Scientific Research Publishing

Journal of Software Engineering and Applications, 2020, 13, 91-103
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Can We Predict the Change in Code in a Software Product Line Project?

Yasser Ali Alshehri

Computer Science and Engineering Department, Yanbu University College, Royal Commission, Yanbu, Saudi Arabia
Email: alshehriya@rcyci.edu.sa

How to cite this paper: Alshehri, Y.A. (2020) Can We Predict the Change in Code in a Software Product Line Project? *Journal of Software Engineering and Applications*, **13**, 91-103.
https://doi.org/10.4236/jsea.2020.136007

Received: March 28, 2020
Accepted: May 25, 2020
Published: May 28, 2020

Abstract

Software programs are always prone to change for several reasons. In a software product line, the change is more often as many software units are carried from one release to another. Also, other new files are added to the reused files. In this work, we explore the possibility of building a model that can predict files with a high chance of experiencing the change from one release to another. Knowing the files that are likely to face a change is vital because it will help to improve the planning, managing resources, and reducing the cost. This also helps to improve the software process, which should lead to better software quality. Also, we explore how different learners perform in this context, and if the learning improves as the software evolved. Predicting change from a release to the next release was successful using logistic regression, J48, and random forest with accuracy and precision scored between 72% to 100%, recall scored between 74% to 100%, and F-score scored between 80% to 100%. We also found that there was no clear evidence regarding if the prediction performance will ever improve as the project evolved.

Keywords

Software Change Proneness, Software Quality, Machine Learning, Decision Tree J48, Logistic Regression, Naïve Bayes, Random Forest, Data Mining

1. Introduction

Predicting the change in software as a product line SPL is vital in the development cycle. In the development cycle, it takes one year for a new release to be deployed. Before that, the new release goes under the testing period to report issues or bugs. In the testing time, it is imperative to predict what files are likely to face change (any change). These units should be given more attention, which helps to plan well and reduce cost by correctly estimating resources to be allo-

cated for the next release.

The predicted change of a software unit can be minor or significant. Our goal in this research is to detect the change regardless of the nature of the size of change. To achieve that, we used machine learning models to learn from files of the current release and use them to predict the change in the next release. We use static code metrics to predict the change. Static code metrics explain the main features of a software unit, such as the complexity, number of methods, and cohesion of different classes. Change metrics are not used to predict the change on the next release because change metrics may help to define the nature of the change but not necessarily can predict if the change will occur.

The way we observe the change to a software unit (*i.e.*, file or class) is based on the change in the lines of code. If the software unit encountered added or deleted lines of code, that would make the software unit a change prone unit (*i.e.*, classified as changed). If the software unit did not face any added or deleted lines, the unit would be classified as not changed. We can determine if a file experienced change or not by observing the Code churn metric, which is the total number of added and deleted lines. When the value of this metric is zero, it means that the file has never been changed. In this paper, we measure the ability of our model to predict the next release by learning from the current release. We will test the performance of different algorithms to explore how the results are consistent with each other. Lastly, we explore the performance across all releases are different, and if they are affected by the evolution of the project. The research questions we address in this work are in the following list:

- RQ1: Can we predict the change in a software product line project?
- RQ2: What releases of the Eclipse project provide good learning to algorithms? Does the size of the dataset improve the training?
- RQ3: Does predicting change improve as the product evolved?
- RQ4: Does any of the machine learning algorithm performs better than others?

The rest of the paper is organized as follows: Related work is discussed next in Section 2. Then, we explain the data mining approach of this work in Section 3, including machine learning algorithms, metrics, datasets, and performance metrics. We discuss the results in Section 4. Threats to validity are explained in Section 5, and the paper is concluded in Section 6.

2. Related Works

There are several features related to change in the code. It can be represented through the number of lines added or deleted to a file, the number of authors contributed to the file, the number of revisions, or the number of refactoring. These features have been successfully used [1]-[8] to predict software fault proneness. In this research, we are interested in using one of the change metrics to predict any change associated with software files. The metric we can use for this purpose is the code churn metric. The code churn metric represents the total number of lines added and deleted to a software class. This metric was used to

predict software faults in [2] [6]. In this study, static code metrics are used as input metrics to predict the change. Static code metrics were also used, along with change metrics to predict software faults. In this section, we highlight related works that are targeted the study of software change proneness and used the metric as a response variable but not as a predictor. Some studies did a statistical analysis to investigate the relationship between different classes and bad smell [9] [10] [11]. Other studies applied prediction models to predict the change in software. Abdi *et al.* [12] used some machine learning algorithms (e.g., J48, Jrip, PART, and NBTree) to predict the change in open source projects. Tsantalis *et al.* [13] predicted the likelihood of change on software when functions.

