Transfer Defect Learning in Large Industrial Projects

Double Blind Review

ABSTRACT

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Maintaining software*; • **Information systems** → Expert systems;

KEYWORDS

Defect Predictions, Fault Fixing, Software Maintenance, Software Evolution

ACM Reference Format:
Double Blind Review. 018. Transfer Defect Learning in Large Industrial Projects. In *Proceedings of ACM ICSE Conference (ICSE'18).* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

2 RELATED WORK

[Related work are for defect prediction, we have to spin it towards transfer defect learning]

2.1 File, Module and Risky Change Prediction

Existing studies for predicting risky changes within a repository rely mainly on code and process metrics. As discussed in the introduction section, Rosen et al. [21] developed Commit-guru a tool that relies on building models from historical commits using code and process metrics (e.g., code complexity, the experience of the developers, etc.) as the main features. There exist other studies that leverage several code metric suites such as the CK metrics suite [2] or the Briand's coupling metrics [1]. These metrics have been used, with success, to predict defects as shown by Subramanyam *et al.* [23] and Gyimothy *et al.* [5].

Further improvements to these metrics have been proposed by Nagappan *et al.* [13, 15] and Zimmerman *et al.* [26, 27] who used call graphs as the main artifact for computing code metrics with a static analyzer.

Nagappan *et al.* et proposed a technique that uses data mined from source code repository such as churns to assess the quality of a change [14]. Hassan *et al* and Ostrand *et al* used past changes and defects to predict buggy locations [6], [18]. Their methods rely on various heuristics to identify the locations that are most likely to introduce a defect. Kim *et al* [10] proposed the bug cache approach, which is an improved technique over Hassan and Holt's approach [6]. Rahman and Devanbu found that, in general, process-based metrics perform as good as code-based metrics [20].

Other studies that aim to predict risky changes use the entropy of a given change [7, 24] and the size of the change combined with files being changed [8].

These techniques operate at different levels of the systems and may require the presence of the entire source code. In addition, the reliance of metrics may result in high false positives rates. We need a way to to validate whether a suspicious change is indeed risky. In this paper, we address this issue using a two-phase process that combines the use of metrics to detect suspicious risky changes, and code matching to increase the detect accuracy. As we will show in the evaluation section, XXX reduces the number of false positives while keeping good recall. In addition, XXX operates at commit-time for preventing the introduction of faults before they reach the code repository. Through interactions with Ubisoft developers, we found that this integrates well with the workflow of developers.

2.2 Automatic Patch Generation

One feature of XXX is the ability to propose fixes that can help developers correct the detected risky commit. This is similar in principle to the work on automatic patch generation. Pan *et al.* and Kim *et al.* proposed two approaches that extract and apply fix patterns [9, 19]. Pan *et al.* identified 27 patterns and were able to fix 45.7% - 63.6% of bugs using one of the proposed patterns. The patterns found by Kim *et al.* are mined from human-written patches and were able to successfully generate patches for 27 out of 119 bugs. The tool by Kim *et al.*, named PAR, is similar to the second part of XXX where we propose fixes. Our approach also mines potential fixes from human-written patches found in the historical data. In our work, we do not generate patches, but instead propose known patches to developers for further assessment. It has also been shown that patch generation is useful in understanding and debugging the causes of faults [25].

Despite the advances in the field of automatic patch generation, this task remains overly complex. Developers expect from tools high quality patches that can be safely deployed. Many studies proposed a classification of what is considered an acceptable quality patch for an automatically generated patch to be adopted in industry [3, 11, 12].

3 DEFECT PREDICTION AT UBISOFT

[Desribe the approach Prediction approach in a few paragraphs and introduce the cold-start problem. Basically https://github.com/PapersOfMathieuNls/misfire/blob/master/combined.md.pdf]

ICSE'18, May 2018, Gothemburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3.1 Clustering Projects

We cluster projects according to their dependencies. The rationale is that projects that share dependencies are most likely to contain defects caused by misuse of these dependencies. In this step, the project dependencies are analysed and saved into a single NoSQL graph database as shown in Figure ??. A node corresponds to a project that is connected to other projects on which it depends. Dependencies can be *external* or *internal* depending on whether the products are created in-house or supplied by a third-party. For confidentiality reasons, we cannot reveal the name of the projects involved in the project dependency graph. We show the 12 projects in yellow color with their dependencies in blue color in Figure 1. In total, we discovered 405 distinct dependencies that are internal and external both. The resulting partitioning is shown in Figure 2.

