Applying Data Analytic Techniques
for Fault Detection

Ha Manh Tran[(✉)], Sinh Van Nguyen, Son Thanh Le, and Quy Tran Vu

Computer Science and Engineering, International University -
Vietnam National University, Ho Chi Minh City, Vietnam
{tmha,nvsinh,ltson,vtquy}@hcmiu.edu.vn

Abstract. Monitoring events in communication and computing systems becomes more and more challenging due to the increasing complexity and diversity of these systems. Several supporting tools have been created to assist system administrators in monitoring an enormous number of events daily. The main function of these tools is to filter as many as possible events and present highly suspected events to the administrators for fault analysis, detection and report. While these suspected events appear regularly on large and complex systems, such as cloud computing systems, analyzing them consumes much time and effort. In this study, we propose 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 other events. The administrators refer to the result of classification to determine proper actions for the suspected events with a high severity level. We have implemented and experimented the approach for various bug report and log event datasets. The experimental results reveal that the accuracy of classifying severity levels by using the decision trees is above 80%, and some detailed analyses are also provided.

Keywords: Event monitoring · Fault data analysis · Fault detection · Classification decision tree · Software bug report

1 Introduction

The increasing complexity and diversity of communication and computing systems makes management operations more and more challenging. Cloud computing systems [1], as an example, facilitate computing resource management operations on large computing systems to provision infrastructures, platforms and software as services. Armbrust [2] has specified 10 hindrances for managing cloud systems and services. Several hindrances including service availability, performance unpredictability and failure control are closely involved with event monitoring, one of the main functions of fault management. Monitoring events on these systems usually deals with a large number of events. The system administrators needs the support of tools that filter out many events and keep non-trivial events. However, these systems provide so many non-trivial events that

the administrators cannot handle. Furthermore, there is no guarantee that trivial events cannot cause system failure, e.g., warning events can become serious problems if there is no a proper action.

We have proposed an approach for evaluating the severity level of log events using classification and regression decision trees (CART trees). The idea of this approach is to determine the severity level of events automatically, thus providing the system administrators a decision whether further actions are needed for fault detection. The approach focuses on constructing a decision tree based on fault datasets and features, such as bug report and log events, and then using this tree to classify the severity level of other events. We have used bug report datasets obtained from existing bug tracking systems (BTSs) and log events obtained from monitoring systems to implement and experiment decision trees. The contribution is thus threefold:

1. Proposing an approach of using the classification decision tree for fault data analysis
2. Applying this approach to fault datasets for classifying the severity level of events
3. Providing the performance and efficiency evaluation of the approach on various fault datasets

In this paper, we have extended the previous study [3] to using large and real datasets of 500.000 bug reports and Mela log events for the evaluation of the proposed approach. The rest of the paper is structured as follows: the next section includes several analysis techniques applied to software maintenance, system failure and reliability, background of classification decision trees in data analysis. Section 3 describes the fundamentals of growing decision trees based on classification decision trees, focusing on entropy splitting rule and tree growing process. Some mathematical formulas and explanations are referred from the study of Breiman et al. [4]. Section 4 presents the characteristics of fault datasets and features. It also includes several processes of constructing decision trees for fault datasets. Several experiments in Sect. 5 report the performance and efficiency evaluation of the fault data analysis approach using the decision tree before the paper is concluded in Sect. 6.

