Journal of Computer and Communications, 2022, 10, 33-45
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Fault Prediction with Static Software Metrics in Evolving Software: A Case Study in Apache Ant

Xue Han, Gongjun Yan

Romain College of Business, University of Southern Indiana, Evansville, USA
Email: xhan@usi.edu, gyan@usi.edu

How to cite this paper: Han, X. and Yan, G.J. (2022) Fault Prediction with Static Software Metrics in Evolving Software: A Case Study in Apache Ant. *Journal of Computer and Communications*, **10**, 33-45.
https://doi.org/10.4236/jcc.2022.102003

Received: November 13, 2021
Accepted: February 22, 2022
Published: February 25, 2022

Abstract

Software testing is an integral part of software development. Not only that testing exists in each software iteration cycle, but it also consumes a considerable amount of resources. While resources such as machinery and manpower are often restricted, it is crucial to decide where and how much effort to put into testing. One way to address this problem is to identify which components of the subject under the test are more error-prone and thus demand more testing efforts. Recent development in machine learning techniques shows promising potential to predict faults in different components of a software system. This work conducts an empirical study to explore the feasibility of using static software metrics to predict software faults. We apply four machine learning techniques to construct fault prediction models from the PROMISE data set and evaluate the effectiveness of using static software metrics to build fault prediction models in four continuous versions of Apache Ant. The empirical results show that the combined software metrics generate the least misclassification errors. The fault prediction results vary significantly among different machine learning techniques and data set. Overall, fault prediction models built with the support vector machine (SVM) have the lowest misclassification errors.

Keywords

Software Engineering, Fault Prediction, Software Metrics, Machine Learning

1. Introduction

Testing is a crucial part of the software development life cycle [1]. Ultimately, the purpose of testing is to expose all faults in the software system. A solid testing strategy can provide a high level of confidence about the correctness of an application after it has been deployed. However, software testing can be resource-demanding [2]. Detecting faults in a system randomly may not be feasible [3] especially when dealing with large-scale projects. Practitioners (developers and testers) want to allocate resources in the most effective ways to find faults.

Prior research [4] shows that a fault found after deployment can be 100 times as costly to fix in an early stage. Researchers strive to find a way to help practitioners to detect software faults as early as possible [5]. The decisions of when and where to put the testing efforts are often based on developers' experience and expertise. This approach might not be reliable. It may not even be sustainable and consistent as developers move in and out of an organization [6]. The experience-based approach also varies a lot since practitioners have different perspectives regarding how to conduct testing.

With recent advancements in applying AI technologies to software engineering problems [7], many research reports promising preliminary results using machine learning techniques to predict faults in software systems [8] [9] [10]. This study explores what software metrics [11] are suitable for constructing fault prediction models and examine how well those machine learning models perform in predicting faults.

Unlike prior research [12] that depends on similar projects to build the prediction model, this study collects training data through different versions of the same project. Out approach outputs a much reliable representation of the application to build fault prediction models. It is also more practical to collect training data as the subject project evolves than to search for similar projects in the wild.

This study aims to answer the following research questions when conducting the empirical study.

- What static software metrics can provide the best faults prediction result?
- Which machine learning models give the best fault prediction results?
- How well do prediction models perform across the continuous versions of the subject program?

We make the following contributions in this paper.

- An empirical study in fault prediction with software metrics.
- An evaluation of four different fault prediction models.
- A publicly accessible data set.
- A publicly accessible machine learning code (in MATLAB).

This paper is organized as follows. In Section 2, we present the overall approach of the empirical study. In Section 3, we discuss the research questions and explain the design of the experiments. In Section 4, we examine the study results. In Section 5, we discuss the sensitivity analysis and the threat to validity. Lastly, we conclude the empirical study in Section 6.

2. Approaches

In this section, we discuss the overall approach adopted by this empirical study. **Figure 1** shows an overview of the approach. In the pre-processing phase, we extract and synthesize software metrics [8] [13] from the subject programs. In the model construction phase, we build fault prediction models and conduct sensitivity analysis to fine-tune the model hyperparameters.

2.1. Data Pre-Processing

We use static code analysis [14] to extract software metrics. Static software metrics is chosen over runtime software metrics for consistency concerns. For instance, instrumentation and monitor tools may be used to get the runtime metrics which may introduce high runtime overhead and disturb the execution of the subject program [15]. Also, depending on the deployment environment (e.g., physical or virtual machines running the subject program), we may get a completely different set of metrics readings [16]. Table 1 lists the static software metrics used in the study.

