2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

Rediscovery Datasets: Connecting Duplicate Reports

Mefta Sadat
Dept. of Computer Science
Ryerson University, Toronto, Canada
mefta.sadat@ryerson.ca

Ayse Basar Bener
Dept. of Mechanical and Industrial Engineering
Ryerson University, Toronto, Canada
ayse.bener@ryerson.ca

Andriy V. Miranskyy
Dept. of Computer Science
Ryerson University, Toronto, Canada
avm@ryerson.ca

***Abstract*—The same defect can be rediscovered by multiple clients, causing unplanned outages and leading to reduced customer satisfaction. In the case of popular open source software, high volume of defects is reported on a regular basis. A large number of these reports are actually duplicates / rediscoveries of each other. Researchers have analyzed the factors related to the content of duplicate defect reports in the past. However, some of the other potentially important factors, such as the inter-relationships among duplicate defect reports, are not readily available in defect tracking systems such as Bugzilla. This information may speed up bug fixing, enable efficient triaging, improve customer profiles, etc.**

In this paper, we present three defect rediscovery datasets mined from Bugzilla. The datasets capture data for three groups of open source software projects: Apache, Eclipse, and KDE. The datasets contain information about approximately 914 thousands of defect reports over a period of 18 years (1999-2017) to capture the inter-relationships among duplicate defects. We believe that sharing these data with the community will help researchers and practitioners to better understand the nature of defect rediscovery and enhance the analysis of defect reports.

I. Introduction

Software engineering research community mines bug repositories to conduct research in various areas. For example, one can detect duplicate reports to speed up report triaging (deduplication) [11], [2] and identification of the root cause of failure [4], or to predict defect rediscoveries in order to proactively eliminate them before a customer finds [1], or to improve resource allocation to optimally manage the workforce [10], or to predict bug priority to improve planning [12], or to build customer profiles to improve quality assurance processes [9], or to automatically assign defect reports to owners to speed up time-to-fix of defects [3].

All of the above researchers leverage information about duplicate reports. There are already datasets that contain some information about duplicate reports (e.g., [7], [8], [6], [3]). However, to the best of our knowledge, no recent datasets containing information on inter-relations between duplicate reports are available. Thus, our **goal** is to create a collection of such datasets and share them with the community so that further research on duplicate defects can be performed. To achieve this goal, we mined bug repositories of three groups of open source software projects (Apache, Eclipse, and KDE), gathering information about duplicate defects, making it easy to identify relations between all of the duplicate defect reports. The datasets contain information about ≈ 914 thousands defects that have been reported in the last 15-18 years (depending on the project). The resulting datasets are located at http://doi.org/10.5281/zenodo.400614

Throughout the paper the following **terminology** (adopted from [4], [11], [9]) is used. Original defect *discovery* can be defined as the moment when a customer encounters a defect in the software for the very first time. Encounter is manifested by a problem or a fault in the software that leads to an undesired outcome or even a software *failure*. The customer then submits a *report* to a bug tracking system describing the problem.

If another customer encounters the same defect again, it is called defect *rediscovery*. This customer will then submit a new report to the bug tracking system. During report triaging, developers identify if a new report relates to a discovery of a new defect or to a rediscovery of an existing one. If it is a rediscovery, then developers typically mark the most recent report as a duplicate and link it to the original report (in some cases the link may be established incorrectly: "to err is human"). They then choose one of the linked reports as a *master report* and the rest of the reports associated with this particular failure will be deemed *duplicates* of the master report. Note that the report associated with the first discovery does not necessarily become a master report – sometimes developers choose a report of one of the rediscoveries as a master one. Given that there can be more than one rediscovery of the same defect, the network linking the original report with duplicate ones (which we call the *graph of rediscoveries*) may become complex. For example, Figure 1 shows the graph of rediscoveries for Eclipse report $\#4671$. Note that the master report in this case is not the original report.

