2015 International Conference on Computational Intelligence & Networks

Mining Github for Novel Change Metrics to Predict Buggy Files in Software Systems

K Muthukumaran, Abhinav Choudhary, N L Bhanu Murthy
Department of Computer Science and Information Systems
BITS Pilani Hyderabad Campus
{p2011415, f2009155, bhanu}@hyderabad.bits-pilani.ac.in

***Abstract*—Code change metrics mined from source control repositories have proven to be the most reliable predictors of bugs in contemporary software engineering research. Yet a definitive modus operandi for obtaining the required data from a particular software configuration management (SCM) repository needs to be put forward. In this paper, we define a modus operandi to extract some popular change metrics from the Eclipse repository on Github, which can be generalized for any open-source Github repository. We define few code change metrics that are intuitively significant for predicting bugs. Bug prediction models built with these metrics along with the existing prominent code change metrics prove to be competent and consistent as per our experiments on five different versions of Eclipse JDT project. We explored Naïve Bayes Tree algorithm to build a prediction model and have found it to perform better than other commonly used algorithms in this problem domain.**

I. INTRODUCTION

Quality has always been a non-compromising priority task for all IT companies and post release bugs experienced by end customer hampers quality heavily. With time and manpower being limited or getting limited for verification and validation phase of IT project, it would be great if one can predict bug prone files so that the team can focus more on those bug prone files. It helps verification and validation team to allocate or manage their critical resources well. Hence the research on bug prediction models has become significant and more than hundred research papers have been published. Kim et al. proposed FixCache model that uses past bugs to search the vicinity of future bugs with the assumption that faults do not occur in isolation, but rather in bursts of several related faults [1]. Zimmermann et al. mapped defects from the bug database of Eclipse to source code locations and built prediction models that showed that the combination of complexity metrics can predict defects, suggesting that the more complex the code, more bugs it has [2]. There is lot of research focus on studying the impact of source code metrics and past defects on bug prediction during last decade. As we all know IT industry continues to deliver software in evolutionary fashion with almost always changing requirements. When it comes to software giants, change is about millions of lines of code, thousands of developers and billions of customers. Changes are inevitable in constantly changing business environment. Some of the recent studies focused on different change metrics and they clearly outperform code complexity and other source code metrics in bug prediction. Studies done by Hassan shows that change metrics are as good as or even better predictor of faults than prior faults and he concludes that the more complex changes to a file, higher the chance that file contains faults [3]. Nagappan et al. found that change burst metrics yield excellent predictive capability in projects with high-quality changes. They claim that precision and recall exceed 90% for Windows Vista: the highest predictive power ever observed [4]. Nagappan and Ball took code churn measures to build defect prediction model and found out that relative code churn measures are excellent predictors of defect density in large industrial software systems [5]. Moser et al. shows that change metrics clearly outperform predictors based on static code attributes for the Eclipse project [6] [7]. They confirm the observations made by other researchers, as change data, and more in general process related metrics, contain more discriminatory and meaningful information about the defect distribution in software than the source code itself. They suggest that while most of the past research effort has been invested in code metrics based approaches and only produced mixed results ,there remains much more to be explored in the area of how the software process impacts the generation of defects during the software development life cycle. Hence our research problem is focused on predicting program files that are going to be bug prone due to various changes that happened in software over period of time. Since literature proves that change metrics contribute more towards bugs, we tried to do similar experiment with existing and some new change metrics on open source platform. Our results are quite impressive in spite of the fact that the change processes in open source projects are not as controlled as it is in commercial applications.

II. CHANGE METRICS

Unlike source code metrics, object oriented metrics and CK metrics, change metrics do not concern themselves with the contents of a source file, but rather with the details of changes made to the code base over time. Change metrics are gathered from the software configuration management repository of a particular project. One advantage of using change metrics over source metrics is that they are independent of the programming language. Change metrics are extracted for the time period between two consecutive major releases of software and are calculated on a per-file basis. For purposes of this paper, we refer to this time-period between two consecutive releases as timeline. In our work we have considered 12 change metrics. We have considered all major releases of Eclipse from 3.0 to 3.5, and calculated these metrics for the time period between any two consecutive releases. Some of these metrics have been widely considered in other bug prediction research and a few are unique to this paper. We explain the metrics below, along with the rationale for considering them in the feature set.