2   Related Work

The authors of the study [5] have proposed an approach for analyzing fault cases in communication systems. The approach exploits the characteristics of semi-structured fault data by using multiple field-value and semantic vectors for fault representation and evaluation. Note that a fault case usually contains administrative field-value and problem description parts. The approach encounters the problem of high computation cost when processing semantic matrices for large fault datasets. Another study [6] from the same authors has reduced the computation problem by analyzing 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. These approaches have been evaluated on software bug datasets obtained from different open source bug tracking systems. Sinnamon et al. [7] has 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 [8] 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 2 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. The authors of the study [9] have presented two new tree-based techniques for refining the initial classification of software failures based on their causes. The first technique uses tree-like diagrams to represent the results of hierarchical cluster analysis. The second technique refines an initial failure classification that relies on generating a classification tree to recognize failed executions. This technique uses classification and regression tree for each subject of programs. Zheng et al. [10] has presented a decision tree learning approach based on the C4.5 algorithm to diagnose failures in large Internet sites. The approach records runtime properties of each request and applies automated machine learning and data mining techniques to identify the causes of failures. The approach has been evaluated on application log datasets obtained from the eBay centralized application logging framework. The study [11] proposes a nine-state model of adaptive behavior to enable fault detection in mobile applications. This model detects faults caused by erroneous adaptation process, and asynchronous update of context information, which leads to inconsistencies between the external physical context and internal representation within an application. The study [12] proposes a dynamic adaptation model that offers increased expressive power to compose complex adaptation rules, and guarantees soundness in fault detection. The recent study [13] introduces an elasticity analytic technique for cloud services. It also defines the concepts of elasticity space and elasticity pathway, and applies these concepts in evaluating the elasticity of cloud services. The Mela tool as the result of this study is an open source tool for monitoring and analyzing the elasticity of cloud services. This tool has been used in this study to collect log events.

Classification and regression trees (CART) [14] have been introduced by Breiman et al. and widely been used in data mining. Two main types of decision trees are classification and regression trees. The former tree predicts the outcome that belongs to one of the classes of the input data, e.g., predicting that today's weather is sunny, rainy or cloudy, while the later tree predicts the outcome that can be considered a real number, e.g., predicting that today's temperature is 25.3, 27.5, or 29.7 °C. Trees used for regression and classification have some similarities and also differences, such as the procedure used to determine

where to split. There are several variants of decision tree algorithms. Iterative Dichotomiser 3 (ID3) [15] was developed in 1986 by J.R. Quinlan. This algorithm creates a multi-level tree that seeks a categorical feature for each node using a greedy method. The features yield the largest information gain for categorical targets. Trees are grown to their maximum size and then applied to generalize to unseen data. The algorithm C4.5 [16] is an extension of the ID3 algorithm that converts the trained trees as the output of the ID3 algorithm into sets of if-then rules. The accuracy of rules is evaluated by determining the order in which these rules are applied. This algorithm uses numerical variables to define a discrete attribute and partitions the continuous attribute values into a discrete set of intervals. It avoids finding categorical features. Chi-squared automatic inter-action detector (CHAID) [17] uses multi-level splits to compute classification trees. This algorithm focuses on categorical predictors and targets. It computes a chi-square test between the target variable and each available predictor and then uses the best predictor to partition the sample into segments. It repeats the process with each segment until no significant splits remain. There are several differences between the CHAID and CART algorithms: (i) CHAID uses the chi-square measure to identify splits, whereas CART uses the Gini or Entropy rule; (ii) CHAID supports multi-level splits for predictors with more than two levels, whereas CART supports binary splits only and identifies the best binary split for complex categorical or continuous predictors; (iii) CHAID does not prune the tree, whereas CART prunes the tree by testing it against an independent (validation) dataset or through n-fold cross-validation.

3   CART Approach

The CART approach [4] uses a binary recursive partitioning process to build a decision tree. This process starts with the root node where data features are split into two children nodes and each of the children node is in turn split into grandchildren nodes based on splitting rules. The process runs recursively until no further splits are possible due to lack of data features and the tree reaches a maximal size. The process deals with continuous and nominal features as targets and predictors.

3.1   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 CART 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 (data) set for which entropy is being calculated. X is a set of classes in S. $P(x)$ is the proportion of the number of elements in class x to the number of elements in set S. When $H(S) = 0$ the set S is perfectly classified.

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

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

where, $H(S)$ is entropy of set S. T is the subset created from splitting set S by attribute A. $P(t)$ is the proportion of the number of elements in t to the number of elements in set 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 the set S on this iteration.

