An Automated Tool for Collection of Code Attributes for Cross Project Defect Prediction

Ruchika Malhotra
Computer Science and Engineering
Delhi Technological University
ruchikamalhotra2004@yahoo.com

Bhavyaa Bansal
Computer Science and Engineering
Delhi Technological University
bhavbansal17@gmail.com

Chitranshi Jain
Computer Science and Engineering
Delhi Technological University
jain.chitranshi1994@gmail.com

Ekta Punia
Computer Science and Engineering
Delhi Technological University
ektapunia12@gmail.com

***Abstract*— This paper presents a tool that automates the process of data collection for defect or change analysis. Prediction of defects in early phases has become crucial to reduce the efforts and costs incurred due to defects. This tool extracts the information from Git Version Control System of open source projects. Two consecutive versions of one single project have been used by the tool to obtain results. The tool generates a matrix containing code churn (added lines, deleted lines, modified lines, total LoC), complexity, pre-release bugs and post-release bugs of each file of source code. The obtained software metrics can be used to measure the development process of a software and therefore in analysis and prediction purposes.**

Keywords— defect prediction, code churn, open source, pre-release bugs, post release bugs.

I. INTRODUCTION

Software industry is rapidly growing and the need for bug- free and reliable software is also increasing. As the size and complexity levels of software systems have increased, the number of faults present in the software system also have increased exponentially. During the maintenance phase of software, each fault costs a lot of time and effort in fixing [7]. In order to reduce this cost, research is being done to estimate the probability of occurrence of faults in the initial phase of Software development lifecycle. One way to deal with the problem is to predict the faulty portions of the software in early phases of software development lifecycle [1]. The probability whether a software entity will have a defect or not is measured by defect proneness. This work comes under the field of defect prediction. Defect prediction can be done using software metrics. Software metrics consists of code churn of source code. Code churn is the modification done in the code, during development [2]. Code churn is obtained by comparing the two consecutive versions of open source projects. Apart from code churn, we need information of pre-release and post-release bugs. Pre-release bugs are those faults that have occurred before the release of the version of the project and post-release bugs are those faults that come after the release of the version of the project. Each commit is analysed for defect data and code churn, this procedure takes time and is not feasible manually for real life projects in industry, due to large size of projects, which could lead to inaccurate data. This process of data collection if done manually, takes a lot of human effort and time. Therefore, there is a need to automate the collection of software defect data for effective defect prediction [8]. The tool has been built to automate the data collection procedure for open source projects. In open source projects, development of the project is done globally by a group of co-developers. Any person around the globe can be a part of the project and every individual participating in the project becomes a contributor to the project.

We have used the Git version control system to collect defect data of open source project from the Git repositories. The reason for choosing Git version control system is that the development and maintenance phase data of projects is globally available. Besides, it has many users. The changes done on open source projects of these repositories are recorded in a well-structured format of change log. Each commit in the change log consists of <change id, contributor id, description, list of changes >. Previous studies have shown that defect data collected from open source projects can be used in research areas such as defect prediction [3].

The work has been done on open source because the data of development and maintenance phase are easily available. The results generated by the tool consists of code churn, complexity, pre-release, post-release bugs of source code. Code churn measures the number of additions, deletions, modifications done to each component of the source code.

The remote repository of source code is required on end-user machine. Hence, cloning of git version control repository on the local machine has been added to tool as a basic functionality. The secondary reports generated by the tool include the descriptive statistics of collected code churn measures. Statistics calculated by tool are: maximum, median and standard deviation of each metric. The summarized data can be used as input for prediction and analysis purposes.

The remaining paper consists of five sections. Section II has

978-1-5386-2989-5/17/$31.00 ©2017 IEEE

detailed description of the tool. Section III explains the working mechanism of tool. This section has the procedural elaboration of each functionality. Section IV discusses the output generated by tool on given input. Section V provides the information about the various applications of the outputs and the scope of future research work on it. Section VI has conclusion.

II. TOOL DESCRIPTION

The tool has been constructed to extract data that provide software metrics for software products and their analysis (as discussed in section V). A software metrics is defined as "The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products" [4].
Tool has been implemented in java 1.7 using NetBeans (NetBeans 8.1). It needs JDK/JRE environment for proper execution of its functionalities.

