] has proposed a semantic search approach for bug reports. The approach includes crawling bug reports from bug tracking systems, extracting semi-structured bug data, and describing a unified data model to store bug tracking data. This model derived from the analysis of the most popular systems is used for semantic search. The model also facilitates fault feature extraction and analysis using machine learning techniques. Another study of Tran and Le [8] has reduced the computation problem by analysing several types of fault classifications and relationships. This approach exploits package dependency, fault dependency, fault keywords, fault classifications to seek the relationships between fault causes. Evaluating these approaches has performed on software bug datasets obtained from different open source bug tracking systems. Tran and Schönwälder [9] introduced the DisCaRia system that applies a distributed case-based reasoning approach to exploring fault solving resources on peer-to-peer networks. The prototyping system has been deployed and currently measured on the EmanicsLab distributed computing testbed [10].

Several other studies have used decision tree and random forest for fault classification and prediction. Sinnamon et al. [11] have applied the binary decision diagram to identify system failure and reliability. Large systems usually produce thousands of events that consume a large amount of processing time. This diagram associated with if-then-else rules and optimized techniques reduces time consuming problem. The study of Reay and Andrews [12] has proposed an analysis strategy aiming at increasing the likelihood of obtaining a binary decision diagram for any given fault tree while ensuring the associated calculations as efficient as possible. The strategy contains two steps: simplifying the fault tree structure and obtaining the associated binary decision diagram. The study also includes quantitative analysis on the set of binary decision diagrams to obtain the probability of top events, the system unconditional failure intensity and the criticality of the basic events. Guo et al. [13] have

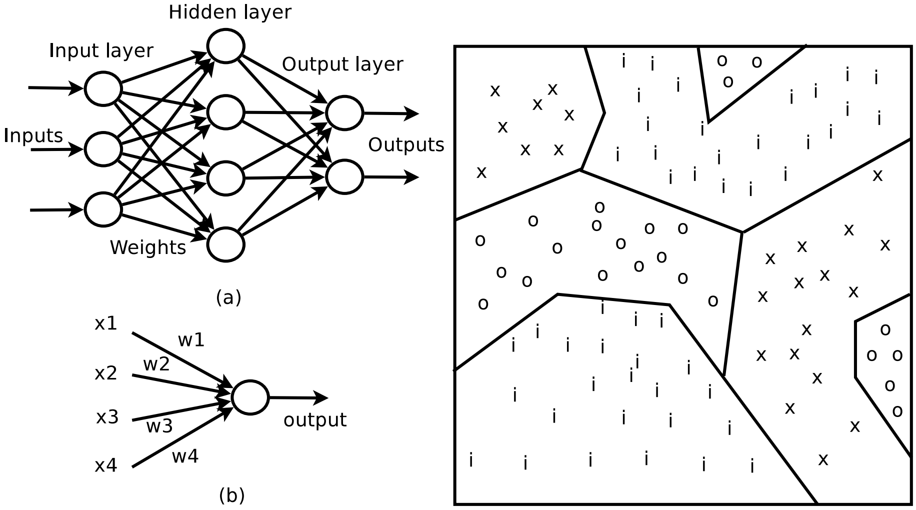

**Fig. 1** An ANN with three layers (**a**) and an artificial neuron (**b**) on the left side; an KNN classification with $K = 1$ on the right side

proposed an approach of using random forests for predicting fault prone modules in software development process. The approach exploits the information of the previous projects including modules, defects, locations, metrics, to predict the current project with an assumption of stable development environment. The approach presents several advantages of running efficiently on large datasets and outperforming the other classifiers in terms of robustness and noise. The study [14] has proposed two tree-based techniques for improving the classification of software failures based on their causes. The first technique uses tree-based diagrams to represent the results of hierarchical cluster analysis. The second technique generates a classification tree to recognize and refine failed executions. Zheng et al. [15] have provided a decision tree learning approach based on the C4.5 algorithm [16] to diagnose failures in large Internet sites. The approach uses the runtime properties of requests and then applies automated data mining techniques to identify the causes of failures. The approach is evaluated on application log datasets obtained from the eBay centralized application logging framework. The recent study of Tran et al. [17] has proposed an approach for evaluating the severity level of events using a classification decision tree. The approach exploits existing fault datasets and features, such as bug reports and log events to construct a decision tree that can be used to classify the severity level of new events. This study includes the prototyping implementation and evaluation of the approach for various bug report and log event datasets. The system operators thus refer to the result of classification to determine proper actions for the suspected events with a high severity level. While the previous approaches focus more on avoiding, detecting and resolving faults on the monitored systems, i.e., passive approaches rely on correct configurations or solutions for the detected faults, this active approach scrutinizes log events from currently running systems and historical bug reports from bug tracking systems in order to classify potential events with high severity that might cause crucial faults on running systems in the near future. The study in this paper is the extension of the previous study [18] that addresses the problem of choosing the best performance technique for analyzing fault datasets. It includes the performance comparison of several machine learning techniques for bug report datasets.