At Ubisoft, dependencies are managed within the framework of a single repository, which makes their automatic extraction possible. The dependencies could also be automatically retrieved if the projects use a dependency manager such as Maven.

Once the project dependency graph is extracted, we use a clustering algorithm to partition the graph. To this end, we choose the Girvan–Newman algorithm [4, 17], used to detect communities by progressively removing edges from the original network. Instead of trying to construct a measure that identifies the edges that are the most central to communities, the Girvan–Newman algorithm focuses on edges that are most likely "between" communities. This algorithm is very effective at discovering community structure in both computer-generated and real-world network data [17]. Other clustering algorithms can also be used.

4 CASE STUDY SETUP

In this section, we present the setup of our case study in terms of repository selection, dependency analysis, comparison process and evaluation measures.

4.1 Project Repository Selection

In collaboration with Ubisoft developers, we selected XX major software projects (i.e., systems) developed at Ubisoft to evaluate the effectiveness of XXX. These systems continue to be actively maintained by thousands of developers. Ubisoft projects are organized by game engines. A game engine can be used in the development of many high-budget games. The projects selected for this case study are related to the same game engine. For confidentiality and security reasons, neither the names nor the characteristics of these projects are provided. We can however disclose that the size of these systems altogether consists of millions of lines of code.

4.2 Project Dependency Analysis

Figure 1 shows the project dependency graph. As shown in Figure 1, these projects are highly interconnected. A review of each cluster shows that this partitioning divides projects in terms of their high-level functionalities. For example, one cluster is related to a particular given family of video games, whereas the other cluster refers to another family. We showed this partitioning to 11 experienced software developers and ask them to validate it. They all agreed that the results of this automatic clustering is accurate and reflects well the various project groups of the company. The clusters are used for decreasing the rate of positive. In addition, fixes mined accross projects but within the cluster are qualitative as show in our experiments.

4.3 Process of Comparing New Models

4.4 Evaluation Measures

5 CASE STUDY RESULTS

5.1 Performance

5.2 Cluster Classifier Performance

The clusters computed with the dependencies of each project help to solve an important problem in defect prediction, known as cold start [22]. [Wahab: you need to explain this in one or two sentences] The clusters computed with the dependencies of each project help to solve an important problem in defect prediction, known as cold start [22]. Indeed, as performant as a classifier can be it, after training, it still needs to be train with historical data. While the system is learning, no prediction can be done.

There exist approaches have been proposed to solve this problems, however, they all require to classify *similar* projects by hand and then, manipulate the feature space in order to adapt the model learnt from one project to another one [16].

[Wahab: you need to develop this section. The text is not clear. The graphs are not either. I think defect learning transfer is another topic you can tackle in another paper. One possibility to fix this is to talk about the result of CLEVER without the clustering step]

With our approach, the *similarity* between projects is computed automatically and the results show that we do not actually need to manipulate the feature space. Figures 3, 4 and 5 show the performance of CLEVER classification in terms of ROC-curve for the first thousand commits of the last project (chronologically) for the blue, yellow and red clusters presented in Figure 2. The left sides or the graph are low cutoff (aggressive) while the right sides are high cutoff (conservative). The area under the ROC curves are 0.817, 0.763 and 0.806 for the blue, yellow and red clusters, respectively. In other words, if we train a classifier with historical data from system in the same cluster as the targeted systems, then we do not have to wait to start classifying incoming commits.

[I have better results since then, this is a placeholder]

6 DISCUSSION

In this section, we propose a discussion on limitations and threats to validity.

6.1 Limitations

We identified two main limitations of our approach, XXX, which require further studies.