The tool operates in four basic phases. The first phase includes two steps, cloning the remote git repository and saving a copy of source code of two consecutive versions of a single project on the user's system. Cloning creates a local copy of git repository on the system of end-user, this copy has details of its own history and all the files of the project. Any type of change or modification that is done on cloned copy requires to be pushed to git repository. In second step, we save the source code of two versions of the open source on local machine, for tool to run on it.

The second phase implements the core functionalities of the tool. This phase collects the relative code churn and complexity of source code (as enlisted in table I).

TABLE I. RELATIVE MEASURES

Measure	**Description**	**Normalized Measure**
Added LoC	LoC added to the source code during observation period.	Added LoC/Total LoC
Deleted LoC	LoC deleted from the source code during observation period	Deleted LoC/Total LoC
Modified LoC	LoC modified in the source code during observation period	Modified LoC/Total LoC
Sum = (Added + deleted + modified) LoC	sum of added, deleted, modified	Sum/(Commits+ 1)
Complexity	The McCabe's complexity of code.	Complexity/Tot al LoC
Pre-release bugs	faults fixed before the release	Bugs/Total LoC

Code churn is measurement of code change taking place within a software unit over a period [2]. We analyse the code churn during the transition between first version to second version. Code churn measures the number of added Lines of code, deleted lines of code, modified lines of code, for each file, from first version to second version of the project. These measures have a varying range. Hence, there is a need of standardized range. For this purpose, relative code churn is computed. Relative code churn measures are normalized values of the various measures obtained during the churn process [2]. The Cyclomatic complexity is calculated by measuring the linearly independent path in the graph [6]. It measures quality and level of difficulty of source code at file level. Complexity is considered to be a strong predictor for code quality [5].

The third phase collects defect data of project. We have collected defect data of projects in two parts, namely pre-release bugs and post-release bugs. Pre-release bugs are the faults encountered before the release of the version in every file of source code and post-release bugs are the faults encountered after the release of the version of every file of source code. The pre-release bugs metric is normalized as ((number of pre-release bugs)/Total LoC) whereas, post-release bugs are stored in the form of binary variable, that is, a non-zero post-release bug count is stored as "Yes" and a zero post-release bug count is stored as a "No".

Number of pre-release bugs and post-release bugs have been calculated with the help of commits. A commit represents the changes merged to the open source git version control repository, it has been assumed that the pre-release bugs for the second version are the commits done to the version between release dates of first version and second version. Similarly, post-release bugs of the second version are the commits done to it after the release dates of second version till the release date of next version.

Fourth phase of tool is for computing descriptive statistics of the data obtained in previous phases. Tool summarizes the data by calculating mean, maximum and standard deviation (referred as sd in the rest of the paper) of each of the normalized measure. There are total six normalized measures: added/total LoC, deleted/total LoC, modified/total LoC, complexity/total LoC, sum/(commits+1) and pre-release bugs/total LoC. Therefore, a total of 18 descriptive characteristics are obtained for the project. Maximum-added, maximum-deleted, maximum-modified, maximum-total, maximum-complexity, maximum-prebug, Median-added, median-deleted, median-modified, median-total, median-complexity, median-prebug, sd-added, sd-deleted, sd-modified, sd-total, sd-complexity, sd-prebug.

TABLE II. STATISTICAL MEASURES

Statistical Measure	**Description**
Mean	Average of all the files for the metric.
Maximum	Maximum value amongst all files for the metric.
Standard Deviation	Deviation of each file from mean for the metric.

Currently, the tool analyses ".java" files in the source code, but it can be extended for other languages. The user interface is concise, familiar and responsive. There are separate interfaces for each phase.

III. WORKING OF TOOL

A. The algorithm is given below. In Fig. 1., workflow of tool is presented:

1) Copy the URL of GitHub repository of the open source project, to be cloned. Select any two consecutive versions of a project and save a copy of their source code on the machine.

2) Create a list of files that are common to both the versions. For each file in the list. Compare the second version file with first version file to: -

a) Calculate the added lines of code to the second version with respect to the first version.

b) Calculate the deleted lines of code to the second version with respect to the first version.

c) Calculate the modified lines of code to the second version with respect to the first version.

d) Compute the complexity of the source code of second version.

e) Calculate the sum of added LoC, deleted LoC and modified LoC, which is represented as sum.