3.2   Tree Growing Process

The tree growing process uses a set of data features as input. A feature can be ordinal categorical, nominal categorical or continuous. The process chooses the best split among all the possible splits that consist of possible splits of each feature, resulting in two subsets of data 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. 1.
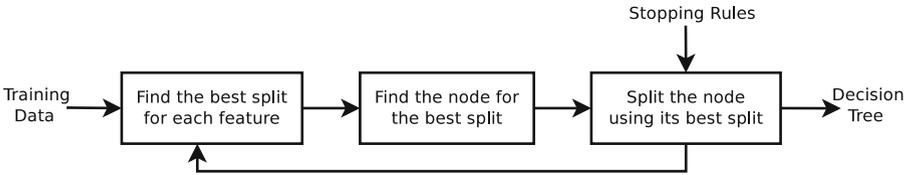

Fig. 1. A process of growing a CART decision tree

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 if the stopping rules are not satisfied. Several stopping rules are used:

– 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.
– If all cases in a node have identical values for each predictor, the node will not be split.
– If the current tree depth reaches the user-specified maximum tree depth limit value, the tree growing process will stop.
– If the size of a node is less than the user-specified minimum node size value, the node will not be split.

– 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.

Figure 6 plots a sample CART tree with 4 levels (refer to the end of the paper). The tree grows enormously as the data size increases.

4   Fault Data Analysis

Fault data analysis in this study focuses on using a decision tree to evaluate the severity level of suspected fault cases, such as bug reports, log events or trace messages. We have used bug report datasets for analysis because bug reports are already verified while log events are usually not verified yet.

4.1   Bug Report Data

Bug report data contains software and hardware bug reports obtained from forums, archives and BTSs. Several tracker sites available on the Internet, such as Bugzilla [18], Launchpad [19], Mantis [20], Debian [21] provide web interfaces to their bug data. Tracker sites differ from data inclusion and presentation, but share several similar administration and description fields. While the administration fields are represented as field-value pairs, such as severity, status, platform, content, component and keyword, the problem description field details the problem and follow-up discussions represented as textual attachments. We have used a web crawler to get as much access to bug data as ordinary users. The crawler retrieves the HTML pages of bug reports, then few parsers extract the content of bug reports and save the content to a database following a unified bug schema [22]. Table 1 reports popular BTSs and numbers of downloadable bug reports for tracker sites.

A bug report contains several features shown in the unified bug schema [22]. Some features cause less impact on determining the severity of the bug report,

Table 1. Popular bug tracking sites (as of November 2014). A plus indicates that the numbers present a lower bound

Tracker site	System	Bugs
bugs.debian.org	Debian BTS	900.000^+
bugs.kde.org	Bugzilla	400.000^+
bugs.eclipse.org	Bugzilla	400.000^+
bugs.gentoo.org	Bugzilla	350.000^+
bugzilla.mozilla.org	Bugzilla	800.000^+
bugzilla.redhat.com	Bugzilla	900.000^+
qa.netbeans.org	Bugzilla	250.000^+
bugs.launchpad.net	Launchpad	$1.200.000^+$

Table 2. List of important features

Feature	Description	Data types
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

such as owner, created time, updated time, etc. Our approach therefore focuses on the features as shown in Table 2. Note that each bug report contains the severity feature with a value. It is necessary to ignore this feature when building the tree to avoid some side effect. The keyword feature that contains the description and discussion of the bug requires further data processing.

4.2   Data Processing

Processing features improves the quality of the training datasets and thus enhance the performance of the decision tree. A bug report contains a textual part of the problem description and some discussions that hide distinct keywords or groups of keywords. We have applied the term frequency–inverse document frequency (tf×idf) method to reveal these keywords for the keyword feature. This method measures the significance of keywords to bug reports in a bug dataset by the occurrence frequency of the keywords in a bug report over the total number of the keywords of the bug report (term frequency) and the occurrence frequency of the keywords in other bug reports over the total number of bug reports (inverse document frequency). A distinct group of keywords contains related keywords with high significance. As a consequence, the keyword feature includes a set of keywords and groups that best describe the bug report. However, since bug reports are obtained from various BTSs, their descriptions and discussions contain redundant words, nonsense words or even meaningless words, such as: memory address, debug information, system path, article, etc. Algorithm 1 filters out these words from the bug dataset. We have implemented this algorithm in Python programming language.