Summing up original discovery and rediscovery count yields *total number or reports for a given failure*. If a given report was discovered in total once, then it means that it was never rediscovered; discovered twice – means that it was rediscovered once, and so on. In the case of Figure 1, report $\#4671$ was rediscovered 14 times. Thus, the total number of reports for a failure associated with report $\#4671$ is 15.

II. Methodology: extraction and transformation

For each group of the software projects, the set of attributes that we extracted from each report are given in Table I. We performed the following four extraction and transformation steps to obtain the attributes.

Step 1: Retrieval of report *id*s. For each of the software projects we selected, we mined its Bugzilla defect tracking

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.50

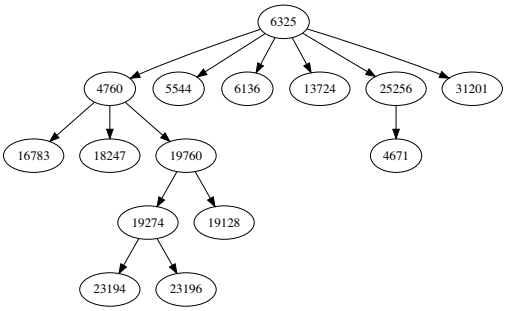


Fig. 1. Graph of rediscoveries of Eclipse report $\#4671$. Report B being duplicate of report A is denoted by $A \rightarrow B$. Note that even though report $\#4671$ is the original discovery, a later report $\#6325$ was chosen by developers as the master report. We can say that the failure associated with report $\#4671$ was discovered 15 times in total (counted as the total number of vertices/reports in the graph) and rediscovered 14 times (total number of duplicate reports).

TABLE I
EXTRACTED ATTRIBUTES

Attribute	Definition
id	The unique integer identifying a report.
product	The name of the software subsystem the report belongs to.
component	The name of the component the report is associated with.
reporter	The unique username of the person who opened the report.
bug_status	The current status of the report.
resolution	The current resolution of the report.
priority	Represents how quickly the defect should be fixed.
bug_severity	Defect's degree of impact on the whole system.
version	The version the defect was observed in.
short_desc	A short textual summary of the report.
opendate	The date when the report was opened.
dup_list	The list of ids of duplicates of a given report; if the report does not have any duplicates – the value is an empty string.
root_id	A derived attribute – the id of the root vertex of the graph of rediscoveries, which typically resembles the master report. If the report does not have any duplicates – the value is an empty string.
disc_id	A derived attribute – the id of the oldest defect report (i.e., the one that is opened first) in the graph of rediscoveries. If the defect does not have any duplicates – the value is an empty string.

system which numbers defect reports sequentially with an integer *id*, with the first *id* set to 1.

Given the sequential nature of the data, we query a given Bugzilla engine for reports opened within the last seven days (at the day of data gathering) and select the maximum *id* value, denoted by $I_{\max}$ returned by the engine. Thus, for a given engine the range of reports *id*s is set to $[1, I_{\max}]$.

Step 2: Data mining and extraction. The data were extracted using a custom-built web scraper. The input to the scraper was the range of *id*s to be mined - identified in the previous step. The scraper outputs all the attributes mentioned in Table I (except the two derived attributes) in *CSV* format (one line per report), saving intermediate results, as the extraction process takes several days to complete.

Step 3: Construction of the dataset. First, we aggregate all intermediate results for a given project in a single *CSV* file.

Second, we eliminate rows from the *CSV* file for which a report either does not exist or is not available. The former may happen because the report may get cancelled by a user before submission or may be erased by a bug tracker administrator. The latter may happen because we do not have sufficient permissions to access a given report. The former case cannot bias our dataset, as the data does not exist. However, the latter case may lead to bias, if the number of reports that we cannot access is large. We built a script that computed the number of *id*s associated with each case (by analysing error messages returned by the bug tracking engine). Details of our analysis are provided in Table I.

