Revisiting the Impact of Concept Drift on Just-in-Time Quality Assurance

Kwabena E. Bennin, Nauman bin Ali, Jürgen Börstler
Department of Software Engineering
Blekinge Institute of Technology
Karlskrona, Sweden
{kwabena.ebo.bennin; nauman.ali; jurgen.borstler}@bth.se

Xiao Yu
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
xyu224-c@my.cityu.edu.hk

***Abstract*—The performance of software defect prediction(SDP) models is known to be dependent on the datasets used for training the models. Evolving data in a dynamic software development environment such as significant refactoring and organizational changes introduces new concept to the prediction model, thus making improved classification performance difficult. In this study, we investigate and assess the existence and impact of concept drift on SDP performances. We empirically asses the prediction performance of five models by conducting cross-version experiments using fifty-five releases of five open-source projects. Prediction performance fluctuated as the training datasets changed over time. Our results indicate that the quality and the reliability of defect prediction models fluctuate over time and that this instability should be considered by software quality teams when using historical datasets. The performance of a static predictor constructed with data from historical versions may degrade over time due to the challenges posed by concept drift.**

***Index Terms*—Defect prediction, Just-in-time Quality assurance, Concept drift**

I. Introduction

Software defect prediction models help practitioners detect potential defective modules in software, thus enabling the effective allocation of scarce testing resources [1], [2]. Several machine learning models [3], [4], [5] and recently deep learning models [6], [7] have been proposed to help software quality teams detect potentially buggy modules in new releases or versions trained on historical versions of a software project. The performance and efficiency of the prediction models are thus dependent on the quality of the training (historical versions) datasets [8], [9].

Traditional defect prediction is usually conducted at the package or file-level where product (static/code) metrics are extracted from software source codes and used for training. To help focus on the most risky changes that can potentially introduce bugs into a software project, Kamei et al. [10] proposed Just-in-Time (JIT) prediction models which focus on identifying defect-prone software changes instead of files or packages. These JIT models are constructed with process metrics extracted from the properties of software changes.

Validation of defect prediction models for within-project scenarios are mostly done using the time-wise validation approach where the chronological order of software modules or versions are considered. The most recent version is used for testing, whereas the least recent versions/modules are considered for training. This widely used approach assumes the stability of software projects during the evolution process. However, software projects vary or undergo several changes over time and this reflects in the distribution of the data instances (metrics) extracted from the projects. These changes in the data distribution over time is referred to as concept drift in the machine learning literature. Factors such as code review process, major refactoring, change of developers, changes in organisation policies such us formal programming language, tools and others can contribute to this concept drift phenomenon.

Very few studies [11], [12], [13], [14] consider the impact of concept drift on software quality. Amongst the common challenges of software defect datasets tackled in literature are class imbalance [15], [16] and feature selection [17]. Ekanayake et al. [18] observed that the prediction quality of models for software projects changes over time as the number of developers editing and fixing defects on the files changes.

To investigate and empirically assess the impact of concept drift on software defect quality, we revisit the study by Ekanayake et al. [18] and conduct comprehensive change-level defect prediction experiments on 55 releases of 5 change-level metrics software projects using 5 prediction models and 3 evaluation measures.

As a contribution, we conduct a replication study in which we reassess previous evidence but achieve similar outcomes. Our analysis reveals that concept drift does impact prediction performance but some prediction models are more robust to the distribution changes.

This paper is organised as follows. Section II presents the related work. In Section III, the datasets, prediction models and experimental settings are presented. The results and a discussion of the results are reported in Section IV. The potential threats to the validity of our study results are presented in Section V. Finally, we present our conclusion and potential future work directions in Section VI.

II. Background and Related Work

In this section, we review the limited related work on concept drift in software defect prediction.

By studying four open source systems, Ekanayake et al. [11] investigated the notion of concept drift and its impact

on defect prediction. Further study by Ekanayake et al. [18] revealed that the number of authors editing software projects contributes to concept drift and results in fluctuating software quality.

Kabir et al. [14] demonstrated the applicability of a drift detection method to identify concept drift for defect prediction. The method helped to identify drift points in the datasets for two open-source projects. Such points can indicate the need for retraining the prediction models.