Can we predict the change of the code in a software product line project3are added to classes using logistic regressions and measuring the performance using accuracy, sensitivity, false-positive ratio, and false-negative ratio. Genetic programming algorithm with object-oriented metrics was used to predict the change in [14]. Object-oriented metrics were also used with 19 projects, including Eclipse in [15]. Code smell related information was used to improve change prediction in [16].

This work aims to investigate how prediction models can work on the software line project. Eclipse is the chosen project for this work as we have access to seven consecutive releases from Eclipse (Eclipse 2.0, 2.1, 3.0, Europa, Ganymede, Galileo, and Helios). There are some other releases between Eclipse3.0 and Europa that we did not have access to in this work.

In this work, we explore the ability of models to learn from a dataset and test change on the following release. This approach should help to identify files that are likely to experience change from the next release. Predicting these files can be helpful in improving code quality ahead of time by identifying files that are likely to experience the change and learn why they need to be changed.

Also, we use the most known algorithms to conduct these experiments. This is particularly important to explore the generalizability of the performance of learning and testing of these algorithms on these datasets and identify any challenges we may have when using them or if one algorithm is performing better than others.

Lastly, we explore if the performance improved due to the evolution of the project. In this sense, we need to see how the performance differs from the old releases to recent ones.

3. Methodology

This section discusses the data mining methodology applied in this research. This includes the type of learners used for prediction, datasets and sampling process, metrics definitions, and performance metrics used to evaluate learners. The model is trained in a release (release n) and tested on the following release (release $n + 1$). This means that we should have a total of six tested models on six releases. We cannot test on release n because this would require access to release

$n-1$, which we do not have. This process is iterated four times as we are using four algorithms to train our models. The total number of experiments of this work is twenty-four, which is the product of six releases by four algorithms. The outcome of each experiment is four performance measures (*i.e.*, accuracy, recall, precision, and F-score).

3.1. Learners

Several learners (e.g., logistic regression, decision tree, and Naive Bayes) have been used in the software fault proneness area [17]. Many of the top learners provided performances that are not significantly different from each other [18]. Our selection of algorithms is based on three main factors: The popularity of algorithms, algorithms fit the data, and algorithms are easy to implement. In this section, we briefly explain some of the learners we used in this study. LR models describe the probability of the existence of a condition (*i.e.*, fault-prone or fault-free) based on a given set of variables X_i. The set of variables is described based on a linear function and then placed into the logit model to calculate the probability ranged between 0 and 1, as shown in Equation (1).

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_i X_i \tag{1}$$

where Y is the response variable (fault prone, fault free), and X_i is the independent variable (*i.e.*, metric).

Naive Bayes classification works based on Bayesian rules, as defined by Equation (2). The classifier is famous for its simplicity and fast computation. The classifier works up a set of input metrics (numerical or categorical) as if they are independent of each other. The probability of the response variable is calculated, as shown in Equation (2).

$$p(X \mid Y_k) = p(Y_k) \prod_{i=1}^{n} p(X_i \mid Y_k) \tag{2}$$

where Y is the response variable (*i.e.*, change prone, non-change prone), and X is the independent variable, k is the number of classes (in our case is two classes), and n is the number of input metrics.

Decision tree J48 works by splitting data based on the most significant splitter (*i.e.*, metric). The splitter is chosen based on the impurity or uncertainty of the data under this subset of data. The decision of splitting is based on calculating the information gain, as shown in Equations (3) and (4). The information gain subtracts the prior entropy of the selected metric X_i. The classifier continues splitting data until a tree is formed, starting from the root (*i.e.*, all metrics) and ending with leaves or terminal nodes (*i.e.*, metrics that were not split).

$$H[D] = -\sum_{j=1}^{|c|} P(c_j) \log 2P(c_j) \tag{3}$$

$$gain(D, A_i) = H[D] - HX_i[D] \tag{4}$$

where C is the desired class, and $H[D]$ is the entropy.

