So You Need More Method Level Datasets for Your Software Defect Prediction? Voilà!

Thomas Shippey, Tracy Hall and Steve Counsell
Computer Science
Brunel University London
Uxbridge, United Kingdom
{thomas.shippey,tracy.hall, steve.counsell}@brunel.ac.uk

David Bowes
Science and Technology Research Institute
University of Hertfordshire
Hatfield, United Kingdom
d.h.bowes@herts.ac.uk

ABSTRACT

Much defect prediction research is based on a small number of defect datasets. Most datasets are at class not method level. Consequently our knowledge of defects is limited. Identifying defect datasets for prediction is not easy and extracting quality data from identified datasets is even more difficult. We identify open source Java systems suitable for defect prediction and extract high quality data from these datasets. We use the Boa September 2013 SourceForge dataset to identify candidate open source systems. We used selection criteria based on the type and quality of both a software repository and its defect tracking system to reduce potentially 50,000 open source systems to 23 suitable for defect prediction. We enhance the SZZ algorithm to extract fault information from these systems and we used JHawk to produce 138 fault and metrics datasets. The data we provide enables future studies to proceed with minimal effort. Our datasets significantly increase the pool of systems currently being used in defect analysis studies. We make these datasets (the ELFF datasets) and our data extraction tools freely available to future researchers.

CCS Concepts

•**Software and its engineering** → **Software defect analysis;** *Search-based software engineering;* •**Computing methodologies** → Machine learning algorithms;

Keywords

Defects, Defect Prediction, Boa, Data Mining, Defect linking

1. INTRODUCTION

In this paper, we present defect and source code metrics data from 23 Java open source systems. We selected systems using a systematic approach for identifying systems suitable for defect prediction. We extracted data from these systems at method level using a rigorous methodology. Over the last 20 years researchers have dedicated a huge amount of effort to developing a variety of software defect[1] prediction models. Most defect prediction models are based on open source systems as commercial source code and defect data is difficult to obtain. Researchers confine themselves to studying a small pool of open source systems. This is not surprising as identifying suitable systems and extracting reliable data is difficult and time consuming. Consequently many of the systems analysed in previous studies are those where data is already available (e.g. from the Promise Repository [2]).

The reasons that collecting usable and reliable defect data is difficult include: First, projects will often not have enough defects stored in repositories to enable the building of defect prediction models. Second, knowledge, skill and care is needed when collecting defect data from project repositories to ensure reliable defect data is extracted. Third, it is difficult to collect sufficient good quality defect data at the method level and so the majority of data sets used in defect prediction are at class or file level. This high level of prediction granularity is of limited use given that a file or class might be hundreds of lines long. To address these difficulties, we present the ELFF dataset which contains systems rich in defect data, with method and class level defect data collected using a rigorous data extraction process which is accompanied by a wide range of source code metrics and for which all source code is available.

Our ELFF dataset was achieved by mining the Boa SourceForge September 2013 open source dataset [7]. In total, this dataset suggested more than 50,000 potential projects; using our bespoke ChallengerELFF tool, we filtered down the number of projects to 23. This filtering was done using criteria based on project maturity and commit frequency to ensure that all 23 systems are mature and contain sufficient defect fixing commits to be usable in defect prediction. We then used another bespoke tool, DefectFinderELFF to extract method and class level defects from multiple versions of the 23 projects. We combined this fault information with source code metrics to create the ELFF dataset: a corpus of new open source datasets for use in defect related research.

The main contribution of this paper is a freely available set of 138 source code metrics and defect data, both at class

[1]There is a distinction between a *fault* and a *defect*. A defect is a direct result of an error by a developer when programming a system. A defect becomes a fault when the error manifests itself during the use of the software product [13].