## Machine Learning Techniques

### Artificial Neural Networks

Artificial neural network (ANN) [19] simulates human nervous system with the connection and communication of many neurons. Similar to the neural network of human brain, ANN learns, records and uses experiences for appropriate circumstances. ANN has been successfully applied to several problems related to prediction and classification in the fields of finance, health, geology and physics. Typical examples include human face recognition, weather and disaster forecast, automatic steering control system, crash prediction system.

The common architecture of ANN includes input layer, hidden layer and output layer as shown in Fig. 1 on the left side. Neurons in the hidden layer connect and receive inputs from neurons in the previous layer, then process and pass outputs to the next layer. ANN can possess several hidden layers.

Data processing in ANN transforms inputs from a layer to another layer to adjust the weights of inputs on neurons for precise results. There are several elements in data processing: (1) inputs are data attributes; (2) outputs are results for a problem; (3) weights present the significance of inputs; (4) summation function sums up the weights of $n$ inputs for each neuron by the following formula: $y = \Sigma_{i=1}^{n} x_i w_i$, where $x_i$, $w_i$

and $y$ are input $i$, weight $i$ and result, respectively; and (5) transfer function referred to as an activation function decides whether a neuron generates an output to another neuron in the next layer.

## *K*-Nearest Neighbors

*K*-nearest neighbor (KNN) [20] is one of the supervised learning algorithms in machine learning. This algorithm is simple and learns less experience from the training process, but obtains much efficiency for some problems. It usually performs all essential computations when predicting the result of new data input. KNN can be applied to classification or regression problems in supervised learning.

The KNN algorithm seeks the result of a new data item using the *K*-nearest data items of the training data. It ignores some imprecise data items from the *K*-nearest data items. For the classification problem, the result of a new data item is directly deduced by the *K*-nearest data items of the training data. The result of a testing data item is possibly decided by the vote of the *K*-nearest data items, or by the weighted vote of the *K*-nearest data items. For the regression problem, the result of a new data item is possibly decided by the result of the nearest known data item ($K = 1$), or the average of the result of the *K*-nearest data items, or the distance function of the *K*-nearest data items. Figure 1 on the right side presents the classification of 1NN. This is a classification problem with three classes: $x$, $o$ and $i$. Each testing data item is assigned by a class that it belongs to.

## Decision Trees

Decision tree (DT) in machine learning is a predictive model that maps observations or phenomenon to concluding the target values of the observations or phenomenon. Each node in a tree corresponds to a variable, and a link between a node and its children node represents a specific value for that variable. Each leaf node represents the predicted value of the target variable, giving the values of the variables represented by the path from the root node to the leaf node. Decision trees are also referred to as decision tree learning technique in machine learning.

Decision tree technique is also a common practice in data mining. A decision tree describes a tree structure in which leaves represent classification classes and branches represent the combinations of attributes that lead to the classification class. A decision tree can be trained by splitting the data set into subsets based on evaluating attribute values. This process is recursively repeated for each derived subset. The recursive process is completed when no further splits are possible, or when a single classification can be applied to each member of the derived subset. Decision tree technique usually uses entropy to compute the homogeneity of a data item and information gain to measure the difference in entropy from before to after a set is split on an attribute.

Random forest technique uses a number of decision trees to improve the precision of regression and classification. This technique is more computational expensive and complex than decision tree technique, but reduces overfit and variance.

## Support Vector Machines

Support vector machine (SVM) [21] is another supervised learning method for classification and regression analysis. A standard SVM classifies input data items into two different classes, the SVM is thus referred to as a binary classification method. With the training dataset of two categories, the SVM training algorithm builds an SVM model that then classifies new dataset into the two categories. The SVM model is a representation of data points in space and selects the boundary between the two categories so that the distance from the training dataset to the boundary is as far as possible. The SVM model also represents the new dataset in the same space and classifies data items in one of the two categories depending on which part of the boundary is located. A general SVM constructs a hyperplane or a set of hyperplanes in a multidimensional or infinite dimensional space for classification and regression analysis. For the best classification, the hyperplane locates furthest away from the data points of all classes to minimize the classification error of the SVM model.

It is difficult in several problems to determine the boundary linearly in the original space. The data points are thus mapped into a more multidimensional space for easy analysis. The SVM method efficiently computes the mapping by the dot product of the data vectors in the new space using the coordinates of the old space. This dot product is defined by a proper kernel function.

## Naive Bayes

Naive Bayes (NB) [22] is a probabilistic method in machine learning, based on Bayes' theorem with the independence assumption of features. This method builds classifiers that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. Classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable. The method is widely used for text classification with low time consumption in the training process due to the independence assumption of features. There are several variants of Naive Bayes.