Random forest is an ensemble tree-based learning algorithm, developed by [19]. Other ensemble classifiers inspired the algorithm (e.g., bagging, random split selection). The algorithm creates multiple trees and takes a majority voting on the predicted class instead of on a single tree decision [20].

3.2. Datasets

In this area, different datasets are used [17]. Eclipse is one of the software projects that are used by 50% of studies reported in [17]. In this study, we use seven releases of the Eclipse project (*i.e.*, Eclipse 2.0, 2.1, 3.0, Europa, Ganymede, Galileo, and Helios). The size of the releases is shown in **Table 1**. In the table, the percentage of change prone files is also presented for each release. The change prone files are the file that had at least one line added, or one line deleted from the file during the development process. The change prone files of early releases (e.g., Eclipse 2.0, 2.1, 3.0) are remarkably high (74% for Eclipse 2.0, 89% for Eclipse 2.1, and 72% for Eclipse 3.0). Then, the percentage of change is dropped to 38% in Europa, 31% in Ganymede, 17% in Galileo, and 14% in Helios.

3.3. Metrics

Static code metrics are associated with the change in software [21]. Therefore, we used only static code metrics. Other earlier works used static code metrics (e.g., [22] [23] [24] [25]). Hall *et al.* [17] found that static code metrics were used by 38% of studies. The earlier work [26] extracted static code metrics used in this work, and [27] extracted the change metrics of this work. Out of all change metrics, we used the Code churn metric and used it in the binary format (*i.e.*, changed/not changed). Changed files are all files that experienced added or deleted lines. Unchanged files are files that had not experienced any change at all. All static metrics are defined in **Table 2**.

3.4. Performance Metrics

Performance metrics are used to measure the performance capabilities of all learners in predicting classes (change prone or not). In this study, we used four major performance measures, accuracy, recall, precision, and F-score. All these measures are extracted from the confusion matrix (see **Table 3**).

Table 1. Number of files and change prone files of every release.

Release	Year	Number of files	Percentage of change prone files
Eclipse 2.0	2002	5016	74%
Eclipse 2.1	2003	6494	89%
Eclipse 3.0	2004	9547	72%
Europa	2007	31,484	38%
Ganymede	2008	32,648	31%
Galileo	2009	22,154	17%
Helios	2010	32,513	14%

Table 2. Metrics definitions.

Metric	Definition
Lines of code LOC	Total number of lines in a file
Statements	Any lines of code terminated by ";"
Percent branch statements	Percentage of statements causing a break in sequential execution, e.g., if, for, try, throw
Methods call statements	All method calls in statements and logical expressions
Percent Lines with Comments	Percentage of comments lines
Classes and Interfaces	Total number of classes and interfaces, including anonymous inner classes
Methods per Class	The total method count divided by the total classes
Ave Statements per Method	Total number of statements found inside of methods divided by the number
Max Complexity	Complexity value of the most complex method
Average Complexity	Sum of all method complexity values divided by the number of methods
Ave Block Depth	Sum of all method block depths divided by the number of methods

Table 3. Confusion matrix.

		Actual class	
		Change prone	Non-Change prone
Predicted class	Change prone	True positive TP	False Positive FP
	Non-change prone	False Negative FN	True Negative TN

Accuracy can measure the total number of correct classifications over the miss-classified instances (Equation (5)). Recall measures the rate of the correct classification over the number of instances that are classified as (change prone), which is the total number of true positive and the false negative as in Equation (6). Precision measures the correct classified instances over the number of instances that are predicted as (change prone), which is the total number of true positive and the false positive instances as in Equation (7). F-score (see Equation (8)) is the harmonic mean of recall and precision.

$$\text{Overall accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{5}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{6}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{7}$$

$$\text{F-score} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{8}$$

4. Results and Discussion

The results discussed in this section are to check the performance of the prediction models, which we developed to predict files that are likely to experience change. The results help to find if the models' performances are good enough to detect any change across all tested releases. Also, the results identify if one algorithm is significantly working better than others or if all algorithms perform with no differences. Lastly, we explore if the performance is affected by the evolution of the software project. In other words, we explore if the last release is significantly higher than previous releases.