   The first step is to load the bug dataset focusing on the keyword feature. The next three steps are to filter useless keywords. The stop-word set is the set of popular keywords that usually appear in textual description such as a, an, the, of, etc. The regular expression contains characters [0–9], [a-f] and [A-F], while the special characters contains _, –, \. The final step is to apply the tf×idf method on the whole keyword set and remove trivial keywords, i.e., keywords with low tf×idf values.

Algorithm 1. Filtering keywords for a bug dataset

Input   : Raw keyword set
Output: Filtered keyword set
1 Load raw keyword set;
2 Remove duplicated words and redundant words by using stop-word set;
3 Remove meaningless words by using regular expression;
4 Remove memory addresses by filtering special characters;
5 Process tf×idf on the whole keyword set;
6 **return** Filtered keyword set;

4.3   Tree Construction

The previous section explains using Entropy splitting rule to grow a decision tree. We present in this section using Scikit Learn library [23] to construct decision trees for bug datasets. Scikit Learn is an open source machine learning library for Python programming language and provides several classification, regression and clustering algorithms. It is designed to interoperate with Python numerical and scientific libraries such as NumPy [24] and SciPy [25]. The CART algorithm is one of the main classification algorithms supported by Scikit Learn. Algorithm 2 presents main steps to construct decision trees using the Scikit Learn library:

Algorithm 2. Constructing a decision tree for a bug dataset

Input   : Processed bug dataset
Output: Decision tree
1 Load the dataset into pandas data-frame and drop the platform feature;
2 Factorize the features;
3 Load sample data and class label;
4 Split the dataset into the training set and testing set;
5 Fit the training set into decision tree classifier;
6 Construct the tree using entropy criterion;

The first step is to load the dataset into pandas data-frame that is a special tabular data structure to prepare data for the CART algorithm. It is also important to drop the platform feature in the data-frame because the dataset is already grouped by this feature. Since the CART algorithm cannot deal with non-numerical values, while the feature values in the bug dataset are non-numeric, i.e., enumerate or text, all the feature values need to be factorized into numerical values in the second step. The pandas library supports for converting non-numerical values to numerical values. Each distinct value is replaced by a unique integer, e.g., the severity feature contains 4 values: feature, minor, normal and critical corresponding to 0, 1, 2, 3 after factorization. The next step is to separate the data-frame into 2 parts. The first part is the sample data that contains the numerical values of all features, while the second part is the class label that marks the numerical classes for each particular bug. The most important step in this algorithm is to partition the sample data and class label into the

training set and testing set. The training set is used for training the decision tree, while the testing set is used for evaluating the decision tree. The percentages of the training and testing sets are 75% and 25% respectively. Finally, the decision tree is trained by a method supported by Scikit Learn library. The input of this method is the training set found in the previous step. Figure 7 plots a part of a decision tree for the Linux platform dataset (refer to the end of the paper).

Since the decision tree contains multiple levels, we only present the first 4 levels. The leaf nodes contains the following values:

1. The first component counts samples that have the severity of feature
2. The second component counts samples that have the severity of critical
3. The third component counts samples that have the severity of minor
4. The fourth component counts samples that have the severity of normal