2375-5822/15 $31.00 © 2015 IEEE
DOI 10.1109/CINE.2015.13

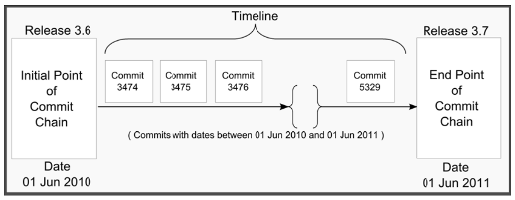
Timeline
Release 3.6
Initial Point of Commit Chain
Date 01 Jun 2010
Commit 3474
Commit 3475
Commit 3476
Commit 5329
(Commits with dates between 01 Jun 2010 and 01 Jun 2011)
Release 3.7
End Point of Commit Chain
Date 01 Jun 2011

Fig. 1. Data Extraction

A. *Commits*

The number of commits within the timeline that have modified a file. Rationale: More a file is changed, more the probability of a defect.

B. *Add*

The total number of lines added to the file due to all the commits within the timeline. Rationale: More the amount of change, more the probability of a defect.

C. *Delete*

The total number of lines deleted from the file due to all the commits within the timeline. Rationale: More the amount of change, more the probability of a defect.

D. *Authors*

The number of unique authors involved in commits that have modified this file within the timeline. Rationale: We assume that more the number of different authors involved in modifications to the file, the greater the chances of a defect.

E. *Commits60*

The number of times the file was changed (the no. of commits to the file) in the last 60 days of the later release. That is, if D is the date of the later release (release 3.7 in figure(1)) and d = D - (60 days), then COMMITS60 is the number of commits within dates d and D that modify this file. Rationale: It is assumed that if a file is changed more just before release, the more the possibility of post-release defects.

F. *Last Commit*

The number of days before the later release date when the file was last changed in a commit. Rationale: A file is expected to be more prone to post-release defects if the last change is closer to the release date.

G. *In Development Bugs*

The number of bug fix commits affecting the file within the timeline (the development period). Rationale: We expect a file to be more defect-prone if it has been frequently fixed for bugs.

H. *Entropy*

Intuition tells us that the distribution of changes (commits) to a file within the timeline should correlate in some way to post-release defects. If we divide the timeline into N equal periods, a flat distribution would have nearly equal number of changes in each of the N periods. On the other hand, changes might be concentrated to one or more regions resulting in peaks and lows in the change distribution graph. We use Shannon's definition of Entropy to estimate this change distribution. If C_i is the number of commits in the i-th period that affect the file and C_{total} is the total number of commits within timeline that affect the file, then we define Entropy of code change for the file as

$$-\sum_{i=1}^{N}\left(\frac{C_i}{C_{total}}\right)\times\log\left(\frac{C_i}{C_{total}}\right) \quad (1)$$

All major Eclipse releases are usually spaced one year apart. So for purposes of this paper, we have considered N as 12. It is to be observed that the way we defined Entropy is fundamentally different from Hasan's Entropy [3].

I. *Mean Period of Change (MPC)*

We divide the timeline into 12 equal periods as in case of ENTROPY. If all changes (commits) for a file are concentrated in one particular period, say period i, then we expect that different values of this period of concentration i should affect post-release defects in different ways. A low value for i implies that the file was last changed in the early periods of the timeline and has been free of changes for the remaining periods. So it can be considered relatively stable, as compared to a file for which period of concentration is localized closer to the date of release. Mean Period of Change is an attempt to estimate this center of concentration of changes.

