Journal of Applied Sciences, Engineering and Technology for Development. Volume 5, Issue 1

Research Article ISSN 2309 - 0936

Analysis of Developers' Network and Change Burst Metrics as Predictors of Software Faults

Malanga Kennedy Ndenga[a], Collins Shikali[b]

[a]*Dedan Kimathi University of Technology, Private Bag – 10143, Dedan Kimathi, Nyeri Kenya.*
[b]*Kibabii University, P.O. Box 1699 – 50200, Bungoma, Kenya.*

Received: November 2020; Revised: March 2021; Accepted: April 2021

Abstract

Introduction: Many software quality metrics that can be used as proxies of measuring software quality by predicting software faults have previously been proposed. However determining a superior predictor of software faults given a set of metrics is difficult since prediction performances of the proposed metrics have been evaluated in non–uniform experimental contexts. There is need for software metrics that can guarantee consistent superior fault prediction performances across different contexts. Such software metrics would enable software developers and users to establish software quality.
Objectives: This research sought to determine a predictor for software faults that requires least effort to detect software faults and has least cost of misclassifying software components as faulty or not given developers' network metrics and change burst metrics.
Methods: Experimental data for this study was derived from Jmeter, Gedit, POI and Gimp open source software projects. Logistic regression was used to predict faultiness of a file while linear regression was used to predict number of faults per file.
Results: Change burst metrics model exhibited the highest fault detection probabilities with least cost of mis-classification of components as compared to the developers' network model.
Conclusion: The study found that change burst metrics could effectively predict software faults.

Keywords:
software faults prediction, software metrics, software quality, developers' network, change burst, prediction effort, misclassification cost

1. Introduction

Measuring quality of a software product is not a simple task. The abstract nature of software and the multiple dimensions of software quality make it difficult to measure quality of a software product. Despite this difficulty, businesses and individuals need information on quality of software when making decisions about adoption of software products. Similarly, software developers need to know whether their software products are of acceptable quality before releasing them. Software quality metrics can be used as proxies of quality to ease the complexity of assessment when measuring quality of software products. Previous studies in Empirical Software Engineering have proposed many software quality metrics as predictors of software bugs in software faults prediction models. Unfortunately, results from most of these studies have not been very useful in helping the community of researchers to arrive at common observations about software quality metrics. The main reason behind this situation has been that the studies have been carried out in extremely varied experimental contexts.

2. Related work

A majority of existing results from software fault prediction studies are derived from experiments whose contexts are not related. As a result, these studies suffer from conclusion validity issues. Ndenga et al. (2015) acknowledge that it is very difficult to confidently accept or reject a hypothesis from a software fault prediction experiment. From literature review of software fault prediction, it was noticed that common observations from already existing research results are not very clear to discern. However, conflicting observations of the same phenomenon could easily be noticed. For example: Nguyen et al.'s (2010) Social Network Analaysis (SNA) observations that contradict observations of Zimmermann and Nagappan (2008); Neuhaus et al. (2007) who believe that vulnerable components are not likely to be vulnerable in future, therefore contradicting the belief that software components that fail now are more likely to fail in future; Malhotra and Jain (2012) and Fukushima et al. (2014) who believe that Random Forest is a superior classifier of software faults while Shin et al. (2011) and Hall et al. (2012) who believe the opposite. Actually, Meneely et al. (2008) question the high success rates claimed by many fault prediction models – most of which cannot demonstrate that they have been replicated successfully with consistent results having been established – or that they are a replication of previous studies achieving consistent results. This situation makes it impossible to compare results from such experiments and draw valid common conclusions about phenomena related to software faults prediction. The extreme variation in experimental context of the studies is exacerbated by uniqueness of software products and the human factor in experiments. Nonetheless, carrying out software fault prediction studies in controlled experimental contexts can help improve conclusion validity of such experiments. Basili et al. (1999) assert that replication of studies, varying of context variables in experiments, and building models that arrive at common observations is the only sure way of creating knowledge form Empirical Software Engineering studies.

Another weakness that has rendered results from existing fault prediction models unreliable is that they have not been evaluated and validated effectively. It is unfortunate that a majority of existing fault prediction studies have evaluated their models only against precision and recall measures. Evaluating a prediction model only against these two measures is not adequate since it does not capture important aspects of a software fault prediction model for example resource utilization during fault prediction and cost of misclassification of components. Software fault prediction models are expected to be economically viable. Therefore the economic viability of models that have only been evaluated using precision, recall and AUC is unknown. According to Jiang, Cukic, and Ma (2008a), precision and recall measures have a weaknesses when used in isolation since they give a one sided story that focuses only on faulty components. A model can actually perform excellently when evaluated with a particular performance measure but perform worse when evaluated with a different kind of measure. This means that software prediction models that have been reported as better performing when evaluated against precision, recall or AUC could as well be worst performers when evaluated against measures like effort of prediction or cost of misclassification of components. Therefore, majority of existing software fault prediction models whose evaluations were inadequately carried out are actually not reliable.

In summary, existing software fault prediction models should be improved so as to enable drawing up of valid common conclusions about phenomena surrounding prediction of software faults. Mende and Koschke (2009) assert that there is a research gap for searching for or improving independent variables for software fault prediction models. Since performance differences between classification algorithms are not necessarily very significant (Arisholm et al., 2010; Lessmann et al., 2008), this study focused on searching for better independent variables i.e., software metrics for predicting software faults.

Although many types of software metrics exist, this research focused on developers' network metrics and change burst metrics. These metrics have been proven to correlate with software faults. At least each category of these metrics has been claimed to be a superior predictor of software faults according to previous studies. For example a study by Nagappan et al. (2010) showed that change burst metrics registered recall of over 90% outperforming other metrics like complexity, code churn and organizational metrics. Zimmermann and Nagappan's (2008) study found out that SNA measures were better predictors of critical binaries than complexity metrics. However according to Nguyen et al. (2010), recall values registered by SNA measures are similar or worse than the recall values registered by complexity metrics when these measures are used as software fault predictors. Ndenga et al. (2019) carried out experiments on Jmeter, gedit, POI and GIMP projects and found out that change burst metrics models registered superior performances for almost all numerical performance measures as compared to change, code churn, organizational and source code metrics. Similarly, Ndenga et al. (2019) found out that change burst metrics models showed the highest fault detection probabilities ranging between 50% and 68% as compared to change, code churn, organizational and source code metrics which exhibited lower probabilities when 20% of code files were examined. Finally, they found out that change burst metrics models had the least cost of misclassification of components in comparison to change, code churn, organizational, and source code metrics for three out of four projects (Ndenga et al., 2019). Studies done by Ndenga et al. (2019) and Nagappan et al. (2010) are all concluding that change burst metrics are superior software fault predictors. Unfortunately, none of the studies has analyzed the performance of change burst metrics to that of developers' network metrics on fault detection probability and the cost of misclassification of components within a common experimental context. Developers' network metrics have previously been shown to also perform well in predicting software faults e.g., by Meneely et al. (2008).

