Delft University of Technology
Software Engineering Research Group
Technical Report Series

Using the Gini Coefficient for Bug Prediction in Eclipse

Emanuel Giger, Martin Pinzger, and Harald C. Gall

Report TUD-SERG-2011-018

TUD-SERG-2011-018

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the proceedings of the International Workshop on Principles on Software Evolution, ERCIM Workshop on Software Evolution, 2011, ACM Press.

Using the Gini Coefficient for Bug Prediction in Eclipse

Emanuel Giger
Department of Informatics
University of Zurich
giger@ifi.uzh.ch

Martin Pinzger
Department of Software Technology
Delft University of Technology
m.pinzger@tudelft.nl

Harald Gall
Department of Informatics
University of Zurich
gall@ifi.uzh.ch

ABSTRACT

The Gini coefficient is a prominent measure to quantify the inequality of a distribution. It is often used in the field of economy to describe how goods, *e.g.*, wealth or farmland, are distributed among people. We use the Gini coefficient to measure code ownership by investigating how changes made to source code are distributed among the *developer population*. The results of our study with data from the Eclipse platform show that less bugs can be expected if a large share of all changes are accumulated, *i.e.*, carried out, by relatively few developers.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures, process measures, software science*

General Terms

Management, Measurement, Reliability, Experimentation

Keywords

Gini coefficient, bug prediction, code ownership

1. INTRODUCTION

Prior work found out that not only properties of the source code itself, *e.g.*, size and complexity, but also the social context of the development process affect the quality of a system. For instance, the number of authors and the contribution structure, *i.e.*, who modified a certain part of the source code, are related to bugs as reported in [2, 16, 18].

In this paper we analyze the relationship between code ownership and bugs in source files using the Gini coefficient. This coefficient is well known in the field of economy to measure the disparity of a good's distribution among individuals [10]. Analogously, we apply the Gini coefficient to historical change data and developer information to quantify how source code changes are distributed among developers. In particular we investigate the following two hypotheses:

H 1: The Gini coefficient based on change data correlates negatively with the number of bugs.
H 2: The Gini coefficient based on change data can classify source files into *bug-prone* and *not bug-prone* files.

Our hypotheses are motivated by the rationale that when a few developers contribute a major portion of all changes—resulting in a high Gini coefficient—possibly less bugs occur as there is a clear responsibility and ownership. Whereas the case of the "*too many cooks*-situation" results in more uncoordinated, fragmented, and bug-prone changes.

Furthermore, we examine the extent to which measuring source code changes at three different levels of granularity, *i.e.*, file revisions (*R*), lines modified (*LM*), and fine-grained source code changes (*SCC*) [7], affects the results of our study.

Our results with data from the Eclipse platform suggest that focusing code changes on a relatively small group of dedicated developers is beneficial with respect to bugs. In addition, using the Gini coefficient we can compute models to successfully identify *bug-prone* files.

2. GINI COEFFICIENT

In this work we understand code ownership by the fact that a relatively small subgroup of developers accumulates a major share of all changes done to a system (or to parts of it). Analyzing this kind of inequality and concentration of a measure is a common task in descriptive statistics when characterizing distributions. Moreover, inequality is often used by economists to describe the disparity of assets in a population. Two examples are: Examining the market concentration by looking at the shares of several companies, and measuring the distribution of wealth among the individuals of a society. In these examples, inequality is usually considered to be unfavorable. Less developed countries typically show a larger inequality in the distribution of wealth among its people. A few big competitors that dominate the market could abuse their power to suppress market mechanisms.

The *Lorenz curve* is a primary graphical method to express inequality and was first used to measure the concentration of wealth [12]. It is a function of the cumulative distribution, *i.e.*, it plots on the x-axis the % of the population against the allocated % of the total wealth on the y-axis. Figure 1 shows an example of a Lorenz curve of the Eclipse Resource plugin project. We used developers as the "*population*" and file revision as the "*wealth*" in this figure. A Lorenz curve equal to the diagonal line represents perfect equality where everyone owns the same share of the total wealth. Deviation from the

diagonal line means inequality; perfect inequality is where one individual owns everything.