Static software metrics undergo a series of pre-processing steps. First, we apply normalization [17] to bring metrics to the same scale while maintaining relative significance. For example, the value of the metric "Lack of Cohesion in Methods

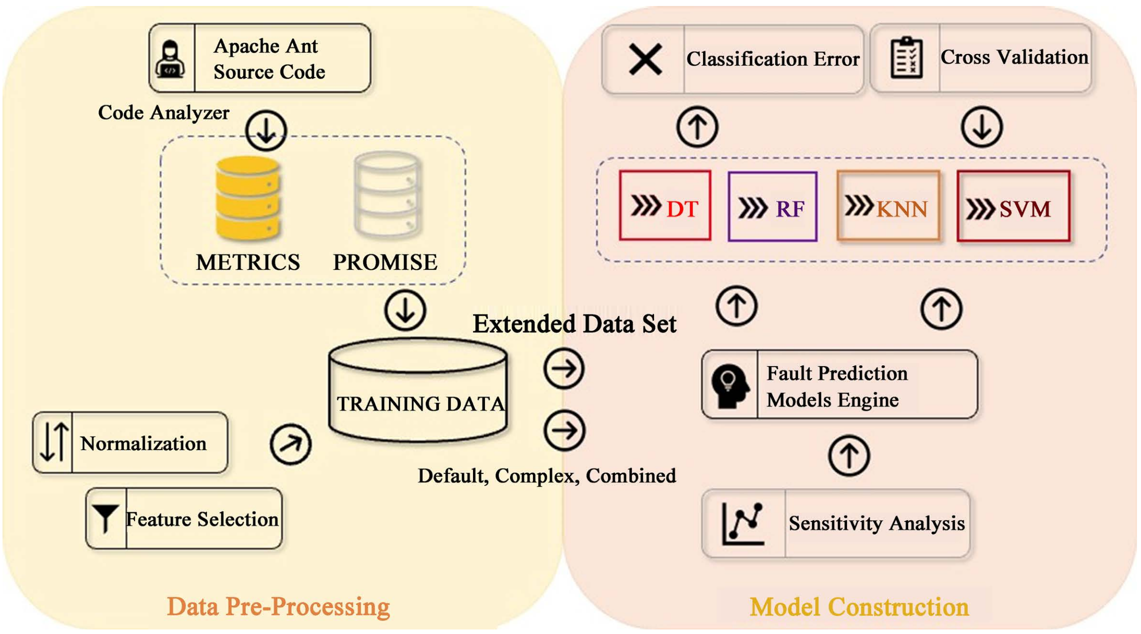

Figure 1. Approach overview.

Table 1. Static software metrics.

METRIC	DESCRIPTION	OO METRIC	DESCRIPTION
Files	# of files	WMC	Weighted method count
Lines	Line of code	DIT	Depth of inheritance tree
AVG-Len	Average code length	NOC	The number of children for a class
Cd/Cm + WS	Code non-code ratio	CA	Afferent coupling
Cd/Cm	Code comments ratio	CE	Efferent coupling
Cd/File	Code file ratio	DAM	Data access metric
Cm/File	Comment file ratio	MOA	Measure of aggregation

(LCOM)" [13] could range from 0 to 2247 in dataset 3 before normalizing to the range of 0 to 1. It reduces the dramatic range in the metrics value space that may otherwise negatively affect the accuracy of the prediction models.

Not all static software metrics are suitable for constructing the fault prediction model. Some of them may even reduce the model's accuracy. Next, a forward and backward feature selection [18] is applied to reduce the feature space dimensionality and to achieve greater generalization.

2.2. Fault Prediction Models

In the second phase, we apply both supervised and unsupervised machine learning techniques to build fault prediction models. Decision Tree (DT) [19] is a classic supervised learning model. The tree is constructed by a recursive binary split on which the selected node maximizes local information gain [13]. We use Gini's Diversity Index [20] $gdi = 1 - \Sigma_i p2(i)$ for tree pruning. Random Forest (RF) [21] is an ensemble method. RF combines an arbitrary number of decision trees. The number of decision trees used for each data set is based on a sensitivity analysis which will be discussed in Section 5. Support Vector Machine (SVM) [22] is a linear classification model that maximizes the decision boundary. The linear kernel is used for two-class learning. $G(x_j, x_k) = x_j^t x_k$ where x_j and x_k are two observations. And an error-correcting output codes (ECOC) model for multi-class learning. K-nearest neighbor (KNN) [23] is an unsupervised learning method. KNN assumes that if two data points are similar, they are likely to be in the same class. We use the euclidean distance to calculate the shortest distance between a data point and the cluster's centroid. We conduct a sensitivity analysis to evaluate different k values and select the k value that gives the least misclassification error [24] to construct KNN. To avoid overfitting [25], ten-fold cross-validation [26] is applied to all four models. Since ten-fold cross-validation randomly samples instances and puts them in ten folds [27], the process is repeated ten times for each model to avoid sampling bias [28].

3. Empirical Study

In this section, we discuss details of the implementation, subjects, and data set design.

3.1. Implementation

The experiment runs on a Mac OS X with a quad-core 2.4 GHz Intel Core-i5 CPU, 16 GB of memory, and 256 GB of SSD. We use CodeAnalyzer [29] to extract static software metrics. CodeAnalyzer is a light-weighted tool for analyzing source code. To build fault prediction models, we use the MATLAB Statistics Toolbox [30].

3.2. Subjects and Data Sets

Apache Ant is an open-source Java-based build tool. Tour continuous versions (v1.4 to v1.7) of the Apache Ant is used for its popularity [31] and availability [32]. Table 2 shows the characteristics of Apache Ant. The first column (METRIC) shows the size of the Apache Ant. The third column (RATIO) shows the proportion for source code. We refer to the online repository Models In Software Engineering (PROMISE) [32] for Apache Ant faults data. Figure 2 shows the distribution of faults in the four versions of Apache Ant. Color schemes are used in the bar chart to indicate different numbers of faults in a class. For example, in the ANT-V4 data set, 78% (565) classes have zero fault and 12.5% (91) classes have one fault.

