An Empirical Study for Prioritizing Quality Assurance : A Replication Work

Md Tajmilur Rahman[1], Louis-Philippe Querel[1]
[1] *Dept. of Computer Science and Engineering, Concordia University, Québec, Canada*

***Abstract*—Quality Assurance by predicting defects in software systems is a highly studied area in the current trend of software engineering research. Predicting defect prone files or modules are still not sufficient to save time to identify actual defects inside the code that needs to be modified to fix a bug. In this case predicting actual defect-inducing change could be a better approach to resolve this insufficiency and may reduce the effort of fixing defects. One research work had already been done which focused on the issue of predicting defect-inducing changes in change/churn level. This influenced us to replicate the work on the small scale open source project Django. We correlate actual bug fixing changes with commits to improve quality of the dataset. A zero-R model has been implied to validate our logistic regression model. The result of our replication on Django project is different for some of the factors than the original work and we found that some of the defect-inducing factors from the original work have a defect-reducing characteristic in our replication with Django.**

I. Introduction

Quality assurance in software development plays a prominent role for a software to turn it into a successful business product. Doing business with a software product by providing services or by selling the software itself demands high level of quality assurance. Quality can be assured in software industries by using unit tests, black-box, white-box, integration, functional and system level testing. Testing is not just the end of the quality assurance but the start of working for assuring continued quality of a software system. Testing produces list of regressions and flaws throughout the system that keeps the development team busy for fixing bugs to make the system stable. Generally this fixing and testing can be practised during the stabilization period once all new feature development works are done. We can estimate development time and cost prior to a development cycle but the effort that a stabilization period consumes for fixing does influence the cost of software product and reputation of the software industry.

In current trend many researchers are focusing on predicting defects in the code-base of software systems [1]. In some contexts this approach can be useful but they have drawbacks. Predicting change level bugs does not specify the amount of work and cannot minimize the effort a great deal. However comparatively predicting the particular change that could be defect inducing would be more efficient to reduce the effort required for fixing bugs around the time of release. This is really important for reducing the required stabilization time and releasing the software product sooner to beat competitors to the market.

As most of the quality assurance research works are based on defect prediction on change level we would like to predict defect inducing changes [2]. We would like to illustrate the prediction of defect inducing changes to enhance software quality by replicating the approach proposed by Kamei et al. [3]. We would like to do this work on Django, a relatively small project. We would follow these two main research questions:

RQ1, Prediction: How well can we predict defect-inducing changes?

Kamei and his co-researchers used six open source projects and five commercial projects. We would like to see how well the model predicts defect-inducing changes in a different software project.

RQ2, Major Characteristics: What are the major characteristics of defect-inducing changes?

To answer this research question we want to observe the factors having the largest impact on the predictions. We want to identify the major characteristics of the defect inducing changes and would like to see if the number of files and whether or not the change fixes a defect are risk-increasing factors, also if the average time interval since the previous change is a risk-decreasing factor in RQ1.

II. Background and Related Work

Software engineers and researchers have been concerned since many years with the quality assurance of software systems. Many approaches have been followed and scientists have tried from many different angle to predict defects in software to enhance it's quality. Earlier in 2005 Nagappan and T. Ball did an interesting research at Microsoft to predict file level defect density [4] based on relative code-churn measures. Subsequently, an approach for predicting defect-prone components was studied by Nagappan et al. In this study they showed that the change bursts can predict defect-prone components in significantly high rate in comparison to other measures like code-churn or organizational structure.

Many other researchers have used change measure, for example Mockus and Weiss [5] used change measures such as the number of subsystems that are changed, number of modified lines of code and the number of modification requests. Now a days churn factor measure became a very traditional approach in predicting defects. From a slightly different angle Sunghun Kim [6] proposes an approach to deal with the noise in defect data as he could realize that automatically collected defect data based on the change logs could include noises.

Many studies have been performed comparing open source and commercial projects. Briand et al. [7] did an analysis to compare the relationship between software design and

