2015 IEEE/ACM 3rd International Workshop on Release Engineering

Performance of Defect Prediction in Rapidly Evolving Software

Davide G. Cavezza*, Roberto Pietrantuono*, Stefano Russo*
*DIETI, Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Napoli, Italy
Email: d.cavezza@studenti.unina.it, {roberto.pietrantuono, stefano.russo}@unina.it

***Abstract*—Defect prediction techniques allow spotting modules (or commits) likely to contain (introduce) a defect by training models with product or process metrics – thus supporting testing, code integration, and release decisions. When applied to processes where software changes rapidly, conventional techniques might fail, as trained models are not thought to evolve along with the software.**

In this study, we analyze the performance of defect prediction in rapidly evolving software. Framed in a high commit frequency context, we set up an approach to continuously refine prediction models by using new commit data, and predict whether or not an attempted commit is going to introduce a bug. An experiment is set up on the Eclipse JDT software to assess the prediction ability trend. Results enable to leverage defect prediction potentials in modern development paradigms with short release cycle and high code variability.

I. Introduction

There is a long tradition regarding techniques, models, metrics, and algorithms for software defect prediction. Much literature has been produced around the "best" set of product metrics (e.g., McCabe cyclomatic complexity, OO class metrics, file metrics) and machine learning algorithms (e.g., decision trees, logistic regression) to predict the defectiveness of a software module [1], [2], [3]. With the spread of evolutive and agile development paradigms, several new metrics have been considered regarding the code change/churn as predictors of module defectiveness [4], [5]. Recently, defect prediction is focusing on commit-level prediction, as this paper does, rather than at module-level, with the goal of predicting whether a commit is likely to introduce a defect or not [6], [7].

Defect prediction is a useful means to have a clearer view about product defectiveness. It can support testing or inspection activities, and more generally the V&V process (e.g., by suggesting where to focus the greatest effort), code integration (e.g., by warning developers whenever a commit is likely to introduce a defect), and release decisions (a high predicted defectiveness might indicate the need for further testing or quality assurance activities).

Current approaches for defect prediction consider a frame of data for training models, and then assess the prediction ability on a set of "test data". Resulting model is then supposed to be used on real process data. However, the practical application of such a model to a context where code changes rapidly, might yield poor performance. Indeed, the rapid evolution of code and high frequency of commits is likely to invalidate any model built "statically", as the model will be soon unaligned with the product/process it is supposed to describe. To exploit the potential of defect prediction in this context, a dynamic approach is required.

In this paper, we investigate defect prediction applicability in a rapidly evolving development context. An approach to continuously refine prediction models with new commit data is implemented, and its performance compared with a static prediction. We set up an experiment on the Eclipse JDT software, in order to determine the evolution of the prediction performance as new data are used to train the model. The data consist of a set of metrics calculated on each commit in the history and a label indicating whether it introduced a defect in the software. The model built on them is intended to predict whether or not an attempted commit is going to introduce a bug and to issue a warning in this case.

Results confirm the superiority of a dynamic over a static approach on an experimental dataset of 26,000 samples spanning a period of 13 years. Besides the extent of the improvement in terms of precision and recall, an important outcome is the evidence that defect prediction potentials can be fully exploited in highly dynamic development contexts (e.g., agile building, continuous integration), provided that its models are thought to be just as much dynamic as the code.

II. Dynamic prediction model

Defect prediction is aimed at assessing the defect-proneness of software components individually, starting from a set of metrics usually computed to assess the overall product and process quality [8]. The output of the prediction may be either a binary value indicating whether a component is defective or non-defective, or a component ranking based on its relative defect-proneness. This is the classical view of software defect prediction and we refer to it as module-level prediction.

More recent papers focus on commit-level prediction. The metrics used as predictors are related to single commits issued to a version control system such as Git or Subversion, and likewise the prediction output says whether or not a commit introduces a defect in the software. Examples of commit-level metrics are lines added/deleted, or number of modified files.

The existing works usually employ machine learning algorithms to build a prediction model starting from the existing data. Studies conducted on different systems have shown that there is no model absolutely superior to another, but every

978-1-4673-7070-7/15 $31.00 © 2015 IEEE
DOI 10.1109/RELENG.2015.12

model has different performance in each system. Starting from this perspective, Song et al. [9] propose a module-level prediction framework consisting in a preliminary selection of the best model among a set of alternatives; the evaluation is performed through cross-validation on historical data and the winning model is used to predict the defectiveness of future modules.