Step 4: Construction of derived attributes. In order to construct derived attributes, we built a directed graph G linking *id* with its duplicates using information stored in the *dup_list* attribute. Going back to example given in Figure 1, report $\#19274$ has two duplicates linked to it ($\#23194$ and $\#23196$), as per the *dup_list* attribute. Thus, we will add to the G two edges: $19274 \rightarrow 23194$ and $19274 \rightarrow 23196$. We repeat this process for each report in a given dataset. We then use Graphviz software [5] to identify all 'connected components' (in the graph theory sense of the term) in the G. The resulting connected components represent the graph of rediscoveries for each of the original defects. An example of such connected component is given in Figure 1.

We then analyze each graph of rediscoveries (connected component) and identify the root vertex (typically, this report is a master report) and the vertex associated with the *id* with the oldest *opendate*. The former becomes *root_id* value for each report associated with a given graph of rediscoveries; the latter value becomes $disc\_id$. For example, in case of Figure 1, the *root_id* value for all the reports will be set to 6325 and *disc_id* to 4671 (since, by design of the Bugzilla defect tracking system, the smaller the defect *id* – the older the defect). Then, we merge the original dataset with the derived attributes and store the resulting dataset in the *CSV*, *SQL*, and *Neo4j* formats.

III. ANALYSIS OF THE DATASET

The summary statistics of the datasets are given in Table II. The number of reports that we gathered (column 'Total accessible reports count') ranges from ≈ 44 thousands for Apache to ≈ 504 thousands for Eclipse. The reports were opened between years 1999 and 2017.

As discussed in Section II, we could not access some of the reports. The percentage of such reports (shown in column 'Inaccessible reports count') is small: 0.002% for Apache, 0.1% for Eclipse, and 1.3% for KDE. These reports also lead to 1, 79, and 33 inaccessible edges in G for Apache, Eclipse, and KDE, respectively. Thus, these missing observations should not bias the datasets significantly and can be ignored.

To gather information about original discoveries and rediscoveries of reports, as discussed in Section II, we analysed graphs of rediscoveries (similar to the one shown in Figure 1).

TABLE II
SUMMARY STATISTICS.

Project name	Total accessible reports count	Inaccessible reports count	Rediscoveries count	Distinct *disc_id* count	Min report *opendate* (YYYY-MM-DD)	Max report *opendate* (YYYY-MM-DD)	Max number of rediscover-ies	Distinct *product*s count	Distinct *productI componen*ts count	Non-rediscovered reports (% of total)
Apache	44,049	1	3,616	2,416	2000-08-26	2017-02-10	19	35	350	86
Eclipse	503,935	560	52,499	31,811	2001-10-10	2017-02-07	128	232	1,486	83
KDE	365,893	4,818	82,359	26,114	1999-01-21	2017-02-13	405	584	2,054	70

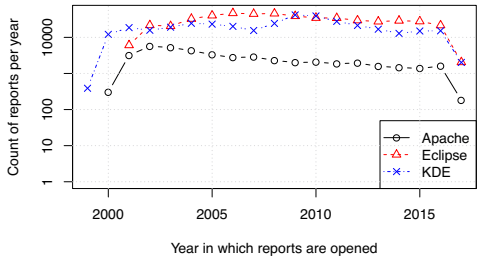
Such graphs can become fairly large: based on Table II, the maximum number of rediscoveries of an original report (per graph of rediscoveries) ranges from 19 for Apache to 405 for KDE. Most graphs are acyclical, with the exception of one cyclical graph in Eclipse and four in KDE datasets.

The percentage of the original reports that were rediscovered at least once ranges from 5% (2416/44049) for Apache to 7% (26114/365893) for KDE. The distributions of the total number of reports (obtained by combining rediscovery and original defect count, as discussed in Section I) for a given failure are given in Figure 4. The distributions are heavy-tailed as evident from the linear structure of the data plotted on the log-log scale. The number of reports per year changes, as seen in Figure 2a. Magnitude-wise, the number of reports ranges from thousands for Apache to tens of thousands for Eclipse and KDE (with the exception of the first and last reporting year for each project).