- Gaussian Naive Bayes (GNB) applies for continuous data with an assumption that data segments representing

**Table 1** An overview of machine learning techniques

| Method | Features | Advantages | Disadvantages |
|---|---|---|---|
| ANN | Black-box algorithm | Handle error well<br>Can learn non-linear models<br>Can learn in real-time | Require complex tuning<br>Sensitive to feature scaling<br>High computational cost |
| KNN | Distance-based algorithm<br>Lazy learner | Simple to implement<br>No training cost | Cannot be interpret<br>Need to define *K*<br>Computational expensive |
| DT | Tree-based algorithm | Easy to interpret<br>Useful in data investigation<br>Regression and classification | Can be over fit<br>Can be impacted by noise data |
| SVM | Discriminative algorithm | Effective in high dimensional spaces | Hard to choose kernel<br>Hard to interpret<br>High computational cost |
| GNB | Probabilistic algorithm | Easy to implement<br>No complicated optimization | Independent classes<br>Impractical assumption |

classes are distributed by a Gaussian distribution. For example, the training data contains a continuous feature $x$. This method segments the data by the classes, and then computes the mean and variance of $x$ in each class.

- Multinomial Naive Bayes applies for text classification that uses feature vectors to represent bags of words. For example, each document is represented by a vector with the same length of all words in dictionary. The use of the term frequency–inverse document frequency (tf–idf) statistics improves the efficiency of this method.
- Bernoulli Naive Bayes applies for data in which each component is a binary value (0 or 1). For example, instead of counting the total times of word appearance in a document, a vector only expresses the occurrence or absence of all words in dictionary by applying a yes (1) or no (0) answer for word appearance.

Table 1 describes the advantages and disadvantages of the mentioned machine learning techniques

## Brief Comparison

We have performed a brief comparison of the mentioned machine learning techniques. This comparison aims at evaluating the capability of these techniques on bug data analysis. Three metrics used for evaluating the classifiers include mean squared error (MSE), median absolute error (MAE) and time consumption. The performance of a classifier can be measured using the testing dataset. The output of a classifier is presented by the resulting vector that contains the severity values of the testing data items. This vector is compared with the true vector of the severity values of the same data items. The bug dataset possesses 100,000 bug reports described below.

**Table 2** Performance comparison of various classification techniques on the bug dataset

| Technique | MSE | MAE | Time (s) |
|---|---|---|---|
| ANN | 0.21 | 0.31 | 85.5 |
| KNN | 0.16 | 0.21 | 0.4 |
| DT | 0.2 | 0.23 | 0.5 |
| SVM | 0.22 | 0.32 | 25.8 |
| GNB | 0.21 | 0.3 | 0.2 |

An efficient classifier must obtain low MSE, MAE and time consumption. In Table 2, while ANN and SVM spend a lot of time for the training process, KNN, DT and GNB consume very low execution time. KNN provides the best classifier that outperforms the remaining classifiers with MSE = 0.16 and MAE = 0.21. This classifier sets parameters by the weighted distances, the ball-tree algorithm, and the optimal number of neighbors. DT is the second best classifier with MSE = 0.2 and MAE = 0.23. This classifier sets parameters by considering all the features before splitting. The ANN, SVM and GNB classifiers perform similarly. We choose the DT technique for further investigation because multiple decision trees can be optimized during the training phase to improve the capability of classification on large datasets.

## Random Forest Approach for Bug Analysis

Random forest is a classifier consisting of a number of decision trees that depend on the independently sampled values of random vectors with the same distribution. The precision of a random forest relies on the strength of the individual trees in the forest and the correlation between trees. The idea of random forest is to select the best decision trees from an

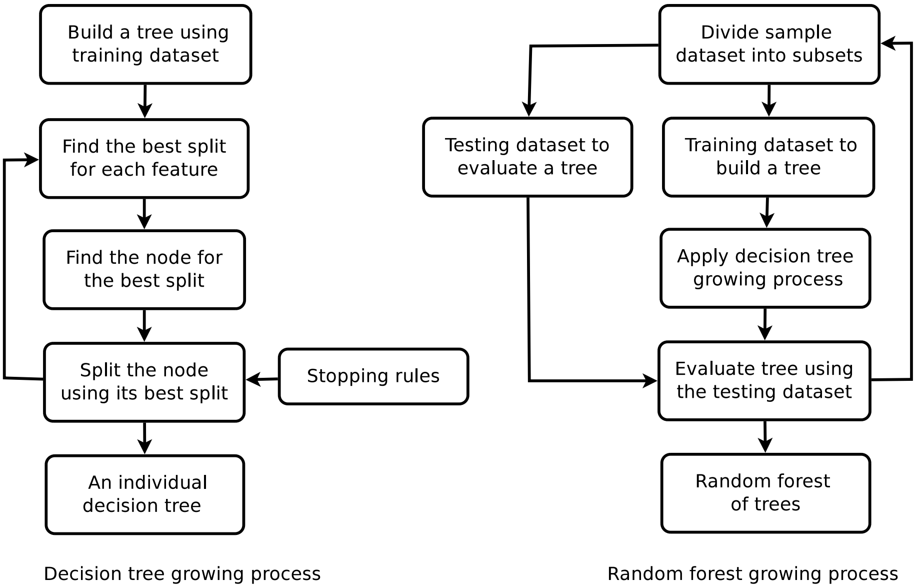
Build a tree using training dataset
Find the best split for each feature
Find the node for the best split
Split the node using its best split
Stopping rules
An individual decision tree
Decision tree growing process
Divide sample dataset into subsets
Testing dataset to evaluate a tree
Training dataset to build a tree
Apply decision tree growing process
Evaluate tree using the testing dataset
Random forest of trees
Random forest growing process

**Fig. 2** Processes of growing a decision tree (left) and a random forest (right)

ensemble of decision trees built by the subsets of the training dataset. A decision tree algorithm divides the training dataset into subsets with similar instances and uses entropy to calculate homogeneous values for instances. The decision tree and random forest growing processes are referred by the previous studies [3].