Table I: Summary of Change Measures

Dim.	Name	Definition	Rationale	Related Work
Diffusion	NS	Number of modified subsystems	Changes modifying many subsystems are more likely to be defect-prone.	The defect probability of a change increases with the number of modified subsystems [5].
	ND	Number of modified directories	Changes touching many files are more likely to be defect-prone.	The higher the number of modified directories, the higher the chance that a change will induce a defect [5].
	NF	Number of modified files	Changes touching many files are more likely to be defect-prone.	The number of classes in a module is a good featureof post-release defects of a module [9].
	Entropy	Distribution of modified code across each file	Changes with high entropy are more likely to be defect-prone, because a developer will have to recall and track large numbers of scattered changes across each file.	Scattered changes are more likely to introduce defects [10], [11].
Size	LA	Lines of code added	The more lines of code added, the more likely a defect is introduced.	Relative code churn measures are good . indicators of defect modules [12], [4]
	LD	Lines of code deleted	The more lines of code deleted, the higher the chance of a defect.	
Purpose	FIX	Whether or not the change is defect fix	Fixing a defect means that an error was made in an earlier implementation, therefore it may indicate an area where errors are more likely.	Changes that fix defects are more likely to introduce defects than changes that implement new functionality [13], [14].
History	NDEV	The number of developers that changed the modified files	The larger the NDEV, the more likely a defect is introduced, because files revised by many developers often contain different design thoughts and coding styles.	Files previously touched by more developers contain more defects [15].
	AGE	The avergae time interval between the last and current change	The lower the AGE (i.e., the more recent the last change), the more likely a defect will be introduced.	More recent changes contribute more defects than older changes [16].
	NUC	The number of unique changes to the modified files	The larger the NUC, the more likely a defect is introduced, because a developer will have to recalland track many previous changes.	The larger the spread of modified files, the higher the complexity [10], [11].
Experience	EXP	Developer experience	More experienced developers are less likely to introduced a defect.	Programmer experience significantly decreases the defect probability [5].
	REXP	Recent developer experience	A developer that has often modified the files in recent months is less likely to introduce defect, because they will be more familiar with recent developments in the system.	
	SEXP	Developer experience on a subsystem	Developers that are familiar with the subsystems modified by a change are less likely to introduce defects.	

quality. Zimmermann et al.[8] showed that there is no single factor that leads to an accurate prediction. They focused on defect prediction from one project to another based on seven commercial projects and four open source projects. Existing studies create the impression that we can predict defects in a software system in many different ways having little variation in accuracy and prediction performance. However this does not help us the reduce the time required to fix the defect. We still need to find out the particular change in the code that is involved in producing the defect.

Previous works have mostly concentrated on predicting defects. Many of them are also focusing on quality assurance by predicting the probability of defects or the number of defects for a particular location in the system back-end. In contrast to many of these studies Kamei et al. [3] focus on predicting the probability of a software change inducing a defect at check-in time. The research of Kamei et al. made us interested in this topic and we are replicating this work of Kamei and his co-workers.

III. Approach

Similarly to Kamei's work, we use a logistic regression model to perform our prediction [3]. We are collecting the historical data of Django from it's git repository and the tables that we generate to store the historical data contains commit information. A commit represents one change here. If a commit contains multiple files then we would like to say these files have got a change at this particular commit. For the SSZ algorithm which we use we also require the data for Django's defect tracker. Django uses the Trac ticket software system to keep track of tickets which are created for the project. We extracted these tickets of which 54% of them have a type assigned to them which identify the category of work which it covers. These types can refer to bugs/defects, new features, enhancements and cleanup/optimization ticket tasks. Of these 54% some of these tickets have their types as uncategorised

We then used an hybrid of the SZZ and ASZZ algorithms[17] which is used to link defect fix commit to the original commit which introduced the defect. This is a three step process which uses the acquired data. In the first step of the SZZ algorithm we attempt to identify the commits which were used to fix a defect. This is done by searching for ticket numbers in the commit's message and for keywords such as "bug", "defect", "fix" and "patch" [3].

The second step involves the use of the defect tracking data to determine whether a commit which was identified in the first step can be linked to a tracked ticket and whether that

Table II: Django Project Statictics

Period	Total number of commits with changes	Percentage of defect inducing commits	Average LOC per file	Avg. LOC per changes	# of modified files per changes	# of changes per day	Max # dev. per file	Avg. # of dev. per file
12/07/2005 - 19/09/2014	26606	28.2%	142.5	272.3	4.6	7.9	143	5.7

ticket is a defect. Given that not all tickets are defects, the identifier of a ticket is not enough to say that a commit was used to fix a defect. This is why the link must be done and the ticket must be validated. However, the reason for our use of an hybrid of SZZ and ASZZ is the lack of information present in the issue tracker. We were able to link 60.0% of the commits to tickets, but for 43.8% of those commits it is not possible to determine what kind of ticket they are given that they have not been categorized. As a result we can only be certain of the category of 33.7% of the commits. Due to this we have to augment our data using the ASZZ. In the event where we are unable to make a link or that the link doesn't have a type we use the keywords such as "bug", "defect", "fix" and "patch" [3] to determine if the commit might have been fixing a defect. As a result we are able to determine that 34.1% of commits were performed to fix a defect.