The results shown in this section are for Eclipse 2.1, 3.0, Europa, Ganymede, Galileo, and Helios. We trained the prediction models using four algorithms using static code metrics on a release and test on the next release. Therefore, Eclipse 2.0 results are not reported in this section, because Eclipse 2.0 was used only for training the model that was tested on Eclipse 2.1.

The overall accuracy results for all algorithms on all tested releases are reported in **Figure 1**. If we excluded the NB performance, we find a consistent pattern of all algorithms on all releases. The logistic algorithm performed higher than other algorithms with Eclipse 2.1. Other than that, differences are not significant. Differences between the logistic performance of different algorithms on Ganymede, Galileo, and Helios are minor. These releases have been tested after training on large datasets (*i.e.*, Europa, Ganymede, and Galileo).

Accuracy is not always enough measure for a model performance. This because the confusion matrix may face many instances that are reported as false positive or false negative and still show a high accuracy. We need to check the recall and precision of each model, which can help to understand a clear pictures and amount of false negative and false positive identified. False positive are unchanged files that were identified as changed files. False negative is changed files that were identified as unchanged.

The recalls are shown in **Figure 2**. With exception of NB, all algorithms demonstrate high recalls in all releases. Naive Bayes algorithm started with incredibly low performance in Eclipse 2.1, then a slightly increase was detected in Eclipse 3.0 and Europa. The performance is sharply increase with Ganymede release and remained high for subsequent releases. This indicates that NB perform high on models that are trained on large datasets. High recalls indicated that the number of false negative events are low and all changed files are predicted.

All recalls are reported between 70% to 100% and logistic regression provided the highest recall compared to all other algorithms. Decision tree J48 came second and random forest at the third place. NB provided incredibly low recalls when trained on small datasets, as shown in the first three releases.

With respect to the precision, the results of all algorithms on all releases are presented in **Figure 3**. All algorithms scored precisions between 72% to 100%. Eclipse2.1, Europa, Ganymede, and Helios reported the highest precisions. High precision means that there are less events reported false positive.

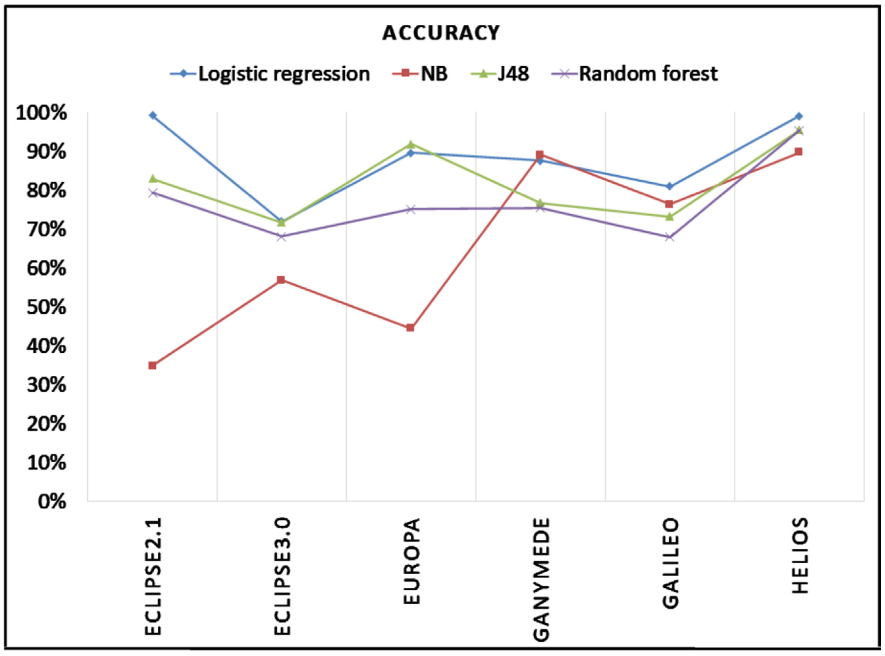

Figure 1. Accuracy of all algorithms on eclipse releases.