2.1. Objectives

This study sought to achieve the following objectives given developers' network metrics and change burst metrics;

i. To determine a software metric that requires least effort to detect faulty software files.

ii. To determine a software metric that has a minimum cost of misclassifying software files when used to discriminate faulty software files from fault free files.

2.2. Research Questions

Given developers' network software metrics and change burst software metrics at file level granularity, this study sought to answer the following research questions;

RQ 1. Which family of software metrics requires least effort when detecting faulty software files?

RQ 2. Which family of software metrics has least cost of misclassification of software files?

3. Methods

This research is an extension of a study carried out by Ndenga et al. (2019). It extended this study by comparing performance of developers' network metrics to change burst metrics as predictors of software faults. Data sources and experimental research method used to derive change burst metrics data were as described by Ndenga et al. (2019).

3.1. Determining developers' network and change burst metrics

The following section presents a discussion on how developers network metrics were computed. This study re-used the data-set for change burst metrics derived in experiments carried out by Ndenga et al. (2019).

3.1.1. Developers' network metrics

The social structure amongst software developers can be established by carrying out a social network analysis on the artifacts of the software. Poor social interaction amongst developers can exacerbate the likelihood of creating software modules that have low degrees of compatibility – thus yielding software faults. This research studied social networks centered around commits made on a file. In these networks, commiters (developers) formed vertices or nodes of the network whereby two commiters were connected only if they made at least a commit to a common file. The edge of the graph was the geodesic (shortest) path between developers (nodes) who made changes on a common file thus creating simple undirected graphs. Meneely et al. (2008)

argue that developers network metrics try to quantify how well–known a developer is in the context of a project. Like in Meneely et al.'s (2008) and Lopez-Fernandez et al.'s (2004) studies, this research studied the Social Network of commiters basing on *centrality* and *connectivity* metrics as predictors of software faults. Pythons NetworkX [1] package (NetworkX-Developers, 2020) was used to derive developers' network metrics from commit histories of the software projects.

Centrality network metrics measure the indirect connectedness of nodes in a network (Meneely et al., 2008). The two metrics that are used to measure centrality are *closeness-centrality* and *betweenness-centrality* as explained below.

Closeness-centrality of a node *(C(u))* is the reciprocal of the sum of the shortest path distances from u to all other $n-1$ nodes (Freeman, 1978). Freeman (1978) adds that closeness is normalized by the sum of minimum possible distances $n-1$ since sum of distances is dependent on sum of nodes. Freeman (1978) formally defines closeness-centrality *(C(u))* of a node as:

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)}, \tag{1}$$

where $d(v,u)$ is the shortest–path distance between v and u, while n is the number of nodes in the graph. Closeness-centrality can be interpreted as a measurement of the influence of a vertex in a graph whereby vertices – in this case developers – with higher values of this measure can easily spread information into their network (Lopez-Fernandez et al., 2004). Freeman (1978) adds that higher values of closeness indicate higher centrality. The study determined values of *sum*, *average* and *maximum* of closeness–centrality metrics for each file calculated from closeness–centrality values of developers who collaboratively modified the files.

Betweenness–centrality (c_B) of a node v is the sum of the fraction of all pairs of shortest paths that pass through v (Brandes, 2008). It is defined as;

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}, \tag{2}$$

where V is the set of nodes , $\sigma(s,t)$ is the number of shortest (s,t) paths, and $\sigma(s,t|v)$ is the number of those paths passing through some node v other than

[1]https://networkx.github.io/

s, t, given that $s = t, \sigma(s, t) = 1$, and $v \in s, t, \sigma(s, t|v) = 0$ (Brandes, 2008). This study determined values of sum, average and maximum of closeness and betweenness centrality metrics for each file calculated from values of developers who collaboratively modified the files.

Algorithm 1 shows the procedure used by this study to determine sum of betweenness centrality.

4. Analysis

Classification models were built to analyze the capability of developers' network metrics and change burst software metrics to predict software faults at the granularity of a file. Files were labeled as buggy or not-buggy so as to create a class attribute. Logistic regression algorithm was used to predict bug status of each file, while linear regression algorithm was used to predict number of bugs per file. Witten et al. (2011) found out that Logistic regression is a powerful classification algorithm that produces better performance as compared to other classiers like zeroR and naïves Bayes when applied on same dataset. According to Hall et al. (2012), fault predictive models that tend to perform well use simple algorithms like regression. Prediction performance of the models built with software metrics as predictors was evaluated against effort of prediction and cost of mis-classification of components.

4.1. Validity and reliability of research instruments

In order to mitigate potential threats resulting from construct validity – which is about the relationship between theory and observation (Romano and Pinzger, 2011),– this research studied four software projects data spread across a period of five years. It was believed that a recent five year period represents the history of the latest versions of the software projects. In this study, internal validity – which refers to the ability of a data collection instrument to measure what it is intended to measure (Saunders, 2011) – was preserved by using deterministic tools to collect data that guaranteed delivery of reliable results. For computing developers' network metrics and change burst metrics, the same internally developed tool was used. This research was guided by known statistical prediction modeling performance evaluation measures in drawing conclusions about the ability of each kind of software metric to discriminate faulty software files from fault free files.

Algorithm 1: Sum of Betweenness per file

```
Input: $D_m$ // Data matrix for file_id with corresponding unique
       commiter_ids.
Input: $S_{date}$ // start date
Input: $E_{date}$ // end date
$D_{set} \subseteq D_m \leftarrow$ dataset within limits of $S_{date}$ and $E_{date}$;
Output: $F_{c_B}[size\ of\ D_{set}]$ //Array of Sum of Betweenness centrality
        per file_id.
$C_n \leftarrow$ Commit connections amongst developers;
$G \leftarrow (V, E)$ //Undirected graph generated from $C_n$ ;
$V \leftarrow$ Nodes in graph $G$;
foreach file_id f in $D_{set}$ do
    $F_{c_B}[\textbf{file\_id}] \leftarrow \emptyset$ // initialization of Betweenness centrality for
    each file;
    $S \leftarrow \emptyset$ // sum of Betweenness initialization;
    while i¡ size of $D_{set}$ do
        repeat
            foreach committer_id c in V do
                $B_{c_B}[\textbf{file\_id}][\textbf{committer\_id}] \leftarrow \emptyset$ // initialization of
                Betweenness of each committer per file;
                while j¡ size of V do
                    repeat
                        $B_{c_B}[i][j] \leftarrow$ developer Betweenness //Determined
                        by NetworkX's Betweeness–Centrality algorithm.
                        $S \leftarrow S + B_{c_B}[i][j]$;
                        $j \leftarrow j + 1$;
                    until $j > V$;
                end
            end
            $F_{c_B}[i] \leftarrow S$
            $i \leftarrow i + 1$;
        until $i > D_{set}$;
    end
end
return $F_{c_B}$;
```

These measures were effort and cost of misclassification performance. Therefore, conclusion validity – which concerns issues that affect the ability of drawing a correct conclusion from analyzed data (Scanniello et al., 2013) – was preserved. To some extent, the external validity of this study was threatened by the fact that the research studied only four OSS projects. According to Scanniello et al. (2013), external validity focuses on the approximate truth from conclusions that involve generalizations from different experimental contexts (Scanniello et al., 2013). To mitigate threats associated with external validity, the study considered heterogeneous projects in the sense that the projects were developed for different purposes, they were of different sizes, and they were developed by different developers.

