Performance Analysis of Machine Learning Approaches in Software Complexity Prediction

Sayed Moshin Reza, Md. Mahfujur Rahman, Hasnat Parvez, Omar Badreddin, and Shamim Al Mamun

Abstract Software design is one of the core concepts in software engineering. This covers insights and intuitions of software evolution, reliability, and maintainability. Effective software design facilitates software reliability and better quality management during development which reduces software development cost. Therefore, it is required to detect and maintain these issues earlier. Class complexity is one of the ways of detecting software quality. The objective of this paper is to predict class complexity from source code metrics using machine learning (ML) approaches and compare the performance of the approaches. In order to do that, we collect ten popular and quality maintained open source repositories and extract 18 source code metrics that relate to complexity for class-level analysis. First, we apply statistical correlation to find out the source code metrics that impact most on class complexity. Second, we apply five alternative ML techniques to build complexity predictors and compare the performances. The results report that the following source code metrics: Depth inheritance tree (DIT), response for class (RFC), weighted method count (WMC), lines of code (LOC), and coupling between objects (CBO) have the most impact on class complexity. Also, we evaluate the performance of the techniques, and results show that random forest (RF) significantly improves accuracy without providing additional false negative or false positive that work as false alarms in complexity prediction.

S. Moshin Reza · O. Badreddin
University of Texas, Austin, TX, USA
e-mail: sreza3@miners.utep.edu

O. Badreddin
e-mail: obbadreddin@utep.edu

Md. Mahfujur Rahman (✉)
Daffodil International University, Dhaka, Bangladesh
e-mail: mrrajuiit@gmail.com

H. Parvez · S. Al Mamun
Jahangirnagar University,Dhaka, Bangladesh
e-mail: hasnatiit847@gmail.com

S. Al Mamun
e-mail: shamim@juniv.edu

M. S. Kaiser et al. (eds.), *Proceedings of International Conference on Trends in Computational and Cognitive Engineering*, Advances in Intelligent Systems and Computing 1309, https://doi.org/10.1007/978-981-33-4673-4_3

Keywords Software complexity · Software quality · Machine learning · Software design · Software reliability

1 Introduction

Software design is a process of creating software artifacts, primitive components, and constraints. Effective software design with object oriented structures facilitates better software quality, reusability, and maintainability [1]. One of the quality factors is complexity. This quality attribute is determined by many factors related to code structures, object-oriented properties, and source code metrics [2]. The less the complexity of a software, the less the cost of software development will be [3, 4]. This motivates us to research on software complexity prediction.

In software life cycle, the more the complexity is, maintenance becomes costly, unpredictable, human-intensive activity [2]. Moreover, high maintenance efforts often affect the software sustainability that many software systems become unsustainable over time [5, 6]. Therefore, software redesign becomes an essential step where complexity of the software needs to be reduced. Such action will enhance software maintainability and reduce the associated costs [7, 8]. Having set the importance of complexity detection for software redesign, we are motivated to predict class-level complexity from source code metrics.

Some studies introduced McCabe complexity, a widely accepted metrics developed by Thomas McCabe to show the level of software complexity [9]. Another approach on calculation of software complexity was based on counting number of operators and operands in software. But the calculation and counting process of total operators and operands are tedious [10].

In this paper, we use machine learning techniques to build complexity predictor. The reason behind using machine learning to get rid of manual process or code rules to detect class complexity. Also, successful research on detecting software defect, vulnerability using ML techniques motivate us [11, 12]. We use five ML classifiers, analyze the performance of the classifiers, and report the best technique in complexity prediction.

The rest of the paper is organized as follows. We present literature reviews in Sect. 2. We describe research methodology in Sect. 3. Results and evaluation are discussed in Sect. 4 and finally, we conclude the paper in Sect. 5.