To prepare the raw training data set, we associate software metrics (features) of the training data with faults (labels) provided in the PROMISE by the module's class name. With each modeling iteration, the training data set is expended and fault prediction models are rebuilt using techniques outlined in Section 2.2.

4. Results and Discussion

In this section, we answer the following research questions and discuss the study results.

- RQ1: What static software metrics can provide the best fault prediction result?

Table 2. Apache ant characteristics.

METRIC	VALUE	RATIO	VALUE
Total Files	228	Code /(Comment + Whitespace) Ratio	0.81
Avg Line Length	34	Code/Whitespace Ratio	4.16
Comment Lines	25,590	Code Lines Per File	113
Total Lines	57,462	Code/Comment Ratio	1.01
Code Lines	25,838	Code/Total Lines Ratio	0.45
Whitespace Lines	6213	Comment Lines Per File	112

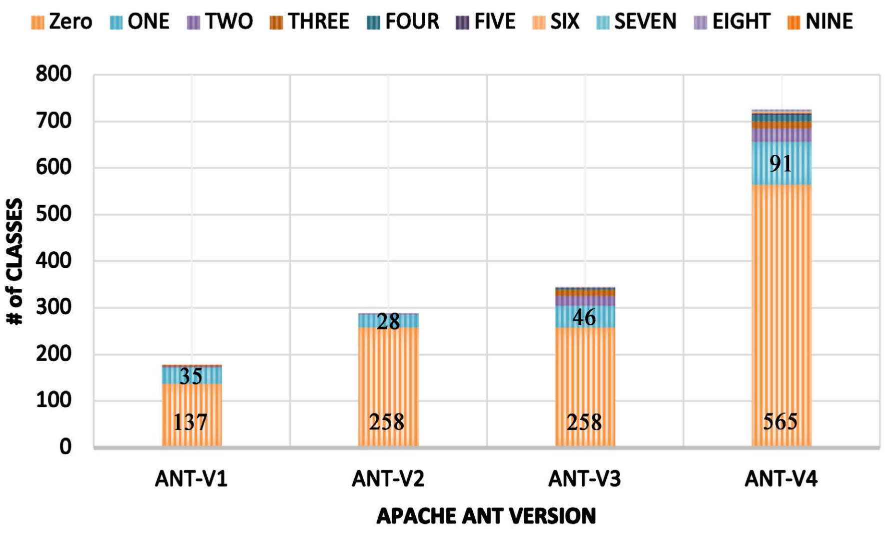

Figure 2. Apache ant faults distribution.

- RQ2: Which machine learning models give the best fault prediction results?
- RQ3: How well do prediction models perform across the continuous versions of the Apache Ant?

4.1. RQ1: Software Metrics

We build models with three sets of metrics (Table 1). For each model, the default, complex, and combined metrics are used as the training data, respectively. Table 3 shows a portion of the training data set for ANT-V1. For example, the "Module" column shows the class name; the "Weighted Methods per Class (WMC)" column is the sum of the complexities of all class methods; the "Bug" column shows the number of faults in the class. Figure 3 shows the performance of fault prediction models with all three sets of metrics. Their performance varies among different data sets. For example, the complex metrics outperform the

Table 3. ANT-V1.

Module	WMC	DIT	NOC	CBO	Cd/WS	Cd/File	Cm/File	Bug
Ant Class Loader	17	2	0	9	1.02	236	231	2
Build Event	11	2	0	7	3.79	53	97	0
Constants	0	1	0	0	3	3	1	0
Main	14	1	0	7	5.32	367	178	1
Project Helper	17	1	0	19	4.63	482	141	3
Zip	22	4	1	15	4.17	192	126	3