In Figure 1 the curve exhibits a serious deviation from the diagonal line, *i.e.*, there is inequality in the distribution of the revisions among the developers that contributed to Eclipse Resource. On the one hand, we can see that 85% of all developers accumulate only 20% of all revisions. On the other hand, 15% of all developers are responsible for almost 80% of all revisions. Numerically expressed: Eclipse Resource strongly depends on two developers—they committed 6'200 revisions out of 7'932 in total.

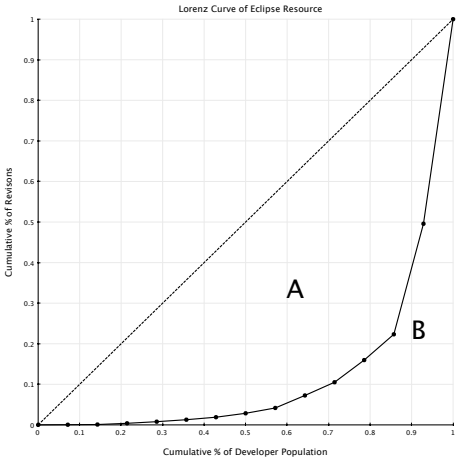

Figure 1: Lorenz curve of the Eclipse Resource plugin project using developers and revisions

The *Gini coefficient* was proposed by Corrado Gini in [10] and is a popular measure of inequality in economy. This coefficient is closely related to the Lorenz curve. In Figure 1, A is the area between the diagonal line of perfect equality and the Lorenz curve, B is the area under the lorenz curve. The Gini coefficient is then defined as $A/(A+B)$ [4]. It can be normalized to the range [0,1]: 0 is perfect equality and 1 reflects perfect inequality. The Gini coefficient is a robust measure and allows the comparison of the disparity of an attribute of differently sized populations [13] since it does not rely on any assumptions regarding the distribution of that underlying attribute. These characteristics are particularly beneficial in the context of software systems where the distributions of attributes and metrics are often heavily skewed and non-Gaussian [6].

3. DATA COLLECTION

For the empirical study in Section 4 and the computation of the Gini coefficient we collected (1) the historical versioning data, *i.e.*, file revisions (*R*) and lines modified (*LM*) including developer information, (2) fine-grained source code changes (*SCC*), and (3) the number of bugs per file (#*Bugs*).

1. Versioning Data: Versioning repositories, *e.g.*, CVS, GIT, or SVN, provide log entries about the history of a system. Those entries contain information about each revision of all files of that particular system including a manually entered commit message and the name of the developer that committed a revision. *LM* is the sum of lines added, deleted, and changed per each file revision. We use EVOLIZER [8] to automatically access the log entries and extract above mentioned information. Using this versioning information we compute the Gini coefficient for each source file, once based on the distribution of file revisions (Gini_R) and once based on the distribution of *LM* (Gini_{LM}) among developers.

2. Fine-Grained Source Code Changes (SCC): Versioning systems record changes solely on file level and handle source code files internally as text files. Therefore, revisions and *LM* can be too coarse-grained or might ignore the semantics of changes to accurately describe all the detailed maintenance activities that occur between two file revisions. Fluri *et al.* developed a tree differencing algorithm based on the abstract syntax tree (AST) structure to extract code changes and their semantics at a fine-grained level, *i.e.*, statement level [7]. Their algorithm is part of CHANGEDISTILLER [8] that compares the ASTs of subsequent file revisions obtained from the versioning system. Including the name of the developer that committed the revision corresponding to a certain version of the AST, we then can compute the Gini coefficient based on the distribution of *SCC* among developers (Gini_{SCC}) for each file.