2 Literature Review

Several research on code quality from source code metrics includes fault-prone modules detection [1], early detection of vulnerabilities [11], improvement of network software security [13, 14], software redesign [9], etc. All of these researches are targeted to reduce the maintenance effort and cost during the software development.

Chowdhury et al. investigate the efficacy of applying cohesion, complexity, and coupling metrics to automatically predict vulnerability and complexity entities [11]. This study used machine learning and statistical approaches to predict vulnerability that learn from the cohesion, complexity, and coupling metrics. The results indicate that structural information from the non-security realm such as cohesion, complexity, and coupling is useful in vulnerability prediction which minimize the maintenance effort.

Another study proposed by Briand et al. [15] analyzed correspondence between object-oriented metrics and fault proneness. This research results are created based upon few number of classes analysis. Gegick et al. [16] developed a heuristic model to predict vulnerable components and complexity. The model was successful on a large commercial telecommunications software and predicted vulnerable components with 8% false positive rate and 0% false negative rate.

In this research, we analyze source code metrics in relation to complexity. Also, we apply ML techniques to predict complexity from source code metrics.

3 Research Methodology

This research has two main goals. First, analyze source code metrics to what extent it is possible to predict complexity. Second, report the best ML approaches evaluating relative effectiveness in prediction of complexity from source code metrics. The details of our research questions, datasets, and machine learning approaches are discussed in the following subsections.

3.1 Research Questions

This research is focused on answering two primary research questions.

Research Question 1: How source code metrics are correlated with quality attribute: class complexity?

This question reveals the relationships between complexity and source code metrics, such as number of attributes, and lines of code. To answer this question, we apply statistical correlation on 18 source code metrics and complexity collected from ten different source code repositories to find out the relationship.

Research Question 2: How accurately can machine learning approaches predict class complexity from source code metrics?

This question is targeted to find out the accuracy of machine learning approaches in class-level complexity detection. We apply five machine learning techniques and evaluate the performance. This question reveals the best technique in detecting class complexity from source code metrics.

3.2 Proposed Research Framework

The proposed research is build upon three steps. First, extracting source code metrics and complexity from classes of large code bases. Second, prepare the dataset for complexity prediction by applying data cleaning process. Third, apply ML techniques and evaluate to find out the best one.

For the first step, we extract source code metrics and quality feature: complexity from a large number of classes. The details of dataset creation process are discussed in Sect. 3.3. In the second step, we apply data cleaning process to get better learned ML model. Uncleaned data fed into machine learning techniques may result to a bad model creation [17]. The details of the process are discussed in Sect. 3.4. For the final step, we select several ML techniques and train the dataset to detect highly complex classes. We also assess ML prediction effectiveness using performance metrics. The detailed picture of the study is shown in Fig. 1.

3.3 Dataset Collection

Dataset for complexity prediction needs diverse set of repositories. We search codebase repositories using ModelMine tool [18] with the following criteria; a repository with primary language Java, a minimum of 5000 commits, at least 100 active contributors, a minimum of 3000 stars and 500 forks. The selected repositories are shown in Table 1 with repository metadata information.

To validate the diversity of repositories, we consider high number of stars and forks as a proxy for popularity of repositories and high number of commits as a proxy of maintenance. Also, we consider repository size as follows: low (1–1000