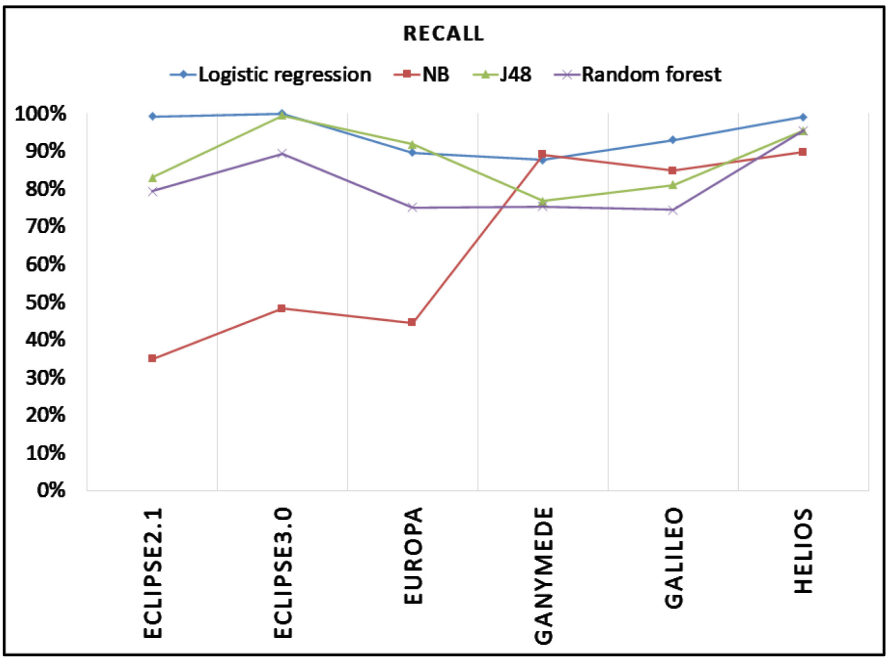

Figure 2. Recall of all algorithms on eclipse releases.

Both recall and precision are important measures. In some occasions, we may face high reading of one of them and low reading from the other. Therefore, it is important to report a third measure which gives us an indication of both. This metric is called the F-score, which reports the harmonic mean of the two measures.

The F-score results of all algorithms on all releases are reported in Figure 4. Naive Bayes reported low F-scores on Eclipse 2.1, 3.0, and Europa. NB on other releases and all algorithms on all releases reported F-scores between 80% to 100%.

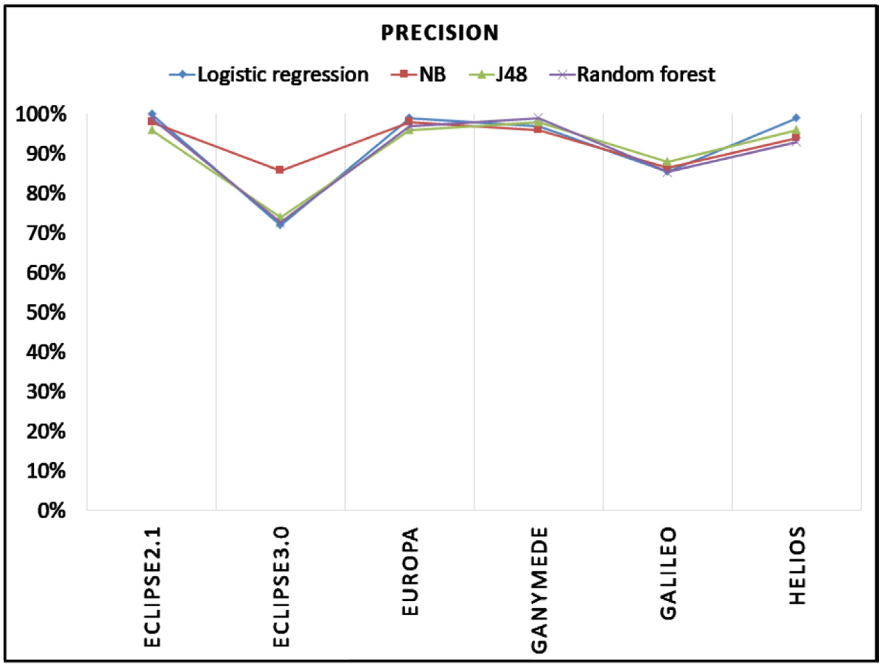

Figure 3. Precision of all algorithms on eclipse releases.