ESEM '16 Ciudad Real, Spain

DOI: 10.475/123_4

and method level, from 23 open source projects and versions of those projects. This contribution to defect prediction will significantly increase the current pool of projects. The identification of new datasets is important for many reasons. Firstly, the ELFF dataset could be used in replication studies as there are many reported challenges currently within software defect prediction [5] that need further work and researchers can apply their new techniques to more projects and test the stability of their conclusions. Secondly, the ELFF corpus provides fault information at both method level and class level and so studies can be replicated at a lower level of granularity to test the stability of their conclusions. The ELFF dataset, ELFF tools and information to enable the work reported in this paper to be replicated can be found at www.elff.org.uk/ESEM2016.

2. BACKGROUND

Software defect prediction is a method for determining potentially defective areas in a particular piece of software code. The predictions make it possible for the developer to focus on areas of the software system before release, reducing both time and effort. Software defect prediction relies on three main components; *dependent* variables, *independent* variables and a *model*. Dependent variables are the defect data for a particular module. The defect data can be binary (faulty or non faulty), or continuous (number of faults). Independent variables are the metrics which describe the software code, how it has changed or who changed it. Independent variables come in two forms, software code metrics: those that can be derived from the software code itself and process metrics: metrics that measure the change of software code or software practices over time. The models often use machine learning approaches which contain the rule(s) or algorithm(s) that predict the dependent variable from the independent variables. These rules can be as simple as the number of independent variables in the model, or as complicated as decision trees[2] and regression[3] techniques. Our previous work Hall et al. [12] identified over 200 papers and the models/metrics used to carry out defect prediction.

Dependent variable information is extracted from software repositories. One way to identify defective software code is to analyse the code's respective software version control system (VCS) and defect tracking system. When a fault is fixed, it is good practice to reference this fault fix in the VCS commit message. Normally, this fault will correspond to a fault that has been reported and logged in a fault tracking system. A defect tracking system is needed to identify where a fault has been fixed because sometimes the commit does not always include the information on why a change was made [29]. When the defect tracking information and the VCS information is combined, the reliability of discovering defect links is increased. If the defect tracking system was used alone, the commit that caused the change would not be known, only the time that the defect report was updated. Similarity, if just the VCS was used, the defect fix numbers reported in a commit message may not actually be defect numbers. One method to identify defective code is to match a particular commit where the defect has been fixed from the VCS to the correct defect number in a defect tracking system. This will give a defect-link and will tell us where a particular fault has been fixed; it is used subsequently to find the defect insertion point by tracking the historical changes to the fixed code. The defect insertion point is an important piece of information. If the insertion point and the fix point (the defect-link) is identified for known defects, then it is possible to know where there is a defect at any point in the history of the code. Many different algorithms have been proposed to find defects from these two software repositories. These algorithms include: SZZ by Śliwerski et al. [29], including its improvements by Kim et al. [16], Bird et al. [3] and Williams and Spacco [31], LINKSTER by Bird et al. [4], BugInfo by Jureczko [14], ReLink by Wu et al. [32], MLink by Nguyen et al. [26], RCLinker by Le et al. [21] and HyLok by Lam et al. [19].

A variety of open source and closed source projects have previously been used in defect prediction. The NASA datasets [23] are an example of closed source projects that have been used extensively in software defect prediction (in approximately 30% of software prediction studies from 2000 to 2010 [12]). However, the code for these datasets is not available and many inconsistencies have been reported with the data contained [28, 10, 11, 27]. Other close source systems have been less used. For example, Microsoft systems have been used by Kim et al. [17], Layman et al. [20], Nagappan and Ball [24], Nagappan et al. [25]. Defect prediction studies mainly use open source systems due to the ease with which researchers can gather the data. Open source code and defect information is freely available and therefore easily mined. There are many different open source projects available, but Eclipse and Apache projects are two of the most frequently used [12]. Researchers can also obtain defect data from the Promise repository [2]. This repository holds around 60 defect datasets, and the datasets held are used frequently by defect prediction studies. Some of the more frequently used Promise datasets are those curated by Jureczko [14], which have been cited five times this year alone [33, 30, 15, 1, 22].