5. Results and discussion

The following subsections present a discussion of results realized in this study.

5.1. To determine a software metric that requires least effort in detecting software faults

Using linear regression, the research predicted the number of bugs for each file for the software projects when change burst metrics and developers' network metrics were used as predictors. Cumulative lift graphs were plotted from the prediction results. Cumulative lift graphs are plots used to predict a model's effort of prediction. A cumulative graph shows the percentage of software faults that can be detected when n% of software components is reviewed (D'Ambros et al., 2012). Figures 1, 2, 3, and 4 show the percentage of faults that developers' networks metrics and change burst metrics models would predict when a particular percentage of source code files are inspected for a given software project. Change burst metrics models showed highest fault detection probabilities when 20% of source code files were inspected for the four software projects as compared to developers' network metrics models. For example as shown in figure 1, when 20% of Jmeter's source code files were examined with developers' networks model, approximately a 30% probability of detecting faulty files was realized. For the same project, change burst metric model, showed approximately a 68% probability of detecting faults.

Figure 2 shows that developers' networks metrics performed dismally – similar to a random classifier by predicting 20% of faulty files when 20% of

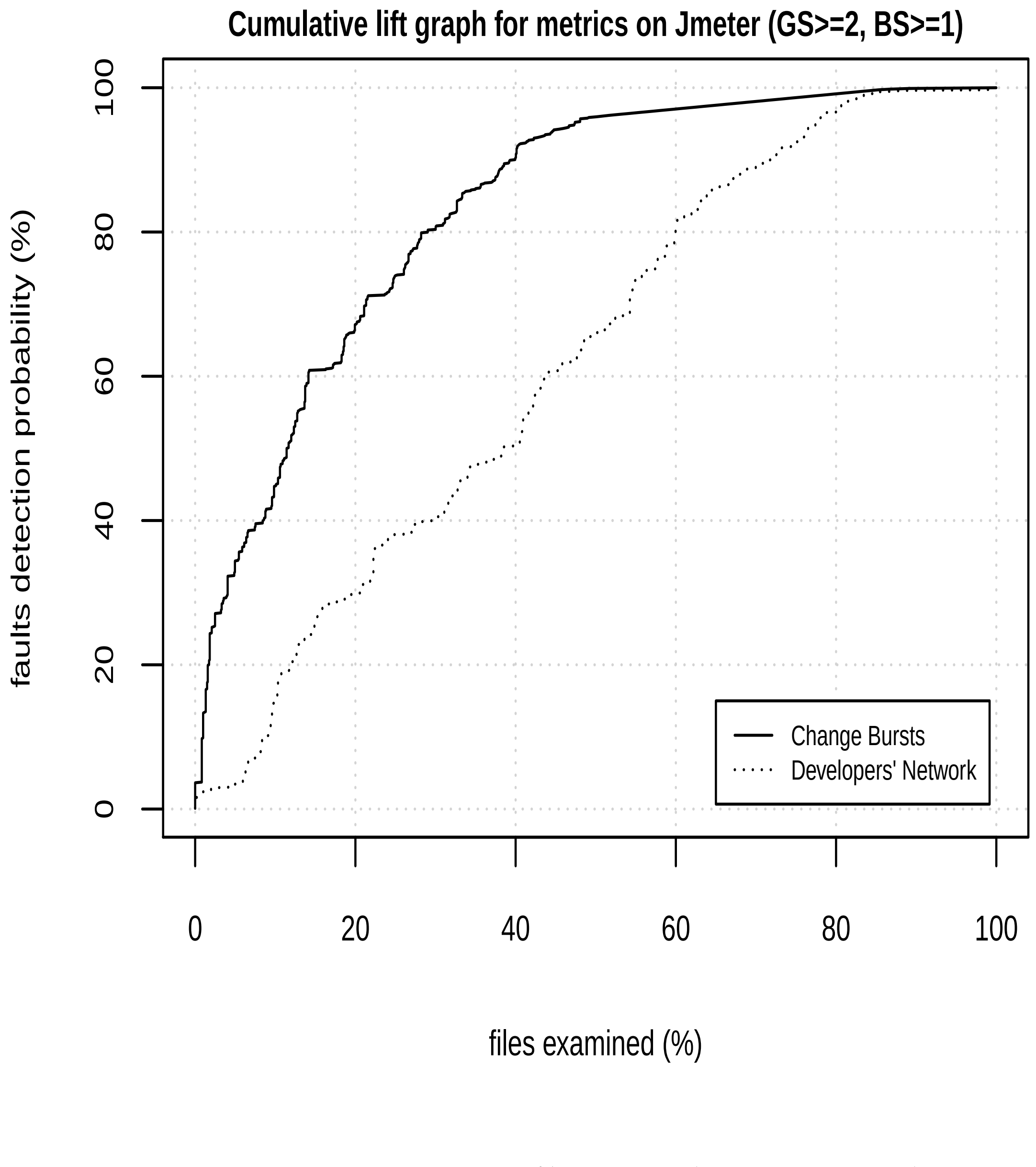

Figure 1: Graph showing effort of inspecting n% of Jmeter ($G_S \geqslant 2$, $B_S \geqslant 1$) files. Examining 20% of Jmeter's source code files with developers' network and change burst metric models showed a 30% and a 68% probability of detecting faults respectively.