## Entropy Splitting Rule

A decision tree is built top-down from a root node and involves partitioning data into subsets that contain instances with similar values (homogeneous). The decision tree algorithm uses entropy to calculate the homogeneity of a sample.

$$H(S) = -\Sigma_{x \in X} P(x) \log P(x), \tag{1}$$

where $S$ is the current dataset for which entropy is being calculated. $X$ is a set of classes in $S$. $P(x)$ is a ratio between the number of elements in class $x$ and the number of elements in set $S$. When $H(S) = 0$, the set $S$ is perfectly classified.

Information gain IG($A$) is the measure of difference in entropy from before to after the set $S$ is split on an attribute $A$. In the other words, how much uncertainty in $S$ is reduced after splitting the set $S$ on the attribute $A$.

$$\mathrm{IG}(A, S) = H(S) - \Sigma_{t \in T} P(t) H(t), \tag{2}$$

where $H(S)$ is entropy of the set $S$. $T$ is the subset created from splitting $S$ by $A$. $P(t)$ is a ratio between the number of elements in $t$ and the number of elements in $S$. $H(t)$ is entropy of subset $t$. Information gain can be calculated (instead of entropy) for each remaining attribute. The attribute with the largest information gain is used to split $S$ in the current iteration.

## Decision Tree Growing Process

This process uses a dataset with features as input. A feature value can be ordinal categorical, nominal categorical, or continuous. The process uses entropy splitting rules to choose the best split among all the possible splits that consist of possible splits of each feature, resulting in two subsets of features. Each split depends on the value of only one feature. The process starts with the root node of the tree and repeatedly runs three steps on each node to grow the tree, as shown in Fig. 2 on the left side.

The first step is to find the best split of each feature. Since feature values can be computed and sorted to examine candidate splits, the best split maximizes the defined splitting criterion. The second step is to find the best split of the node among the best splits found in the first step. The best split also maximizes the defined splitting criterion. The third step is to split the node using its best split found in the second step, then the process repeats the first step if the stopping rules are not satisfied. The process generates a decision tree when the stopping rules are satisfied, as follows:

1. If a node becomes pure; that is, all cases in a node have identical values of the dependent variable, the node will not be split.
2. If all cases in a node have identical values for each predictor, the node will not be split.
3. If the current tree depth reaches the user-specified maximum tree depth limit value, the tree growing process will stop.
4. If the size of a node is less than the user-specified minimum node size value, the node will not be split.

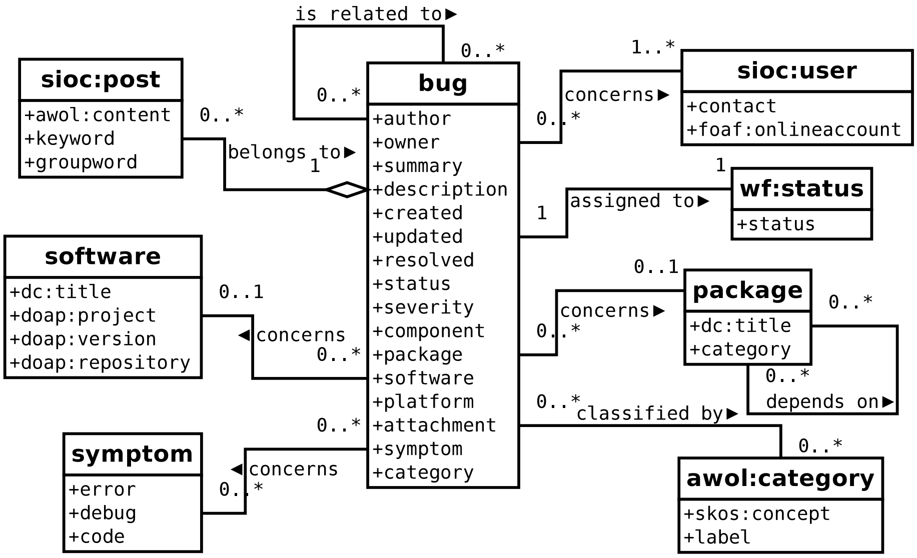

**Fig. 3** Unified bug schema represented as a UML diagram

5. If the split of a node results in a child node whose node size is less than the user specified minimum child node size value, the node will not be split.

## Random Forest Growing Process

The decision tree growing process is one of the main steps of the random forest growing process that also uses a dataset with features as input. The dataset is randomly partitioned into the training and testing datasets. The training dataset is the input of the decision tree growing process to construct a decision tree. The testing dataset is used to evaluate the constructed tree, as shown in Fig. 2 on the right side. This process repeats to select an sufficient number of trees. We have used Scikit Learn [23], NumPy [24] and SciPy [25] libraries for constructing a random forest. Algorithm 1 presents several steps to build a random forest for the bug dataset with the support of these libraries.