Defect prediction is performed at different levels of granularity, with the majority being at a high level of granularity, e.g. file and class level. All 60 defect datasets on the Promise repository [2] are at this high level. Hall et al. [12] show that only 12% (22/182) of defect prediction models from 2000 to 2012 are performed at method, function or procedure level and the performance range of these models is much wider than models at higher levels of granularity performed. Method level predictions are much more valuable to developers than predictions at file or class level. Files and classes can be very large, whilst defects can be very small, meaning greater additional effort to find the defect.

3. METHODOLOGY

3.1 SourceForge project extraction

We systematically generated a list of open source projects from which defect data can be extracted. To do this we used Boa which is a domain-specific programming language for analysing ultra-large-scale software repositories [7]. We chose Boa as it substantially decreases the effort required to mine software repositories and experiments are easily replicable. Boa allows you to mine several datasets, for our study we chose the September 2013 SourceForge dataset [7]. We

[2]A decision tree algorithm is one that creates a graph of decisions based on the chance of an event happening.

[3]Regression analysis seeks to determine best fit of independent value(s) based on a dependent value(s).

Criteria	Reason
Is a Java project	Our implementation of SZZ works with Java code
An SVN project	Our current implementation of the SZZ algorithm is currently set up to use SVN.
Have defect linking commits	To perform defect linking, we need defect linking commits.
SourceForge defect Tracking system	Defect id's are needed to run defect linking.
At least 100 fixed defects	Needs to be lots of fault information to combat data imbalance.

Table 1: The criteria with which the final projects will be chosen

chose this dataset as it was the most recent repository that has information stored about possible SVN projects. We applied criteria to establish the project's suitability for inclusion in the ELFF dataset provided in Table 1.

We focus on SVN and Java projects because we have created a tool that extracts defects from projects with these two sources of information, we intend to extend this tool to extract information from other repositories in the future. A potential project had to have a SourceForge defect tracker because to perform this analysis systematically and automatically, we extract defect information from the SourceForge Rest API[4]. Other projects could have different defect tracking systems, but this would be time-consuming to manually check. The projects have to be rich in fixed defects. If there are not enough fixed defects, then we could encounter data imbalance problems when running our defect prediction due to the lack of defective modules and this would impair the results of our prediction algorithms.

To identify SVN projects on SourceForge we ran a Boa script[5] on the September 2013 dataset to extract all SVN projects and the number of defect fixing commits that fix Java files. To determined if the project had a defect tracking system in SourceForge, we created ChallengerELFF, which examined the SourceForge Rest API. ChallengerELFF was developed in Java and extracts information from the JSON representation of a SourceForge project. Each SourceForge project's JSON representation was parsed to determine if a project had a defect tracker. This was determined by examining if the project had a ticket system called "Bug" or "Issue". ChallengerELFF then extracted fixed defects tracker JSON. For SourceForge projects with defect trackers, we analysed the commits of the SVN to determine if they had defect-linking commits. A defect-linking commit was determined by using the SZZ algorithm [29]. This would give us the total number of commits that fixed defects reported. This coverage statistic is vital, since projects that have low defect coverage could contain many false positive faults when we applied our defect-linking technique.

3.2 Fault data extraction

For each of the SourceForge projects we compiled a dataset of faulty methods. We found which methods were faulty at the time of release by finding the fault insertion and fix points. To identify faulty methods we used the SZZ approach as it has been used in many previous studies [6, 8, 16, 17, 31, 34]. SZZ is a fault linking algorithm described by Śliwerski, Zimmermann, and Zeller [29]. SZZ was based on work by Cubranic and Murphy [6] and Fischer et al. [8, 9], who inferred links between Bugzilla defect reports with CVS commit messages. We have created our own SZZ implementation called DefectFinderELFF[6]. The SZZ algorithm matches the fault fix described in a defect tracking system with the corresponding commit in a version control system that 'removed' the fault. By backtracking through the version control records, it is possible to identify earlier code changes which ended up being 'fixed'. It is assumed that the earlier code changes inserted the fault. The module of code is therefore labeled as faulty between the time the fault was inserted and the time it was fixed. Using this technique it is possible to identify for a particular snapshot of the code base, which modules are faulty and which are not. Obviously there will be defective modules which have not yet been reported. It is therefore important to carry out the fault mapping after sufficient time has passed for users to report most faults. It is unlikely that all defects will be reported and therefore there will be false negatives. Kim suggests that as long as the number of false negatives and false positives is less than 20% in total, defect prediction can be carried out [18]. This is an important point, early work by Zimmermann et al. [34] only managed to map about 50% of faults reported in the defect tracking systems to changes in the code base. Later Bird et al. [3] improved the mapping by removing some of the constraints that Zimmermann had introduced, for example the requirement to have matching defect IDs in a predefined format. The implementation of SZZ used in this paper was improved slightly from the original. It has a higher weighting for those numbers found in commit messages that are in the defect database and takes into account the "Fix for" prefix. The implementation was verified by us, independently of this study, by manually checking ALL defect links found for Eclipse JDT 3.0.