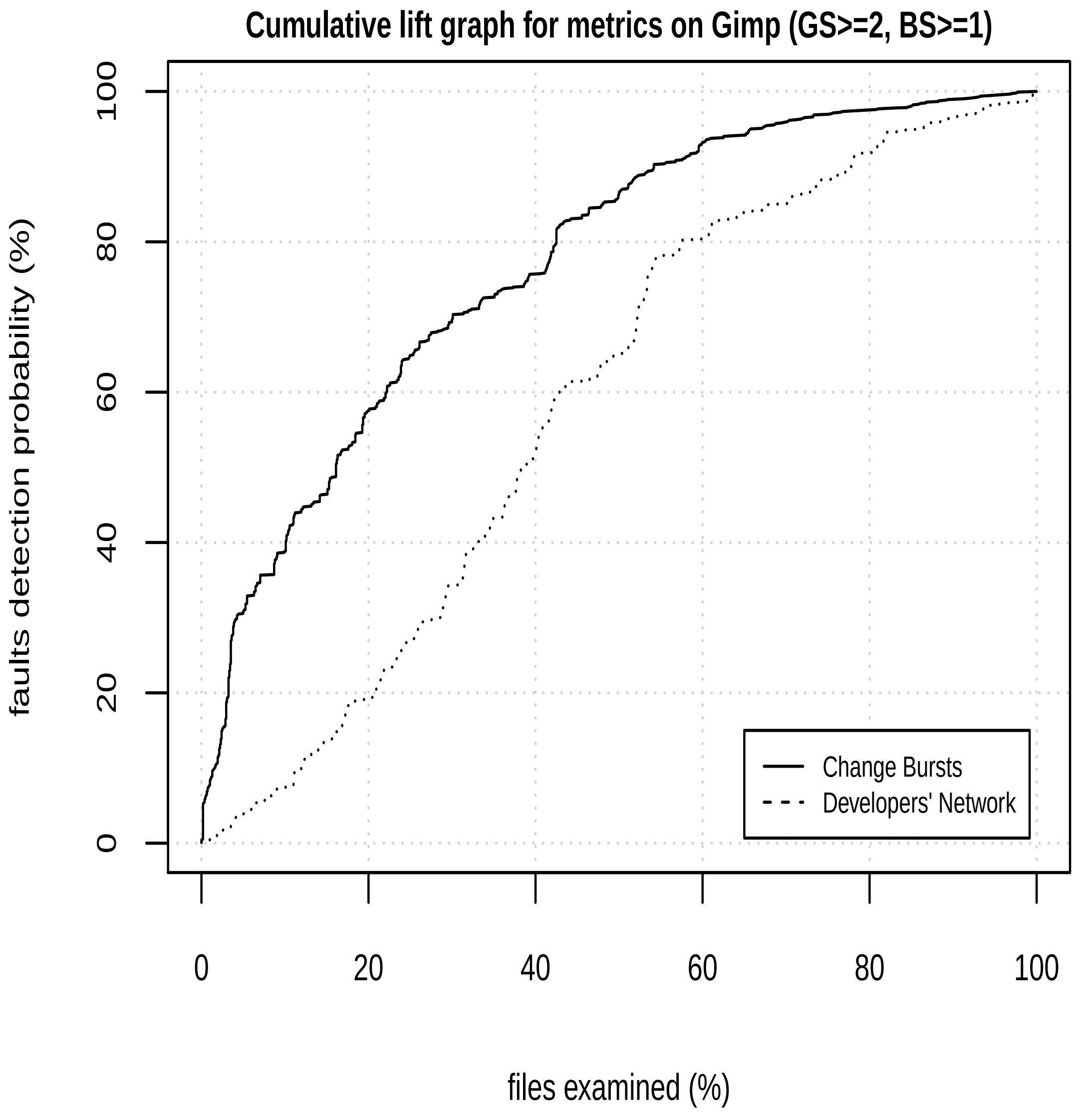

Figure 2: Graph showing effort of inspecting n% of Gimp ($G_S \geqslant 2$, $B_S \geqslant 1$) files. Examining 20% of Gimp's source code files with developers' network and change burst metric models yielded a 20% and a 58% probability of detecting faults respectively.

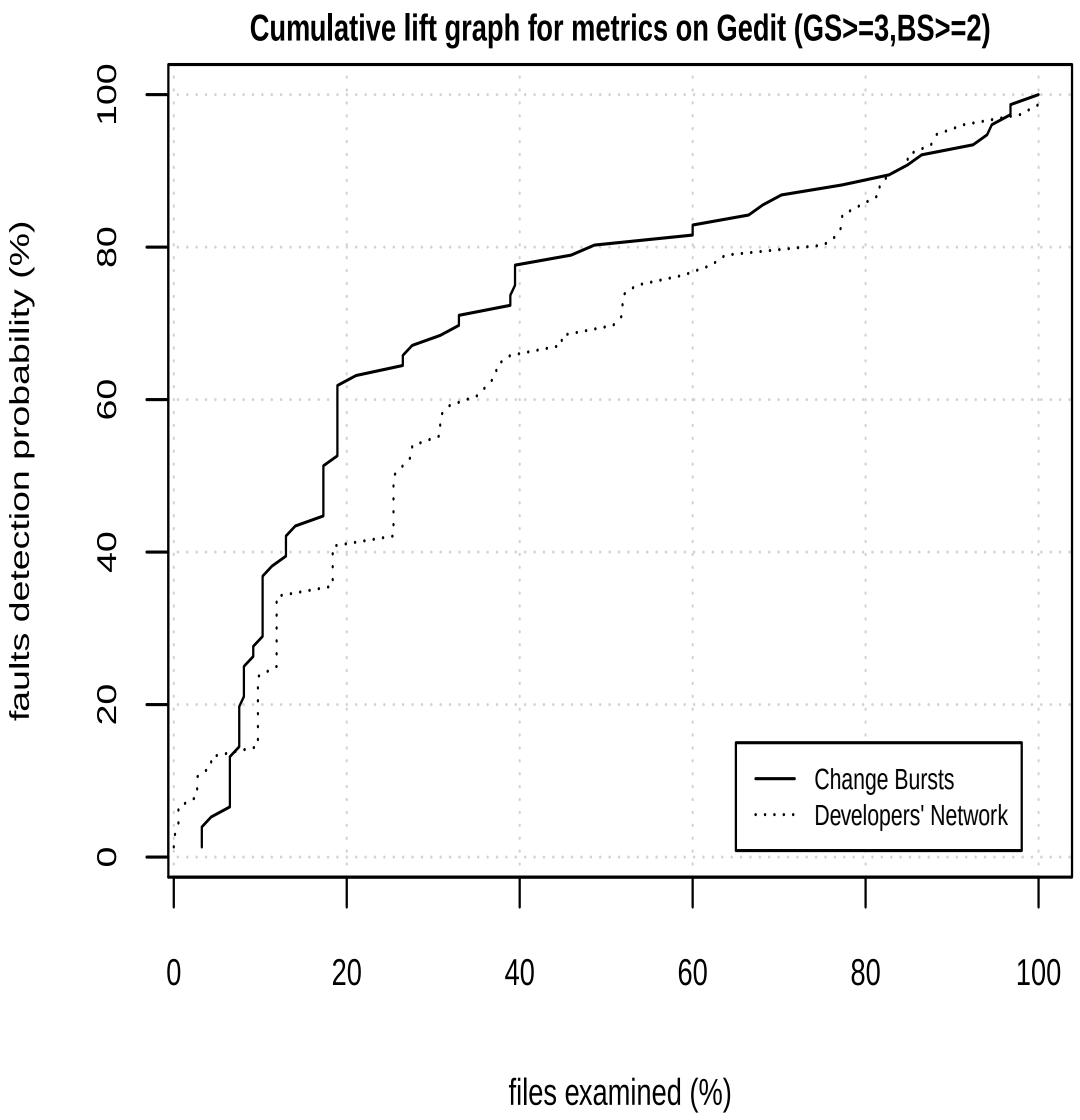

Figure 3: Graph showing effort of inspecting n% of Gedit ($G_S \geqslant 3$, $B_S \geqslant 2$) files. Examining 20% of Gedit's source code files with developers' network and change burst metrics models yielded a 41% and a 62% probability of detecting faults respectively.