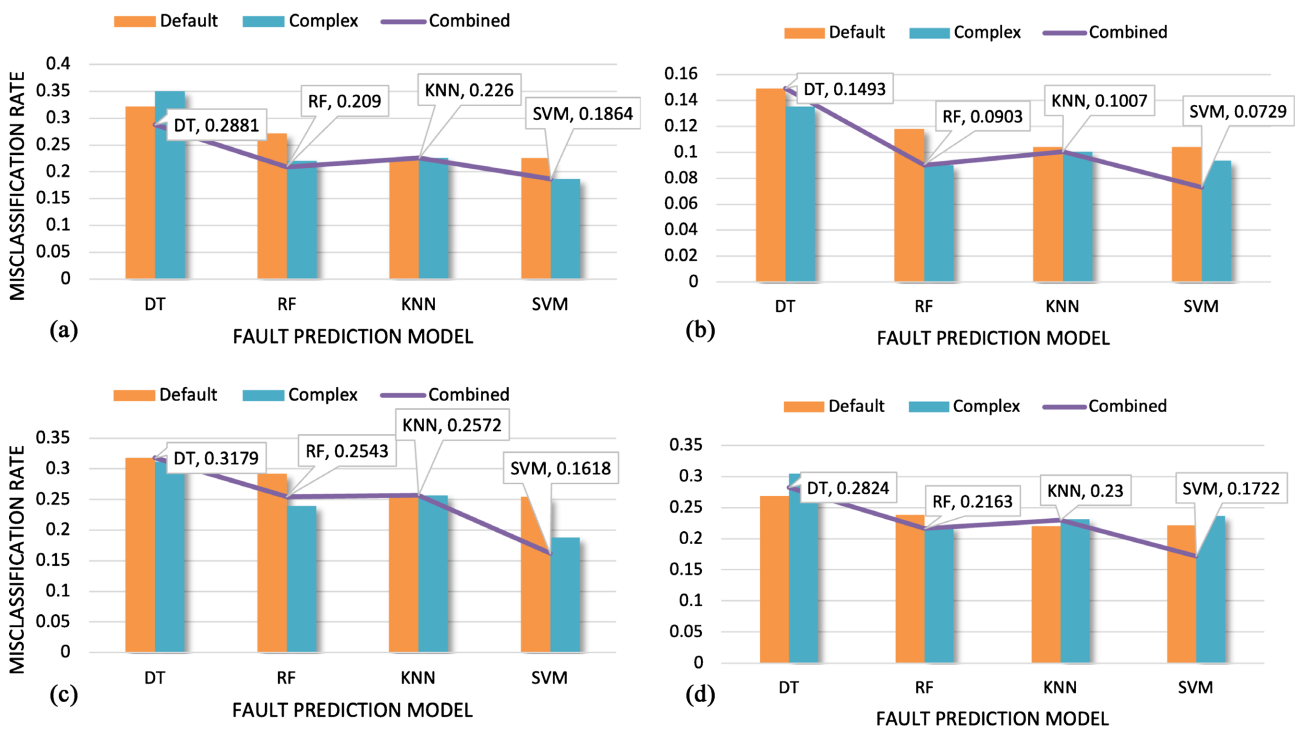

Figure 3. Software Metrics Performance. (a) ANT-V1; (b) ANT-V2; (c) ANT-V3; (d) ANT-V4.

default metrics in ANT-V1 with DT but fall short in ANT-V2 compared to default metrics. On average, models built with combined metrics has the lowest misclassification error (0.2).

4.2. RQ2: Fault Prediction Model Performance

To answer RQ2, we compare the performance of models built with individual Apache Ant versions in Figure 4. The performance of fault prediction models varies across Apache Ant versions. For example, RF has a misclassification error of 0.09 in ANT-V2 compared to a misclassification error of 0.254 in ANT-V3. Overall, SVM has the least misclassification error (0.148) followed by RF (0.192), KNN (0.203), and DT (0.259). Figure 4 shows models trained with ANT-V2 have the best performance with an average misclassification error of 0.103 compared to ANT-V1 (0.227), ANT-V3 (0.248), and ANT-V4 (0.225). It is our observation that for linearly separable spaces, KNN is preferred for its interpretability. KNN does require a larger data set for it to work accurately.

4.3. RQ3: Cross Program Training and Fault Prediction

To answer RQ3, we examine whether training data from other project versions can improve fault prediction performance. To prepare the expended data set, we construct a new data set with all previous training data sets. For example, ANT-DS-2 contains data for ANT-DS-1 plus ANT-V2; and ANT-DS-3 contains data for ANT-DS-2 plus ANT-V3. Figure 5 illustrates the performance of each fault prediction model with the expanded data set. Overall, the model prediction misclassification error is equivalent to the regular data set ($MC_{expanded} = 0.206$ v.s. $MC_{regular} = 0.2$). The misclassification error of models built with expanded data set outperform the regular data set in ANT-DS-3 ($MC_{expanded} = 0.198$ v.s. $MC_{regular} = 0.248$), ANT-DS-4 ($MC_{expanded} = 0.209$ v.s. $MC_{regular} = 0.225$) and underperform the regular data set in ANT-DS-2 ($MC_{expanded} = 0.214$ v.s. $MC_{regular} = 0.103$). The results imply in cases when training data for a subject is unavailable, we may

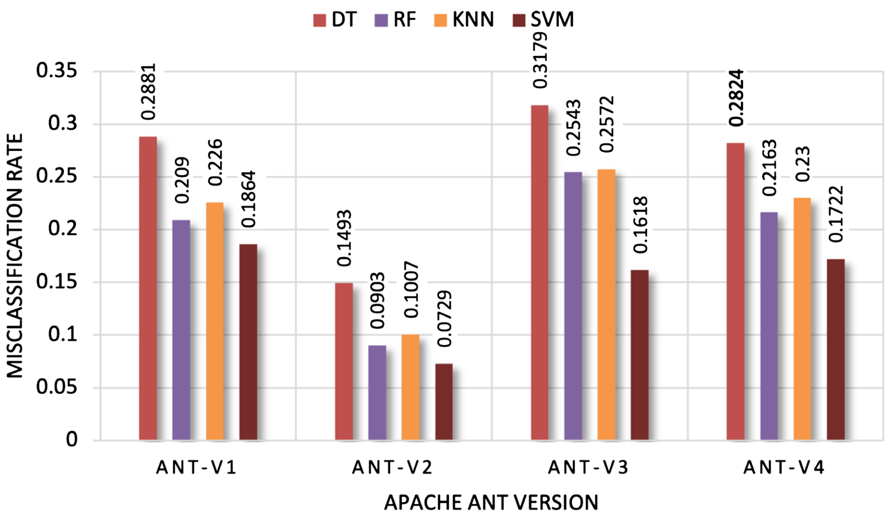

Figure 4. Fault prediction models performance.