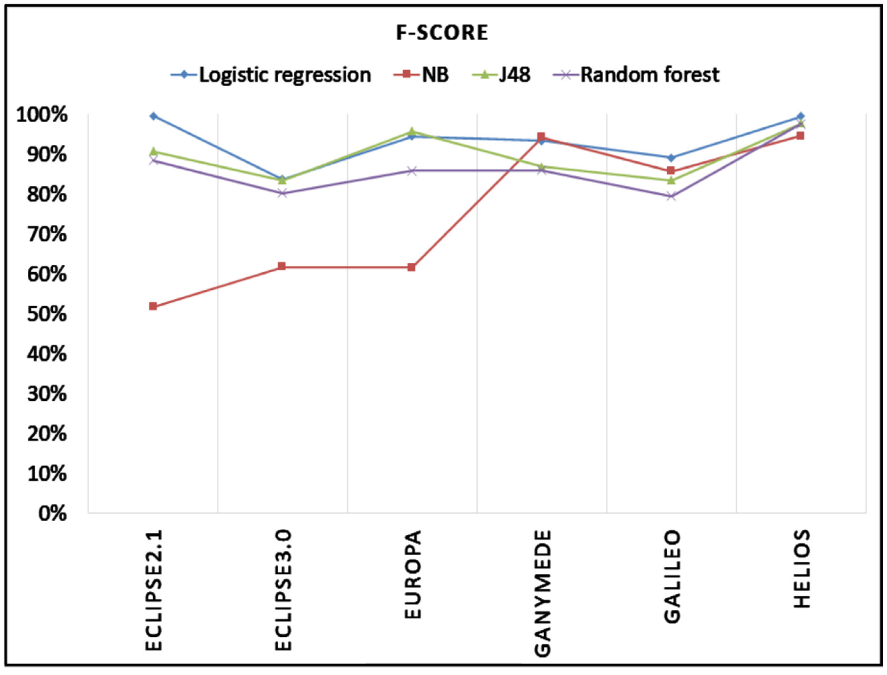

Figure 4. F-score of all algorithms on eclipse releases.

There is no clear pattern regarding the performance of a specific dataset. Except for NB, all algorithms perform at an almost similar level in all datasets with no clear distinction. The performance of NB increases as the software evolved. This increase is associated with an increase in the size of the training set.

The results of the two research questions RQ1, RQ2, RQ3, and RQ4 are listed below:

– RQ1: Can we predict the change in a software product line project?

Predicting changed or unchanged files require a balanced distribution of the number of changed and unchanged files. In this study, we managed to predict changed files in Eclipse 2.1 and 3.0, and we predicted unchanged files in Europa, Ganymede, Galileo, and Helios. When changed or unchanged files are rare events, then predicting any of them will be unsuccessful due to bad classification. To overcome this problem, we need to apply the oversampling method to get a balanced distribution.

– RQ2: What releases of the Eclipse project provide good learning to algorithms? Does the size of the dataset improve the training?

We found that all datasets provide similar learning because the performance of all tested release is almost at the same level of performance. Only one algorithm (*i.e.*, Naive Bayes) provided different patterns as the learner works well when learning from large datasets (e.g., Europa, Ganymede, Galileo, and Helios). The algorithm provided a low level of accuracy, recall, precision, and F-score when the algorithm trained on small datasets (e.g., Eclipse 2.0, Eclipse 2.1, Eclipse 3.0).

– RQ3: Does predicting change improve as the product evolved?

When we used the naive Bayes algorithm, the performance (accuracy, recall, and F-score) increased linearly started at 35% accuracy of the first release until it reached 90% accuracy in the last release. The same pattern exists with the recall and F-score. The reason could be due to the sensitivity of the NB algorithm to dataset size and has nothing to do with the evolution of the project.

– RQ4: Does any of the machine learning algorithms perform better than others?

In terms of accuracy, logistic regression performed better than other algorithms on three releases but without a significant difference.

5. Threats to Validity

This research took all steps to ensure that no threat affects the internal, construct, conclusion, and external validity.

Internal validity is concerned with the quality of the data. The confidence in the data is high as we conduct sanity checks on them to ensure their quality, and they reflect the actual source files.

The construct validity is concerned with that the experiment measured what is intended to measure. We explained what we intended to measure in the introductory part with some research questions. We developed the experimentation on this basis, and we gathered all results, we explained to them and addressed all research questions clearly at the end of the work. We predicted changed files in Eclipse 2.0, 2.1, and 3.0. In other releases, we reported the performance of the models when they predict unchanged files because they were the majority class. When predicting unchanged files, this means we decided that these groups of files will not require change.