When ran DefectFinderELFF over different versions of each of the SourceForge projects to create a corpus of faulty methods. We combined this fault information with both class and method level metrics extracted using JHawk[7], to create datasets of fault information for each version of a SourceForge project. The metrics and faults can be found at www.elff.org.uk/ESEM2016/.

4. RESULTS

Using our Boa script, we extracted around 50,000 possible Java systems using SVN. Using the information gathered from this initial extraction, we were then able to analyse each of the possible systems further to determine if they were good candidates for defect prediction studies using the SourceForge Rest API. Table 2 shows the final 23 possible systems that could be used to extract fault information using the subversion and defect tracking systems from Sourceforge, selected using our criteria. The tables are sorted by the level of fixed defect coverage. The defect coverage of the system is important to the SZZ algorithm as it relies on the defect id's to have been mentioned in the commit message. Without

[4]https://sourceforge.net/p/forge/documentation/Allura\%20API/

[5]Our script can be found at www.elff.org.uk/ESEM2016.

[6]Available at www.elff.org.uk/ESEM2016

[7]Version 5.0

the defect id, the algorithm may not be as accurate since it has to rely on commits that have certain keywords.

Table 2 gives the context and defect information for each project. There is a lot of variation between the projects. The average age of the projects is 12 years. The oldest project is jEdit, which started in 1999. Autopilot is the youngest project at eight years old. The domain for each of the projects, ranges from 3D modelling to simple integrated development environments. The projects vary in size from large projects like JMRI, with 549 KLOC to smaller projects like Autopilot, which has 16 KLOC. The average size of the projects is around 180 KLOC.

BRL-CAD has the best defect linkage of the projects extracted from the SourceForge dataset with nearly 100% defect linkage. The worst project linkage is jEdit with only around 8%. The selected 23 projects have approximately 41% defect linkage on average. The defects in the commit messages ranges between 106 (jTDS) and 484 (RunaWFE). BRL-CAD in its latest release had no Java files, but in its history there have been files changed due to a defect fix so we include it as it does fit our criteria.

We ran DefectFinderELFF on multiple versions for each of the 23 projects. We have fault information for 69 versions of the projects discovered using Boa. Table 3 shows a sample of the fault information for randomly selected projects. Some of the projects did not have easily identifiable releases, in this instance, we chose a particular date as the snapshot. For example, 1st January 2008 would have a version number of 01012008. Jmol 6 has the highest level of faults with around 13% of methods and around 42% of classes defective. The results show that a high linkage of defect-fixing commits does not always translate into a high number of faults in a particular version of a project. This could be due to a couple of reasons. The found defect-links might not actually be around the time of the versions identified, the projects could have picked up the defect links in later releases or became lax in reporting defect fixes properly. Another reason could be that even though the defect was fixed, but as SZZ traces the fix to its insertion point, the method may have not actually been defective at the time of the release.