A recent study by Amasaki [12] on the feasibility of applying a cross-project defect prediction approach for cross-version project defect prediction revealed that the nearest neighbor filter method used for cross-project defect prediction did not improve the performance of models trained on cross-versions. The author also observed that most projects suffered from concept drift and that was the main challenge impeding performance improvement. We supplement the findings of these studies by empirically identifying concept drift in five additional projects. The projects we have used in the analysis are significantly larger than the ones used before and therefore are more likely to be representative of some real-world systems.

III. Experimental Methodology

A. Datasets

Five large software projects with change-level metrics which were made publicly available by Kamei et al. [10] were extracted and used for the experiments. The basic information of these software projects with the number of changes (modules) and percentage of defects are presented in Table I. These datasets comprise 14 change-level metrics and an additional metric called bug which shows the number of bugs found per module. Table II shows the summary of the metrics for the datasets used. Further information on these metrics and how they were extracted can be found in the original study by Kamei et al. [10]. The ND and REXP metrics were removed from the datasets before constructing the models because these two metrics were found to be highly correlated by Kamei et al. [10].

TABLE I
SUMMARY OF THE STUDIED SOFTWARE PROJECT

Project	Period	# changes	% Defect	Total LOC
Bugzilla	08/1998 – 12/2006	4620	36	173250
Columba	11/2002 – 07/2006	4455	14	665577
Eclipse Platform	05/2001 – 12/2007	64250	5	4638850
Mozilla	01/2000 – 12/2006	98275	25	10466287.5
PostgreSQL	07/1996 – 05/2010	20431	20	203960.3

B. Prediction Models and Performance Measures

To assess the impact of concept drift on prediction performance, we used five prediction models. Four of them are models commonly used in defect prediction studies; Naive-Bayes, Neural Networks, K-Nearest Neighbor (n=3) and Random Forest [19]. Furthermore, we adopted a recently proposed boosting model called XGBoost [20]. XGBoost is an improved implementation of a gradient boosting framework optimized to be highly efficient, effective and better performance [21], [22]. The models are implemented and executed using scikit-learn[1], a machine learning package in Python. We used the default configurations for each model.

TABLE II
SUMMARY OF CHANGE MEASURES

Name	Definition
NS	Number of modified subsystems
ND	Number of modified directories
NF	Number of modified files
Entropy	Distribution of modified code across each file
LA	Lines of code added
LD	Lines of code deleted
LT	Lines of code in a file before the change
FIX	Whether or not the change is a defect fix
NDEV	The number of developers that changed the modified files
AGE	The average time interval between the last and the current change
NUC	The number of unique changes to the modified files
EXP	Developer experience
REXP	Recent developer experience
SEXP	Developer experience on a subsystem
bug	Indicates the existence of a bug or not

The prediction models are evaluated using three measures: recall or probability of detection (*pd*), probability of false alarms (*pf*) and the Area Under the Curve (AUC). Both (*pd*) and (*pf*) are computed from a confusion matrix (III) and the Area Under the Curve (AUC) value is calculated from the Receiver Operating Characteristics (ROC) curve. Recall (*pd*) measures the percentage of actually detected defects and achieves a value of 100% if false negatives are zero. *Pf* computes the rate of predicted modules that are wrongly predicted as defect-prone. Higher values of *pd* and lower values of *pf* denote better prediction performance. Equations 1 and 2 show the mathematical computation for these measures. AUC handles the trade-offs between true and false positive rates [23] and has been shown to be more suitable for evaluating prediction models trained on imbalanced defect datasets [16], [24]. As a preliminary study, a direct comparison of performance values are compared.