$$MPC = \sum_{i=1}^{N} i * \left(\frac{C_i}{C_{total}}\right) \quad (2)$$

J. *Maximum Burst*

Nagappan et al. define a change burst as a “sequence of consecutive changes” to a file. Two parameters are used to detect change bursts gap size and burst size. Gap size determines the minimum time gap between two changes (commits) to a file. For a file, commits with time gap less than the gap size will belong to the same change burst. The burst size determines the minimum no. of changes (commits) in a change burst. If the no. of commits in a change burst is less than the burst size, the change burst will not be considered [4]. . MAXBURST represents the maximum change burst for a file, and is the maximum no. of commits in any change burst for the file, i.e. $max|B|/B\epsilon bursts(file)$. We used gap size as 3 and max burst as 3 in our experiments.

K. *Maximum Change Set*

Both Moser et al. [8] and Krishnan et al. [9] consider two change metrics MAXCHANGESET and AVGCHANGESET (described next) that relate to the number of files changed along with the current file. For every file, the MAXCHANGESET is the maximum number of files changed along with this file in any commit within the timeline. If C is the subset of commits within the timeline that affect this file, then MAXCHANGESET is

$$\max_{c \epsilon C} n(c) \tag{3}$$

where $n(c)$ is the number of files modified by commit c, other than the current file.

L. Average Change Set

For every file, the AVGCHANGESET is the average of the number of files changed together with this file, calculated over all commits affecting this file. If C is the subset of commits within timeline that affect this file and $|C|$ is the size of this subset, the AVGCHANGESET is

$$\frac{1}{|C|} \sum_{i=1}^{N} n\left(c_i\right) \tag{4}$$

where $n\left(c_i\right)$ is number of files modified by commit, other than the current file.

III. DATA EXTRACTION

GitHub is a web-based code hosting platform that uses the Git revision control system developed by Linus Torvalds. Some popular projects hosted in GitHub are rails, jquery, node, html5-boilerplate, homebrew and diaspora. In our research, we have focused on extracting data from open-source software projects hosted on GitHub because of the growing popularity of the platform and also because of the easy availability of the GitHub API in different programming languages. To access the Github API, we have found the open-source Python library PyGithub[1] to be extremely helpful. In revision control terminology, a commit represents a change to the code repository/database. In GitHub, every commit is uniquely represented by a 40 character hex-string, generated from the commit data using SHA-1 hash algorithm.

A. Extracting six change metrics

We list the details of the procedure we followed in data extraction of first six metrics (explained in section II-A through II-F). We assume we are gathering change metrics between release 3.2 (release date D1) and release 3.3 (release date D2) of some open-source repository on Github.

- All commits made to the repository are scanned sequentially and commits with date between D1 and D2 are filtered out. The SHA strings of these commits are listed to a file.
- For each commit SHA in the file, the following details of the commit are retrieved:
  - Author, the person responsible for the change.
  - Date and Time, when the change was updated to the repository.
  - Number of files modified as a part of the change.
  - Names of the modified files.
  - Number of lines added and deleted from each modified file.

  The above details for a particular commit are output to a file in an easily parseable format, with the commit SHA as the filename, to guarantee uniqueness.
- All Java files from the two releases are listed separately, and only files common to both releases are considered.
- The commit data files generated in previous steps are processed to extract the first 6 metrics for each Java file common to both releases:

At this point, we have the name of each Java file along with the 6 generated metrics for each file. Files that have not been changed at all during the timeline will have zero values for all 6 metrics. Currently we do not consider such files in our study, because we focus only on how changes to a file affect the post-release bugs. We filter out such files.

B. Extracting number of in-development bug fixes

In-development bugs refer to the number of bug fix commits made to a file during the period between the two releases. To recognize a commit as a bug fix, we parse the commit message string for the words ‘Bug’, ‘Fix’, ‘fixed’ [10]. If any of the words are matched in the commit message, the bug fix count for that file is incremented by one.

C. Calculating the code change entropy

We divide the timeline into N periods and calculate the start and end dates for each period. For each file, we loop through the list of commits (considering only commits affecting the particular file) and distribute them in the N periods according to their commit dates. So, for every file, we know C_i, the number of commits to the file in period i. Also, C_{total} is the number of commits out of all the commits in the timeline that modify the file under consideration. Entropy for the file is then calculated according to the definition in Section II-H.