5   Evaluation

We have used a dataset of 500.000 bug reports approximately for experiments. A large dataset usually yields a large decision tree that possibly causes the performance problem due to the complexity and memory consumption of the tree. The authors of the study [26] have already proposed an approach to construct decision trees from large datasets. This approach builds a set of decision trees based on tractable size training datasets which are the subsets of the original dataset. As a result, the approach reveals the over-fitting problem of large decision trees. We have separated bug reports into 4 smaller datasets following the platform feature: 180.000 bug reports occurring on all platforms (All platform), 160.000 bug reports occurring on Windows platform (Win platform), 80.000 bug report occurring on Linux platform (Linux platform) and 80.000 bug report occurring on Macintosh platform (Mac platform). We have built several decision trees and performed all experiments on laboratory workstations with Intel Core i5-4590 CPU 3.30 GHZ, 8 GB of RAM and Ubuntu 14.04 LTS. In addition, we have used the Mela tool to collect log events with an interval of 5 s. The interval is sufficient to capture changes on the monitored system, but it also produces several duplications on idle time. We have filtered duplicated events in the Mela dataset for experiments.

The first experiment measures time consumption for constructing decision trees over various datasets. Time consumption linearly increases as the size of datasets increases, as shown in Fig. 2. It takes 340 ms to 380 ms approximately to build a decision tree of 160.000 bug reports. Note that time consumption depends on numbers of events and features of bug reports. Bug reports in the Win platform dataset contain less one feature than bug reports in the whole dataset, i.e., the platform feature is eliminated in a platform specific dataset, thus time consumption for both datasets is slightly different. It takes 900 ms approximately to construct a decision tree of 500.000 bug reports. However, log files usually contain millions of events, reducing processing time is an important task.
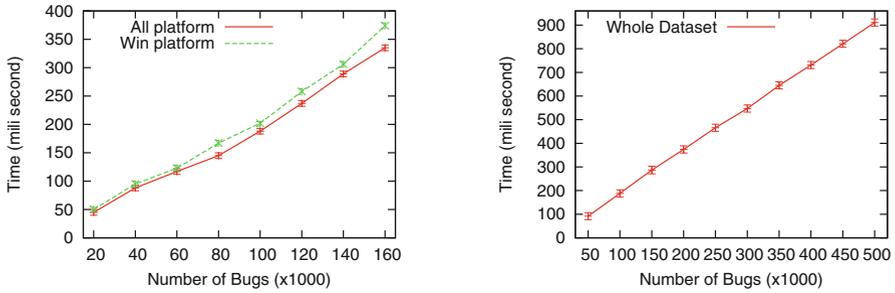
Fig. 2. Time consumption for constructing decision trees over various datasets

The datasets contain thousands of bug reports that possibly miss several features. It is necessary to apply the median imputation method for these datasets to fill the missing features. Dealing with the missing values is one of the most common problems in data training process. This problem occurs when data values are unavailable for observations due to the lack of responses: data is provided for neither several features nor a whole case. Lacking values are sometimes caused by researchers due to collecting data improperly or making mistakes in data input.

Using training datasets with the missing values can affect accuracy in classification. Several prevailing methods that are capable of dealing with this issue have been developed before. Case deletion method discards cases with the missing values for at least one feature. A variant of this method only eliminates cases with a high level of the missing values while determining the extent of features for cases with a low level of the missing values. Mean imputation method replaces the missing values for features by the mean of all known values of these features in the class to which the case with the missing values belongs. Similar to the mean imputation method, the median imputation method replaces the missing values for features by the median of all known values of the features in the class to which the case with missing values belongs. Using median avoids the presence of outliers and also assures the robustness of the method. This method is suitable for datasets that the distribution of the values of a certain feature is skewed. Modified K-nearest neighbor method determines the missing values for a case by considering a certain number of the most similar cases. The similarity of two cases is measured by a distance function.