TABLE III
CONFUSION MATRIX FOR THE TWO-CLASS PROBLEM

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

$$Recall\ (pd) = \frac{TP}{TP + FN} \tag{1}$$

$$FalseAlarm\ (pf) = \frac{FP}{FP + TN} \tag{2}$$

C. Experiments

Three sets of experiments were conducted to assess and examine the impacts of different training datasets developed in different time periods on prediction quality. To determine the evolution and drift changes in the software projects, the

[1]http://scikit-learn.org/

change-level datasets are chronologically arranged based on the commit dates. They are then divided into groups where changes made in the same month are grouped together. Similar to the approach by Yang et al. [25] and in conformance with popular intuition that a release/version of a software is completed within 6–8 weeks [18], changes in two groups (months) are merged to create one dataset. Thus, 11 versions/releases were created from each software project where version 1 represents the very first two months of changes (old version) and version 11 represents the latest version (current changes). In addition to replicating the procedure followed by Ekanayake et al. [18] where they keep the target project (latest version) constant and predict the defects using models trained on all of the possible combinations of training datasets collected within a period earlier than that of the latest target project, we conduct two different sets of cross-version experiments where the target (test) changes per every experimental run imitating a more practical scenario. Below is a description of the three approaches followed.

(1) **Single cross-version defect prediction for varying target period:** For each software project, we conduct cross-versions experiments. Prediction models are trained on the earlier version and used to predict the bug-proneness (classify) of the subsequent version. As an example, for each project with n (n=11) versions, the models are first trained on version 1 and used to predict version 2. Afterwards, models will be trained on version 2 and then tested on version 3. The latest version (11) is not used for training. Since each software project has 11 versions, 10 i.e (11–1) cross-versions experiments are conducted per each model. By following this process, an overall 250 (10*5*5) models across all datasets.

(2) **Aggregated cross-version defect prediction for varying target period:** Similar to the first experiment, we conduct cross-versions predictions. However, the training set is gradually expanded by the subsequent versions after being used as a target set in the previous experimental run. The target project is moved one version forward and the preceding versions are combined as the training sets. We begin by using the oldest version to predict the next oldest version. Afterwards, both versions are merged to form one training set and tested on the subsequent older version. As an example, for each project with n (n=11) versions, the first experimental run uses version 1 as the training set and the model is tested on version 2. Subsequently, versions 1 and 2 are merged and used as the training sets and version 3 is used as the testing set. The procedure is followed until all versions but the last version are merged as the training set.

(3) **Aggregated cross-version defect prediction for constant target period:** We follow the same procedure implemented by Ekanayake et al. [18] for our datasets and prediction models. For this experiment, all combinations of training datasets are created to predict a constant target project (most recent version being version 11).

IV. Results and Discussions

To give an overview over the impact of drift changes on prediction performance, the results are presented using line plots.

A. Single cross-version defect prediction for varying target periods

As explained in the experimental approach in Section III-C, we conduct cross-version predictions where each old version is used to predict the subsequent recent version once. Only the last (*x*th) recent version is not used for training but it was used as a testing dataset for the *x–1*th version. Figure 1 shows the results of using different projects developed in different time periods on prediction performance. The versions used for training the models are displayed on the x-axis with the performance values on the y-axis.

For most prediction models especially for the Naive-Bayes (NB) and Neural Network (NNET) models, recall and *pf* values fluctuate as the training data set changes. The performance increases and decreases at different times. The Random Forest (RF) and XGBoost (XGB) models are more robust to changes. AUC values are almost stable across all datasets and models.

The results indicate that it is a challenge to construct a stable predictor that will perform well across new software releases developed in different time periods. More specifically, constructing defect prediction models using a single version may or may not improve prediction performance. To find a more stable predictor that will perform well, the second set of experiments where several releases of software are combined are analysed in the section below.

B. Aggregated cross-version defect prediction for varying target periods

Figure 2 displays the results of using more than one software releases as a training set for predicting a new release. Apart from the very first (T1) release which was used individually to predict the next subsequent release, all quality predictions were made with models trained on 2 or more releases. The last prediction was made with models trained on the previous 10 releases (T1 to T10) and tested on T11.

The performance for all measures initially decreases as the training data set is expanded to include new versions produced in different time periods for most models. Similar to the previous observations in the first experiment, AUC values were almost stable across all datasets and models. Recall and *pf* values fluctuate especially for the NB and NNET models. However, there is a bit of stability for the NB and NNET models compared to the single-set cross version experiment results.