**Algorithm 1:** Constructing a random forest for the bug dataset

```
Input  : Processed bug dataset
Output: Random forest
Import random forest and caret libraries;
Load the dataset into data-frame;
Factorize the data-frame to numeric;
Parse factor to class label;
Partition the dataset to the training and testing datasets;
Construct random forest with the dataset;
return Random forest;
```

The first step is to load the random forest and caret libraries to build a random forest and a confusion matrix for validating the accuracy of the random forest. The second step is to load the bug dataset into data-frame that is a special tabular data structure to prepare for training and testing the random forest. The feature values of the bug dataset are factorized by numeric values in the third step because the algorithm cannot deal with text and enumerate values. An integer represents a distinct value, e.g., the priority feature contains 4 values: P1 (urgent), P2 (high), P3 (normal), and P4 (low) corresponding to 1, 2, 3, and 4 after factorization. The fourth step is to parse the class label of the bug dataset to factor variable type, which is a vector of integer values with a corresponding set of character values to use when the factor is displayed. This parsing step also decides the classification response of the random forest. We have used 25% of the bug dataset for evaluation and 75% of the bug dataset for building the random forest. The later dataset includes the testing and training datasets used to classify observations into severity or priority level. The algorithm also relies on a variable that specifies a certain number of random features to grow a single tree.

## Bug Data Processing

Bug dataset contains software bug reports obtained from several open source BTSs such as Bugzilla [26], Launchpad [27], Mantis [28], Debian [29]. Bug reports from different BTSs share several common features. Administration features, such as severity, status, platform, content, component, and keyword, are represented as field-value pairs. Description features, such as problem description, and follow-up discussion, are represented as textual attachments. The unified bug schema shown in Fig. 3 aims to support for semantic bug search [2]. This schema contains several features extracted by BTSs and new features to minimize the loss of bug report information, for example, the new relation feature establishes the relationships between bugs, or the new category feature provides for more sophisticated

**Table 3** List of extracted features

| Feature | Description | Data types |
|---|---|---|
| Severity | The effect level of the bug | Enumerate |
| Priority | The fixing order of the bug | Enumerate |
| Status | The open, fixed or closed status of the bug | Enumerate |
| Component | The component contains the bug | Enumerate |
| Software | The software contains the bug | Enumerate |
| Platform | The platform where the bug occurs | Enumerate |
| Keyword | The list of keywords that describe the bug | Text |
| Relation | The list of bugs related to the bug | Numeric |
| Category | The category of the bug | Enumerate |

bug classification. We have used a web crawler to get access to BTSs, retrieve and parse the HTML pages of bug reports to extract their content following the unified bug schema that allows various types of bug reports to be stored in one database. Most of bug features can be extracted from bug content. However, BTSs provide very different classifications for some features including severity, priority, status. The model classifies the severity feature into critical, normal, minor, feature, and the priority feature into urgent, high, normal, low.

Several bug features are concerned with severity and priority. Table 3 presents a list of essential features extracted from the bug schema. These features cause profound impact on determining the severity and priority of a bug report. The keyword feature contains the result of processing the description and discussion features. Figure 4 depicts the decision tree of a small fault dataset (see below). The decision tree contains multiple levels, we only present the first 4 levels.

We have applied the term frequency–inverse document frequency (tf × idf) method to process the description and discussion features that only include the textual data of a bug report. This method measures the significance of keywords to bug reports in a bug dataset by the occurrence frequency of keywords in a bug report over the total number of keywords of the bug report (term frequency) and the occurrence frequency of keywords in other bug reports over the total number of bug reports (inverse document frequency). This process results in a set of keywords that best describe the bug report, i.e., a set of distinct keywords with high significance. Algorithm 2 describes several steps to process the textual data of the bug dataset. The first step is to load the bug dataset as raw keyword set. The next three steps are to remove redundant words, meaningless words, and special characters using stop-word set and regular expression. The last step is to apply the tf × idf method for the remaining keywords.

```
Algorithm 2: Filtering keywords for a bug dataset
  Input   : Raw keyword set
  Output: Filtered keyword set
  Load raw keyword set;
  Remove duplicated and redundant words by using stop-word set;
  Remove meaningless words by using regular expression;
  Remove memory addresses by filtering special characters ;
  Process tf×idf on the keyword set;
  return Filtered keyword set;
```

## Evaluation

We have used a bug dataset of 300,000 bug reports occurring on Windows platform (Win platform) and 300,000 bug reports occurring on other platforms (All platform), and a computer with Intel Core I7, 3.8 GHZ each core, 8 GB RAM and Ubuntu 16.04 LTS 64-bit to build decision trees and random forests. We have verified the priority and severity of bug reports obtained from both the bug tracking systems and the fault detection system operating on a cluster of 100 workstations at International University–Vietnam National University's computing center. This cluster provides computing and storage services based on the OpenStack platform [30]. These bug reports have already included priority and severity values assessed by system operators for comparison. We have run experiments for several times and then computed the average evaluation scores with errors.