The second experiment fills the missing values for bug reports using the mean imputation method. It then compares cross-validation scores for both datasets with and without imputation. Figure 3 on the left side reports the difference of cross-validation scores for the All platform dataset with and without imputation. The All platform dataset with imputation obtains the average cross-validation score of 0.68 that improves considerably a number of the missing values from the All platform dataset without imputation. Especially with imputation, the All platform dataset of 160.000 bug reports reaches the cross-validation score of 0.7.
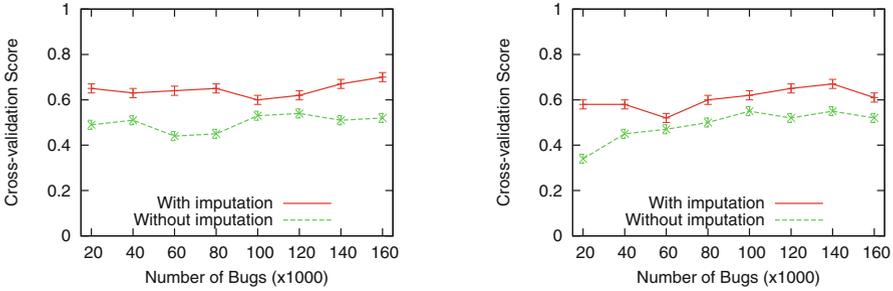
Fig. 3. Cross-validation comparison for the All platform (left) for the Win platform (right) with and without imputation

Similarly, the Win platform dataset with imputation obtains the average cross-validation score of 0.62 that slightly improves a number of the missing values from the Win platform dataset without imputation, as shown in Fig. 3 on the right side. With imputation, the Win platform dataset of 160.000 bug reports reaches the cross-validation score of 0.6. The Win platform dataset performs worse than the All platform dataset. We observe that the missing values of bug reports in the All platform dataset are less specific than that the missing values of bug reports in the Win platform dataset. Note that the number of the missing values increases as the size of datasets increases, thus using the imputation technique can improve the accuracy score of classification.

The third experiment evaluates the accuracy of the decision trees using various sizes of datasets. The idea is to divide the original dataset with imputation into the training and testing datasets. While the training dataset is used to build a decision tree, the testing dataset is used to evaluate the accuracy of this decision tree. The extreme case of cross-validation, namely leave-one-out cross-validation, has been used for this experiment. We have used the decision trees to classify bug reports from the testing dataset into severity levels, then compared these classified severity levels with the correct severity levels of the testing dataset. Accuracy score is calculated based on the number of matching severity levels.

Figure 4 on the left side reports high and stable accuracy scores for the All and Win platform datasets with the average score of 0.9 approximately. Both datasets perform similarly. We observed that bug reports for these platforms tend to be common problems that can be easily reproduced, determining severity levels for these bug reports is rather straightforward and precise. However, some bug reports from a specific platform are very specific and difficult to be classified into severity levels properly. These bug reports reduces accuracy scores. Figure 4 on the right side also presents the similar accuracy scores of the Linux and Mac platform datasets with the average score of 0.85 approximately. The All and Win platform datasets are larger than the Mac and Linux platform datasets that possibly cause an impact on accuracy scores as the size of these datasets
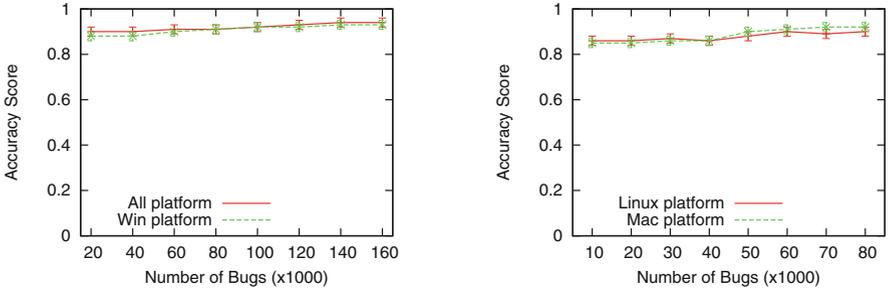
Fig. 4. Accuracy comparison between the All and Win platforms (left) and between the Linux and Mac platforms (right)
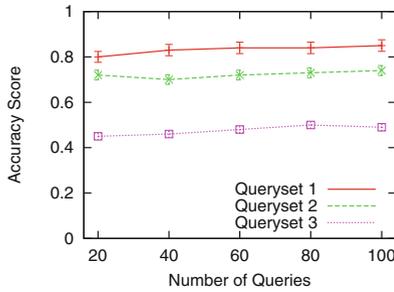

Fig. 5. Accuracy of the decision tree constructed by the whole dataset using various querysets

increases. In addition, average time to train a decision tree of 160.000 bug reports is considerably fast.