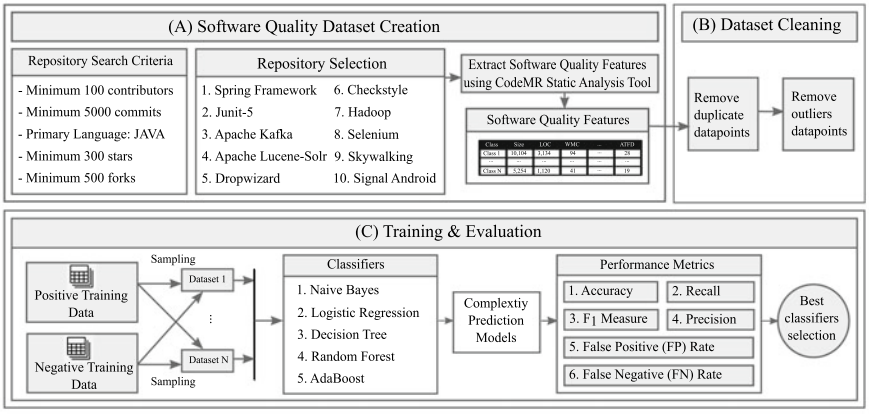


Fig. 1 Proposed methodology

Table 1 Selected repositories with metadata information

Serial	Repository name	Repository link	Commits	Contributors	Stars	Forks	Lines of code	Classes
1	Spring framework	https://github.com/spring-projects/spring-framework	21,154	491	38,200	25,800	232,447	5628
2	Junit5	https://github.com/junit-team/junit5/	6286	146	4000	899	16,856	659
3	Apache Kafka	https://github.com/apache/kafka	7787	691	16,300	8700	119,299	2463
4	Apache Lucene-Solr	https://github.com/apache/lucene-solr	33,899	194	3600	2500	602,185	8850
5	Dropwizard	https://github.com/dropwizard/dropwizard	5448	345	7700	3200	14,268	508
6	Checkstyle	https://github.com/checkstyle/checkstyle	9408	232	5400	7400	26,030	454
7	Hadoop	https://github.com/apache/hadoop	24,001	280	10,600	6600	695,992	10,496
8	Selenium	https://github.com/SeleniumHQ/selenium	25,354	518	18,100	5800	36,031	1175
9	Skywalking	https://github.com/apache/skywalking	5753	245	14,000	4100	61,588	2531
10	Signal-Android	https://github.com/signalapp/Signal-Android	5777	206	13,400	3400	116,268	2861

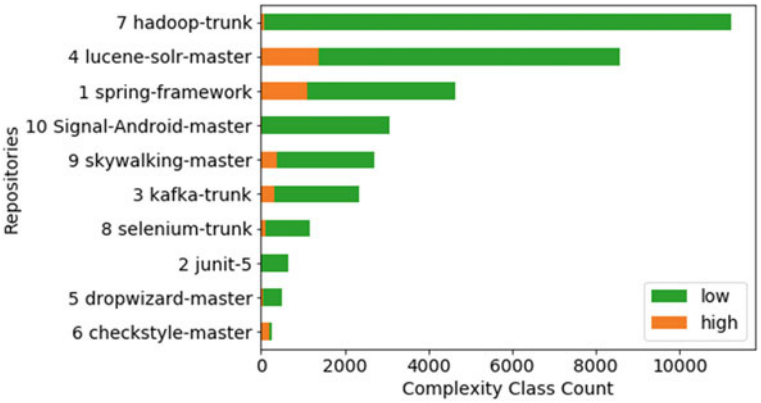


Fig. 2 Complexity distribution among repositories

classes), medium (1001–5000 classes), and high (more than 5000 classes) in size. This selection implies diversity in complexity of classes. Figure 2 shows number of complexity classes against each selected repository where three of them are selected from low, four of them are selected from medium, and rest of them are selected from high volume of category.

After extracting code repositories, we extract source code metrics for each class in the repository using CODEMR tool [19]. The tool provides 18 unique source code metrics for each class. The details of the source code metrics are described in Table 2. The target variable data is collected also for each class using same tool with different process. The data is then combined using the class file name for training and testing purpose.