D. Calculating Mean Period of Change

As in case of entropy above, the C_i values are calculated for each file ($i \epsilon [1 \ldots 12]$) and then MPC (Mean Period of Change) is calculated as per definition in Section II-I.

E. Calculating MAXBURST

MAXBURST is calculated as per definition in Section II-J

F. Calculating MAXCHANGESET and AVGCHANGESET

For every file, three variables are maintained: Max – Holds the MAXCHANGESET value. Initialized to zero. Sum – Holds the sum of the number of files changed along with this file by commits. Initialized to zero. $Count$ - Holds the number of commits in the timeline that affect this file. Initialized to zero. For every commit in the timeline:

- For every file that the commit modifies, increment the file’s $Count$ by 1
- N is the total number of files modifies by the commit. For every file modified by the commit, add $(N-1)$ to the file’s Sum. Also for every file that it modifies, replace max by $(N-1)$, if max is less than $(N-1)$

[1]https://github.com/jacquev6/PyGithub

The final max value of a file is the MAXCHANGESET value of the file. The AVGCHANGESET value for the file is the Sum divided by the $Count$.

G. Calculating post release bugs

Bug reports are traced backwards from bug fixes by identifying the bug number from the commit message and then looking up the report date for that particular bug in a bug repository. We make an important assumption here: "Any bugs reported (for the projects mentioned in this paper) are fixed within six months from the report date" With this assumption, all bugs reported within six months post-release will have been fixed within one year post-release. So we need to consider all bug fix commits within one year from the later release date. We maintain a list of files along with the no. of post-release bug reports, and initialize the no. of reports to zero for every file. For every bug fix commit within 1 year post-release:(1) The commit message is analyzed to strip out the bug ID number for the bug that was fixed in the commit. (2) The bug repository for our project[2] is then consulted and the document for the particular bug ID is parsed to extract the report date.(3) The report date is checked to see if it lies between the release date of the later release and within 6 months from that. (4) If it does, the bug report counts for each file modified by the commit is incremented by 1.

At this point, we have, for each file, the number of post-release defects (bugs) which is a numeric. Since we would like the output class to be binary nominal (YES or NO), all zero values for post-release defects are replaced by a 'NO' indicating the absence of post-release defects and all non-zero values are replaced by a 'YES' indicating the presence of post-release defects for that file.

IV. EXPERIMENT AND RESULTS

For our experiments, we have used the core Java Development Tools[3] repository of Eclipse from GitHub. Metrics were extracted for all releases starting from Eclipse 3.0 to Eclipse 3.5. The Eclipse archives[4] provides the release dates:

TABLE I. ECLIPSE DATA SET

Release Version	Release Date
3.0	25-Jun-04
3.1	27-Jun-05
3.2	29-Jun-06
3.3	25-Jun-07
3.4	17-Jun-08
3.5	11-Jun-09

The reader should notice that the period between 2 consecutive releases from the above table is nearly 1 year. Keeping this in mind, we have taken every 2 consecutive releases (3.0 to 3.1, 3.1 to 3.2 and so on) and generated change metrics for the time-period.

A. TRAINING CLASSIFIERS

A study of previous bug prediction research shows that the most popular classification algorithms used in bug prediction are Naïve Bayes (NB), Decision Trees and Logistic regression. Menzies et al. [11] used NB successfully for bug prediction using static code attributes and concluded that it performs better than Decision Trees. Similarly, Moser et al. [6] did a comparative analysis of the performance of NB, Logistic regression and Decision Trees while using change metrics to predict defects. Zimmerman et al. [2] used static code metrics for the Eclipse project and logistic regression models for classification. For the sake of comparison, we have tested the datasets by 10-fold cross-validation using the three classification models separately. We have also considered the NB tree algorithm from Ron Kohavis work [12], which is a hybrid of the NB and Decision Trees algorithms and supposedly has better predictive capabilities than the component algorithms alone. As per our knowledge, this algorithm has not been used in bug prediction models.