C. Aggregated cross-version defect prediction for constant target period

Figure 3 displays the results of using more than one software releases as a training set to predict the latest release. Different from the observations of the previous two experiments, more fluctuations were observed across all prediction models and

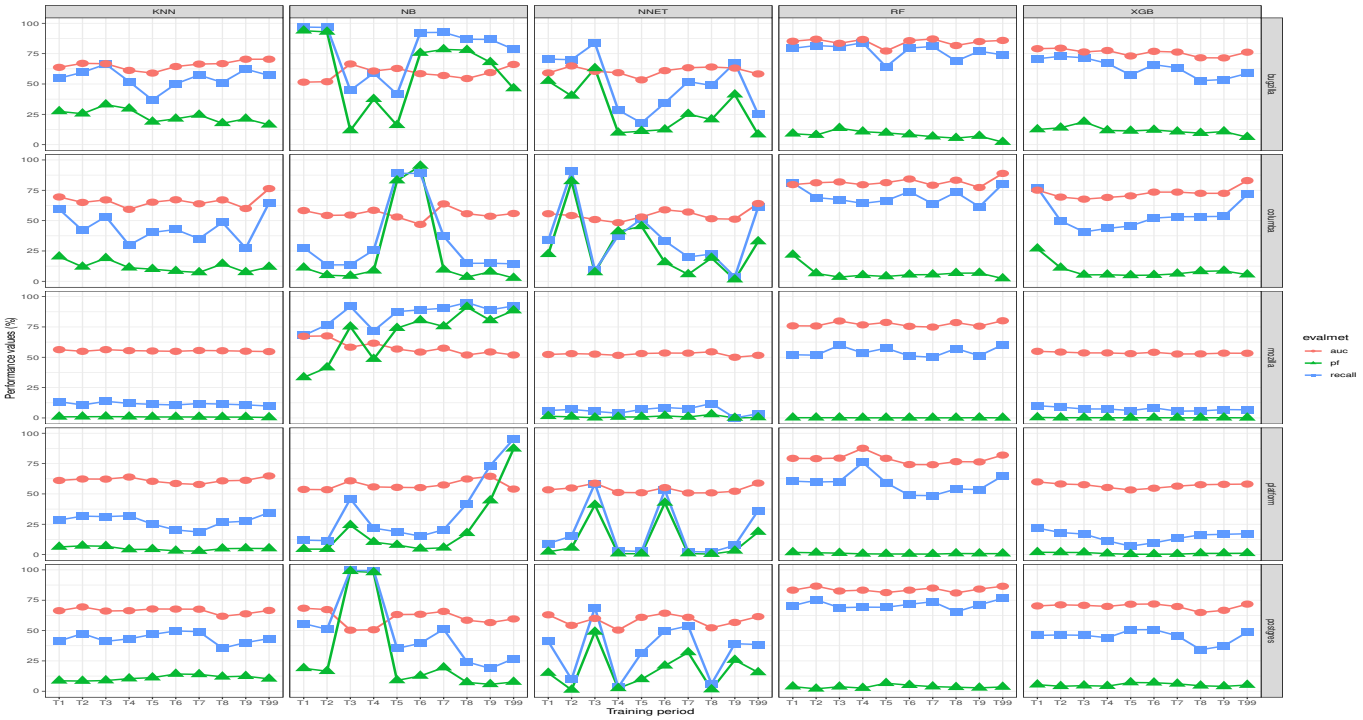

Fig. 1. Prediction quality using different single training datasets developed at different time periods to predict changing target project for different classification models