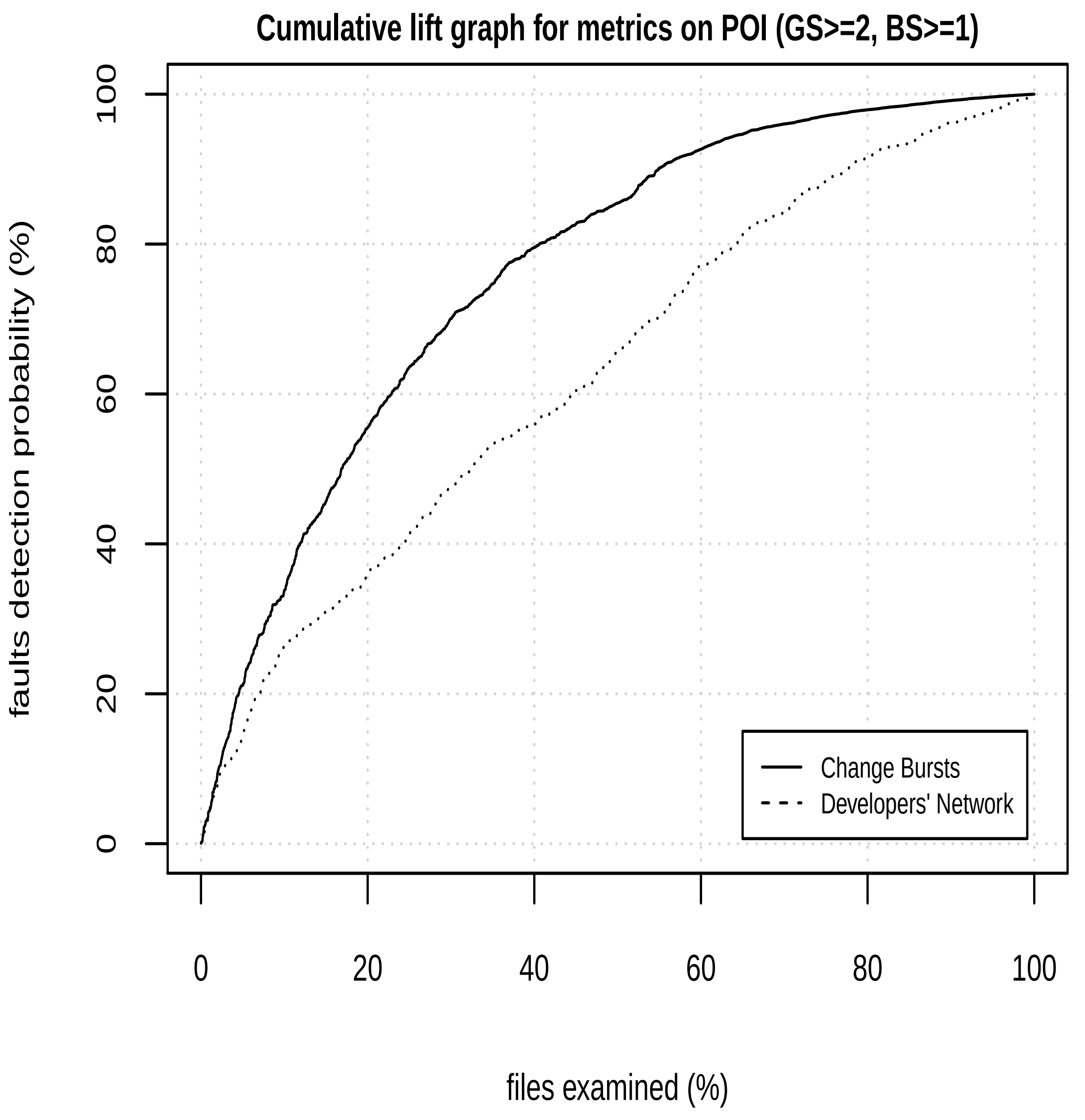

Figure 4: Graph showing effort of inspecting n% of POI ($G_S \geqslant 2$, $B_S \geqslant 1$) files. Examining 20% of POI's source code files with developers' network and change burst metric models yielded 35% and 55% probabilities of detecting faults respectively.

Gimp's source code files were inspected while change burst metrics were able to detect 58% of faulty files when 20% of source code files were inspected for the same project.

According to the graphs in Figures 1, 2, 3, and 4, over 70% of software faults could be detected with models built using change burst metrics upon inspection of only 40% of files. This finding reveals that change burst metrics models require least effort to detect many faulty software files therefore addressing the first research question (**RQ 1**).

5.2. To determine a software metric that has minimum cost of misclassification of components when predicting software faults

During classification process, some faulty components could erroneously be classified as fault free or vice versa. Such a misclassification is costly. Misclassifying a fault-free component as faulty may call for unnecessary quality checks that result into higher development costs. Similarly, classifying a faulty component as fault-free may result in software system failure. Drummond and Holte (2006) proposed the use of cost curves as a reliable means of establishing the cost associated with using a particular classifier. A cost curve describes the performance of a classifier based on the cost of misclassification of components where the x–axis represents the probability cost function, while the y–axis represents the normalized expected cost of misclassification (Drummond and Holte, 2006; Jiang et al., 2008b). A prediction of number bugs was determined using each family of software metrics. Using R's ROCR package (Sing et al., 2007) and the predictions data, cost curves were plotted for each model showing the cost of misclassification of components associated with the model. According to Jiang et al. (2008b), the purpose of cost curve analysis is to enable developers to select software prediction models which minimizes overall misclassification cost, i.e., minimize the area under the lower envelope boundary.

Scrutiny of graphs in figures 5, 6, 7, and 8 reveals that cost curves for change burst metrics models are closer to the probability cost function axis. This finding validates that as compared to developers' networks models, change burst models have the least normalized cost of misclassification of source code files. The finding addressed the second research question (**RQ 2**).