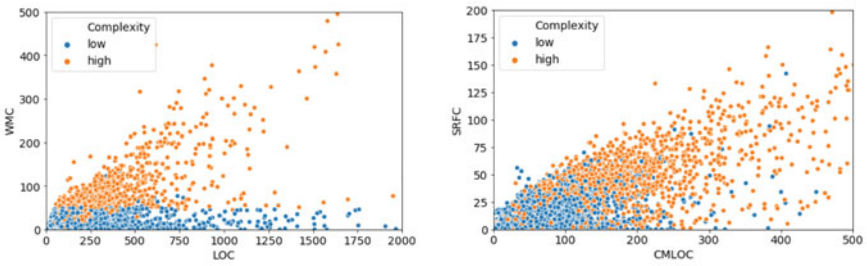
3.4 Dataset Cleaning and Analysis

Data cleaning is critically important step for the complexity prediction. To get optimistic performance result of ML approaches, we clean the data in two stages. First, by identifying column variables that have single value or very few unique values. In this stage, we also remove the duplicate observations. In second stage, we apply box plot for each source code metrics and find the outliers. This technique helps to remove the bias datapoints from the dataset.

After cleaning the dataset, we have come up with much more differential and clear dataset for complexity prediction. Figure 3a visualizes the relationship between weighted method count, lines of code, and complexity. Figure 3b visualizes the relationship between response for class, method lines of code, and complexity.

Table 2 Source code metrics

No	Source code metric name	Description
1	Class lines of code (CLOC)	The number of all non-commented and nonempty lines of a class
2	Weighted method count (WMC)	The weighted sum of all class' methods
3	Depth of inheritance tree (DIT)	The location of a class in the inheritance tree
4	Number of children (NOC)	The number of associated sub-classes of a class
5	Coupling between object classes (CBO)	The number of classes that another class is coupled to
6	Response for a class (RFC)	The number of the methods that can be potentially invoked in response by an object of a class
7	Simple response for a class (SRFC)	The number of the methods that can be potentially invoked in response by an object of a particular class
8	Lack of cohesion of methods (LCOM)	Measure how methods of a class are related to each other
9	Lack of cohesion among methods (LCAM)	Measure cohesion based on parameter types of methods
10	Number of fields (NOF)	The number of fields (attributes) in a class
11	Number of methods (NOM)	The number of methods in a class
12	Number of static fields (NOSF)	The number of static fields in a class
13	Number of static methods (NOSM)	The number of static methods in a class
14	Specialization index (SI)	Measures the extent to which sub-classes override their ancestor's classes
15	Class-methods lines of code (CMLOC)	Total number of all nonempty, non-commented lines of methods inside a class
16	Number of overridden methods (NORM)	The number of methods that are inherit from a super-class and has return type as the method that it overrides
17	Lack of tight class cohesion (LTCC)	Measures cohesion between the public methods of a class and subtract from 1
18	Access to foreign data (ATFD)	The number of classes whose attributes are directly or indirectly reachable from the a class



(a) WMC, LOC vs Complexity (b) SRFC, CMLOC vs Complexity

Fig. 3 Relationship of input variables with target variable

3.5 Machine Learning Classifiers and Evaluation Metrics

This subsection provides a brief overviews of five alternative machine learning classifiers used to build class complexity predictors. The machine learning classifiers are as follows: (1) Naive Bayes (NB), (2) Logistic Regression (LR), (3) Decision Tree (DT), (4) Random Forest (RF), and (5) AdaBoost (AB). These classifiers are well-known classifiers in building vulnerability predictors and used in several similar research [11, 20, 21]. The statistical performance of selected ML classifiers is calculated by performing ten-fold cross-validation technique. Cross-validation is a technique for assessing how accurately a predictive model will perform in practice after generating the model [22]. The objective of such operation is to reduce the variability of the results.

4 Result and Discussion

This section describes the results of correlation analysis, complexity prediction using ML models and compares the performance of ML classifiers.