3. Bugs: Bug repositories, such as Bugzilla, record the information about bug reports of a system. Currently Bugzilla does implicitly not provide direct links to versioning repositories. Therefore, the information which file was affected by a specific bug and when, *i.e.*, in which revision that bug was fixed, is usually missing. However, developers often manually enter a bug report reference, *e.g.*,"*fixed bug 16745*" or "*bug1859*", into the commit messages of file revisions when fixing bugs. Prior research developed matching techniques to query those references and to establish the missing links between bugs and file revisions, *e.g.*, [20]. Again, we use EVOLIZER to automate this linkage process. Based on this information we then count the number of bugs per file (#*Bugs*).

Gini_R, Gini_{LM}, Gini_{SCC}, and #*Bugs* are then stored in one dataset on file level granularity.

Table 1: Eclipse dataset used in this study

Eclipse Project	Files	Rev.	*LM*	*SCC*	#*Bugs*	Time [M,Y]
Compare	278	3'736	140'784	21'137	665	May01-Sep10
jFace	541	6'603	321582	25'314	1'591	Sep02-Sep10
JDT Debug	713	8'252	218'982	32'872	1'019	May01-July10
Resource	449	7'932	315'752	33'019	1'156	May01-Sep10
Runtime	391	5'585	243'863	30'554	844	May01-Jun10
Team Core	486	3'783	101'913	8'083	492	Nov01-Aug10
CVS Core	381	6'847	213'401	29'032	901	Nov01-Aug10
Debug Core	336	3'709	85'943	14'079	596	May01-Sep10
jFace Text	430	5'570	116'534	25'397	856	Sep02-Oct10
Update Core	595	8'496	251'434	36'151	532	Oct01-Jun10
Debug UI	1'954	18'862	444'061	81'836	3'120	May01-Oct10
JDT Debug UI	775	8'663	168'598	45'645	2'002	Nov01-Sep10
Help	598	3'658	66'743	12'170	243	May01-May10
JDT Core	1'705	63'038	2'814K	451'483	6'033	Jun01-Sep10
OSGI	748	9'866	335'253	56'238	1'411	Nov03-Oct10
UI Workbench	3'723	38'505	1'427K	168'988	5'000	Sep02-Oct10

4. STUDY

We investigated our two research hypotheses using data extracted from 16 plug-in projects of the Eclipse platform. Table 1 gives an overview of the dataset: *Files* denotes the number of unique `*.java` files. *Rev.* denotes the total number of file revisions. *LM* is the sum of the total number of lines added, deleted, and changed. *SCC* represents to total number of fine-grained source code changes. #*Bugs* is the total number of bugs. *Time* represents the observation period.

4.1 Correlation Analysis

In this section we investigate the correlation between the Gini coefficient and the number of bugs on file level (**H 1**). Furthermore, we analyze if there is a difference in the correlation when calculating the Gini coefficient based on *R*, *LM*, and *SCC*. For the correlation analysis we used the Spearman rank correlation ρ. It is more robust than the Pearson correlation since it does not make any assumption regarding the distribution of the data and is not restricted to linear relations between two measured variables [5]. Spearman values of +1 and -1 indicate exceptionally strong correlations, whereas a value of 0 denotes the absence of any correlation. We consider correlation values of $-0.5 >= \rho >= 0.5$ as substantial and values of $-0.7 >= \rho >= 0.7$ as strong correlations [18].

Table 2 shows the correlation values of the Gini coefficient based on *R* (Gini_R), *LM* (Gini_{LM}), and *SCC* (Gini_{SCC}) on source file level for each project. We can see that all correlation values are *negative*. This means that an increase in the inequality of one of the three change measures among all developers—resulting in a larger Gini coefficient—comes with a decrease in the number of bugs. In other words, the more changes in a source file are done by a small group of developers, the less bugs it will have. In contrast, if the changes of a file are scattered more evenly among the developers it is more likely to have bugs.