3) To calculate the pre-release bugs of second version, the following dates are required:
D1: Release date of the first version.
D2: Release date of the second version.
To calculate the post-release bugs of current version the following dates are required:
D3: Release date of the next version with respect to the second version.

4) For each file in the list*:*

a) Find pre-release bugs by counting the number of commits done during D1 and D2.

b) Find post-release bugs by counting number of commits done during D2 and D3

5) Total commits = total commits done on the second version.

6) for each metric (added, deleted, modified, complexity , pre-release bug)
for each file:
metric = metric/ (total LoC).

7) Sum is normalized as:
sum = sum/ (commits+1).

8) The output generated is stored in CSV file format containing all the measures for each file. This is csv_normalized.

9) The next interface is of descriptive statistics, csv_normalized is input to it.

10) For each of the 6 metrics in csv_normalized, mean, maximum and standard deviation are calculated. 18 statistics are obtained.

11) Descriptive statistics are stored in a new CSV file, named, csv_statistics.

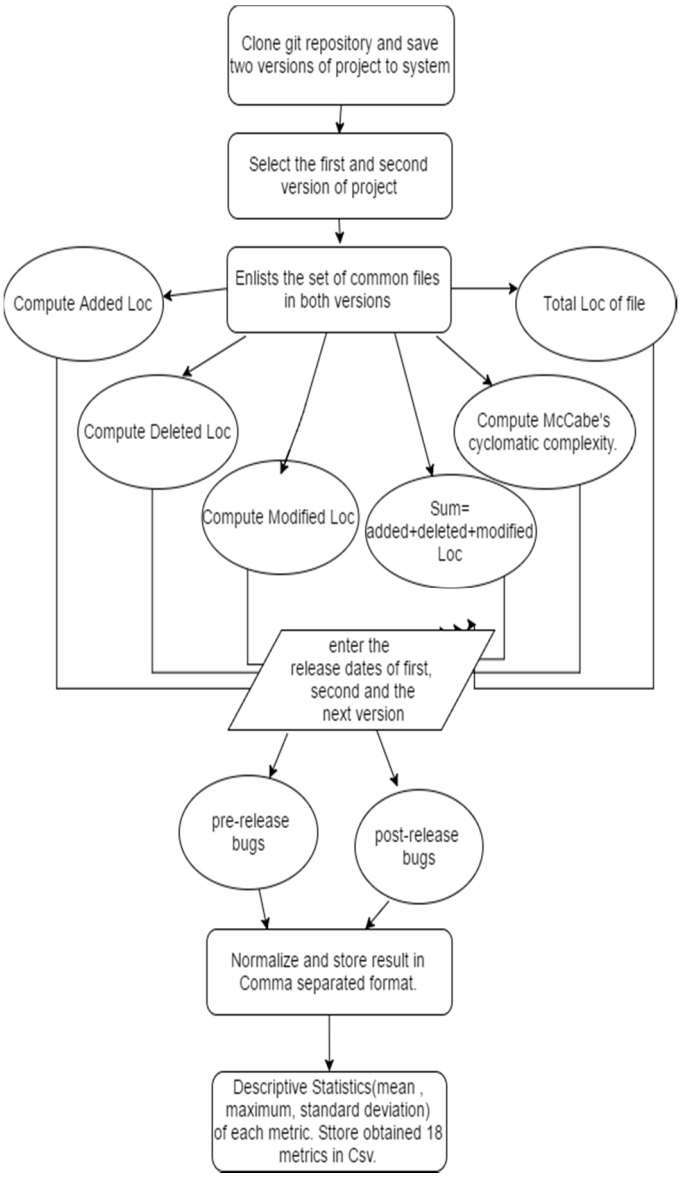

Fig. 1. The graphical description of working of tool.

B. Instances of the tool

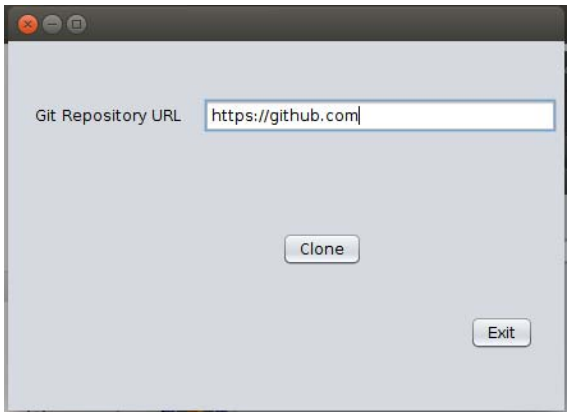

Fig. 2. Interface for cloning the git repository to the local machine.