Several bug reports contain incomplete values for the extracted features due to the lack of responses during data collection. There are several methods, such as case deletion, mean imputation, median imputation, and $K$-nearest neighbor, to fill incomplete values for improving accuracy. We have applied the median imputation method for filling incomplete values by the mean of all known values of the features in the class to which the bug report with incomplete values belongs. The method approximately increases the average cross-validation scores of the Win platform and All platform datasets to 0.65, as shown in Fig. 5. The dataset of the All platform suffers less impact by incomplete values than that of the Win platform.

The first experiment compares severity accuracy between decision tree and random forest for Win and All platforms. For the Win platform, Fig. 6 on the left side reports the average accuracy scores of 0.65 and 0.75 for decision tree and random forest, respectively. While decision tree line slightly decreases as the size of datasets increases, random forest line starts with high scores, reduces and then linearly increases again as the size of datasets increases. For the All platform, Fig. 6 on the right side reports the average accuracy scores of 0.7 and 0.8 for decision tree and random forest, respectively. The dataset of the All platform is more complete than that of the Win platform. The lines of the accuracy scores in both figures are similar. Random forest outperforms decision tree

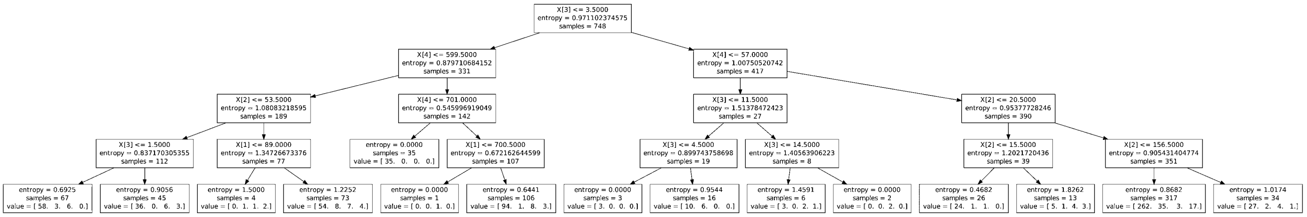

**Fig. 4** A decision tree for a fault dataset

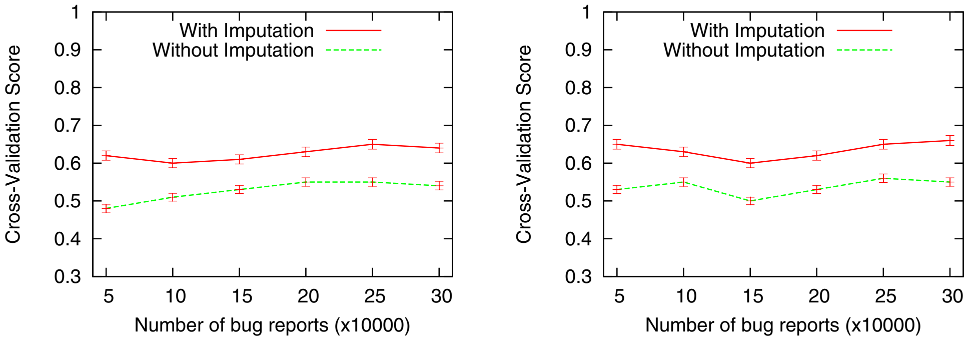

**Fig. 5** Cross-validation comparison between using and not using imputation on Win platform (left) and All platform (right)

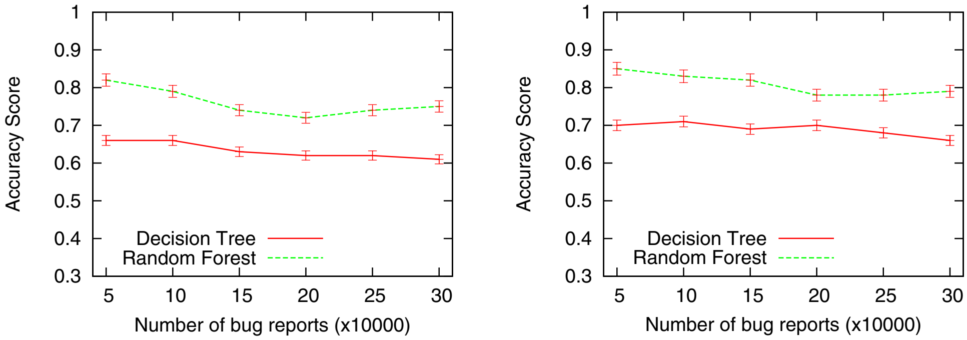

**Fig. 6** Severity accuracy comparison between decision tree and random forest on Win platform (left) and All platform (right)

on both the Win and All platforms. Random forest and decision tree performs well with a small number of bug reports. Random forest also performs well with a large number of bug reports while decision tree does not.