For all three Gini coefficients the median correlations are below -0.5. With a median of -0.58 Gini_{SCC} has the strongest correlation of all three coefficients and the strongest correlation in 10 out of 16 projects. Furthermore, 10 projects show a substantial correlation, five out of these are even strong correlations. The second highest median of -0.55 has Gini_R. Compared to Gini_{SCC} it shows only for three projects the highest correlations. Gini_R has for nine projects substantial correlation values out of which three are strong. CVS Core, Help, and OSGI show no correlation at all. Gini_{LM} exhibits a slightly lower median correlation (-0.54). It has 10 substantial and two strong correlations on project level, and in four cases it shows the highest values.

We used a *Related Samples Friedman Test* to examine these differences between the correlation values of the three Gini coefficients. This test is the non-parametric, rank-based alternative of the *One-Way ANOVA* procedure for comparing related samples. Therefore, we can relax any assumptions regarding the distribution of the data [5]. Since the test was not significant at $\alpha = 0.05$ we conclude that the observed differences of the correlation values in Table 2 are not significant.

To assess the strength of the correlations, we performed three *One-Sample Wilcoxon Signed Rank Tests* [5] for each Gini coefficient against the hypothesized value of -0.5, *i.e.*, substantial, negative correlation. Similarly to the afore used *Friedman Test*, this test is the non-parametric counter-part of the *One-Sample T-Test*. Again, there are no required assumptions with respect to the distribution of the data. The test only requires a certain degree of symmetry, *i.e.*, approximately the same number of examples above and below the median which is true in our dataset. Furthermore, it can also be applied to smaller sized samples. All three tests were not significant at $\alpha = 0.05$, *i.e.*, the median correlations of all three Gini coefficients are not significantly different from -0.5. Therefore, we accept **H 1** stated in Section 1—*Gini coefficients based on change data correlate (substantially) negatively with the number of bugs.*

Table 2: Non-parametric Spearman rank correlation between #*Bugs* and the Gini coefficients based on *R*, *LM*, and *SCC* on source file level. (* significant at $\alpha = 0.01$)

Eclipse Project	Gini_R	Gini_{LM}	Gini_{SCC}
Compare	-0.68*	-0.69*	**-0.74***
jFace	-0.66*	-0.63*	**-0.71***
Resource	-0.55*	**-0.57***	**-0.57***
Team Core	-0.28*	-0.36*	**-0.4***
CVS Core	-0.05	**-0.5***	-0.4*
Debug Core	-0.35*	-0.34*	**-0.49***
Runtime	-0.33*	-0.42*	**-0.43***
JDT Debug	-0.63*	-0.4*	**-0.7***
jFace Text	**-0.54***	-0.52*	-0.51*
JDT Debug UI	-0.6*	-0.63*	**-0.7***
Update Core	-0.72*	**-0.75***	-0.69*
Debug UI	-0.48*	-0.55*	**-0.59***
Help	-0.08	**-0.34***	-0.29*
JDT Core	**-0.74***	-0.67*	-0.67*
OSGI	-0.09	-0.37*	**-0.47***
UI Workbench	**-0.79***	-0.74*	-0.76*
Median	-0.55	-0.54	**-0.58**

4.2 Predicting Bug-Prone Files

In the previous Section 4.1 we observed a substantial, negative correlation between the Gini coefficient based on the distribution of change data among developers. The purpose of **H 2** is to analyze whether the Gini coefficient of a file can be used to identify *bug-prone* files with reasonable performance. For that we applied six different machine learning algorithms: *Logistic Regression* (LogReg), *Random Forest* (RndFor), and *J48 Decision Tree* (J48) as implemented by the WEKA toolkit [23], *Neural Network* (NN), *Naive Bayes* (NB), and *Support Vector Machine* (LibSVM) as implemented by the RapidMiner toolkit [15]. The rationale for choosing several learning algorithms stems from previous results presented in [11] that discovered that *more sophisticated* algorithms, such as NN, LibSVM, and RndFor, or Bayesian Methods [14] achieve possibly better classification performance. We binned all files of each project into the observed, target classes *not bug-prone* and *bug-prone* using the median of the number of bugs per file (#*Bugs*) of that particular project, *i.e.*, equal frequency binning using two classes:

$$bugClass = \begin{cases} not\ bug-prone & : \#Bugs <= median \\ bug-prone & : \#Bugs > median \end{cases}$$