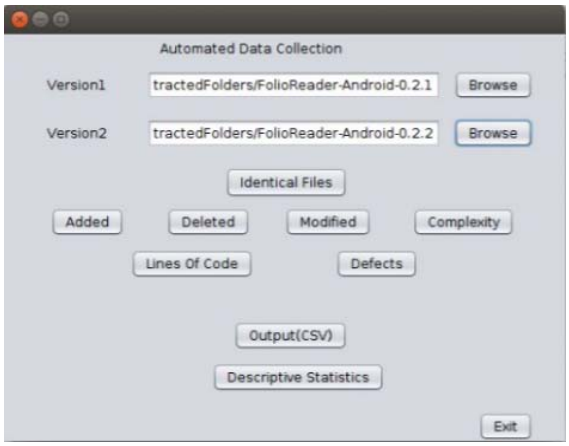

Fig. 3. Interface displayed at the initialization of the tool.

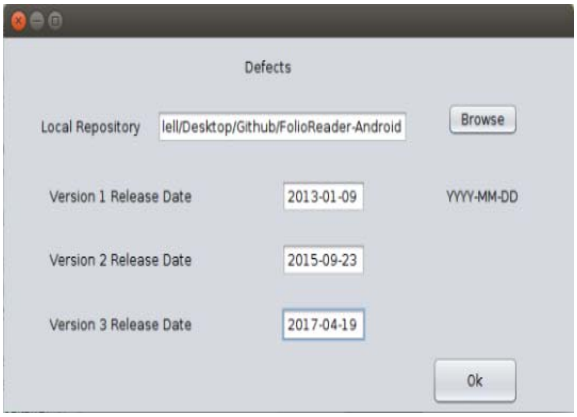

Fig. 4. Interface displayed on clicking Defects, to calculate bugs.

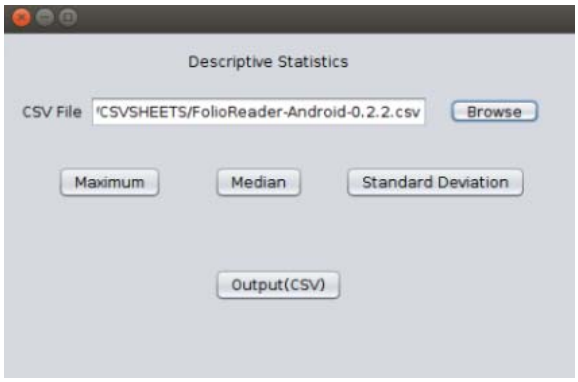

Fig. 5. Interface displayed on clicking Descriptive Statistics, to calculate the statistics data of the software measures.

Fig. 2,3,4,5 are the instances of tool's user interface for data collection. For calculating each metric, there is a separate button. On clicking each button, the respective metric is computed and stored.

The user should select the path to first version and second version of the same open source project. The browse button is for browsing to local repository saved on the user's machine. The buttons present in the UI are:

TABLE III. FUNCTIONALITY OF TOOL BUTTONS

Button Name	**Function of the button**
Identical Files	Enlists all the files that are common to both the versions
Added	Computes the number of lines added to second version with respect to first version
Deleted	Computes the number of lines deleted in second version with respect to first version
Modified	Computes the number of lines modified in second version with respect to first version
Complexity	Computes the Cyclomatic complexity of second version.
Lines of code	Gives the total number of lines in the source code of second version
Defects	Computes the pre-release and post-release bugs
Output(CSV)	This puts all the calculated metrics in a CSV file
Descriptive Statistics	This generates a CSV file containing maximum, median and standard deviation

IV. TOOL DEMONSTRATION

We operate our tool on open source projects. We have chosen Dagger for demonstration. The selected versions of Dagger are Dagger 1.0.1 and Dagger 1.0.2. Dagger is a fast dependency injector for Android and java [11]. Table IV shows a sample of the normalized code churn obtained for Dagger 1.0.2. Table V shows the Descriptive statistics of Dagger 1.0.2. These characteristics describe the development process of a project and hence can be used for further experimentation.