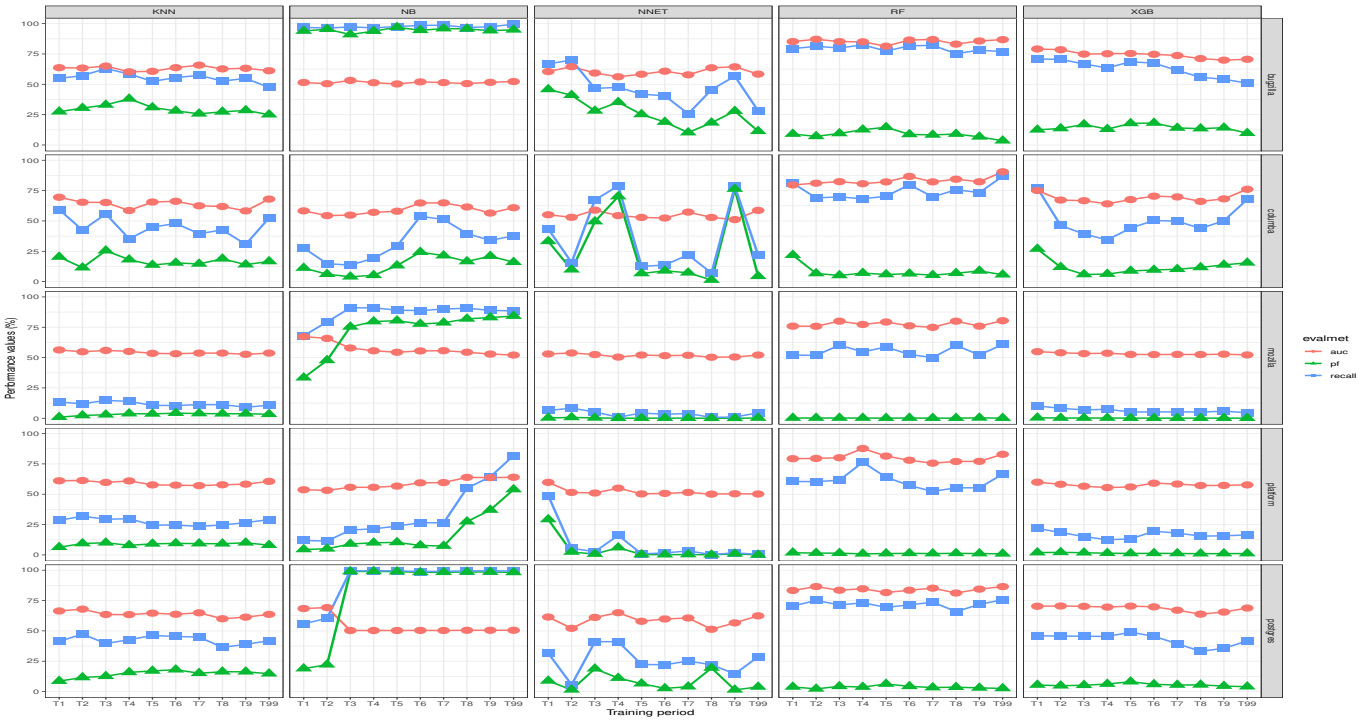

Fig. 2. Prediction quality using different combined datasets developed at different time periods to predict changing target project for different classification models

some datasets. AUC values were as volatile as the *pd* and *pf* values for the mozilla, platform and postgre projects. For these projects, the prediction quality were more unstable throughout all prediction periods. The NNET model was the most unstable across all datasets. Considering the bugzilla and columba datasets, the prediction performance is initially low and stable