B. 10-Fold Cross Validations

Cross-validation is a general model validation technique for measuring the predictive ability, usually expressed in terms of accuracy or f-measure, of a classification model.

The 10-fold cross-validation technique, as used in this paper, works by partitioning the training data set into 10 nearly equal subsets. Taking one of the subsets as testing/validation set, the classifier is trained on the remaining 9 subsets and the number of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN) are calculated. This is repeated by taking each of the 10 subsets as testing/validation set once. The values for each of the 10 subsets are then combined to get the mean precision, recall and f-measure.

Precision(P): Precision measures what fractions of instances that the classifier predicts as belonging to a particular class actually belong to that class. It is defined as

$$\left(\frac{np}{nn} * \frac{TP}{TP+FP}\right) + \left(\frac{nn}{n} * \frac{TN}{TN+FN}\right) \tag{5}$$

where np = number of instances of positive class in the test set, nn =number of instances of negative class in the test set, n = total number of instances in the test set

The precision for the Positive output class is:

$$P_{(pos)} = \left(\frac{TP}{TP+FP}\right) \tag{6}$$

and that for the Negative output class is:

$$P_{(neg)} = \left(\frac{TN}{TN+FN}\right) \tag{7}$$

So the final precision of the classification is a weighted average of the precision for the individual output classes.

Recall(R): Recall measures what fractions of instances that actually belong to a class are predicted by the classifier as belonging to that class. It is defined as

$$\left(\frac{np}{n} * \frac{TP}{TP+FN}\right) + \left(\frac{nn}{n} * \frac{TN}{TN+FP}\right) \tag{8}$$

Where, The recall for the positive output class is:

$$R_{(pos)} = \frac{TP}{TP+FN} \tag{9}$$

[2]http://bugs.eclipse.org
[3]http://eclipse.jdt.core
[4]http://archive.eclipse.org/eclipse/downloads

The recall for the negative output class is:

$$R_{(neg)} = \frac{TN}{TN + FP} \tag{10}$$

F-measure(F): F-measure combines Precision and Recall into a single value while assigning different weights to each. In the most general sense, F-measure is given as:

$$F(\alpha) = \frac{1}{\alpha \frac{1}{P} + (1-\alpha)\frac{1}{R}} \tag{11}$$

where P = precision R = recall α = factor that determines the weights assigned to P and R; 01 When $= 0.5$, F-measure is just the harmonic mean of P and R.

$$F = \frac{2PR}{P + R} \tag{12}$$

If $\alpha > 0.5$, the F-measure is precision-oriented and when $\alpha < 0.5$, the F-measure is recall-oriented. Keeping in accordance with Precision and Recall calculations, we calculate F-measure separately for each output classes.

$$F_{(pos)} = \frac{1}{\alpha \frac{1}{P_{(pos)}} + (1-\alpha)\frac{1}{R_{(pos)}}} \tag{13}$$

And

$$F_{(neg)} = \frac{1}{\alpha \frac{1}{P_{(neg)}} + (1-\alpha)\frac{1}{R_{(neg)}}} \tag{14}$$

The final F-measure calculated for the classification is a weighted average of the two:

$$\frac{np}{n} * F_{(pos)} + \frac{nn}{n} * F_{(neg)} \tag{15}$$

We have done several experiments with α taking different values but the F-measure values revolve around the same value as that of F-measure with α as 0.5. Hence we conducted all our experiments with α as 0.5.