XXX is designed to work on multiple related systems. Applying XXX on a single system will most likely be less effective. The the two-phases classification process of XXX would be hindered by the fact that it is unlikely to have a large number of similar bugs within the same system. For single systems, we recommend the use of metric-based models. A metric-based solution, however, may turn to be ineffective when applied across systems because of the

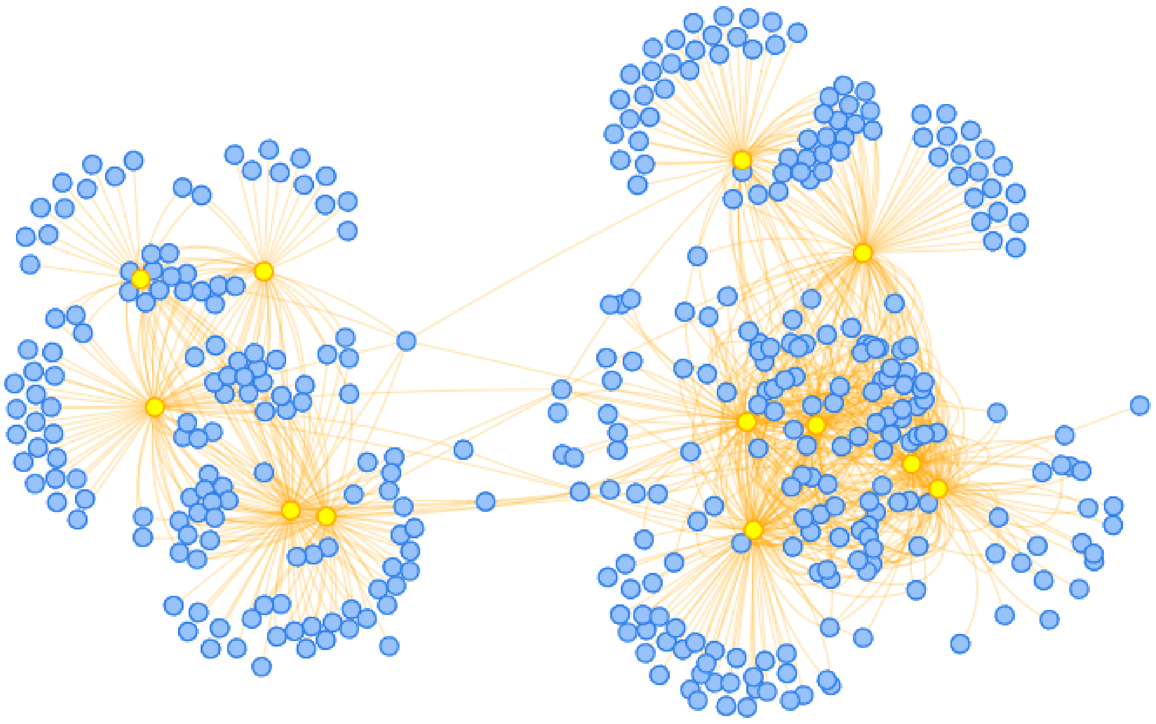

Figure 1: Dependency Graph

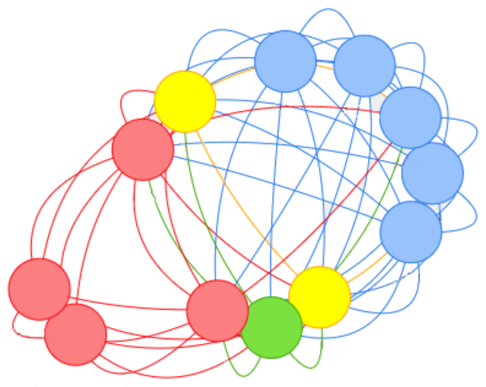

Figure 2: Clusters

difficulty associated with identifying common thresholds that are applicable to a wide range of systems.

The second limitation we identified has to do with the fact that XXX is designed to work with Ubisoft systems. Ubisoft uses C#, C, C++, Java and other internally developed languages. It is however common to have other languages used in an environment with many inter-related systems. We intend to extend XXX to process commits from other languages as well.

6.2 Threats to Validity

The selection of target systems is one of the common threats to validity for approaches aiming to improve the analysis of software systems. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. Because of the industrial nature of this study, we had to work with the systems developed by the company.

The programs we used in this study are all based on the C#, C, C++ and Java programming languages. This can limit the generalization of the results to projects written in other languages,

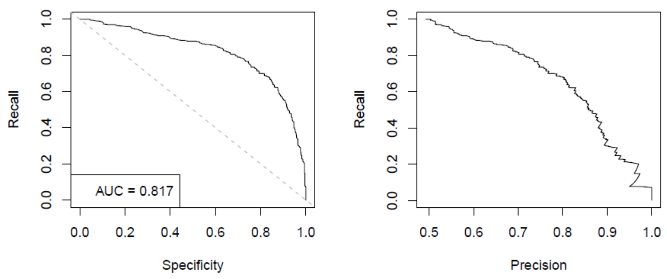

Figure 3: Performances of CLEVER while cold-starting the last project in the blue cluster