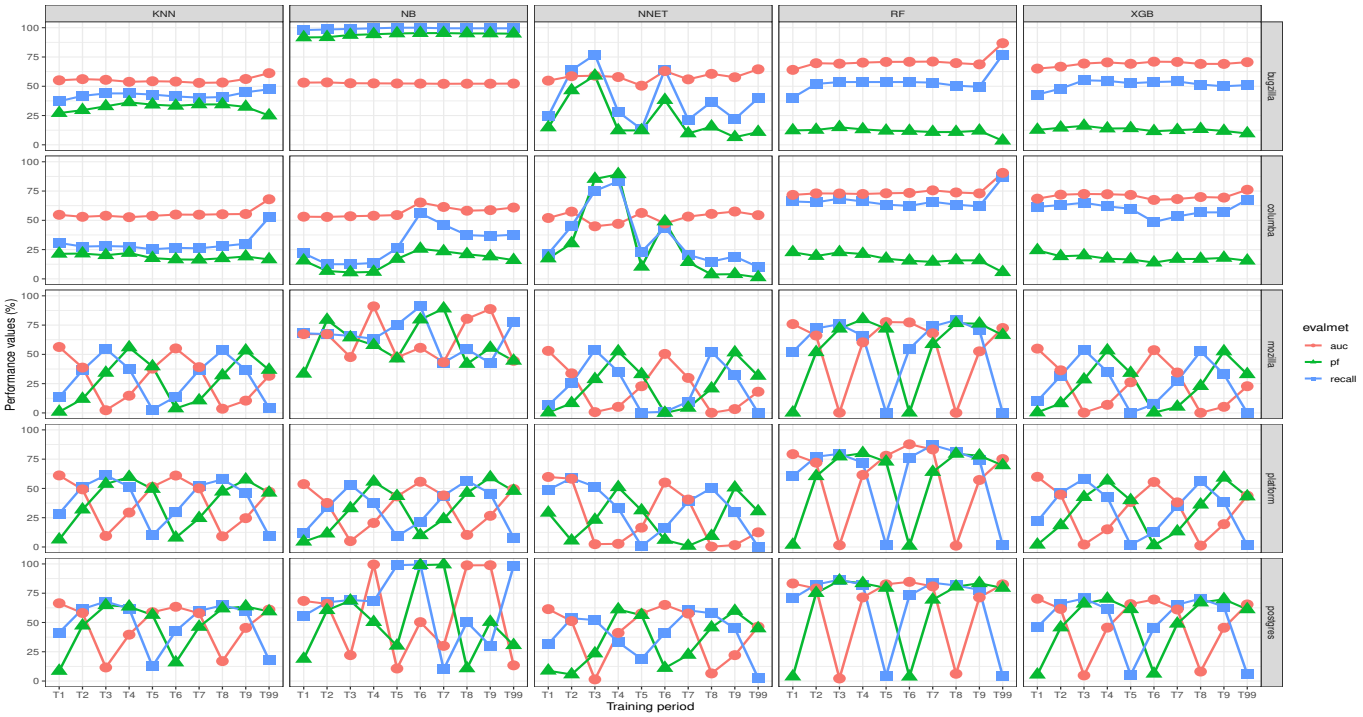

Fig. 3. Prediction quality using different combined datasets developed at different time periods to predict a stable target project for different classification models