TABLE II. MOSER'S (RECALL) RESULTS

	Algorithms			
Metrics	NB	DT	LR	DT with CSA
Change Metrics	24%	60%	28%	80%
Source Code Metrics	40%	38%	27%	65%
Change + Code Metrics	36%	58%	38%	74%

C. Results

As explained in Section III, we obtained all code change metrics for the timeline between two consecutive releases and performed 10-fold cross-validation on the data acquired between different releases. We calculated the precision, recall and F-Measure according to the definitions in section IV-B. We have used Gaussian NB, CART Decision Tree, Logistic Regression and NB Tree algorithms to build bug prediction models for all considered versions of Eclipse JDT project. NB Tree algorithm has not been explored for bug prediction models though many researchers have used first three algorithms frequently. The results of different algorithms with our change metrics set for versions 3.0 through version 3.5 are depicted in Table III. The consistency of results across five release by all four algorithms shows that our metrics set consisting of novel metrics like Entropy, MPC and other prominent metrics is an ideal set to build bug prediction models. The recall average of 76.44% is quite impressive in contemporary bug predictions models. NB Tree algorithm is better than all other three algorithms individually for each release individually. The precision, recall, F-measure averages of NB Tree are 75.02, 76.44, 74.62 and they are better than all other three algorithms. The standard deviation is around 1 and hence, we can say that results are very consistent across the releases. Based on our experience we recommend this algorithm to be used more often in tandem with other algorithms for all comparison studies. Nagappan et al., claim recall of 92% on windows vista using change burst metrics [4]. It is worthwhile to note that these predictions differ from our predictions as described below: (1)The granularity for their prediction is binaries rather than file. A binary may contain on an average of 40 files and if a binary is predicted to be buggy then all files in it are to be reviewed / acted upon. If granularity is file level as in our case then the only predicted files need to be reviewed and this is much more cost effective than earlier model. The bug prediction models with granularity such as binary or package level may have good recall, precision values but they may not be desirable with respect to cost effectiveness. (2)Windows Vista is developed under very much controlled environment where in processes are very much streamlined and are adhered to the core by all Microsoft developers. But whereas Eclipse being an open-source project the change process may be not as controlled as it is in Microsoft projects and also there are fundamental differences between the two developments / change processes.

Nagappan et al. conducted experiments on Eclipse 2.0 with same set of metrics and their results (Recall 51% and Precision 67%) are in no way comparable to the experiments they conducted on Windows Vista (Recall 92% and Precision 91%) [4]. Hence we believe that our metrics set is worth considering and our results on Eclipse Releases are competitive and quite impressive. For us to compare the effectiveness of our model with other models, we have experimented on Eclipse JDT Release 2.0 to Release 2.1 as well. We compare all the models by making use of the measure, Recall, as it is common across all works and significant measure for our research problem. Moser et al. performed experiments on Eclipse 2.1 Release with 31 source code metrics and 18 change metrics. They have built three models with (1) source code metrics alone (2) change metrics alone (3) both source code and change metrics. They have used NB Classifier, Logistics Regression (LR), Decision Tree (DT) and also improvised their results using cost-sensitive analysis on Decision Tree (CT with CSA) [6] to build bug prediction models. Their results are depicted in Table II.

Schroter et al. considered the impact of importing packages / classes in files on bug prediction and they performed experiments on Eclipse 2.1 using Support Vector Machines (SVM) [13]. Two bug prediction models have been built using SVM - the imported classes inside a file as feature in the first model and imported packages inside a file as feature in the second model. Recall of 16.9% is observed in first model and 9.69% is observed in second model.

Zimmerman et al. studied the impact of forty odd source code complexity metrics on bug prediction using Eclipse 2.1

TABLE III. Bug Prediction Results with change metrics on five Eclipse JDT Releases