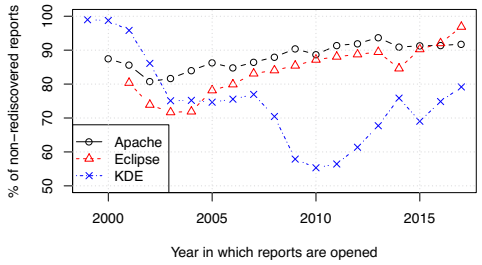
Overall, percentage of reports that are not rediscovered ranges between 70% for KDE and 86% for Apache. However, these values change from year to year, as shown in Figure 2b. This figure may suggest that for the last seven years percentage of non-rediscovered reports grows up (albeit non-monotonically). For example, for defects opened in 2016, the percentage of non-rediscovered defects ranges from 75% for KDE to 92% for Eclipse (compare these numbers with the average values of 70% and 86%, respectively).

However, in the future, users may encounter and report some of the defects discussed in these non-rediscovered reports. This will lead to reduction of the number of non-rediscovered reports opened in previous years. To confirm this conjecture, we plot the distribution of time intervals between the opening dates of the original discovery and the latest rediscovery, shown in Figure 3. The figure suggests that some reports get rediscovered years after the original discovery. Even for the graph of rediscoveries shown in Figure 1, the time interval between open dates of the original report #4671 and its latest rediscovery #31201 was ≈ 1.3 years.

The number of *product*s per project ranges from 35 for Apache to 584 for KDE; the number of *productIcomponent* tuples per project – from 350 for Apache to 2054 for KDE. The percentage of reports that are not rediscovered per product-component is given in Figure 5. The median percentage ranges between 84% for KDE to 96% for Eclipse. However, there are outliers with low percentage of non-rediscovered defects, suggesting that different components may exhibit different



(a) Number of reports per year.



(b) Percent of reports that have not been (yet) rediscovered.

Fig. 2. Per-year analysis. The data are current as of February 2017, thus the dataset for year 2017 is incomplete, hence the "dip" in reports for year 2017. By construction, zero observations for a given year are not shown.

behaviour. Therefore, various *productIcomponent*s may be studied independently.

IV. Relevance of the Dataset

Based on the analysis of the datasets given in Section III, we believe that these datasets provide a rich ground for researchers interested in analyzing defects for various purposes discussed in Section I. For example, they can be used for cross-product verification of the models built by researchers to speed up triaging and identification of root causes of failures, predict defect rediscoveries, or assign owners to reports. The data are provided in the *CSV*, *SQL*, and *Neo4j* formats, enabling easy investigation of the datasets.

V. Challenges and Limitations

We do not have access to a number of reports, which may bias our dataset (as discussed in Section III). However, given

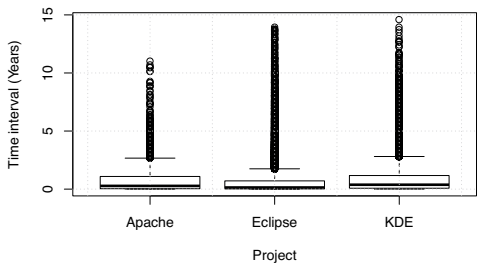


Fig. 3. Distributions of time intervals between the original discovery and the latest rediscovery for a given graph of rediscoveries.