A question not fully explored yet is the impact of software changes on the validity of a prediction model. As software is modified over time, a prediction model built at an initial stage of the project may no longer be suitable to represent the correlation between metrics' values and defectiveness. This is particularly critical in agile methodologies, in which changes can be in the order of more than one commit a day.

We propose a commit-level approach that takes into account such dynamics. It is based on the periodic evaluation of prediction performance and the retraining of the employed model if it does not reach a minimum value; the new training is performed using the most recent commit data at one's disposal. More formally, it consists of four steps, repeated cyclically during a system's lifetime:

- *Model selection*: select the best performing model on the most recent training set;
- *(Re)training*: train the selected model;
- *Prediction*: use the trained model to predict defective commits;
- *Evaluation*: evaluate periodically the prediction performance; start a new iteration if the performance is under a defined threshold.

III. Experimental setting

A. Eclipse JDT data collection

As a case study, we choose the popular open-source Java IDE Eclipse JDT. It is one of the most frequently used systems in defect prediction studies [1], [4], so we have a great amount of baseline data to which we can compare the performance of our approach.

Eclipse development data are kept in a public *Git* repository. We use the *CVSAnalY*[1] tool to extract all the commit information from the repository and save it into an SQL database. A total of 26,009 commits are extracted, spanning over more than 13 years from 2001-06-05 to 2014-12-13.

We use the SZZ algorithm [10] to distinguish between defective and non-defective commits. The algorithm consists in the following steps:

1) Find all the bug fixing commits.
2) For each bug fixing commit, locate the files and the lines affected by it.
3) For each modified line, search for the commit that last touched it; this commit is marked as defective, since it introduced a bug.
4) By elimination, all the commits not marked in the previous step are labelled as non-defective.

[1]Available at *https://github.com/MetricsGrimoire/CVSAnalY*

TABLE I
Eclipse dataset summary

Total commits	Timespan	Defective commits	Non-defective commits
26,009	2001-06-05 - 2014-12-13	13,984 (53.77%)	12,025 (46.23%)

To perform step 1, we look in the commit messages for the keywords `bug`, `defect`, `fix`, `patch`, or their variations. To perform step 2, we use the *diff* utility, which lists the lines modified by a commit. Finally, step 3 is performed via the *git blame* utility, that marks each line with the commit that last modified it.

In analyzing diff's output, other studies, like [7], consider only removed or modified lines as buggy lines. However, there are some bugs that do not imply the elimination of code, but only additions: for instance, if-else statements lacking the else branch are corrected by adding new lines without deletions; such bug would not be considered by the algorithm, since there are no deleted lines associated with its fix. Therefore, we choose to consider also the context lines reported by diff, typically 2 or 3 lines before and after the change.

The results of the labelling, along with the other properties of the dataset, are summarized in Table I. We can notice that the commits are almost equally distributed between the defective and the non-defective class.

On each commit, we calculate a set of metrics that we intend to use as predictors of defectiveness. The metrics are a subset of the ones used by Kamei et al. in [6]. Some of them are related to commit complexity, e.g. measured as number of lines added or deleted: the rationale is that more complex commits are more likely to be defective. Other metrics measure the experience of the developer that performed a commit: a more experienced committer is expected to issue more non-defective commits than a less experienced one.

The list of metrics calculated is shown in Table II; some of them, like LA and LD, are automatically computed by *CVSAnalY*, while we need to implement a *Python* script to calculate the remaining ones.

At the end of the data collection process, we obtained a set of 26,009 rows, each corresponding to a commit and containing its date, the values of the 10 metrics computed on it, and a label indicating whether it is defective or not.

B. Experiment execution

To test our approach, we first divide the dataset into several training and test sets. Each training set covers a period of 9 months, while a test set spans over 3 months. The repartition is made so that the i-th training set contains the data immediately preceding the i-th test set. Figure 1 illustrates the repartition.