Following the second step we shall now have a dataset where we know whether a commit served to fix a defect. Going into the third step we now want to determine from which commits these defects originated from. This is achieved using Git's ability to know from which commit a line originates from. Where we know that a commit replaced lines which contained a defect we can compare the fix commit to the previous commit to be able to blame those replaced lines. As a result we are able to determine the origin of the lines which were replaced. This however can result in false positives as other tasks such as refactoring might have been performed as part of the commit, but we do not have a way to discriminate against those. Following the execution we were able to determine that 28.2% of commits were defect-inducing.

From our data we also generated the change measurement factors as defined in the original paper. Table I is a copy of factors table provided in the original paper [3]. We then generated a logistic regression model based on these factors which were generated from our Django dataset. For each commit, the logistic regression model outputs a probability represented by a value ranging from 0 to 1. Like Kamei we also set a threshold value of 0.5 to make our prediction easier like: defect-inducing "yes" or "no" [18], [13].

Kamei et al. selected a minimal set of factors to include as independent variables to avoid over-fitting the models. We followed a similar approach at this point as we are also removing highly correlated factors manually but then instead of using Stepwise variable selection we are using the CfsSubsetEval evaluator [**?**] to remove the remaining collinear metrics and those metrics that do not contribute to the model. We used accuracy, precision, recall, and F1-measures to evaluate the performance of our model. The confusion matrix has been represented in Table III. A change True Positive (TP) indicates that the corresponding change is Defect inducing and False Negative (FN) indicates that the change is non-defect-inducing.

IV. Results

In this section we will be presenting the results of our research questions which replicated the original paper. Django, the project which was selected for this replication is of similar size to other open source projects in the original paper (Table II). While Bugzilla is also a web application, it is programmed in a different programming language. As such Django which is a web development framework written in python will serve to comparision to other open source projects which we being used in the original paper.

A. RQ1: How well can we predict defect-inducing changes?

To avoid the overfitting of factors within the model we need to remove factors which are highly correlated. We used Weka's CfsSubsetEval attribute evaluator using the exhaustive search approach to identify the factors to retain. Like with subsequent models we used a 10 fold cross-validation approach to reduce the error rate of our model that could result from the data. Following it's execution it was determined that the factors NS, Entropy, LA, NUC and REXP would have the least coupling. We therefore deleted other factors from our model and retained the factors which were specified.

Using this model which does not contain the

We present the results of the model in Tables III and IV. The confusion matrix in table III illustrate the performance of the model which used logistic regression and compares their results to the known values of which change introduced a defect. As illustrated we can see that our model performs better at identifying changes which cahnges are not defect-inducing. On changes which had no defects there was a recall of 95.8% which reduce the risk of having a false positive. Of the 28.2% (Table II) of changes which are identified as defect-inducing in our dataset, only 14.6% of these are recalled by the model. While there would be more instances of false negative, this would result in an increase in the number of defect-inducing changes which might go unnoticied until a defect ticket is openned by a user of the application.

Table III: Django Confusion Matrix

true / classified	No defect	defect
No defect	19121	832
defect	6413	1096

Table IV: Prediction Performance

Accuracy	Prec.	Recall	F1	AUC	% improvement	
					Prec.	AUC
73.6%	70.0%	73.6%	67.4%	69.0%	32.6%	38.0%

We used a ZeroR model with 10 folds cross-validation as our baseline to compare against our logistic regression model. Following the execution of both models we have their comparisation in table IV. Overall there is a 32.6% precision increase in the logistic regression model compared to the baseline. This improvement indicate that our model performs better then the baseline and this is also confirmed with the improved area under the curve (AUC) of 38.0%.

B. RQ2: What are the major characteristics of defect-inducing changes?

Using all the factors from our Django model (Table I) we can determine which factors are defect-inducing. Compared to the original paper this replication study only covers one open source project. Due to this we will be comparing it to the results of the original paper.

We followed the approach presented in the original paper of using positive and negative signs to illustrate the effect which a factor as of inducing defects in a change. Factors with a positive sign (+) will result in a higher probability that a change is defect-inducing. On the other hand factors with a negative sign (-) have the opposite effect and reduce the probability that a change is defect inducing.