TABLE IV. SAMPLE OUTPUT ON DAGGER (ADDED(ADD), DELETED(DEL), MODIFIED(MOD), SUM(SUM), COMPLEXITY(COM), PRE-RELEASE BUGS (PRE), POST-RELEASE BUGS(POST))

ADD	DEL	MOD	SUM	COM	PRE	POST
0.16	0.05	0.09	1.80	0.04	0.04	NO
0.04	0.07	0.02	1.14	0	0.04	NO
0.12	0.02	0.12	2.83	0.14	0.02	YES
0.12	0.02	0.12	1.73	0.08	0.03	YES
0	0	0	1	0	0	NO
0	0	0	1	0.05	0	NO
0.16	0	0	1.09	0.16	0.04	NO
0	0	0	1	0.21	0	NO
0.04	0.02	0.02	1.09	0.02	0.04	YES
0	0	0	1	0.08	0	NO
0	0	0	1	0.14	0	NO
0	0	0	1	0.28	0.00	NO
0	0	0	1	0.11	0	NO
0.18	0.05	0.19	4.95	0.13	0.06	YES
0.02	0.04	0.01	1.26	0.11	0.00	YES
0	0	0	1	0.08	0	NO
0.01	0	0.01	1.14	0.15	0.01	YES
0.07	0.02	0.08	1.61	0.08	0.02	YES
0.02	0.02	0.02	1.07	0.15	0.02	NO
0.06	0.09	0.06	1.23	0.04	0.06	YES
0.16	0.13	0.16	3.45	0.07	0.06	YES
0.52	0.05	0	1.23	0.11	0.06	YES
0	0	0	1	0.14	0	NO
0	0	0	1	0.14	0	NO
0	0	0	1	0.14	0	NO
0	0	0	1	0.16	0	NO
0.19	0	0.00	2.11	0.07	0.00	NO
0	0.02	0	1.07	0.10	0.01	NO

TABLE V: DESCRIPTIVE STATISTICS OF DAGGER 1.0.2

S No	Metric	Maximum	Median	Standard Deviation
1	Added	0.407	0	0.066
2	Deleted	0.025	0	0.004
3	Modified	0.840	0	0.133
4	Sum	0.050	0	0.012
5	Complexity	1.040	0.016	2.180
6	Pre-release bugs	0.900	0	0.167

V. APPLICATIONS

The automation tool reports transition from first version to second version of a project. These collected code attributes can be used to predict defects of the future versions as well as for Cross Project Defect Prediction algorithms [1]. The tool can be used to collect data of open source, for analysis purposes and various prediction models.

VI. CONCLUSION

Automation has been achieved for optimizing the process of data collection as well as to increase data accuracy. Automation provides the speed and scale of data collection for projects. Even though the current scope is limited to open source projects, Tool can also be expanded for closed source projects too.

REFERENCES

[1] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process,” in Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2009, pp. 91–100.

[2] N. Nagappan, T. Ball, 2005. Use of Relative Code Churn Measures to Predict System Defect Density. Proc. of 27th International Conference on Software Engineering (St. Louis, MO, USA, May 15–21, 2005), ICSE ’05

[3] R. Malhotra, “A Defect Prediction Model for Open Source Software”, Proceedings of the World Congress on Engineering (2012) Vol II, London, U.K, 2012.

[4] P. Goodman, “Practical Implementation of software Metrics”, McGraw Hill Book Company, UK, 1993.

[5] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict

component failures," in *International Conference on Software Engineering*, 2006, pp. 452-461.

[6] T. McCabe, “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric,” NIST Special Publication, (1996) September, pp. 500-235.

[7] N. Alija, “Justification of Software Maintenance Costs”, International Journal of Advanced Research in Computer Science and Software Engineering 7(3), March- 2017, pp. 15-23.

[8] Zimmermann T., Nagappan N., Zeller A. (2008) “Predicting Bugs from History” . In: Software Evolution. Springer, Berlin, Heidelberg

[9] https://github.com

[10] http://www.locmetrics.com

[11] https://github.com/square/dagger