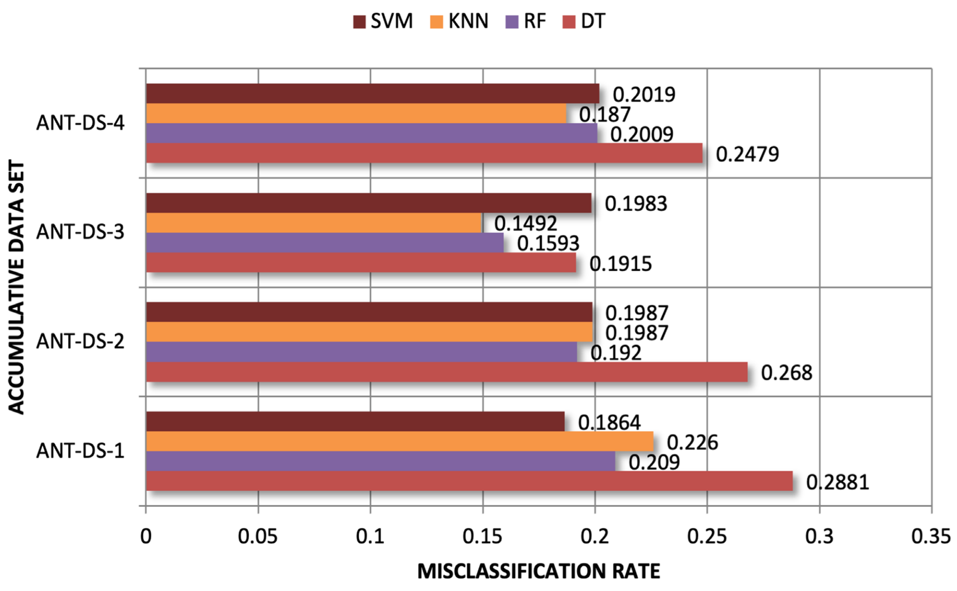

Figure 5. Fault prediction with expanded data set.

utilize training data of a different version of the same subject.

5. Discussions

In this section, we discuss the sensitivity analysis and the threats to validity of the empirical study.

5.1. Sensitivity Analysis

One challenge of using machine learning techniques is that we need to find proper values for the hyperparameters. To get a better fault prediction results, we try out different values to fine-tune the model. For example, Figure 6 shows the influence on the number of random trees used in RF. For KNN, a different number of neighbors (Figure 7) were selected to minimize the classification errors. Empirical data indicates for Apache Ant the best number of neighbors fall between 13 and 16.

5.2. Internal Validity

A threat to internal validity for this study is the possible faults in the implementation of our approach and the tools that we use to perform the evaluation. We control this threat by extensively testing our tools and verifying their results against a small program for which we can manually determine the correctness of the results. Another threat involves the selection of hyperparameters [33] used in machine learning techniques. We use the recommended settings for each modeling technique and conduct a sensitivity analysis to fine-tune the parameters. The accuracy of each fault prediction model may also be different with a different implementation. For example, the RF may report a different misclassification rate in scikit-learn [34] and weka [8] [35]. We choose the statistics and machine learning toolbox in MATLAB for its simplicity to use and its popularity (MATLAB has