Using the odds ratio results obtained from the model which are presented in table V we can determine the effect which these factors have on the probability that it is defect-inducing. We can see that most factors have a negligable effect when the Django data is modelled. The number of directories in a change (ND) is the biggest defect-inducing factor for Django. Where a change includes modifications to more directories it shall be expected that there will be a higher probability of this change introducing a defect. More of the factors can be said to have a negative effect in inducing defects in Django. The number of subsystem (NS), Entropy and Fix fit within this category for Django. These results would indicate that the more subsystems are modified, the greater the risk of introducing a defect is reduced. The presence of Entropy as a defect-reducing factor would illustrate that the greater the distribution of changes in a Django change the lower the probability that a defet will be introduced. And finally where a change is made to correct an existing defect there is subsequent reduced probability of a defect being present in the lines affected by the change.

The result of these three factors with a negative effect present a rejection of their rationales as defined in Table I. The rationales in Table I indicate the expected behaviors of the factors on the probability of inducing-defects, however the project is not required to follow these assumptions as these results illustrate. The results can also be briefly compared to that of the original paper. A more throurough comparision is performed later in section V. The presence of NS and Fix as defect-reducing factors in the Django data appear to contradict the results which were presented in the original paper.

Table V: The Impact of Change Factors on Defect-Inducing Changes

Metrics name	Impact	Django
NS	-	0.6779
ND	+	1.0129
NF		1.0007
Entropy	-	0.6078
LA		1
LD		1
FIX	-	0.9536
NDEV		0.997
AGE		1.0003
NUC		0.965
EXP		1.0001
REXP		1
SEXP		0.9998

Table VI: The Regression Coefficients of Change Factors

Metrics name	Impact	Django
NS	-	-0.3887
ND	+	0.0128
NF		0.0007
Entropy	-	-0.4978
LA		0
LD		0
FIX	-	-0.0475
NDEV		-0.003
AGE		0.0003
NUC	-	-0.0356
EXP		0.0001
REXP		0
SEXP		-0.0002

The change factor's regression coefficients which are present in Table VI also illustrate similar results to the odds ratios of Table V. The finding which ND is defect-inducing in changes for Django is consistent with the results of the odds ratio in table V. A consistent selection of defect-reducing factors are also present in the regression coefficients in comparision to the odds ratio. The factors NS, Entropy, Fix and NUC would all reduce the probability of a change inducing defects in Django. The defect-reducing effect of NUC would indicate that the more changes which have been applied to a file the less probable it is that this file contains a defect. The defect-reducing effects of NS, Entropy and Fix were presented in a earlier paragraph of this section.

V. Comparison of Results

In comparison to the original work we would like to compare our RQ1 with RQ1 of the original paper and our RQ2 with the RQ3 of the original paper. Kamei et al. have used both open source and commercial projects for their study but we replicated it on one single open source project. So, the comparison would be based on the results that we find in the original work for open source projects compared to our results and findings.

Table VII: Comparison of Prediction Performance

	Acc.	Prec.	Rec.	F1	AUC	% improvement	
						Prec.	AUC
Ours	73.6%	70.0%	73.6%	67.4%	69.0%	32.6%	38.0%
Original OSS Avg.	71%	37%	67%	45%	76%	92.6%	51.5%

A. RQ1-RQ1 Comparison

Research question 1 for both papers wanted to determine how well we can predict defect-inducing changes. The original paper provided the performance prediction metrics for it's 6 open source project and 5 commercial projects. We replicated their process and have obtained the same performance prediction metrics for Django which we can use for the comparison. Given that we did not analyse a commercial project we will disregard those metrics and will perform our comparison on the other open source projects.

Table IV represents the prediction result for Django. Our accuracy and precision are better than any of the open source projects that have been used in the original work as illustrated in the comparison table VII. Given that we do see in the other results that we are getting the better performance. Were the linear regression model's performance is compared to the baseline we notice that our results have a lower percentage improvement then that of the original paper. We believe that this would be a result of different baselines. The original paper had few details regarding the model which they used for their random defect-inducing baseline. We attempted to replicate this baseline using the ZeroR, but as the results illustrate this was probably not the one which they were using.

B. RQ2-RQ3 Comparison

Research question 2 from our paper and research question 3 from the original paper wanted to determine what are the major characteristics of defect-inducing changes. At this point Kamei et al. found that "Entropy" is a risk reducing factor. It has the same indication in our research as well. However, "Fix" has an opposite indication in our study. Compared to the original work it is a risk increasing factor while it's a defect reducing factor for us.

Surprisingly the number of sub-systems "NS" is a defect reducing factor in our study while it didn't indicate anything in the original paper. This further contradicts table I. Given that Django is written in a different programming language and is a framework for web applications which different compared to the other open source projects that have been studied by Kamei et al.. This could explain the observed differences. Number of directories "ND" is a poor defect-inducing factor in our study, but it is the largest one which we have observed for Django.