We then conducted three different classification experiments each using one of the Gini coefficients as input variable at a time. In each experiment we trained the six learning algorithms on all projects and computed for each project the following performance measures using 10-fold cross-validation: area under the receiver operating characteristic curve statistic (AUC) [14], precision (P), and recall (R).

We mainly use AUC in the performance discussion. AUC is a robust performance measure for classification models since it is independent of prior probability and therefore facilitates the comparison of different approaches [1].

Table 3 lists the median of AUC, P, and R over all projects of each learning algorithm for a given Gini coefficient. Except for RndFor and J48, both using Gini_{LM} as input variable, all prediction models obtain AUC values above 0.7, what Lessman *et al.* denote as *promising results* [11]. The models computed with the six machine learning algorithms differ in their performance as indicated by the different median AUC values. In the following we discuss these differences with respect to each Gini coefficient. For that, we use the *Related-Samples Friedman Test* using $\alpha = 0.05$. In the case of obtain-

Table 3: Median AUC, Precision, and Recall of prediction models computed with each machine learning algorithm (M-Learner) for the three Gini coefficients

	Gini_R			Gini_{LM}			Gini_{SCC}		
M-Learner	AUC	P	R	AUC	P	R	AUC	P	R
NN	0.74	0.65	0.8	0.75	0.66	0.83	0.78	0.7	0.84
LogReg	0.73	0.64	0.78	0.74	0.67	0.81	0.78	0.71	0.82
RndFor	0.81	0.79	0.78	0.69	0.66	0.6	0.72	0.72	0.66
NB	0.74	0.7	0.78	0.74	0.67	0.81	0.77	0.71	0.83
LibSVM	0.73	0.64	0.79	0.74	0.67	0.81	0.77	0.71	0.83
J48	0.75	0.81	0.29	0.65	0.73	0.33	0.74	0.76	0.2

ing a significant probability, *i.e.*, there is an overall significant difference regarding the AUC values among all learning algorithms, we apply pairwise-post hoc tests including an adjustment of the α-level to investigate between which two learners the differences actually occur:

Gini_R. RndFor performs the best with a median AUC of 0.81. All other learners exhibit lower values between 0.73 and 0.75. The *Friedman Test* was barely significant. The post-hoc tests showed that this is due to the better performance of RndFor.

Gini_{LM}. NN shows the highest AUC median (0.75). LogReg, NB, and SVM perform only slightly lower than NN. RndFor and J48 have median values below 0.7. The comparable low performance of RndFor was confirmed by the result of the pairwise post-hoc tests as it had the lowest mean rank of all learners. This is surprising as RndFor was the best method in case of Gini_R.

Gini_{SCC}. NN and LogReg both perform the best with a median AUC of 0.78. With a median AUC of 0.77 NB and LibSVM are second. Similarly to the case of Gini_{LM}, RndFor and J48 exhibit the lowest prediction performance. Again, this is confirmed by the *Friedman Test* and the pairwise post-hoc tests. The other algorithms do not exhibit significant differences in terms of their AUC values.

The comparison of the performance of different machine learning methods in our work supports the findings made in [11]: Some methods might perform better than others but in most cases not significantly. Consequently, the selection of a particular machine learning technique should not be based on performance alone but also on other criteria, *e.g.*, readability of the resulting prediction models.