As in Song et al.'s work [9], the algorithms compared at each *model selection* step are: *OneR*, *J48*, and *NaiveBayes*. These algorithms are all provided by the machine learning software tool *Weka*[2], which we used to run the experiment. The

[2]Available at: *http://www.cs.waikato.ac.nz/ml/weka/*

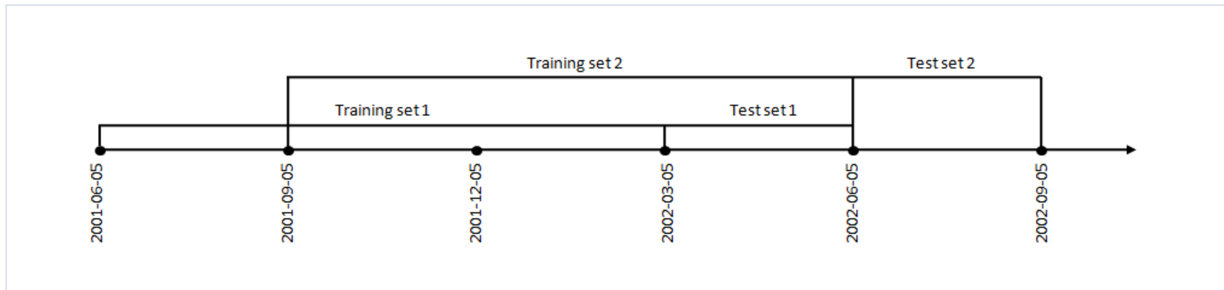

Fig. 1. Repartition of training and test sets

TABLE II
LIST OF METRICS USED

Metric	Description
Number of modified files (NF)	Number of files modified in the commit
Entropy	Scattering of modifications throughout the modified files. A commit requiring the modification of many lines in a single file has a lower entropy than one modifying few lines in many files
Lines added (LA)	Number of lines added in the commit
Lines deleted (LD)	Number of lines deleted in the commit
FIX	Binary value indicating whether or not the commit is a bug fix
Number of developers (NDEV)	Number of developers that changed the files touched by the commit before the commit was issued
AGE	Average time interval between the current and the last change across all the involved files
Number of unique changes (NUC)	Number of unique commits that last changed the involved files
Experience (EXP)	Experience of the developer, measured as the number of changes previously committed by him
Recent experience (REXP)	Number of past commits of the same developer, each weighted proportionally to the number of years between that commit and the measured one

selection is performed by means of a 10-fold cross-validation of the three models, each one iterated 10 times, using one of the training sets as validating data; the model with the best performance is selected. As a performance measure, we use the F-measure, a common metric in defect prediction studies [11], defined as:

$$F\text{-}measure = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

We start the experiment by selecting the best model for the training set 1, and evaluate the F-measure on each of the following test sets. We keep using the same model while the F-measure remains above 0.6. Whenever it goes below the threshold for the test set *i*, we select a new model trained on the training set *i* and use this model on the following test sets.

Finally, we compare the dynamic approach with a conventional "static" one. The latter is obtained by training a model on the training set 1, and using it to make predictions in all the successive test sets.

C. Results

Figure 2 shows the prediction performance over the test sets. The labels on the curve indicate the test sets in which the predictor had to be retrained. It is immediately evident that the dynamic approach significantly outperforms the static one. We had to perform retraining 9 times. J48 was the winning model in all the *model selection* phases. Therefore, the entire experiment was performed with that model. In almost every case, retraining helped to keep performance over the threshold. There were only two exceptions: in test set 3, even after the retraining, the performance stayed below the threshold, although improving consistently; in test set 5, the performance was slightly reduced after the retraining.

Some models performed well for a considerably long period of time: the ones trained on training set 5, 15 and 26 could be successfully used on 10 consecutive test sets each, corresponding to 30 months in Eclipse's history. In those cases, nine months of data were sufficient to predict with good accuracy the defectiveness of changes over a three times longer period.

IV. DISCUSSION AND CONCLUSION

The experiment produced very promising results about the applicability of a dynamic approach to defect prediction model learning. Our next work will be aimed at further investigating and refining the following aspects.

- **Extension of the training window.** We used 9-month-long training sets that allowed us to predict faulty commits in much longer periods. Equal or even better performance could have been reached by properly tuning the training window.
- **Frequency of evaluations.** We chose to evaluate the prediction model on test sets of 3 months. Some projects, however, may require more frequent evaluations, in order to pinpoint performance degradations earlier. This may cause problems in periods of scarce work (e.g. holidays), in which there may be too few commits to obtain a reliable evaluation. A possible solution may involve flexible evaluation intervals, based on the number of commits actually submitted to the system since the last evaluation. The frequency should also be affected by the cost of the evaluations and the eventual retrainings.
- **Lack of knowledge on recent commit defectiveness.** The *retraining* and *evaluation* phases need the actual information on whether or not each commit introduced a defect. In real settings, this could be a problem for the most recent commits, which may have introduced defects