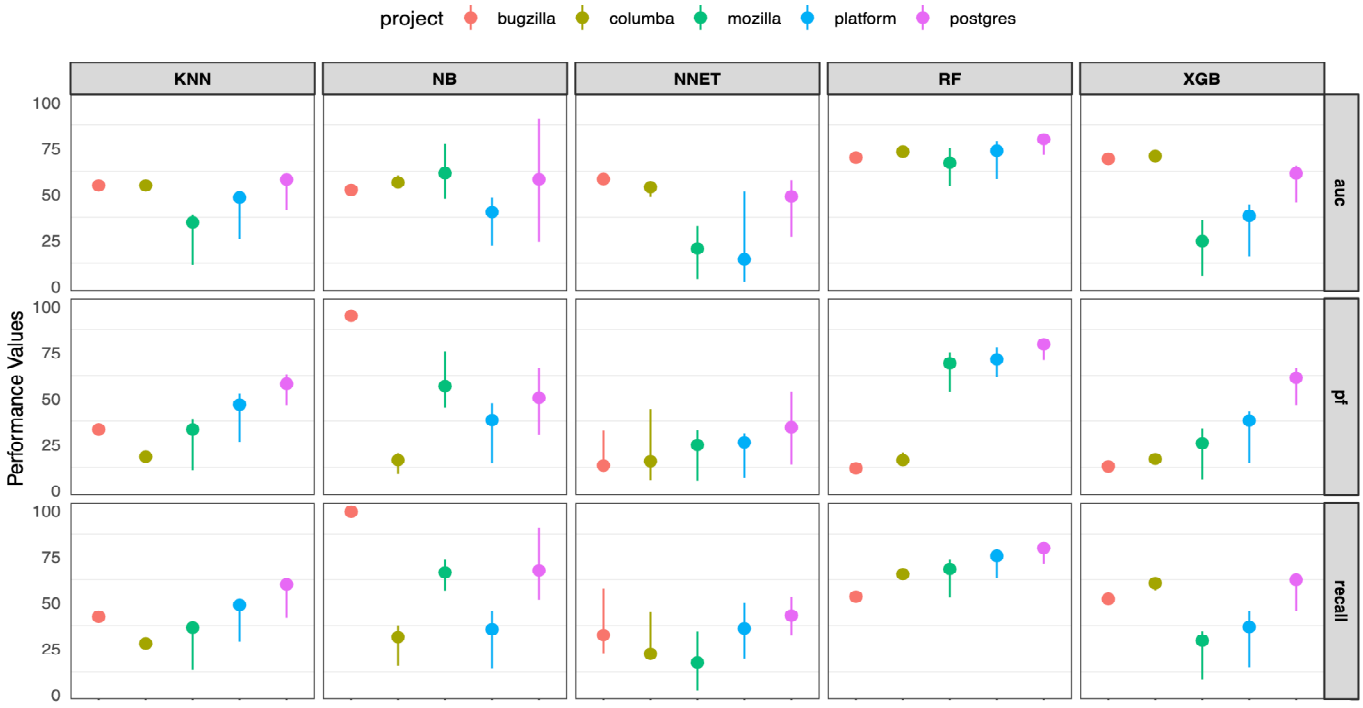

Fig. 4. Quartile plot of prediction quality using different combined datasets developed at different time periods to predict a stable target project for different classification models

but increases towards the end of the prediction prediction. To better demonstrate the variability in prediction results, quartile plots are presented in Figure 4. The quartile plots are generated by sorting the prediction results and extracting the median, lower (25th percentile) and upper (75th percentile) quartiles for each dataset project. In the figure, the solid dot

represents the median whereas the limits represents the lower and upper quartiles. From the quartile plots, we observe that all prediction models with the exception of the NNET model trained on the bugzilla and columba projects produced more stable recall and pf values. All models trained on these two projects produced stable AUC values. These two projects had few code changes (less than 5K) compared to the other three projects with more than 20K changes (from Table I) and may explain why prediction models trained on them were more stable. This indicates that concept drift may exist in the project but the drift is not significant. We also observe that the Random Forest model was the most robust model to concept drift changes as very few variations of results were observed across all training periods and datasets.

D. Discussion

Despite the concept drift impact on prediction performance, it is not always negative as noted by a previous study [14]. Model performance at different time periods (using versions) vary. Regarding the first experiment, the instability in performance is due in part to the changes in the training dataset per each prediction. For the second and third experiments, the improvement or degradation in performance is caused by the expansion in training set. Although the versions belong to the same software project, the training datasets were recorded and extracted in different time periods. As such, different factors such as change in developers, code review systems and others may affect the quality of each software version subsequently affecting the data distribution. Concepts available in old version may not be available in the new version and vice versa. The different and new concepts existing in the different versions once introduced into the training set are not easily detected and learned by the prediction models. Interestingly, the Bugzilla dataset which was collected over a period of 8 years was one of the most stable project across all prediction models in all three experiments. The usual intuition will be that there will be a significant concept drift. Obviously, the factors such as code reviews, tools used and developers may have changed during the 8-year period but the concepts or distribution did not significantly change (drift). We attribute this stability to the few number of changes made during the 8-year period the data was collected. Other factors may have led to this stability and further research beyond our current study may aid unravel the factors. This is left for a future study.

We also observed different types of concept drift based on the prediction performance. Whilst most the drifts observed are gradual, we observe sudden drifts for the NB and NNET models. These models are less robust to distribution changes in the training datasets. Additionally, the recall and *pf* values are almost the same and highly correlated. A good model should achieve high recall and low *pf* [1], thus we do not recommend using NB and NNET for defect prediction trained on several releases of software projects.

Most studies suggest discarding the old historical data when drift changes are observed and impacts prediction performance negatively. However, our analysis reveal that not all the data should be discarded; not only will it lead to data shrinkage but the data instances that do not contribute to the model should be the only ones discarded keeping the beneficial models. Determining the beneficial instances is a challenge and we leave this as a future study where we will conduct in-depth analysis of the data instances and keep the beneficial ones.