4.1 Correlation Results

The results of Pearson correlation reveal the impact of source code metrics on quality attribute: complexity. Figure 4 visualizes the correlation between source code metrics and complexity. It is clear in the figure that not any single metric highly impact on complexity. This quality attribute is formed based on a combined behavior of source code metrics. Among the code metrics, DIT, SRFC, RFC, WMC, CMLOC, and CBO have moderately high impact on complexity. Generally, classes with higher number of WMC, LOC, or DIT associated with high number of defect in the software,

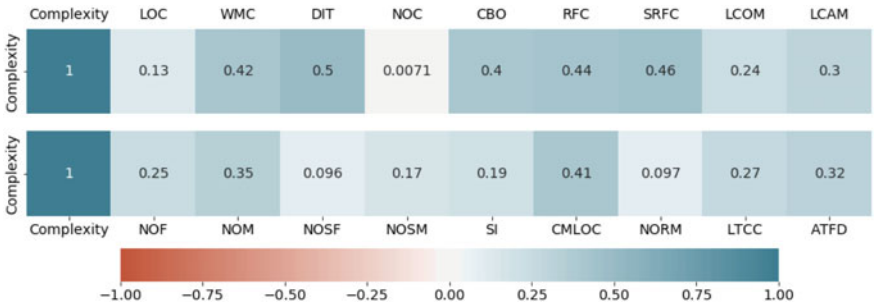


Fig. 4 Correlation among source code metrics and quality attribute

and it becomes hard to maintain over time [12]. This issue is also mentioned by Subramanyam et al. that DIT and CBO have influenced class complexity [12]. In another research, Chowdhury et al. experimentally showed that WMC, DIT, RFC, and CBO code-level metrics are strongly correlated to vulnerabilities which are directly generated from file complexity [23]. This answers research question 1.

4.2 Performance Results

In this subsection, we discuss the performance of ML complexity predictors. We use the following evaluation metrics: accuracy, precision, recall, F_1 score, FP rate, and FN rate to compare the performances. At first, we generate confusion matrices from the validation set. Table 3 visualizes the confusion matrices of the classifiers for predicting software complexity.

We evaluate the techniques using following metrics: accuracy, precision, recall, F_1 score, FP and FN rate, and the results are visualized in Table 4. Accuracy and precision are most used measurement in comparing the performance. Table 4 and Fig. 5 shows the accuracy and precision value of the selected classifiers. The result implies decision tree and random forest classifier have the highest accuracy and precision than other classifiers. We also observe random forest has highest recall and F_1 score.

Table 3 Confusion matrices of classifiers for predicting software complexity

Classifier names	Naive Bayes		Logistic regression		Decision tree		Random forest		Ada Boost	
	Predicted									
Actual	Low	High	Low	High	Low	High	Low	High	Low	High
Low	6416	475	6766	125	6832	59	6820	71	6813	78
High	330	434	173	591	223	541	58	706	82	682

Table 4 Prediction performance of machine learning models

Serial	Classifier name	Accuracy	Precision	Recall	F1 score	FP rate	FN rate
1	Naive Bayes	89	71	75	73	6.88	42.11
2	Logistic regression	96	91	86	88	1.44	26.08
3	Decision tree	98	95	96	96	0.90	7.53
4	Random forest	98	95	99	97	1.00	1.95
5	Ada Boost	97	94	93	94	1.13	12.27