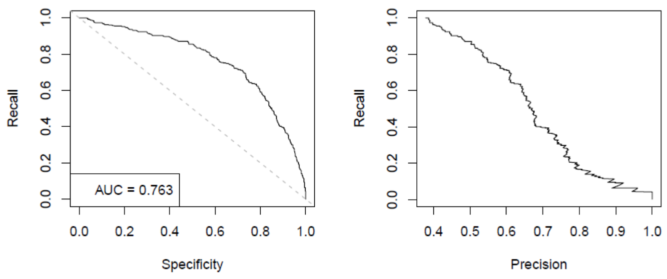

Figure 4: Performances of CLEVER while cold-starting the last project in the yellow cluster

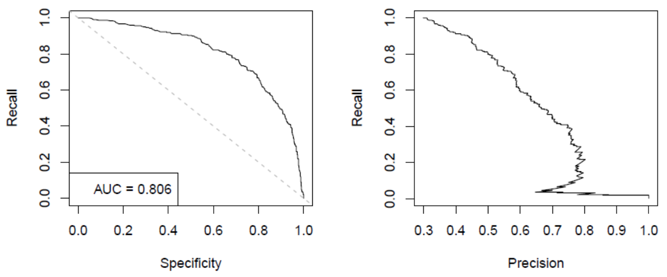

Figure 5: Performances of CLEVER while cold-starting the last project in the red cluster

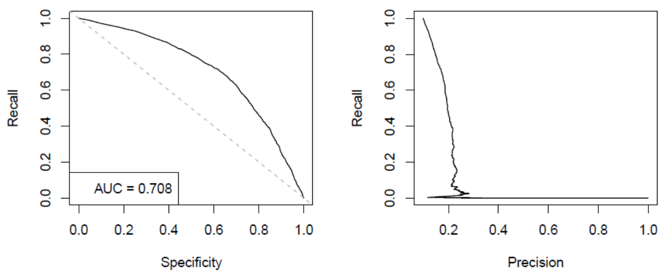

Figure 6: Performances of CLEVER while cold-starting the last project in the blue cluster with model built with data from the red cluster

especially that the main component of XXX is based on code clone matching.

Finally, part of the analysis of the XXX proposed fixes that we did was based on manual comparisons of the XXX fixes with those proposed by developers with a focus group composed of experienced engineers and software architects. Although, we exercised great care in analysing all the fixes, we may have misunderstood some aspects of the commits.

In conclusion, internal and external validity have both been minimized by choosing a set of 12 different systems, using input data that can be found in any programming languages and version systems (commits and changesets).

7 CONCLUSION

In this paper, we presented XXX (Combining Levels of Bug Prevention and Resolution Techniques), an approach that detects risky commits (i.e., a commit that is likely to introduce a bug) with an average of 79.10% precision and a 65.61% recall. XXX combines code metrics, clone detection techniques, and project dependency analysis to detect risky commits within and across projects. XXX operates at commit-time, i.e., before the commits reach the central code repository. Also, because it relies on code comparison, XXX does not only detect risky commits but also makes recommendations to developers on how to fix them. We believe that this makes XXX a practical approach for preventing bugs and proposing corrective measures that integrate well with the developer's workflow through the commit mechanism.

As future work, we want to build a feedback loop between the users and the clusters of known buggy commits and their fixes. If a fix is never used by the end-users, then we could remove it from the clusters and improve our accuracy. We also intend to improve XXX to deal with generated code. Moreover, we will investigate how to improve the fixes proposed by XXX to add contextual information to help developers better assess the applicability of the fixes.

8 REPRODUCTION PACKAGE

For security and confidentiality reasons we cannot provide a reproduction package that will inevitably involve Ubisoft's copyrighted source code. However, the XXX source code is in the process of being open-sourced and will be soon available at https://github.com/ubisoftinc.

ACKNOWLEDGMENTS

We are thankful to the software development team at Ubisoft for their participations to the study and their assessment of the effectiveness of XXX.

REFERENCES

[1] Briand, L. et al. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*. 25, 1 (1999), 91–121. DOI:https://doi.org/10.1109/32.748920.

[2] Chidamber, S. and Kemerer, C. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*. 20, 6 (Jun. 1994), 476–493. DOI:https://doi.org/10.1109/32.295895.