V. Threats to Validity

We discuss below the potential threats to validity that might influence our results and findings. The prediction models and the Python library used are potential threats to validity and we acknowledge that using different prediction models and libraries might produce different results. We used the default parameters of the prediction models during training and different configurations would produce different results. We cannot generalize our results for all datasets especially commercial projects since we studied only open-source projects. We limit the results of our study to the five open-source projects studied. Additionally, investigations of projects with metrics other than process metrics will be considered in a future study. The timeframe used to divide the projects into several versions is also a potential threats to validity. We followed an approach by Yang et al. [25] where we assume software releases are done between 6–8 weeks but do acknowledge that different time periods may produce different results.

VI. Conclusions

Construction of reliable software quality models that span different versions (releases) developed in different time periods is a challenge facing software quality teams. This study investigated the impact of concept drift in software defect prediction. We assess and examine the impacts of different training datasets developed in different time periods on prediction quality. By considering 5 prediction models and 5 open source projects comprising 55 versions, three sets of cross-version experiments are conducted. The results indicate that quality of prediction performance is not stable over time and the random forest and XGBoost models are robust to concept drift when the target project is not constant.

Software quality teams should consider the instability factor when using historical datasets for prediction training. It will be interesting to design a tool that detects concept drift in prediction models before model training. This could be done by leveraging the benefits of incremental learning to reduce the misclassification rate caused by concept drift and this is left as a possible future work.

Acknowledgements

This work has been supported by ELLIIT, a Strategic Area within IT and Mobile Communications, funded by the Swedish Government. The work has also been supported by a research grant for the VITS project (reference number 20180127) from the Knowledge Foundation in Sweden.

REFERENCES

[1] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[2] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[3] F. Xing, P. Guo, and M. R. Lyu, "A novel method for early software quality prediction based on support vector machine," in *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, 2005. ISSRE 2005.* IEEE, 2005, pp. 10–pp.

[4] T. M. Khoshgoftaar and N. Seliya, "Comparative assessment of software quality classification techniques: An empirical case study," *Empirical Software Engineering*, vol. 9, no. 3, pp. 229–257, 2004.

[5] J. Nam, "Survey on software defect prediction," *Master's Thesis*, 2009.

[6] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[7] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.

[8] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on software engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[9] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "The misuse of the nasa metrics data program data sets for automated software defect prediction," *in Proceedings of the 15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*, pp. 96–103, 2011.

[10] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[11] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Tracking concept drift of software projects using defect prediction quality," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 51–60.

[12] S. Amasaki, "On applicability of cross-project defect prediction method for multi-versions projects," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2017, pp. 93–96.

[13] L. L. Minku and X. Yao, "Can cross-company data improve performance in software effort estimation?" in *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*. ACM, 2012, pp. 69–78.

[14] M. A. Kabir, J. W. Keung, K. E. Benniny, and M. Zhang, "Assessing the significant impact of concept drift in software defect prediction," in *Proceedings of the IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 53–58.

[15] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 6, pp. 534–550, 2017.

[16] K. E. Bennin, J. W. Keung, and A. Monden, "On the relative value of data resampling approaches for software defect prediction," *Empirical Software Engineering*, vol. 24, no. 2, pp. 602–636, 2019.

[17] T. M. Khoshgoftaar, L. A. Bullard, and K. Gao, "Attribute selection using rough sets in software quality classification," *International Journal of Reliability, Quality and Safety Engineering*, vol. 16, no. 01, pp. 73–89, 2009.

[18] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Time variance and defect prediction in software projects," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 348–389, 2012.

[19] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[20] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[21] T. Chen, T. He, M. Benesty, V. Khotilovich, and Y. Tang, "Xgboost: extreme gradient boosting," *R package version 0.4-2*, pp. 1–4, 2015.

[22] D. Nielsen, "Tree boosting with xgboost-why does xgboost win" every" machine learning competition?" Master's thesis, NTNU, 2016.

[23] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[24] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *IEEE Transactions on Software Engineering*, 2018.

[25] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 157–168.