	Algorithms											
	Gaussian Naïve Bayes			CART Decision Tree			Logistic Regression			Naïve Bayes Tree		
	P	R	F	P	R	F	P	R	F	P	R	F
Eclipse 3.0 - 3.1	**73.31**	**68.448**	**66.074**	**71.019**	**71.034**	**71.023**	**73.3**	**73.3**	**73.2**	**76.4**	**76.4**	**76.3**
Eclipse 3.1 - 3.2	**77.445**	**78.168**	**76.788**	**70.621**	**71.016**	**70.8**	**76.3**	**77.2**	**75.7**	**75.4**	**76.4**	**75**
Eclipse 3.2 - 3.3	**72.726**	**76.149**	**73.387**	**70.105**	**70.678**	**70.38**	**73.2**	**77.2**	**71.9**	**74.6**	**77.9**	**73.6**
Eclipse 3.3 - 3.4	**73.532**	**74.1**	**71.001**	**70.466**	**70.863**	**70.641**	**72.5**	**73.7**	**71.8**	**75.3**	**76.1**	**75.3**
Eclipse 3.4 - 3.5	**72.132**	**74.177**	**72.448**	**70.461**	**69.873**	**70.146**	**71.2**	**73.9**	**70.8**	**73.4**	**75.4**	**72.9**
Average	**73.829**	**74.2084**	**71.94**	**70.5344**	**70.6928**	**70.598**	**73.3**	**75.06**	**72.68**	**75**	**76.4**	**74.62**
S D	**2.0934**	**3.62765**	**3.9098**	**0.33045**	**0.48014**	**0.3446**	**1.875**	**1.9655**	**1.8913**	**1.11**	**0.91**	**1.363**

version data. Recall of 16% is observed when prediction model is built using Eclipse 2.1 version data and tested again on the same data [2]. We have gathered all our change metrics in the timeline between Eclipse JDT Release 2.0 and Release 2.1 and post release bug data for six months after the release date of Release 2.1.
We have experimented with all four algorithms, Gaussian NB, CART Decision Tree (DT), Logistic Regression (LR) and NB Tree. The results are displayed in Table IV.

TABLE IV. Recall of Eclipse JDT 2.1 Release

	Algorithms			
Metrics	NB	DT	LR	NBT
12 Code,Change Metrics (mix of novel and existing metrics) as proposed in this paper	83.7	85	90.4	86.8

Our change metrics leads to exceptional results as compared to any other models with Recall of 90.4% using LR. Even the other three algorithms have performed exceedingly well as compared to any other model.

V. Threats to Validity

For comparing the effectiveness of our model with other models we have conducted our experiments on Eclipse JDT 2.1 Release where as all other models might have conducted on all Packages of Eclipse 2.1 Release. If we do experiments on all packages of Eclipse 2.1 Release the results might not be the same. Though we confirmed the correctness of our data set with repeated experiments there could still be some issues. For example, if developer made many overlapping edits to a file in single check-out/check-in then we could have wrongly captures related metrics. There are chances that we could have missed some bug data due to lack of proper documentation which is quite natural in open source platforms. Another limitation of our work is we have done the experiments only in different versions of Eclipse project and not with any other project. Hence the similar metrics for other projects may not yield same consistent results.

VI. Conclusion

A definitive procedure along with working software has been put forth to get some popular code change metrics for any project that has been hosted on GitHub. We have proposed couple of new change metrics to figure out whether commits are uniformly distributed over the timeline between the two consecutive releases. The bug prediction models have been built with these novel metrics along with some existing prominent change metrics by making use of four different algorithms and these models yield consistent and competent results for five consecutive releases (Release 3.0 through Release 3.5) of Eclipse JDT project. NB Tree Algorithm that has not been explored in this problem domain found to be the best to be used. We compared our results on Eclipse 2.1 Release with the results of other existing models on the same version and discovered that our model is doing exceedingly well.

References

[1] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 489–498.

[2] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*. IEEE, 2007, pp. 9–9.

[3] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 78–88.

[4] 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*. IEEE, 2010, pp. 309–318.

[5] 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*. IEEE, 2005, pp. 284–292.

[6] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 181–190.

[7] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 31–41.

[8] R. Moser, W. Pedrycz, and G. Succi, "Analysis of the reliability of a subset of change metrics for defect prediction," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 309–311.

[9] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova, "Are change metrics good predictors for an evolving software product line?" in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. ACM, 2011, p. 7.

[10] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[11] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[12] R. Kohavi, "Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid." in *KDD*, 1996, pp. 202–207.

[13] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 2006, pp. 18–27.