To ensure the conclusion validity, we applied the algorithms that are common in the area. We provided that the algorithms are fit for the data we used. Our response variable is dichotomous, and the input metrics are in numerical and dichotomous format. The models were evaluated using very common measures, which can help to address all research questions mentioned in the introduction.

External validity can be violated if we claim the generalizability of the results. Our results are valid for the specific releases used from the Eclipse project. We do not generalize the results on other software projects.

6. Conclusions

In this work, we predicted the change in software files in one of the software product line projects (*i.e.*, Eclipse). We used four algorithms, trained on six releases, and tested on six releases. The training release is the release right before the tested release. We found that predicting changed and unchanged files are possible for all releases. The only problem that could face the software manager is that the balanced distribution of the two classes of the response variable. We found that all algorithms are performing at the same level, except for naive Bayes algorithm when trained small datasets. Lastly, we found that there is not enough evidence to prove that the evolution of the project improves learning.

Our future work will consider predicting the level of change and the type of change that software files are likely to face at every release of Eclipse. Also, we need to consider methods to improve performance (e.g., parameter tuning). We will apply to replicate the work on other software projects to explore the generalizability. Further, we will apply explanatory work to quantify the contribution of explanatory metrics on the response variable.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Alshehri, Y.A., Goseva-Popstojanova, K., Dzielski, D.G. and Devine, T. (2018) Applying Machine Learning to Predict Software Fault Proneness Using Change Metrics, Static Code Metrics, and a Combination of Them. *SoutheastCon* 2018, St. Petersburg, 19-22 April 2018, 1-7. https://doi.org/10.1109/SECON.2018.8478911

[2] Bell, R.M., Ostrand, T.J. and Weyuker, E.J. (2011) Does Measuring Code Change Improve Fault Prediction? *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Banff, 20-21 September 2011, Article No. 2. https://doi.org/10.1145/2020390.2020392

[3] Goseva-Popstojanova, K., Ahmad, M. and Alshehri, Y. (2019) Software Fault Proneness Prediction with Group Lasso Regression: On Factors That Affect Classification Performance. 2019 *IEEE 43rd Annual Computer Software and Applications Conference*, Volume 2, 336-343. https://doi.org/10.1109/COMPSAC.2019.10229

[4] Krishnan, S., Strasburg, C., Lutz, R.R. and GosevaPopstojanova, K. (2011) Are Change Metrics Good Predictors for an Evolving Software Product Line? *Proceedings of the 7th International Conference on Predictive Models in Software Engi-*

neering, Banff, 20-21 September 2011, Article No. 7.
https://doi.org/10.1145/2020390.2020397

[5] Moser, R., Pedrycz, W. and Succi, G. (2008) A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. *ACM/IEEE* 30*th International Conference on Software Engineering*, Leipzig, May 2008, 181-190.
https://doi.org/10.1145/1368088.1368114

[6] Nagappan, N. and Ball, T. (2005) Use of Relative Code Churn Measures to Predict System Defect Density. *Proceedings of the* 27*th International Conference on Software Engineering*, St Louis, 15-21 May 2005, 284-292.
https://doi.org/10.1145/1062455.1062514

[7] Ostrand, T.J., Weyuker, E.J. and Bell, R.M. (2010) Programmer-Based Fault Prediction. *Proceedings of the* 6*th International Conference on Predictive Models in Software Engineering*, Timisoara, 12-13 September 2010, Article No. 19.
https://doi.org/10.1145/1868328.1868357

[8] Weyuker, E.J., Ostrand, T.J. and Bell, R.M. (2008) Do Too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models. *Empirical Software Engineering*, **13**, 539-559.
https://doi.org/10.1007/s10664-008-9082-8

[9] Khomh, F., Di Penta, M. and Gueheneuc, Y.-G. (2009) An Exploratory Study of the Impact of Code Smells on Software Change-Proneness. 2009 16*th Working Conference on Reverse Engineering*, Lille, 13-16 October 2009, 75-84.
https://doi.org/10.1109/WCRE.2009.28

[10] Khomh, F., Di Penta, M., Gueheneuc, Y.-G. and Antoniol, G. (2009) An Exploratory Study of the Impact of Software Changeability.