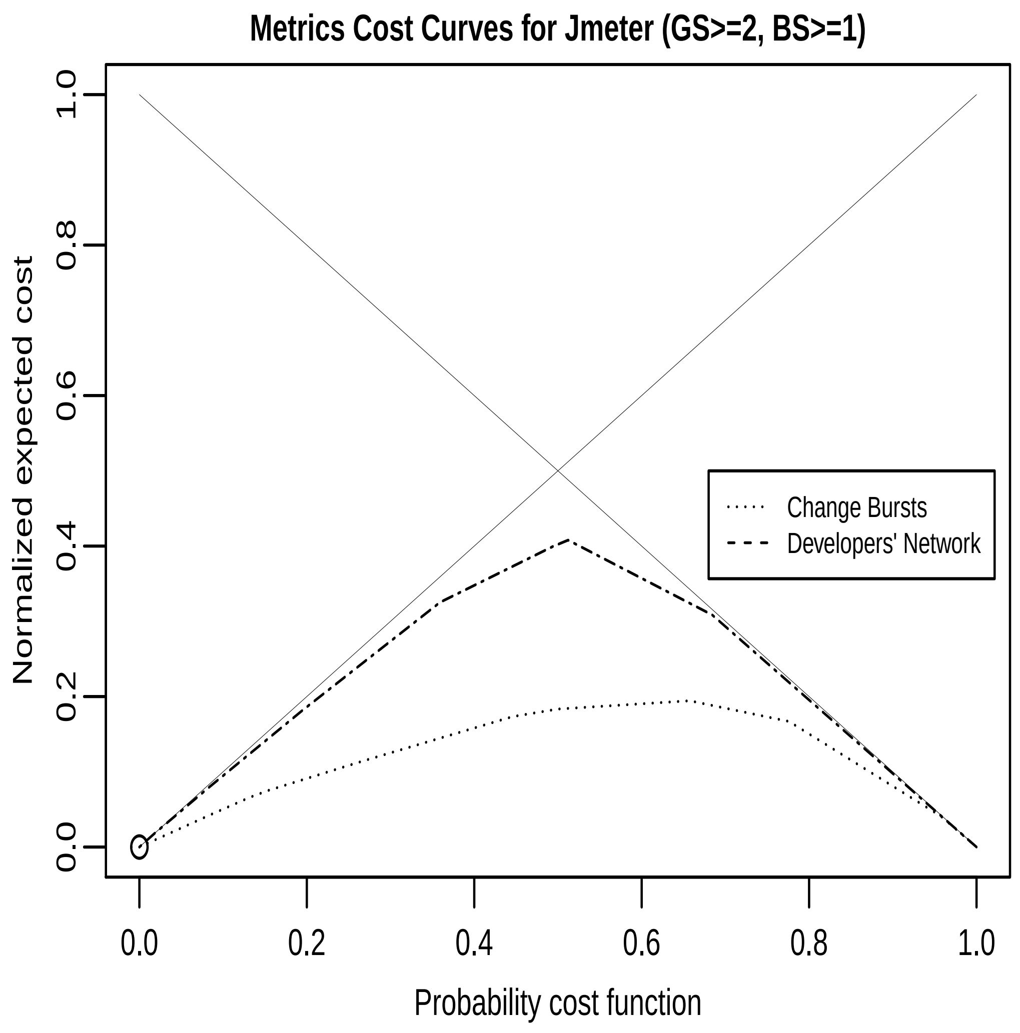

Figure 5: Cost curves for Jmeter project at $G_S \geqslant 2$ and $B_S \geqslant 1$. The cost curve for change bursts metrics' model is closest to the probability cost function axis, hence it has minimum normalized cost of misclassification of source code files.

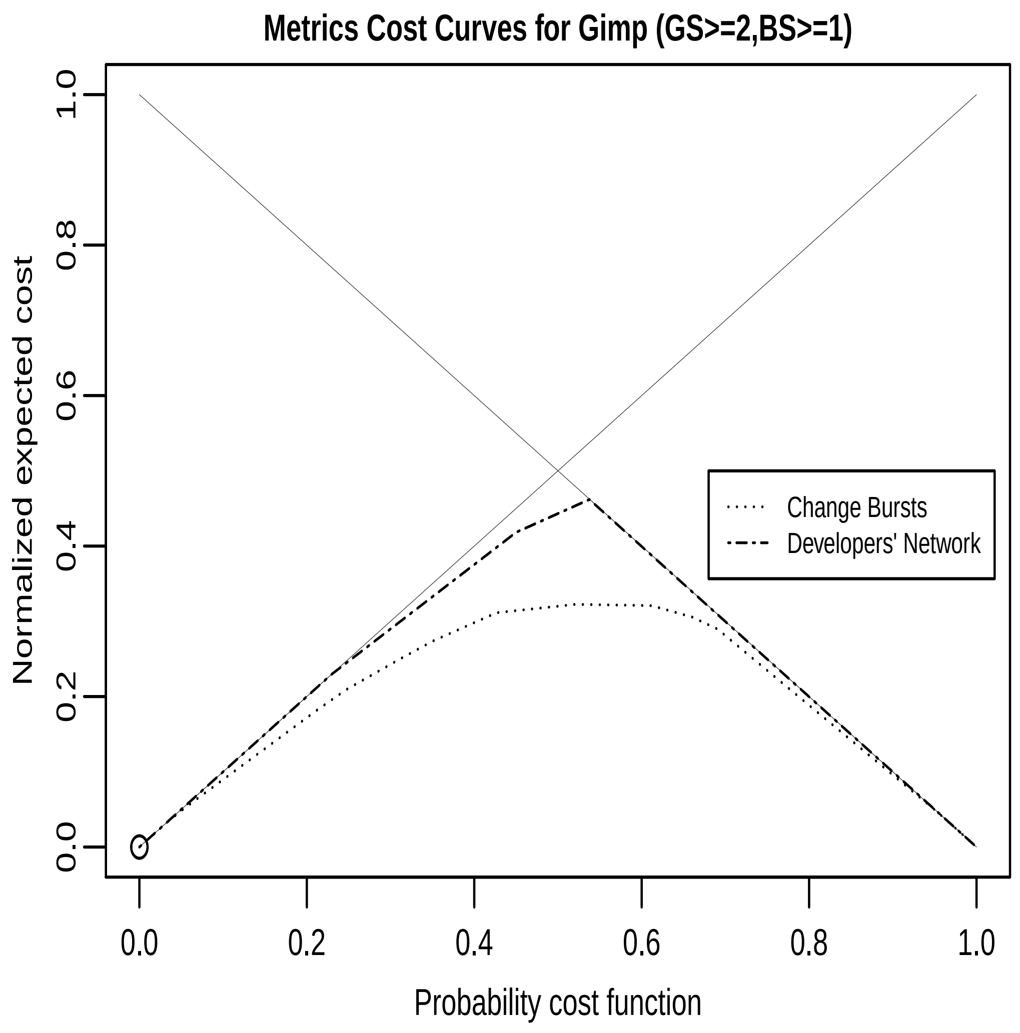

Figure 6: Cost curves for Gimp project at $G_S \geqslant 2$ and $B_S \geqslant 1$. The cost curve for change bursts metrics' model is closest to the probability cost function axis, hence it has minimum normalized cost of misclassification of source code files.