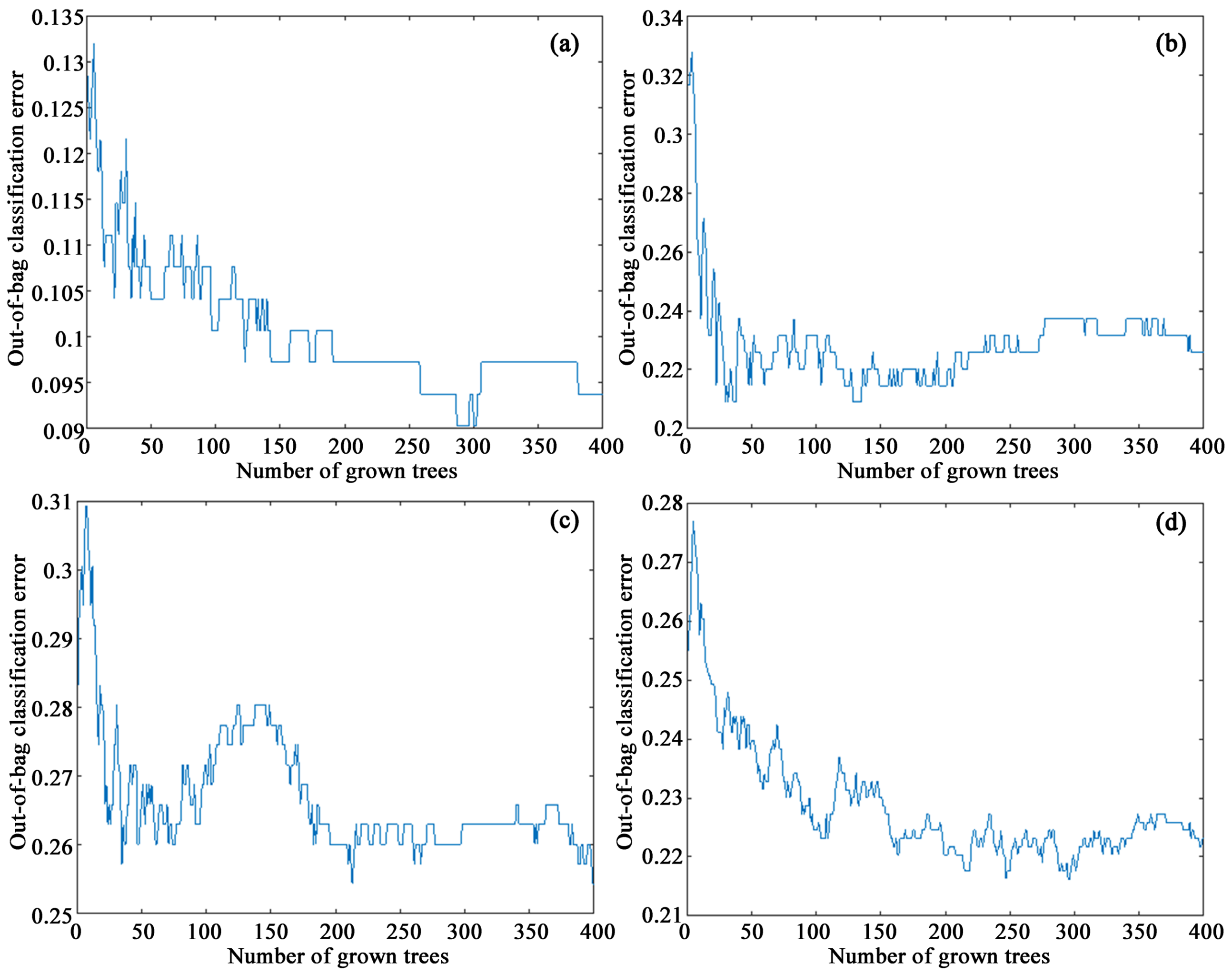

Figure 6. Number of RF Trees. (a) ANT-V1; (b) ANT-V2; (c) ANT-V3; (d) ANT-V4.

been widely used in both industry and academia).

5.3. External Validity

The primary threat to external validity for this study involves the representativeness of the selected subjects and modeling techniques. Other subjects may exhibit different characteristics and lead to other conclusions [36]. We reduce this threat by studying multiple versions of the subject program. In addition, we apply four different modeling techniques on seven data sets to generalize conclusions.

5.4. Construct Validity

The primary threat to construct validity involves the dataset and software metrics used in the study. To mitigate this threat, we use data sets that are publicly available, well understood, and widely used [32]. We have also applied well-known software metrics in the data set that is straightforward to compute and is less error-prone.