The second experiment compares priority accuracy between decision tree and random forest for Win and All platforms. For the Win platform, Fig. 7 on the left side reports the average accuracy scores of 0.66 and 0.73 for decision tree and random forest, respectively. While decision tree line linearly decreases as the size of datasets increases, random forest line remains stable as the size of datasets increases. For the Win platform, Fig. 7 on the right side reports the average accuracy scores of 0.69 and 0.81 for decision tree and random forest, respectively. Priority lines are more stable than severity lines, thus severity to some extent is more difficult to determine than priority. The dataset of All platform is more consistent than that of Win platform because of the big gap between the lines of accuracy scores on All platform. The shape of accuracy scores is also similar.

The third experiment presents time consumption for constructing decision tree and random forest for the whole dataset of 600,000 bug reports, as shown in Fig. 8 on the left side. Decision tree line linearly increases and random forest line exponentially increases as the size of datasets increases. It takes approximately 8 s and 33 s to build a decision tree and a random forest for 600,000 bug reports, respectively. Time consumption depends on the number of bug reports and features. Time consumption for random forest also depends on constructing multiple decision trees and selecting the best decision trees. Processing large bug datasets consumes much time. Figure 8 on the right side reports the accuracy score of random forest for severity and priority classification using the whole dataset. Severity and priority lines slightly decrease as the size of datasets increases. The accuracy scores of severity and priority are similar on

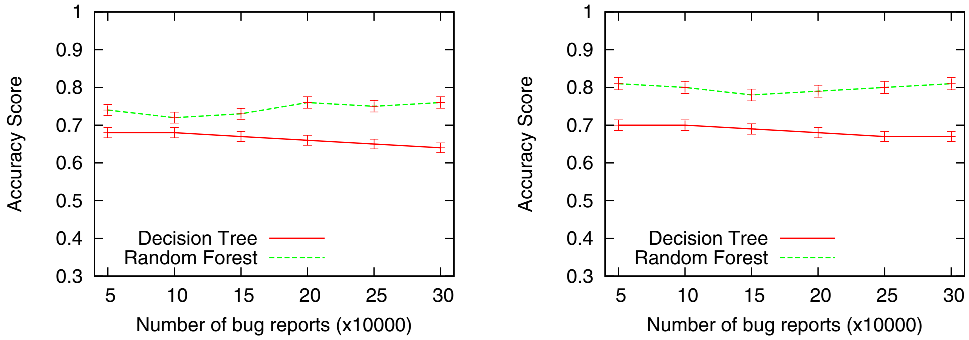

**Fig. 7** Priority accuracy comparison between decision tree and random forest on Win platform (left) and All platform (right)

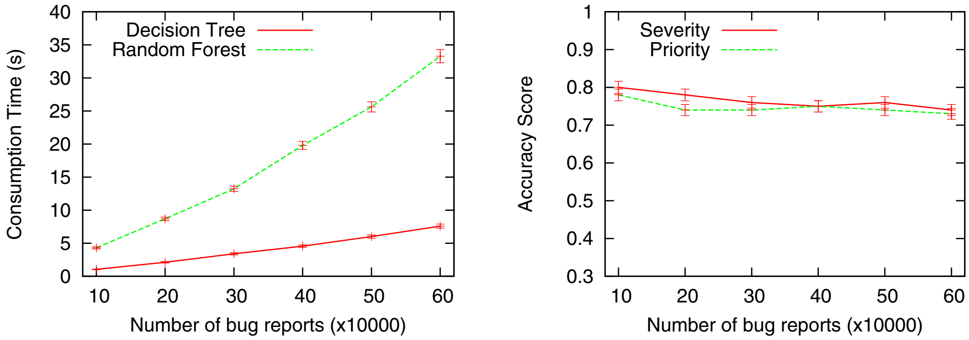

**Fig. 8** Time consumption comparison between decision tree and random forest (left) and random forest accuracy comparison between severity and priority classification (right) on the whole dataset

average because they are similar features in terms of data type and determination process. The average accuracy score of 0.75 can be sufficient for assisting system operators in determining these features.

## Conclusion

We have compared the performance of machine learning techniques for software bug report analysis, then chosen the random forest approach for further investigation on bug datasets. This approach targets to determining the severity and priority of a bug report automatically. Evaluating these features heavily depends on human being. This approach can be applied to evaluating a large number of log events for fault detection and forecast in large, complex communication networks and distributed systems. The log event dataset is so huge that system operators and even supporting tools cannot process quickly, thus resulting in neglecting potentially critical errors or warning events leading to critical errors. Instead of constructing a decision tree for learning from the training dataset and assessing the testing dataset, a random forest constructs a number of decision trees from the subsets of the training dataset and selects the best decision trees for assessing the testing dataset. We have used bug datasets obtained from open source BTSs for experiments. Bug reports to some extent contain the same features as log events including severity and priority. Evaluating the approach focuses on the performance and accuracy of random forest. We have measured the time consumption and accuracy of random forest for bug datasets. The experimental results reveal that random forest outperforms decision tree, i.e., approximately 10% for various datasets. Random forest, however, consumes more processing time than decision tree. Various bug datasets, such as bug reports on the Windows platform or on the other platforms, provide certain impact on accuracy score due to the consistency and completeness of bug reports. Future work focuses on exploring further bug report features and log event datasets to improve the accuracy of random forest and applying the fault detection and forecast tool for the realistic systems.