5. CONCLUSION

Good defect data is hard to obtain, especially at the method level. Of the defect data on the Promise repository, none have method level defects. Method level defect prediction is important as developers do not have time to search through files to find potentially defective code. Finding appropriate open source projects that have rich defect repositories and accompanying subversions is hard and labourious. Consequently, researchers have repeatedly used the same projects in defect prediction. We applied a systematic approach to determine open source projects on which defect prediction could be performed. These projects were extracted from the SourceForge September 2013 dataset [7] and then further analysed with the SourceForge Rest API. We narrowed down 50,000 potential projects, to 23. These 23 projects were chosen as they have the best potential for extracting good defect information due to the defect linking in their SVN repositories. With these 23 projects, we extracted fault data for 69 versions. This fault data showed that a high defect-linkage in the commits, does not always correlate well to having a high number of faults in a particular version. The datasets that we have curated in this paper, the ELFF dataset, is freely available for other researchers to use. The corpus significantly increases the amount of method level datasets available. The ELFF dataset will help increase the diversity of datasets being used, so that researchers in defect prediction can make better global conclusions, perform study replications and test the stability of their conclusions on a larger set of systems.

Acknowledgements

This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/L011751/1.

References

[1] Defect prediction: Accomplishments and future challenges.

[2] The promise repository of empirical software engineering data, 2015.

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2.

[4] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein. Linkster: Enabling efficient manual inspection and annotation of mined data. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 369–370, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. . URL http://doi.acm.org/10.1145/1882291.1882352.

[5] D. Bowes, T. Hall, and D. Gray. Dconfusion: A technique to allow cross study performance evaluation of fault prediction studies. *Automated Software Engineering*, 21(2):287–13, 4 2014. ISSN 0928-8910. .

[6] D. Cubranic and G. Murphy. Hipikat: recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 408 – 418, may 2003.

[7] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25 (1):7:1–7:34, Dec. 2015. ISSN 1049-331X. . URL http://doi.acm.org/10.1145/2803171.

[8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23 – 32, sept. 2003.

[9] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 23–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1905-9.

Project	Domain	KLOC	Age (Years)	Committers	Releases	Defects Reported	Fixed in Commits	Coverage
BRL-CAD	3d Modelling		12	9	42	276	259	93.84
JPPF	Distributed Computing	45	11	77	120	235	208	88.51
EclEmma	Quality Assurance	55	9	26	35	138	109	78.99
GenoViz	Data Visualisation	193	11	9	57	245	165	67.35
Jikes RVM	Compilers	179	11	47	35	648	430	66.36
Jmol	3D Rendering	255	15	4	15	535	286	53.46
TANGO	Human machine interfaces	288	13	21	6	684	335	48.98
RunaWFE	Business Management	584	11	9	49	994	484	48.69
CMU Sphinx	Speech Recognition	68	16	189	26	333	151	45.35
JUMP	GIS	182	11	17	24	392	144	36.73
DrJava	IDE	161	14	7	62	825	299	36.24
JMRI	Gaming	550	15	30	49	502	181	36.06
UNICORE	Security	47	12	48	32	773	239	30.92
XAware	Development	104	8	11	16	702	209	29.77
ControlTier	Config Management	127	10	6	18	612	177	28.92
Autoplot	Visualisation	16	8	18	60	715	200	27.97
Saros	Agile development tool	88	9	55	29	766	202	26.37
OmegaT	Text Processing (CAT)	64	13	13	11	673	173	25.71
Jitterbit	Integration Tool	458	10	57	11	1110	202	18.2
CDK	3D Rendering	140	15	77	38	1225	204	16.65
HtmlUnit	HTML Manipulation	249	14	11	38	1637	265	16.19
jTDS	Database	133	14	75	37	658	106	16.11
jEdit	Text Editor	115	16	25	39	3727	283	7.59

Table 2: The contextual information for the 23 SourceForge projects identified as being suitable to apply the SZZ algorithm. N.B. BRL-CAD has been included as it fixed Java files in the past, however its suitability is impaired as currently it has no Java files.