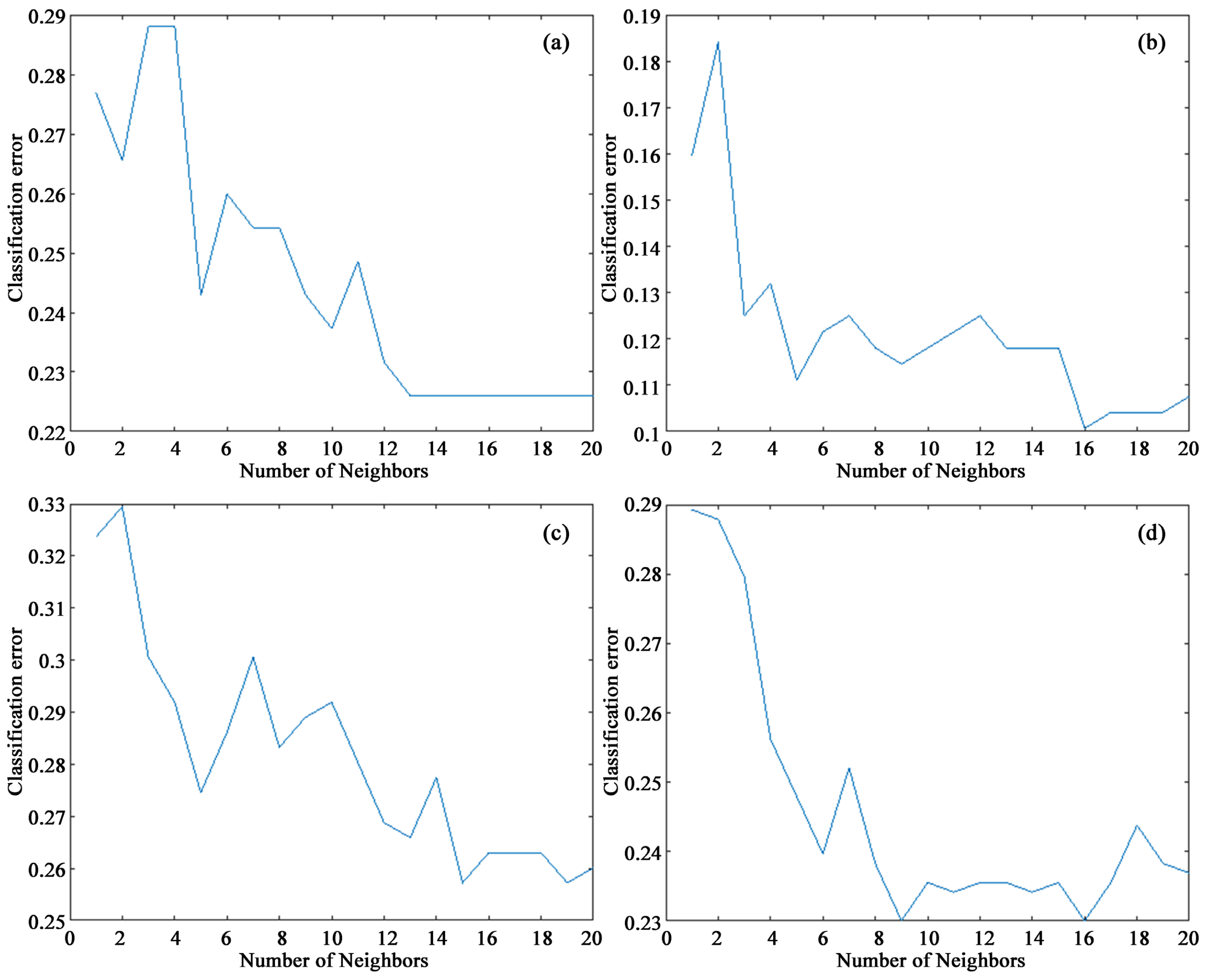

Figure 7. Number of Neighbors. (a) ANT-V1; (b) ANT-V2; (c) ANT-V3; (d) ANT-V4.

5.5. Limitations

The first limitation of this work is that our approach requires the source code to get the training data. In some cases, especially for a legacy program, the source code may not always be available [2]. Second, when preparing for the training data, it is not fully automated. Our approach first extracts static metrics from the source code, and then we manually combine the PROMISE labels (faults) to get the training data set. One solution is to automate the fault prediction model construction as part of the continuous integration (CI) [37]. We can leverage the fault information from the issue tracker to automatically append the labels to the training data set.

6. Conclusion

We conduct an empirical study to examine the effectiveness of building fault prediction models with static software metrics. We examine the effectiveness of metrics to build fault prediction models. We study four different types of fault prediction models with four continuous versions of the Apache Ant. We evaluate the performance of fault prediction models across multiple Apache Ant versions. Our results suggest the fault prediction models built with combined software metrics have the lowest overall misclassification error (0.2). Among all fault prediction models, SVM has the least misclassification error (0.148). Lastly, our results show the fault prediction models built with the expanded data set are equally powerful. In cases when training data for a subject is unavailable, we may utilize training data of a different version of the same subject.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Ammann, P. and Offutt, J. (2016) Introduction to Software Testing. Cambridge University Press, Cambridge. https://doi.org/10.1017/9781316771273

[2] Han, X., Carroll, D. and Yu, T. (2019) Reproducing Performance Bug Reports in Server Applications: The Researchers' Experiences. *Journal of Systems and Software*, **156**, 268-282. https://doi.org/10.1016/j.jss.2019.06.100

[3] Sen, K., Marinov, D. and Agha, G. (2005) Cute: A Concolic Unit Testing Engine for C. *ACM SIGSOFT Software Engineering Notes*, **30**, 263-272. https://doi.org/10.1145/1095430.1081750

[4] Dawson, M., Burrell, D.N., Rahim, E. and Brewster, S. (2010) Integrating Software Assurance into the Software Development Life Cycle (SDLC). *Journal of Information Systems Technology and Planning*, **3**, 49-53.

[5] Boberg, J. (2008) Early Fault Detection with Model-Based Testing. *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, Victoria, 27 September 2008, 9-20. https://doi.org/10.1145/1411273.1411276

[6] Dore, T.L. (2004) The Relationships between Job Characteristics, Job Satisfaction, and Turnover Intention among Software Developers. Argosy University/Orange County, Atlanta.

[7] Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B. and Zimmermann, T. (2019) Software Engineering for Machine Learning: A Case Study. 2019 *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice* (*ICSE-SEIP*), Montreal, 25-31 May 2019, 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[8] Yu, T., Wen, W., Han, X. and Hayes, J.H. (2016) Predicting Testability of Concurrent Programs. 2016 *IEEE International Conference on Software Testing, Verification and Validation* (*ICST*), 11-15 April 2016, Chicago, 168-179. https://doi.org/10.1109/ICST.2016.39

[9] Rathore, S.S. and Kumar, S. (2017) An Empirical Study of Some Software Fault Prediction Techniques for the Number of Faults Prediction. *Soft Computing*, **21**, 7417-7434. https://doi.org/10.1007/s00500-016-2284-x

[10] Yu, T., Wen, W., Han, X. and Hayes, J.H. (2018) Conpredictor: Concurrency Defect Prediction in Real-World Applications. *IEEE Transactions on Software Engineering*, **45**, 558-575. https://doi.org/10.1109/TSE.2018.2791521

[11] Schach, S.R. (2007) Object-Oriented and Classical Software Engineering. Vol. 6,

McGraw-Hill, New York.

[12] Chidamber, S.R. and Kemerer, C.F. (1994) A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, **20**, 476-493. https://doi.org/10.1109/32.295895

[13] Aggarwal, K., Singh, Y., Kaur, A. and Malhotra, R. (2006) Empirical Study of Object-Oriented Metrics. *Journal of Object Technology*, **5**, 149-173.