**Acknowledgements** This research activity is funded by the Vietnam National University in Ho Chi Minh City (VNU-HCM) under the Grant number C2019-28-06.

## References

1. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M. A view of cloud computing. ACM Commun. 2010;53(4):50–8.
2. Tran HM, Lange C, Chulkov G, Schönwälder J, Kohlhase M. Applying semantic techniques to search and analyze bug tracking data. J Netw Syst Manag. 2009;17(3):285–308.
3. Breiman L. Random Forests. Mach Learn. 2001;45(1):5–32.
4. Wang T, Zhang W, Wei J, Zhong H. Fault detection for cloud computing systems with correlation analysis. In: Proceedings of IFIP/IEEE international symposium on integrated network management IM'15; 2015. p. 652–8.

5. Ferreira VC, Carrano RC, Silva JO, Albuquerque CVN, Muchaluat-Saade DC, Passos DG. Fault detection and diagnosis for solar-powered wireless mesh networks using machine learning. In: Proceedings of IFIP/IEEE symposium on integrated network and service management (IM'17); 2017. p. 456–62.
6. Duenas JC, Navarro JM, Parada HA, Andion J, Cuadrado F. Applying event stream processing to network online failure prediction. Commun Mag. 2018;56(1):166–70.
7. Tan JS, Ho CK, Lim AH, Ramly MR. Predicting network faults using Random Forest and C5.0. Int J Eng Technol. 2018;7(2.14):93–6.
8. Tran HM, Le ST. Software bug ontology supporting semantic bug search on peer-to-peer networks. New Gen Comput. 2014;32(2):145–62.
9. Tran HM, Schönwälder J. Discaria—distributed case-based reasoning system for fault management. IEEE Trans Netw Serv Manag. 2015;12(4):540–53.
10. Hausheer D, Morariu C. Distributed Test-Lab: EMANICSLab. In: The 2nd international summer school on network and service management (ISSNSM '08). Switzerland: University of Zurich; 2008.
11. Sinnamon RM, Andrews JD. Fault tree analysis and binary decision diagrams. In: Proceedings in reliability and maintainability annual symposium; 1996. p. 215–22.
12. Reay KA, Andrews JD. A fault tree analysis strategy using binary decision diagrams. Reliab Eng Syst Saf. 2002;78(1):45–56.
13. Guo L, Ma Y, Cukic B, Singh H. Robust prediction of fault-proneness by Random Forests. In: Proceedings of 15th international symposium on software reliability engineering (ISSRE'04). Washington, DC: IEEE; 2004. p. 417–28.
14. Francis P, Leon D, Minch M, Podgurski A. Tree-based methods for classifying software failures. In: Proceedings of 15th international symposium on software reliability engineering (ISSRE'04). Washington, DC: IEEE; 2004. p. 451–62.
15. Zheng AX, Lloyd J, Brewer E. Failure diagnosis using decision trees. In: Proceedings of 1st international conference on autonomic computing (ICAC'04). Washington, DC: IEEE Computer Society; 2004. p. 36–43.
16. Quinlan JR. C4.5: programs for machine learning. San Francisco: Morgan Kaufmann Publishers; 1993.
17. Tran HM, Nguyen SV, Le ST, Vu QT. Applying data analytic techniques for fault detection. Trans Large Scale Data Knowl Cent Syst (TLDKS). 2017;31:30–46.
18. Tran HM, Nguyen SV, Ha SVU, Le TQ. An analysis of software bug reports using Random Forest. In: Proceedings of 5th international conference on future data and security engineering (FDSE'18). Springer; 2018. p. 1–13.
19. Bishop CM. Neural networks for pattern recognition. New York: Oxford University Press Inc; 1995.
20. Aha DW, Kibler D, Albert MK. Instance-based learning algorithms. Mach Learn. 1991;6(1):37–66.
21. Cortes C, Vapnik V. Support-vector networks. Mach Learn. 1995;20(3):273–97.
22. Rish I. An empirical study of the Naive Bayes classifier. In: IJCAI 2001 workshop on empirical methods in artificial intelligence, vol 3(22). 2001. p. 41–6.
23. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn: machine learning in python. J Mach Learn Res. 2011;12:2825–30.
24. Oliphant T. A guide to NumPy, vol. 1. New York: Trelgol Publishing; 2006.
25. Silva FB. Learning SciPy for numerical and scientific computing. Birmingham: Packt Publishing; 2013.
26. Mozilla bug tracking system. https://bugzilla.mozilla.org/. Accessed Aug 2017.
27. Launchpad bugs. https://bugs.launchpad.net/. Accessed Aug 2017.
28. Mantis bug tracker. https://www.mantisbt.org/. Accessed Aug 2017.
29. Debian bug tracking system. https://www.debian.org/Bugs/. Accessed Aug 2017.
30. OpenStack Cloud Software. http://www.openstack.org/ (2010). Accessed Aug 2017.