Project	Version	Classes	Classes Faulty	% Classes Faulty	Methods	Methods Faulty	% Methods Faulty
jmol	6	170	72	42.35	2,217	294	13.26
htmlunit	01012008	280	45	16.07	2,979	343	11.51
genoviz	6.1	687	197	28.68	7,815	800	10.24
jmol	7	187	50	26.74	2,432	248	10.2
genoviz	6	708	202	28.53	8,131	824	10.13
genoviz	5.4	722	205	28.39	8,489	807	9.51
drjava	01012008	918	153	16.67	14,123	919	6.51
jmol	4	134	28	20.9	1,367	81	5.93
saros	1.0.6	135	18	13.33	1,293	69	5.34
drjava	01012009	1,033	169	16.36	16,617	724	4.36
unicore	1.2	346	39	11.27	2,108	90	4.27
eclemma	2.1	117	9	7.69	919	37	4.03
jikesrvm	3	1,399	117	8.36	17,007	440	2.59
jmol	3	122	21	17.21	1,393	35	2.51
unicore	1.3	392	32	8.16	2,422	54	2.23
drjava	01012010	1,099	73	6.64	18,482	385	2.08
htmlunit	01012009	606	30	4.95	8,018	72	0.9
controltier	3.1	1,323	42	3.17	13,534	117	0.86
jitterbit	1.2	6,351	44	0.69	49,885	143	0.29
cmusphinx	3.6	416	4	0.96	4,751	13	0.27
jedit	5.2	556	8	1.44	7,524	13	0.17
cmusphinx	3.7	415	3	0.72	4,758	8	0.17
runawfe	3.6	3,325	5	0.15	32,221	5	0.02
cdk	1.2	696		0	7,317	0	0
cdk	1	1,016		0	10,195	0	0
controltier	3.2	1,344		0	13,723	0	0

Table 3: The fault information for a random selection of 69 versions the selected SourceForge projects. The full list is available at http://www.elff.org.uk/ESEM2016

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

[11] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Reflections on the nasa mdp data sets. *Software, IET*, 6(6):549 –558, dec. 2012. ISSN 1751-8806.

[12] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012. ISSN 0098-5589.

[13] IEEE. IEEE standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.

[14] M. Jureczko. Significance of different software metrics in defect prediction. *Software Engineering: An International Journal*, 1(1):86–95, 2011.

[15] K. Kawata, S. Amasaki, and T. Yokogawa. Improving relevancy filter methods for cross-project defect prediction. In *Applied Computing & Information Technology*, pages 1–12. Springer, 2016.

[16] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2.

[17] S. Kim, T. Zimmermann, E. Whitehead, and A. Zeller. Predicting faults from cached history. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 489 –498, may 2007.

[18] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 481 –490, may 2011.

[19] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481, Nov 2015. .

[20] L. Layman, N. Nagappan, S. Guckenheimer, J. Beehler, and A. Begel. Mining software effort data: preliminary analysis of visual studio team system data. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 43–46. ACM, 2008.

[21] T.-D. B. Le, M. Linares-Vásquez, D. Lo, and D. Poshyvanyk. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 36–47, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2820282.2820290.

[22] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu. Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology*, 69:50–70, 2016.

[23] T. Menzies and J. Di Stefano. How good is your blind spot sampling policy. In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 129–138, March 2004. .

[24] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[25] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. accept, 2010.

[26] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Multi-layered approach for recovering links between bug reports and fixes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 63:1–63:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. . URL http://doi.acm.org/10.1145/2393596.2393671.

[27] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo. The jinx on the nasa software defect data sets. In *The 20th International Conference on Evaluation and Assessment in Software Engineering (EASE'16)*, 2016.

[28] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the nasa software defect datasets. *Software Engineering, IEEE Transactions on*, 39(9):1208–1215, Sept 2013. ISSN 0098-5589. .

[29] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30 (4):1–5, May 2005. ISSN 0163-5948.

[30] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *The International Conference on Software Engineering (ICSE)*, page To appear, 2016.

[31] C. C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *DEFECTS*, pages 32–36, 2008.

[32] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011.

[33] F. Zhang. Towards generalizing defect prediction models. 2016.

[34] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, May 2007.