In Section 4.1 we could not observe a significant difference regarding the correlation of Gini_R, Gini_{LM}, and Gini_{SCC} with #*Bugs*. Analogously, we applied a *Related-Samples Friedman Test* to the AUC values of the best performing learners of each Gini coefficient, *i.e.*, RndFor for Gini_R, NN for Gini_{LM}, and NN and LogReg for Gini_{SCC}. The test was not significant: The Gini coefficients based on the distribution of *R*, *LM*, and *SCC* among developers can equally well discriminate between *not bug-prone* and *bug-prone* files in our dataset. RandFor using Gini_R obtained the highest median AUC and resulted also in adequate values for precision and recall (0.79 and 0.78 respectively). Therefore, we accept **H 2** stated in Section 1—*Gini coefficients based on change data can be used to classify source files into bug- and not bug-prone files.*

4.3 Discussion of the Results

In this work we empirically investigated the relation of how changes are distributed among developers and the number of bugs in source files. For that, we computed the Gini coefficient—a prominent economic measure for the inequality of distributions—using three different change measures, *i.e.*, revisions, lines modified, and fine-grained source code changes. The results show that the more changes of a source file are done by a relativly small group of developers, the less likely it will have bugs. The findings suggest that code ownership in terms of changes should be enforced (at least to a certain degree) and contributions made to a file should be focused on a few dedicated developers. Moreover, our models can list potentially *bug-prone* files whose ownership should be re-organized in order to reduce the likelihood of bugs. Since the collection of all required data can be fully automated in a straightforward way and the tools, such as EVOLIZER and CHANGEDISTILLER exist, our models could be integrated into a versioning system for tool support.

We see two main potential threats to the validity our work that need to be discussed: First, our study used only data from the Eclipse platform. Therefore, our results are possibly biased by the unique characteristics of the Eclipse development process. This fact might threaten the generalizability to systems other than Eclipse. Nevertheless, Eclipse is a mature system that has been subject of numerous studies before, *e.g.*, [9, 16, 24]. As such, we can benefit from and contribute to prior knowledge. Furthermore, our results confirm prior findings that the contribution structure of source code, *e.g.*, [2, 18], is related to bugs.

Second, different commit policies and behaviors among the developers can influence the measurement of the change data. For example, some developers might regularly commit (small) individual changes for each modification task, *e.g.*, a single bug fix, while others only commit larger changes, *e.g.*, refactorings including bug fixes. In addition, some projects follow certain commit policies that allow only a set of core developers to commit changes to the versioning repository. The original developers of the changes are not mentioned.

For both threats, additional studies with different systems are required to sustain our findings in this work.

5. RELATED WORK

Vasand *et al.* were among the first to describe the distribution of software engineering data using the Gini coefficient [13]. They collected a number of product metrics on class level, *e.g.*, *Number of Methods*, and used the Gini coefficient to analyze how those metrics are distributed among the classes on several systems. Similarly to our work, Winston computed the Gini coefficient based on change data and its distribution among developers [22]. However, he used it as a project risk measure rather than an indicator for bugs: The higher the Gini coefficient, the more a project depends on a few key developers—a situation often termed as *key man risk*. Closer to our work are studies that relate the contribution structure of source code to defects. Pinzger *et al.* found out that the position of Windows Vista binaries in the developer contribution network is an indicator of failures [18]. Bird *et al.* investigated the effect of minor contributors on failures in such networks [2]. Somewhat contrary, other work stated that the number of developers does not significantly affect bugs [21]. Moser *et al.* presented a comparative study using change and product metrics to predict defects in Eclipse [16]. Among others, their change metrics include the number of authors that committed a file. Schroeter *et al.* showed that import relations of files and packages in Eclipse correlate with their bug-proneness [19]. In addition to traditional complexity metrics, the structure of the abstract syntax tree of Eclipse source code was used to predict defects [24].

Another study on Eclipse investigated the relation between post-release failures and dependency network measures [17].

Fenton and Neil provide critical survey on several defect prediction methods [6]. An extensive review and comparison of more recent bug prediction approaches is given in [3].