[11] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. and De Lucia, A. (2018) On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation. *Empirical Software Engineering*, **23**, 1188-1221.
https://doi.org/10.1007/s10664-017-9535-z

[12] Abdi, M.K., Lounis, H. and Sahraoui, H. (2006) Analyzing Change Impact in Object Oriented Systems. 32*nd EUROMICRO Conference on Software Engineering and Advanced Applications*, Cavtat, 29 August-1 September 2006, 310-319.
https://doi.org/10.1109/EUROMICRO.2006.20

[13] Tsantalis, N., Chatzigeorgiou, A. and Stephanides, G. (2005) Predicting the Probability of Change in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, **31**, 601-614. https://doi.org/10.1109/TSE.2005.83

[14] Marinescu, C. (2014) How Good Is Genetic Programming at Predicting Changes and Defects? 2014 16*th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, 22-25 September 2014, 544-548.

[15] Giger, E., Pinzger, M. and Gall, H.C. (2012) Can We Predict Types of Code Changes? An Empirical Analysis. 2012 9*th IEEE Working Conference on Mining Software Repositories*, Zurich, 2-3 June 2012, 217-226.
https://doi.org/10.1109/MSR.2012.6224284

[16] Catolino, G., Palomba, F., Fontana, F.A., De Lucia, A., Zaidman, A. and Ferrucci, F. (2020) Improving Change Prediction Models with Code Smell-Related Information. *Empirical Software Engineering*, **25**, 49-95.
https://doi.org/10.1007/s10664-019-09739-0

[17] Hall, T., Beecham, S., Bowes, D., Gray, D. and Counsell, S. (2012) A systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, **38**, 1276-1304.
https://doi.org/10.1109/TSE.2011.103

[18] Lessmann, S., Baesens, B., Mues, C. and Pietsch, S. (2008) Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering*, **34**, 485-496. https://doi.org/10.1109/TSE.2008.35

[19] Breiman, L. (2001) Random Forests. *Machine Learning*, **45**, 5-32. https://doi.org/10.1023/A:1010933404324

[20] Liaw, A. and Wiener, M. (2002) Classification and Regression by Random Forest. *R News*, **2**, 18-22. https://doi.org/10.1109/SYNASC.2014.78

[21] Gune, A., Koru, S. and Liu, H.F. (2007) Identifying and Characterizing Change-Prone Classes in Two Large-Scale Open-Source Products. *Journal of Systems and Software*, **80**, 63-73. https://doi.org/10.1016/j.jss.2006.05.017

[22] Alshayeb, M. and Li, W. (2003) An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes. *IEEE Transactions on Software Engineering*, **29**, 1043-1049. https://doi.org/10.1109/TSE.2003.1245305

[23] Li, W. and Henry, S. (1993) Object-Oriented Metrics That Predict Maintainability. *Journal of Systems and Software*, **23**, 111-122. https://doi.org/10.1016/0164-1212(93)90077-B

[24] Romano, D. and Pinzger, M. (2011) Using Source Code Metrics to Predict Change-prone Java Interfaces. 2011 27*th IEEE International Conference on Software Maintenance*, Williamsburg, 25-30 September 2011, 303-312. https://doi.org/10.1109/ICSM.2011.6080797

[25] Zhou, Y.M., Leung, H. and Xu, B.W. (2009) Examining the Potentially Confounding Effect of Class Size on the Associations between Object-Oriented Metrics and Change-Proneness. *IEEE Transactions on Software Engineering*, **35**, 607-623. https://doi.org/10.1109/TSE.2009.32

[26] Devine, T., Goseva-Popstojanova, K., Krishnan, S. and Lutz, R.R. (2014) Assessment and Cross-Product Prediction of Software Product Line Quality: Accounting for Reuse across Products, Over Multiple Releases. *Automated Software Engineering*, **23**, 1-50. https://doi.org/10.1007/s10515-014-0160-4

[27] Krishnan, S., Strasburg, C., Lutz, R.R., GosevaPopstojanova, K. and Dorman, K.S. (2013) Predicting Failure-Proneness in an Evolving Software Product Line. *Information and Software Technology*, **55**, 1479-1495. https://doi.org/10.1016/j.infsof.2012.11.008