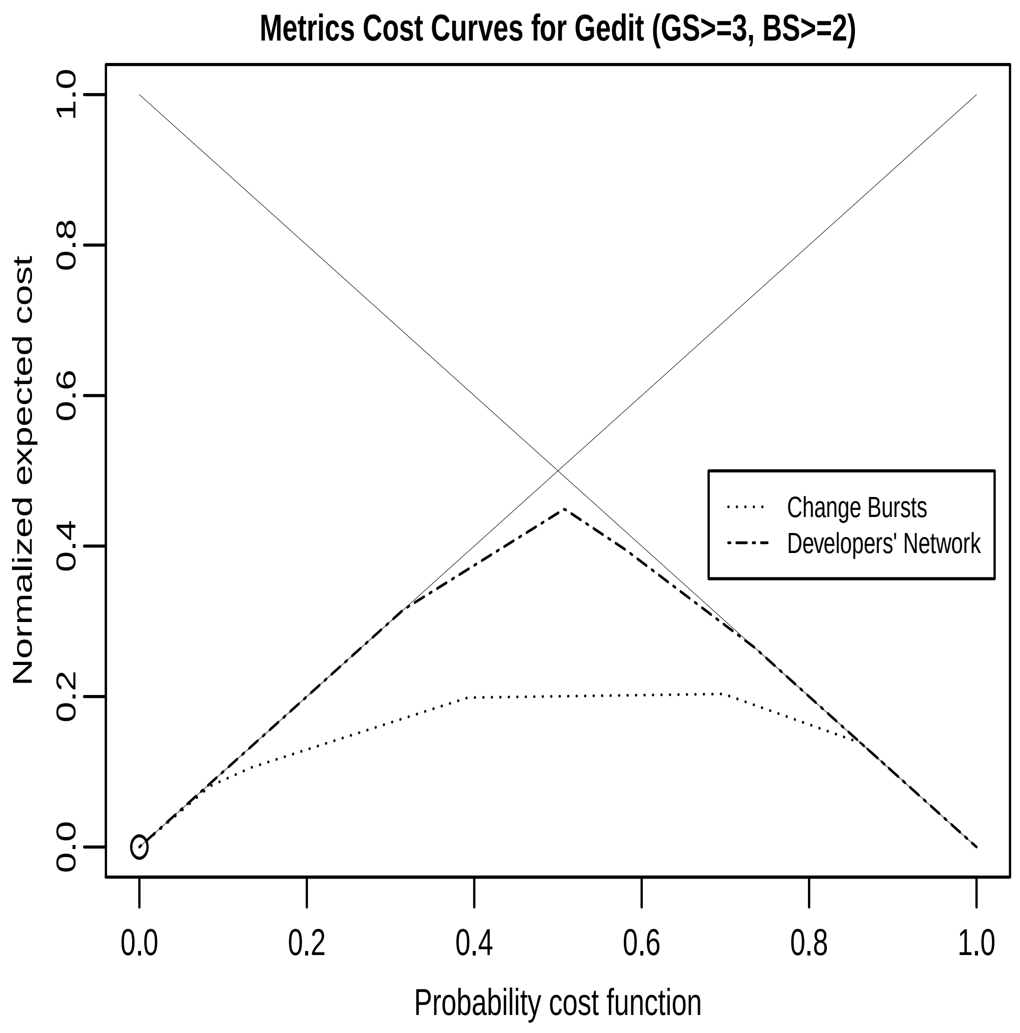

Figure 7: Cost curves for Gedit project at $G_S \geqslant 3$ and $B_S \geqslant 2$. The cost curve for change bursts metrics' model is closest to the probability cost function axis, hence minimum normalized cost of misclassification of source code files.

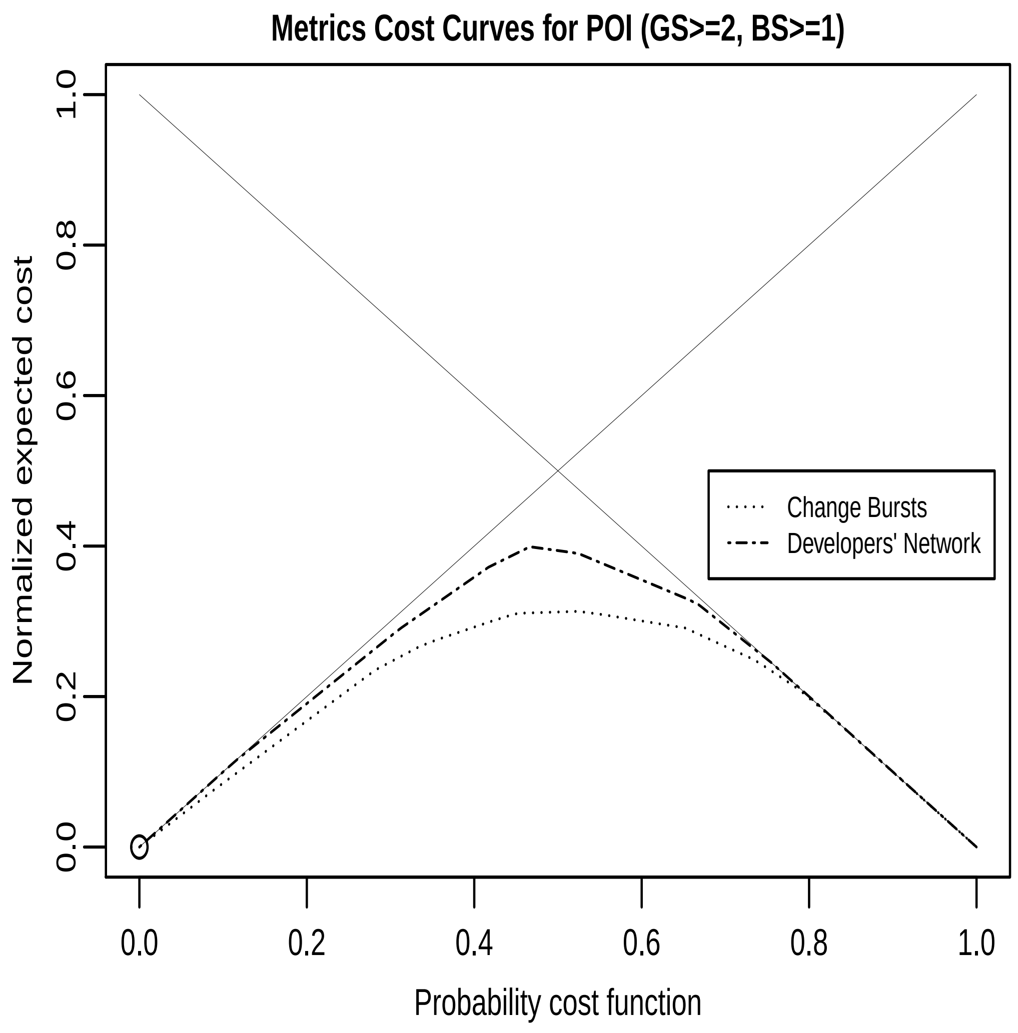

Figure 8: Cost curves for POI project at $G_S \geqslant 3$ and $B_S \geqslant 2$. The cost curve for change bursts metrics' model is closest to the probability cost function axis, hence it has minimum normalized cost of misclassification of source code files.

6. Conclusion

As mentioned at the beginning, several software characteristics for example software metrics have been found to be predictors of software faults according to results from existing studies. The first objective of this study was to determine a predictor for software faults between developers' network metrics and change burst metrics that requires least effort to detect software faults at file–level. Fixing software bugs is a tedious and expensive process that accounts for over 25% of the global software development cost (Britton et al., 2013). Therefore, software faults prediction models should be able to minimize the effort required to detect software faults. Easily detecting and subsequently fixing software faults has a net effect of reducing overall bug fixing cost. Taking longer to fix software faults generally escalates the cost of fixing the faults (Baziuk, 1995; Boehm et al., 1976). Therefore, prediction models that require least effort in detecting majority of software faults should be adopted in Software Engineering industry. These concerns lead this study to seek addressing the first research question.