6. CONCLUSIONS & FUTURE WORK

We empirically investigated the relationship between code ownership and bugs in source files. For that we computed the Gini coefficient for each file in our dataset based on the distribution of the changes of that particular file among all developers. We measured changes at three different granularity levels: Revisions, lines modified, and fine-grained source code changes. A high Gini coefficient for a given file means that a relatively small group of developers is responsible for a large amount of changes, *i.e.*, there is a high degree of code ownership for that file with respect to changes. Summarized, the results of our study are:

- The number of bugs in a file correlates negatively with the Gini coefficient: The more changes of a file are done by a few dedicated developers (high Gini coefficient) the less likely it will have bugs (**H 1**).
- The Gini coefficient on file level can be used to identify *bug-prone* files with adequate performance. The best results (AUC of 0.81) are obtained with a Random Forest prediction model using the Gini coefficient based on the distribution of revisions among developers (**H 2**).

Future work is basically concerned with extending our experiments to systems other than Eclipse. Furthermore, to shed more light on the characteristics of source code changes, we plan to include information about the types of changes. For instance, is it more critical with respect to bugs that declaration changes are done by only a few developers than it is in the case of source code statement changes, *e.g.*, assignments.

7. REFERENCES

[1] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *Proc. Int'l Workshop on Principles of Softw. Evolution*, pages 11–18, 2007.

[2] C. Bird, N. Nagappan, H. C. Gall, P. Devanbu, and B. Murphy. An analysis of the effect of code ownership on software quality across windows, eclipse, and firefox. Tech. Report 140, Microsoft Research, October 2010.

[3] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proc. Int'l Workshop on Mining Softw. Repositories*, pages 31–41, 2010.

[4] R. Dorfman. A formula for the gini coefficient. *The Review of Economics and Statistics*, 61(1):146–149, February 1979.

[5] S. Dowdy, S. Weardon, and D. Chilko. *Statistics for Research*. Probability and Statistics. John Wiley and Sons, Hoboken, New Jersey, third edition, 2004.

[6] N. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25:675–689, September 1999.

[7] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. on Softw. Eng.*, 33(11):725–743, November 2007.

[8] H. C. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *IEEE Software*, 26(1):26–33, January/February 2009.

[9] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proc. Int'l Workshop on Mining Softw. Repositories*, page to appear, 2011.

[10] C. Gini. Variabilità e mutabilità. *Memorie di metodologica statistica*, 1912.

[11] S. Lessmann, B. Baesens, C. M. Swantje, and Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. on Softw. Eng.*, 34(4):485–496, July 2008.

[12] M. O. Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association*, 9(70):209–219, June 1905.

[13] R. V. M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *Proc. Int'l Conf. on Softw. Maintenance*, pages 179 –188, 2009.

[14] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. on Softw. Eng.*, 33(1):2–13, January 2007.

[15] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In *Proc. Int'l Conf. on Knowledge Discovery and Data Mining*, pages 935–940, 2006.

[16] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. Int'l Conf. on Softw. Eng.*, pages 181–190, 2008.

[17] T. Nguyen, B. Adams, and A. Hassan. Studying the impact of dependency network measures on software quality. In *Proc. Int'l Conf. on Softw. Maintenance*, pages 1 –10, 2010.

[18] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proc. ACM SIGSOFT Symposium on the Foundations of Softw. Eng.*, pages 2–12, 2008.

[19] A. Schroeter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proc. Int'l Symposium on Empirical Softw. Eng.*, pages 18–27, 2006.

[20] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Workshop on Mining Softw. Repositories*, pages 1–5, 2005.

[21] E. Weyuker, T. Ostrand, and R. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Softw. Eng.*, 13(5):539–559, October 2008.

[22] R. Winston.
The gini coefficient as a measure of software project risk.
http://www.theresearchkitchen.com/blog/archives/219.

[23] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Data Management Systems. Morgan Kaufmann, second edition, June 2005.

[24] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proc. Int'l Workshop on Predictor Models in Softw. Eng.*, pages 9–15, 2007.

TUD-SERG-2011-018
ISSN 1872-5392