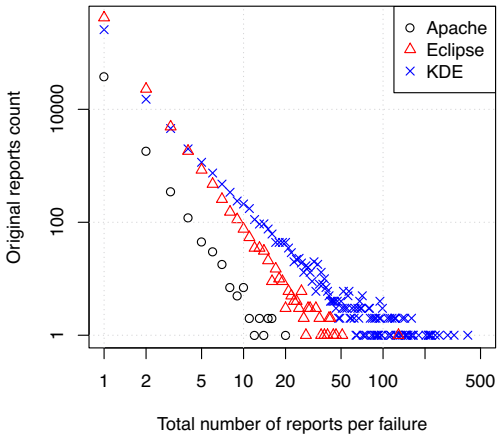


Fig. 4. Count of the total number or reports for a given failure vs. count of original reports. If a given failure was reported once, then it means that it was never rediscovered; reported twice – means that it was rediscovered once, and so on (see Section I for details). For example, Apache dataset has 38017 reports that were never rediscovered (i.e., discovered once) and 1825 reports that were rediscovered once (i.e., discovered twice).

that the percentage of such reports is low (0.002% for Apache, 0.1% for kclipse, and 1.3% for KDk), the dataset should not be affected significantly.

Our list of attributes does not cover all of the defect reports' attributes available in Bugzilla. However, our dataset helps researchers to narrow down a set of the defect reports that have to be mined to gather such additional attributes (e.g., comments associated with a given defect report). For example, if researchers are interested in the analysis of duplicate defects of kclipse dataset, they can focus on mining just 17% $((52499 + 31811)/503935)$ of the reports (as shown in Table II), with report *id*s being readily available in our datasets. Thus, this would allow them to save time and computational resources on the costly extraction and transformation process.

In addition, some of the reports that are currently non-rediscovered may be rediscovered in the future (as discussed

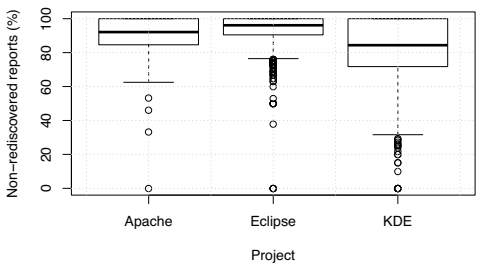


Fig. 5. Distribution of non-rediscovered reports per *product-component*.

in Section III). This has to be taken into consideration during data analysis.

VI. Summary

In this paper, we present datasets collected from three groups of projects (Apache, kclipse, and KDk), aimed at capturing information associated with duplicate / rediscovered defects. We describe the schema of the datasets, extraction and transformation process, and present analysis of the datasets. We believe that these datasets will aid researchers and practitioners in gathering insight into usage of duplicate reports in various areas of software engineering.

AckNOWLkDGMkNT

This work is partially supported by NSkRC grants 402003-2012 and RGPIN-2015-06075.

RkFkRkNCkS

[1] k. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[2] A. Alipour, A. Hindle, and k. Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *Proc. of the 10th Working Conf. on Mining Softw. Rep.*, pages 183–192, 2013.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of the 28th Int. Conference on Softw. Eng.*, pages 361–370, 2006.

[4] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate Bug Reports Considered Harmful... Really? In *Proc. Int. Conf. on Softw. Maintenance*, pages 337–345, 2008.

[5] k. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[6] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proc. of the Int. working conf. on Mining soft. rep.*, pages 145–148, 2008.

[7] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proc. of the 22 Int. Conf. on Automated Softw. Eng.*, pages 34–43, 2007.

[8] A. Lamkanfi, J. Pérez, and S. Demeyer. The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information. In *Proc. of the 10th Working Conf. on Mining Softw. Rep.*, pages 203–206, 2013.

[9] A. V. Miranskyy, k. Cialini, and D. Godwin. Selection of customers for operational and usage profiling. In *Proc. of the 2nd Int. Workshop on Testing Database Systems*, pages 7:1–7:6, 2009.

[10] A. V. Miranskyy, M. Davison, and M. Reesor. Metrics of risk associated with defects rediscovery. *arXiv preprint arXiv:1107.4016*, 2011.

[11] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. of the 29th Int. Conf. on Softw. Eng.*, pages 499–510, 2007.

[12] Y. Tian, D. Lo, and C. Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *Proc. of Int. Conf. on Softw. Maintenance*, pages 200–209, 2013.