VI. Conclusion

This paper studies prioritizing quality assurance approach by replicating a similar work done previously to verify the existing approach. Our study results with higher accuracy, precision and recall of 73.6%, 70.0% and 73.6% with F-measure of 67.4%. We had 33.7% records that we knew whether they are defect or they are of any other type as we were able to identify only 60% of the regression data. Our prediction could give us be much better result if we would have higher number of regression data that we know what category they belong to.

Similarly to the original work we could predict the defect-inducing changes using 10 fold data sets. We found some differences for couple of factors in our results while determining the major characteristics of defect-inducing changes. Over all, the change level prediction is an effective approach for prioritizing the actions to take for quality assurance of a software project. This replication work validates that a model which uses logistic regression could be used to provide recommendations for commits which may have a higher probability of being defect-inducing.

Acknowledgment

We would like to thank Dr. Bram Adams for providing us with the Django data and for his continued support throughout the completion of this replication paper. We would also give a special thanks to Dr. Peter Rigby who's minimum supervision has allowed us to progress with our work.

References

[1] I. S. T. Gyimothy, R. Ferenc, "Empirical validation of object-oriented metrics on open source software for fault prediction," in *IEEE Trans, Software Engineering*, October 2005, pp. 897–910.

[2] Y. Z. S. Kim, E.J. Whitehead Jr., "Classifying software changes: Clean or buggy?" in *IEEE Trans. Software Eng.*, vol. 32, no. 2, March 2008.

[3] B. A. Yasutaka Kamei, Emad Shihab, "A large-scale empirical study of just-in-time quality assurance," in *Software Engineering, IEEE Transactions on*, vol. 39, May 2013, pp. 757–773.

[4] T. B. N. Nagappan, "Use of relative code churn measures to predict system defect density," in *International Conference of Software Engineering*, 2005, pp. 284–292.

[5] D. W. A. Mockus, "Predicting risk of software changes," in *Bell Labs Technical Journal*, vol. 9, no. 2, 2000, pp. 169–180.

[6] R. W. Sunghun Kim, Hongyu Zhang, "Dealing with noise in defect prediction," in *International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 481–490.

[7] S. I. L.C. Briand, J. Wu st, "Investigating quality factors in object-oriented designs: An industrial case study," in *International Conference on Software Engineering (ICSE)*. IEEE, 1999, pp. 345–354.

[8] H. G. T. Zimmermann, N. Nagappan, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *European Software Eng. Conf. and Symposium*, 2009, pp. 91–100.

[9] A. Z. N. Nagappan, T. Ball, "Mining metrics to predict componenet failure," in *International Conference of Software Engineering*, 2006, pp. 452–461.

[10] R. R. M. D'Ambros, M. Lanza, "An extensive comparisation of bug prediction methods," in *International Working Conference Mining Software Repositories (MSR)*, 2010, pp. 31–41.

[11] A. Hassan, "Predicting faults using the complexity of code changes," in *International Conference on Software Engineering (ICSE)*, 2009, pp. 16–24.

[12] G. S. R. Moser, W. Pedrycz, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *International Conference on Software Engineering (ICSE)*, 2008, pp. 181–190.

[13] N. N. P.J. Guo, T. Zimmermann, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *International Conference on Software Engineering*. IEEE, May 2010, pp. 495–504.

[14] D. P. R. Purushothaman, "Toward understanding the rhetoric of small source code changes," in *IEEE Trans. Software Engineering*, vol. 31, no. 6, June 2005, pp. 511–526.

[15] A. M. S. Matsumoto, Y. Kamei, "An analysis of developer metrics for fault prediction," in *Proc. International Conference Predictive Models in Software Engineering (PROMISE)*, 2010, pp. 18:1–18:9.

[16] J. M. T.L. Graves, A.F. Karr, "Predicting fault incidence using software change history," in *IEEE Trans. Software Engineering*, vol. 26, no. 7, July 2000, pp. 653–661.

[17] A. Z. J. Sliwerski, T. Zimmerman, "When do changes induce fixes?" in *Proc. Int'l Conference Mining Software Repositories (MSR)*, 2005, pp. 1–5.

[18] I. S. T. Gyimothy, R. Ferenc, "Empirical validation of object-oriented metrics on open source software for fault prediction," in *IEEE Transactions on Software Engineering*, vol. 31, pp. 897–910.

[19] D. L. C. Peter L. Flom, "Stopping stepwise: Why stepwise and similar selection methods are bad, and what you should use," in *NESUG'07*, November 2007.