The last experiment measures the accuracy of the decision tree constructed by the whole dataset using three querysets. The first queryset contains queries extracted from the testing dataset. The second queryset contains queries extracted from the existing BTSs. These queries are bug reports in open status and some features, such as, keyword, relation, category and severity are insufficient. The last queryset contains queries extracted from some monitoring tools, such as Nagios [27], Ganglia [28] and Mela [13]. These queries are only warning and error events that possess different features from bug reports. Figure 5 presents the accuracy score of the decision tree with three querysets. The first queryset outperforms the other querysets and its accuracy score considerably matches with the results of the above experiments because queries are from the testing dataset. The accuracy score decreases as the decision tree is large. The second queryset obtains the average accuracy score of 0.7. Queries from this queryset contain the same bug format with incomplete features, but still achieve reasonable accuracy scores. The third queryset performs poorly with the average accuracy score lower than 0.5. There are several differences in features between bug reports and log events that prevent the decision tree from
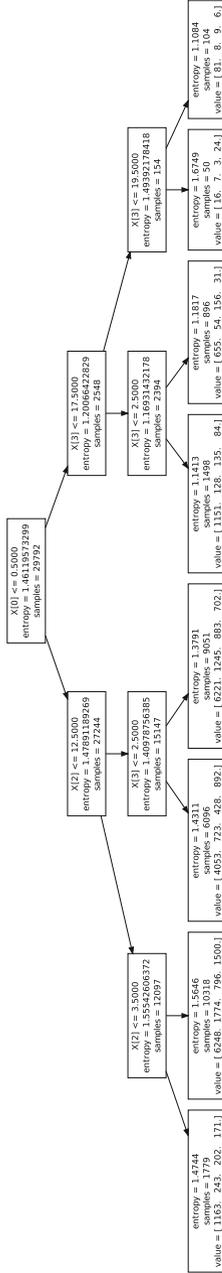
Fig. 6. A sample CART tree

Fig. 7. Decision tree for the Linux platform dataset

classifying these queries precisely. Log events need additional data referred to as context-ware data including system load, network status, memory usage, number of processes, etc. to fulfill the missing features.

6   Conclusions

We have proposed an approach of using the CART decision tree for fault data analysis that can be applied to monitoring and detecting faults in communication networks and distributed systems. The log event data is so huge that system administrators and even supporting tools might miss critical signs, messages or events accidentally. This decision tree is characterized by the capability of learning from the training fault dataset and then determining the severity level of log events from the testing dataset. We have used a bug report dataset for evaluating the decision tree. Bug reports obtained from BTSs are to some extent related to log events with a severity level. Evaluating the approach focuses on the performance and efficiency of the decision tree. We have computed the time consumption of constructing the decision tree, the accuracy of classification and the imputation of the missing features. The experimental results reveal that the accuracy of classifying severity levels by using the decision trees is above 80%, especially 90% for the All platform dataset. Applying methods to deal with the missing features in the training dataset improves efficiency. Moreover, the decision tree with a tractable size training dataset consumes less processing time and possibly yields high efficiency. The decision tree constructed by bug reports does not perform well with log events due to the lack of significant features. Future work focuses on exploiting more common features in bug reports or log events, especially exploiting distinct keywords from textual description. While monitoring systems, log events can be extended to include additional context-aware features, such as system load, network status, memory usage, number of processes, etc. to evaluate precisely the severity level of log events [12,13].