The motivation of the first research question was to determine and compare the effort required by developers' network metrics and change burst metrics in discriminating between faulty and fault free files within the same experimental context. Different studies have claimed superiority of the two metrics in prediction of software faults. For example Zimmermann and Nagappan's (2008) study found out that Social Network Analysis measures could identify twice as many critical binaries as identified by Complexity metrics. But Nguyen et al. (2010) claimed that recall values observed when Social Network Analysis measures are used, are either equivalent or worse than recall values when complexity metrics are used. Nagappan et al. (2010) found out that change burst metrics registered Recall of over 90% outperforming other metrics like complexity, code churn and organizational metrics. However none of the studies tested the software fault prediction performance of the two metrics in the same experimental context. It should be recalled that most of the reviewed software fault prediction studies did not specify the effort required by their models in predicting software faults. In the long run, it has been difficult to determine the economic viability of such models before adopting them. Findings from this study showed that change burst metrics exhibited the highest fault detection probabilities as compared to developers' network metrics. Fault detection probabilities for change burst metrics ranged from 55% to 68% while those of developers' network metrics ranged from 20% to 41% when only 20% of source code files for the four projects were examined. Therefore change burst metrics required least effort to detect software faults as compared to developers' network metrics.

To determine a predictor for software faults that has a minimum cost of misclassification of software components between developers' network metrics and change burst metrics was the second specific objective of this study. Misclassifying a software component as faulty or not faulty has a cost associated with it. For example, classifying a fault free component as faulty would result into unnecessary costly quality assurance activities on the component. Similarly, classifying a faulty software component as not faulty may result into costly software system failures. The risk associated with adopting a particular software fault prediction model should be known in advance by users. Models with the least cost of misclassification of components are desirable. In achieving this objective, the study addressed the second research question. The motivation for this question was to determine and compare the cost of misclassification of software files associated with change burst metrics and developers' network metrics when they are used to discriminate faulty files from fault free files within the same experimental context. The study found out that models built with change burst metrics had an overall least cost of misclassification of software source code files in comparison to models built with developers' network metrics.

In conclusion, this study found out that change burst metrics are superior predictors of software faults as compared to developers' network metrics. Considering this superiority, the study recommends that change burst metrics should be considered by practitioners in Software Engineering when predicting software faults.

Contributions: This research has made contributions to theory and practice in Empirical Software Engineering. Through synthesis of literature review, the study has contributed to knowledge by facilitating a better understanding of software fault prediction. The technique of generating software metrics developed by this research is a methodological contribution since it can be reused by other researchers to advance research in this field. This study has also made a theoretical contribution by introducing new view points by analyzing data from Gedit, Jmeter, POI and GIMP which are OSS projects that are rarely studied. Finally, this study has contributed to practice through the technique of predicting software faults that it proposes. If this technique is applied in software development industry, it could be used to prioritize software files for inspection purposes.

Future work: Future work will entail replicating this study in contexts where many heterogeneous software projects will be studied with the hope of finding results that can be generalized.

Funding/supporting sources: This study was supported by Dedan Kimathi University of Technology and University of Paris 8.

References

Arisholm, E., Briand, L. C., Johannessen, E. B., 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software 83 (1), 2–17.

Basili, V. R., Shull, F., Lanubile, F., 1999. Building knowledge through families of experiments. IEEE Transactions on Software Engineering 25 (4), 456–473.

Baziuk, W., 1995. Bnr/nortel: path to improve product quality, reliability and customer satisfaction. In: Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on. IEEE, pp. 256–262.

Boehm, B. W., Brown, J. R., Lipow, M., 1976. Quantitative evaluation of software quality. In: Proceedings of the 2nd international conference on Software engineering. IEEE Computer Society Press, pp. 592–605.

Brandes, U., 2008. On variants of shortest-path betweenness centrality and their generic computation. Social Networks 30 (2), 136–145.

Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T., 2013. Reversible debugging software. University of Cambridge-Judge Business School, Tech. Rep.

D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Software Engineering 17 (4-5), 531–577.

Drummond, C., Holte, R. C., 2006. Cost curves: An improved method for visualizing classifier performance. Machine learning 65 (1), 95–130.

Freeman, L. C., 1978. Centrality in social networks conceptual clarification. Social networks 1 (3), 215–239.

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., Ubayashi, N., 2014. An empirical study of just-in-time defect prediction using cross-project models. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 172–181.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2012. A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering 38 (6), 1276–1304.

Jiang, Y., Cukic, B., Ma, Y., 2008a. Techniques for evaluating fault prediction models. Empirical Software Engineering 13 (5), 561–595.

Jiang, Y., Cukic, B., Menzies, T., 2008b. Cost curve evaluation of fault prediction models. In: Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on. IEEE, pp. 197–206.

Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering 34 (4), 485–496.

Lopez-Fernandez, L., Robles, G., Gonzalez-Barahona, J. M., et al., 2004. Applying social network analysis to the information in cvs repositories. In: International workshop on mining software repositories. IET, pp. 101–105.

Malhotra, R., Jain, A., 2012. Fault prediction using statistical and machine learning methods for improving software quality. Journal of Information Processing Systems 8 (2), 241–262.

Mende, T., Koschke, R., 2009. Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering. ACM, p. 7.

Meneely, A., Williams, L., Snipes, W., Osborne, J., 2008. Predicting failures with developer networks and social network analysis. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, pp. 13–23.

Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., Murphy, B., 2010. Change bursts as defect predictors. In: Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on. IEEE, pp. 309–318.

Ndenga, M. K., Ganchev, I., Mehat, J., Wabwoba, F., Akdag, H., 2019. Performance and cost-effectiveness of change burst metrics in predicting software faults. Knowledge and Information Systems 60 (1), 275–302.

Ndenga, M. K., Jean, M., Ganchev, I., Franklin, W., 2015. Assessing quality of open source software based on community metrics. International Journal of Software Engineering and Its Applications 9 (12), 337–348.

NetworkX-Developers, 2020. Overview networkx. `https://networkx.github.io/`, accessed: 2020-05-04.

Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A., 2007. Predicting vulnerable software components. In: Proceedings of the 14th ACM conference on Computer and communications security. ACM, pp. 529–540.

Nguyen, T. H., Adams, B., Hassan, A. E., 2010. Studying the impact of dependency network measures on software quality. In: Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, pp. 1–10.

Romano, D., Pinzger, M., 2011. Using source code metrics to predict change-prone java interfaces. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE, pp. 303–312.

Saunders, M. N., 2011. Research methods for business students, 5/e. Pearson Education India.

Scanniello, G., Gravino, C., Marcus, A., Menzies, T., 2013. Class level fault prediction using software clustering. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, pp. 640–645.

Shin, Y., Meneely, A., Williams, L., Osborne, J. A., 2011. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering 37 (6), 772–787.

Sing, T., Sander, O., Beerenwinkel, N., Lengauer, T., 2007. Package rocr: visualizing the performance of scoring classifiers. See: http://rocr.bioinf.mpi-sb.mpg.de.

Witten, I. H., Frank, E., Hall, M. A., 2011. Data Mining: Practical Machine Learning Tools and Techniques. Elsevier.

Zimmermann, T., Nagappan, N., 2008. Predicting defects using network analysis on dependency graphs. In: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. IEEE, pp. 531–540.