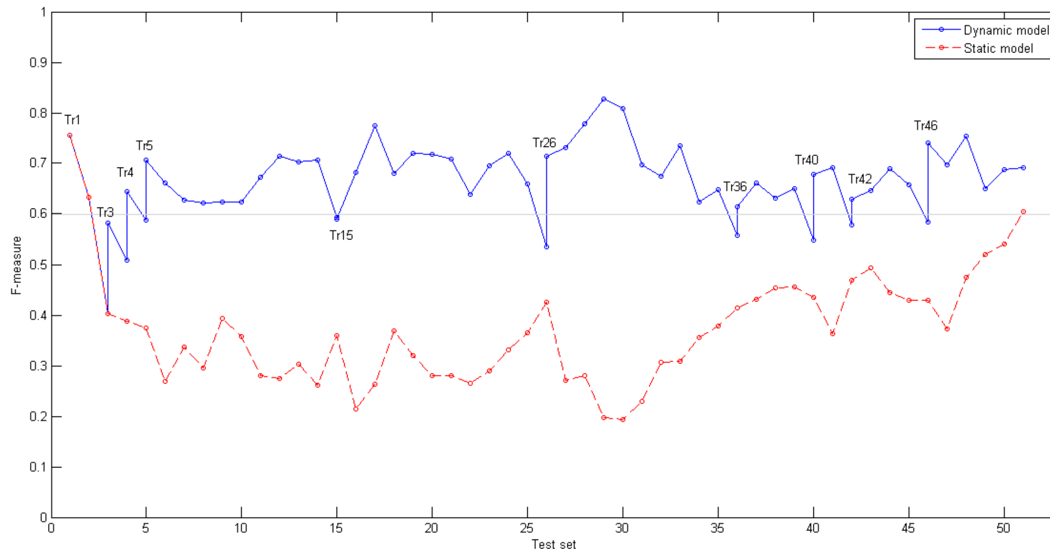

Fig. 2. Prediction performance over the test sets

still undiscovered at the time of evaluation. This is a common issue for the practical applicability of commit-level prediction [7], [6]. Some variations to our approach may be devised, consisting in calculating the expected time between the insertion of a defect and its deletion, and in excluding the most recent commits from evaluation.

- **Choice of the performance measure.** We employed F-measure as it summarizes information on precision and recall in a single value. Some projects may have different requirements in terms of precision and recall, if, for instance, the cost associated to a missed defect is different from the one of a defect-free commit marked as defective. In those cases, a parameterized version of F-measure can be used [11], that weighs precision and recall differently.

The final aim will be to provide a reliable instrument that supports quality assessment and decisions about releases. Commit-level prediction is useful to pinpoint the changes' characteristics that are the most correlated to defect injection in a project (for instance, changes that modify more than 5 files at once are likely to be defective). Once these features are recognized, inspection rules can be enforced before each commit to warn developers against risky violations and to foster the recheck of changes that are likely to be defective. This would help to locate defects as soon as they are introduced. At a later stage, statistics on the commit violations can be combined with other quality measures to take decisions about the opportunity to release the software.

Acknowledgments

This work has been supported by the SVEVIA and DISPLAY research projects of the COSMIC public-private laboratory funded by MIUR under grant no. PON02_00485.

References

[1] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, May 2007, pp. 9–9.

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

[3] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. New York, NY, USA: ACM, 2006, pp. 452–461.

[4] S. Krishnan, C. Strasburg, R. R. Lutz, K. Goseva-Popstojanova, and K. S. Dorman, "Predicting failure-proneness in an evolving software product line," *Information and Software Technology*, vol. 55, no. 8, pp. 1479 – 1495, 2013.

[5] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. New York, NY, USA: ACM, 2008, pp. 181–190.

[6] 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," *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 757–773, 2013.

[7] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1009–1039, Aug. 2014.

[8] G. Carrozza, R. Pietrantuono, and S. Russo, "Defect analysis in mission-critical software systems: a detailed investigation," *Journal of Software: Evolution and Process*, vol. 27, no. 1, pp. 22–49, 2015.

[9] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 356–370, 2011.

[10] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[11] Y. Ma and B. Cukic, "Adequate and precise evaluation of quality models in software engineering studies," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, May 2007, pp. 1–1.