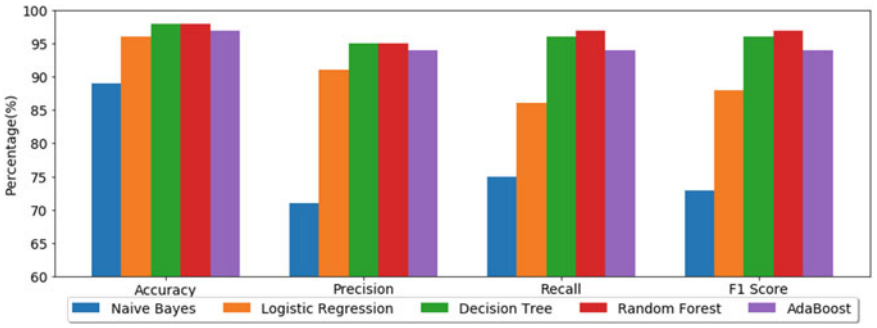


Fig. 5 Relative performance of ML classifiers

However, we evaluate the classifiers with another set of metrics: false positive rate and false negative rate. The higher the FN rate, the model generates more false alarms. This implies high complex classes are detected as low complex classes which are very risky. Figure 6 shows the relative performance of classifiers in terms of false positive rate and false negative rate. One may have to tolerate many false positives to ensure reduced number of complex classes left undetected. As such, if the target is to predict a larger percentage of high complexity class files, then Naive Bayes classifier can be evaluated favorably although in overall prediction, random forest and decision tree classifier'performance are better.

On the other hand, if the target is to predict a fewer percentage of high complex files as low to avoid risk, then obviously random forest might be the good choice as it has the lowest false negative rate. We focus more on false negative rate to reduce the risk of detecting high complex class as low. RF results indicate that it is much better model in prediction of complexity because of its bootstrapping random re-sample technique and working with significant elements. On the other hand, DT is working with all elements, and as a result, it creates more false alarms than RF. Therefore, random forest is the best complexity predictor among selected ML techniques.

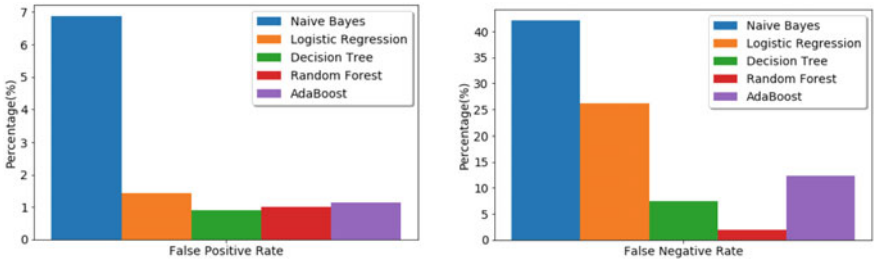


Fig. 6 Relative FP and FN rate of ML classifiers

5 Conclusion

In this study, we analyze the software source code metrics which are mostly impacted the class complexity. It is undoubtedly necessary to take proper action before classes become more complex. Otherwise, it will become more expensive to test and fix if large number of classes become highly complex. To reduce such risk and cost, it is necessary to build complexity predictor.

We start with extracting 38,778 classes of dataset with 18 source code metrics, we use five different machine learning approaches to train the dataset to classify high or low complex classes. In evaluation, we compare the performance of the approaches using the evaluation metrics. The result shows that RF classifier predicts high complexity classes with an accuracy of 98% and also having lowest FN rate of 1.95. Therefore, random forest is considered as best classifier to detect class complexity. In summary, we have made the following observation from our study. First, cross-validation implies low variance of performance metrics detecting software complexity. Second, FN rate needs to be reduced as much as possible to avoid the risk of detecting high complex class as low complex class.

Finally, the observations and results from this study can be useful in software quality research. Using ML automatic prediction on code quality will allow quality managers, practitioners to take preventive actions against bad quality, faults, and errors. Such proactive actions will allow software redesign and maintenance which ensure better software quality during the development.

References

1. Alakus, T.B., Das, R., Turkoglu, I.: An overview of quality metrics used in estimating software faults. In: 2019 International Artificial Intelligence and Data Processing Symposium (IDAP), pp. 1–6. IEEE (2019)
2. Ogheneovo, E.E., et al.: On the relationship between software complexity and maintenance costs. J. Comput. Commun. **2**(14), 1 (2014)