References

1. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. **25**(6), 599–616 (2009)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. ACM Commun. **53**(4), 50–58 (2010)
3. Tran, H.M., Nguyen, S., Le, S.T., Vu, Q.T.: Fault data analytics using decision tree for fault detection. In: Dang, T.K., Wagner, R., Küng, J., Thoai, N., Takizawa, M., Neuhold, E. (eds.) FDSE 2015. LNCS, vol. 9446, pp. 57–71. Springer, Heidelberg (2015). doi:10.1007/978-3-319-26135-5_5

4. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Chapman & Hall/CRC, New York (1984)

5. Tran, H.M., Schönwälder, J.: Fault representation in case-based reasoning. In: Clemm, A., Granville, L.Z., Stadler, R. (eds.) DSOM 2007. LNCS, vol. 4785, pp. 50–61. Springer, Heidelberg (2007). doi:10.1007/978-3-540-75694-1_5

6. Tran, H.M., Le, S.T., Ha, S.V.U., Huynh, T.K.: Software bug ontology supporting bug search on peer-to-peer networks. In: Proceedings of 6th International KES Conference on Agents and Multi-agent Systems Technologies and Applications (AMSTA 2013). IOS Press (2013)

7. Sinnamon, R.M., Andrews, J.D.: Fault tree analysis and binary decision diagrams. In: Proceedings in Reliability and Maintainability Annual Symposium, pp. 215–222 (1996)

8. Reay, K.A., Andrews, J.D.: A fault tree analysis strategy using binary decision diagrams. Reliab. Eng. Syst. Saf. **78**(1), 45–56 (2002)

9. 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 2004), pp. 451–462, Washington, DC, USA. IEEE (2004)

10. Zheng, A.X., Lloyd, J., Brewer, E.: Failure diagnosis using decision trees. In: Proceedings of 1st International Conference on Autonomic Computing (ICAC 2004), Washington, DC, USA, pp. 36–43. IEEE Computer Society (2004)

11. Sama, M., Rosenblum, D.S., Wang, Z., Elbaum, S.: Model-based fault detection in context-aware adaptive applications. In: Proceedings of 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, pp. 261–271. ACM (2008)

12. Xu, C., Cheung, S.C., Ma, X., Cao, C., Jian, L.: Detecting faults in context-aware adaptation. Int. J. Softw. Inf. **7**(1), 85–111 (2013)

13. Moldovan, D., Copil, G., Truong, H.L., Dustdar, S.: MELA: monitoring and analyzing elasticity of cloud services. In: Proceedings of 5th International Conference on Cloud Computing, pp. 80–87. IEEE Press (2013)

14. Breiman, L., Friedman, J., Stone, C., Olshen, R.: Classification and Regression Trees. Chapman & Hall, New York (1984)

15. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)

16. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Francisco (1993)

17. Kass, G.V.: An exploratory technique for investigating large quantities of categorical data. Appl. Stat. **29**(2), 119–127 (1980)

18. Mozilla bug tracking system. https://bugzilla.mozilla.org/. Accessed Jan 2015

19. Launchpad bugs. https://bugs.launchpad.net/. Accessed Jan 2015

20. Mantis bug tracker. https://www.mantisbt.org/. Accessed Jan 2015

21. Debian bug tracking system. https://www.debian.org/Bugs/. Accessed Jan 2015

22. Tran, H.M., Lange, C., Chulkov, G., Schönwälder, J., Kohlhase, M.: Applying semantic techniques to search and analyze bug tracking data. J. Netw. Syst. Manag. **17**(3), 285–308 (2009)

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. **12**, 2825–2830 (2011)

24. Oliphant, T.: A Guide to NumPy, vol. 1. Trelgol Publishing, Spanish Fork (2006)

25. Silva, F.B.: Learning SciPy for Numerical and Scientific Computing. Packt Publishing, Birmingham (2013)

26. Hall, L.O., Chawla, N., Bowyer, K.W.: Decision tree learning on very large data sets. In: Proceedings of IEEE International Conference on Systems, Man and Cybernetics, vol. 3, pp. 2579–2584. IEEE (1998)
27. The industry standard in IT infrastructure monitoring (1999). http://www.nagios.org/. Accessed Nov 2015
28. Ganglia monitoring system (2000). http://ganglia.info/. Accessed Nov 2015