[14] Bardas, A.G. (2010) Static Code Analysis. *Romanian Economic Business Review*, **4**, 99-107.

[15] Uh, G.-R., Cohn, R., Yadavalli, B., Peri, R. and Ayyagari, R. (2006) Analyzing Dynamic Binary Instrumentation Overhead. WBIA Workshop at ASPLOS, Citeseer.

[16] Potdar, A.M., Narayan, D., Kengond, S. and Mulla, M.M. (2020) Performance Evaluation of Docker Container and Virtual Machine. *Procedia Computer Science*, **171**, 1419-1428. https://doi.org/10.1016/j.procs.2020.04.152

[17] Zheng, A. and Casari, A. (2018) Feature Engineering for Machine Learning: Principles and techniques for Data Scientists. O'Reilly Media, Inc., Sebastopol.

[18] Chandrashekar, G. and Sahin, F. (2014) A Survey on Feature Selection Methods. *Computers & Electrical Engineering*, **40**, 16-28. https://doi.org/10.1016/j.compeleceng.2013.11.024

[19] Brijain, M., Patel, R., Kushik, M. and Rana, K. (2014) A Survey on Decision Tree Algorithm for Classification. *International Journal of Engineering Development and Research*, **2**, 1-5.

[20] Jost, L. (2006) Entropy and Diversity. *Oikos*, **113**, 363-375. https://doi.org/10.1111/j.2006.0030-1299.14714.x

[21] Biau, G. and Scornet, E. (2016) A Random Forest Guided Tour. *Test*, **25**, 197-227. https://doi.org/10.1007/s11749-016-0481-7

[22] Noble, W.S. (2006) What Is a Support Vector Machine? *Nature Biotechnology*, **24**, 1565-1567. https://doi.org/10.1038/nbt1206-1565

[23] L. E. Peterson (2009) K-Nearest Neighbor. *Scholarpedia*, **4**, 1883. https://doi.org/10.4249/scholarpedia.1883

[24] Moisen, G.G. (2008) Classification and Regression Trees. In: Jørgensen, S.E. and Fath, B.D., Eds., *Encyclopedia of Ecology*, Vol. 1, Oxford, Elsevier, 582-588. https://doi.org/10.1016/B978-008045405-4.00149-X

[25] Hawkins, D.M. (2004) The Problem of Overfitting. *Journal of Chemical Information and Computer Sciences*, **44**, 1-12. https://doi.org/10.1021/ci0342472

[26] Lee, T., Nam, J., Han, D., Kim, S. and In, H.P. (2011) Micro Interaction Metrics for Defect Prediction. *Proceedings of the* 19*th ACM SIGSOFT Symposium and the* 13*th European Conference on Foundations of Software Engineering*, Szeged, 5-9 September 2011, 311-321. https://doi.org/10.1145/2025113.2025156

[27] Alpaydin, E. (2020) *Introduction to Machine Learning*. MIT Press, Cambridge. https://doi.org/10.7551/mitpress/13811.001.0001

[28] Zadrozny, B. (2004) Learning and Evaluating Classifiers under Sample Selection Bias. *Proceedings of the* 21*st International Conference on Machine Learning*, Banff, 4-8 July 2004, 114. https://doi.org/10.1145/1015330.1015425

[29] Code Analyzer (2013). https://sourceforge.net/projects/codeanalyze-gpl/

[30] (2013) MATLAB and Statistics Toolbox Release. The MathWorks, Inc., Natick.

[31] Nistor, A., Song, L., Marinov, D. and Lu, S. (2013) Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. 2013 35*th International Conference*

on Software Engineerin (*ICSE*), San Francisc, 18-26 May 2013, 562-571. https://doi.org/10.1109/ICSE.2013.6606602

[32] Sayyad Shirabad, J. and Menzies, T. (2005) The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Ottawa.

[33] Probst, P., Boulesteix, A.-L. and Bischl, B. (2019) Tunability: Importance of Hyperparameters of Machine Learning Algorithms. *The Journal of Machine Learning Research*, **20**, 1934-1965.

[34] Han, X., Yu, T. and Pradel, M. (2021) Confprof: White-Box Performance Profiling of Configuration Options. *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, Virtual, 19-23 April 2021, 1-8. https://doi.org/10.1145/3427921.3450255

[35] Han, X., Yu, T. and Lo, D. (2018) Perflearner: Learning from Bug Reports to Understand and Generate Performance Test Frames. 2018 33*rd IEEE/ACM International Conference on Automated Software Engineering (ASE*), Montpellier, 3-7 September 2018, 17-28. https://doi.org/10.1145/3238147.3238204

[36] Sjøberg, D.I., Hannay, J.E., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N.-K. and Rekdal, A.C. (2005) A Survey of Controlled Experiments in Software Engineering. *IEEE Transactions on Software Engineering*, **31**, 733-753. https://doi.org/10.1109/TSE.2005.97

[37] Fitzgerald, B. and Stol, K.-J. (2017) Continuous Software Engineering: A Roadmap and Agenda. *Journal of Systems and Software*, **123**, 176-189. https://doi.org/10.1016/j.jss.2015.06.063