[3] Dallmeier, V. et al. 2009. Generating Fixes from Object Behavior Anomalies. *Proceedings of the international conference on automated software engineering* (2009), 550–554.

[4] Girvan, M. and Newman, M.E.J. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*. 99, 12 (Jun. 2002), 7821–7826. DOI:https://doi.org/10.1073/pnas.122653799.

[5] Gyimothy, T. et al. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*. 31, 10 (Oct. 2005), 897–910. DOI:https://doi.org/10.1109/TSE.2005.112.

[6] Hassan, A. and Holt, R. 2005. The top ten list: dynamic fault prediction. *Proceedings of the international conference on software maintenance* (2005), 263–272.

[7] Hassan, A.E. 2009. Predicting faults using the complexity of code changes. *Proceedings of the international conference on software engineering* (May 2009), 78–88.

[8] Kamei, Y. et al. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*. 39, 6 (Jun. 2013), 757–773. DOI:https://doi.org/10.1109/TSE.2012.70.

[9] Kim, D. et al. 2013. Automatic patch generation learned from human-written patches. *Proceedings of the international conference on software engineering* (May 2013), 802–811.

[10] Kim, S. et al. 2007. Predicting Faults from Cached History. *Proceedings of the international conference on software engineering* (May 2007), 489–498.

[11] Le Goues, C. et al. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. *Proceedings of the international conference on software engineering* (2012), 3–13.

[12] Le, X.-B.D. et al. 2015. Should fixing these failures be delegated to automated program repair? *Proceedings of the international symposium on software reliability engineering* (2015), 427–437.

[13] Nagappan, N. and Ball, T. 2005. Static analysis tools as early indicators of pre-release defect density. *Proceedings of the international conference on software engineering* (New York, New York, USA, May 2005), 580–586.

[14] Nagappan, N. and Ball, T. 2005. Use of relative code churn measures to predict system defect density. *Proceedings of the international conference on software engineering* (2005), 284–292.

[15] Nagappan, N. et al. 2006. Mining metrics to predict component failures. *Proceeding of the international conference on software engineering* (New York, New York, USA, May 2006), 452–461.

[16] Nam, J. et al. 2013. Transfer defect learning. *Proceedings of the international conference on software engineering* (May 2013), 382–391.

[17] Newman, M.E.J. and Girvan, M. 2004. Finding and evaluating community structure in networks. *Physical Review E*. 69, 2 (Feb. 2004), 026113. DOI:https://doi.org/10.1103/PhysRevE.69.026113.

[18] Ostrand, T. et al. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*. 31, 4 (Apr. 2005), 340–355. DOI:https://doi.org/10.1109/TSE.2005.49.

[19] Pan, K. et al. 2008. Toward an understanding of bug fix patterns. *Empirical Software Engineering*. 14, 3 (Aug. 2008), 286–315. DOI:https://doi.org/10.1007/s10664-008-9077-5.

[20] Rahman, F. and Devanbu, P. 2013. How, and why, process metrics are better. *Proceedings of the international conference on software engineering* (2013), 432–441.

[21] Rosen, C. et al. 2015. Commit guru: analytics and risk prediction of software commits. *Proceedings of the joint meeting on foundations*

of software engineering (New York, New York, USA, Aug. 2015), 966–969.
[22] Schein, A.I. et al. 2002. Methods and metrics for cold-start recommendations. *Proceedings of the annual international onference on research and development in information retrieval* (New York, New York, USA, 2002), 253.
[23] Subramanyam, R. and Krishnan, M. 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*. 29, 4 (Apr. 2003), 297–310. DOI:https://doi.org/10.1109/TSE.2003.1191795.
[24] Sunghun Kim, S. et al. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*. 34, 2 (Mar. 2008), 181–196. DOI:https://doi.org/10.1109/TSE.2007.70773.
[25] Tao, Y. et al. 2014. Automatically generated patches as debugging aids: a human study. *Proceedings of the international symposium on foundations of software engineering* (2014), 64–74.
[26] Zimmermann, T. and Nagappan, N. 2008. Predicting defects using network analysis on dependency graphs. *Proceedings of the international conference on software engineering* (New York, New York, USA, May 2008), 531.
[27] Zimmermann, T. et al. 2007. Predicting Defects for Eclipse. *Proceedings of the international workshop on predictor models in software engineering* (May 2007), 9.