3. Yu, S., Zhou, S.: A survey on metric of software complexity. In: 2010 2nd IEEE International Conference on Information Management and Engineering, pp. 352–356. IEEE (2010)
4. Reza, S.M., Rahman, M.M., Parvez, M.H., Shamim Kaiser, M., Mamun, S.A.: Innovative approach in web application effort & cost estimation using functional measurement type. In: 2015 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT), pp. 1–7. IEEE (2015)
5. Durdik, Z., Klatt, B., Koziolek, H., Krogmann, K., Stammel, J., Weiss, R.: Sustainability guidelines for long-living software systems. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 517–526. IEEE (2012)
6. Reza, S.M., Rahman, M.M., Mamun, S.A.: A new approach for road networks-a vehicle xml device collaboration with big data. In: 2014 International Conference on Electrical Engineering and Information & Communication Technology, pp. 1–5. IEEE (2014)
7. Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M.: Graph-based analysis and prediction for software evolution. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 419–429. IEEE (2012)
8. Paul, M.C., Sarkar, S., Rahman, M.M., Reza, S.M., Shamim Kaiser, M.: Low cost and portable patient monitoring system for e-health services in Bangladesh. In: 2016 International Conference on Computer Communication and Informatics (ICCCI), pp. 1–4. IEEE (2016)
9. Moreno-León, J., Robles, G., Román-González, M.: Comparing computational thinking development assessment scores with software complexity metrics. In: 2016 IEEE Global Engineering Education Conference (EDUCON), pp. 1040–1045. IEEE (2016)
10. Singh, G., Singh, Dilbag, Singh, V.: A study of software metrics. IJCEM Int. J. Comput. Eng. Manage. **11**, 22–27 (2011)
11. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Arch. **57**(3), 294–313 (2011)
12. Subramanyam, R., Krishnan, Mayuram S.: Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. IEEE Trans. Softw. Eng. **29**(4), 297–310 (2003)
13. Moshtari, S., Sami, A., Azimi, M.: Using complexity metrics to improve software security. Comput. Fraud Sec. **2013**(5), 8–17 (2013)
14. Rahman, S., Sharma, T., Reza, S.M., Rahman, M.M., Kaiser, M.S., et al.: Pso-nf based vertical handoff decision for ubiquitous heterogeneous wireless network (uhwn). In: 2016 International Workshop on Computational Intelligence (IWCI), pp. 153–158. IEEE (2016)
15. Briand, L.C., Wüst, J., Daly, J.W., Victor Porte, D.: Exploring the relationships between design measures and software quality in object-oriented systems. J. Syst. Softw. **51**(3), 245–273 (2000)
16. Gegick, M., Williams, L., Osborne, J., Vouk, M.: Prioritizing software security fortification throughcode-level metrics. In: Proceedings of the 4th ACM workshop on Quality of protection, QoP '08, pp. 31–38. Association for Computing Machinery, New York, NY, USA, Oct 2008
17. Munappy, A., Bosch, J., Olsson, H.H., Arpteg, A., Brinne, B.: Data management challenges for deep learning. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 140–147. IEEE (2019)
18. Reza, S.M., Badreddin, O., Rahad, K.: Modelmine: a tool to facilitate mining models from open source repositories. In: 2020 ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS). ACM (2020)
19. Shaheen, A., Qamar, U., Nazir, A., Bibi, R., Ansar, M., Zafar, I.: Oocqm: object oriented code quality meter. In: International Conference on Computational Science/Intelligence & Applied Informatics, pp. 149–163. Springer (2019)
20. Zhang, Y., Lo, D., Xia, X., Xu, B., Sun, J., Li, S.: Combining software metrics and text features for vulnerable file prediction. In: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 40–49. IEEE (2015)
21. Jimenez, M., Rwemalika, R., Papadakis, M., Sarro, F., Traon, Y.L., Harman, M.: The importance of accounting for real-world labelling when predicting software vulnerabilities. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 695–705 (2019)

22. Yadav, S., Shukla, S.: Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification. In: 2016 IEEE 6th International Conference on Advanced Computing (IACC), pp. 78–83. IEEE (2016)
23. Chowdhury, I., Zulkernine, M.: Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 1963–1969 (2010)