Generating Duplicate Bug Datasets

Alina Lazar, Sarah Ritchey, Bonita Sharif
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio USA 44555
alazar@ysu.edu, sritchey@student.ysu.edu, bsharif@ysu.edu

ABSTRACT
Automatic identification of duplicate bug reports is an important research problem in the mining software repositories field. This paper presents a collection of bug datasets collected, cleaned and preprocessed for the duplicate bug report identification problem. The datasets were extracted from open-source systems that use Bugzilla as their bug tracking component. The systems used are Eclipse, Open Office, NetBeans and Mozilla. For each dataset, we store the initial data and the cleaned data in separate collections in a mongoDB document-oriented database. For each dataset, in addition to the bug data collections downloaded from bug repositories, the database includes a set of all pairs of duplicate bugs together with randomly selected pairs of non-duplicate bugs. Such a dataset is useful as input for classification models and forms a good base to support replications and comparisons by other researchers. We used a subset of this data to predict duplicate bug reports but the same data set may also be used to predict bug priorities and severity.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Management**]: Productivity H.2.4 [**Systems**]: Textual databases; H.2.8 [**Database Applications**]: Data Mining.

General Terms
Experimentation, Standardization, Measurement.

Keywords
Duplicate Bug Reports, Duplicate Bug Pairs, Data Repositories.

1. INTRODUCTION
Open source software projects usually have a bug or issue tracking system associated with them. These projects have been around for more than a decade, thus the total number of bugs reported has grown significantly. Many times, the same bug is reported by different people, using different words and sometimes in different contexts. Triaging bugs is usually a manual process and time intensive in itself. Having duplicate bugs makes the problem even worse and is a waste of the triager's time. In the past few years, researchers propose several methods to automate the identification of duplicate bug reports.

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597128

Few studies use proprietary bug data from software systems like Sony Ericson [7] and Blackberry [2]. These datasets are not widely available which means that the reported results are not reproducible. The rest of the published research presents results obtained on open-source software systems such as Android [1], (Eclipse, Mozilla, Open Office) [3, 10, 11] and NetBeans [9]. The large open-source projects mentioned above have been in existence for more than ten years, and have collected over a hundred thousand bugs. It is much easier to replicate studies performed on these systems, however not all research studies use the exact same data. Most of the time methods are compared using different subsets which is not a fair comparison due to lack of consistency between datasets. We explain some below.

Only Sun et al. [10] and Nguyen et al. [6] use the same three datasets in their evaluation of the duplicate bug detection problem. Also, there is no systematic way in which different researchers generate duplicate pairs of bugs. Sun et al. do not explicitly state how the pairs were generated and the dataset used in this case was much smaller. It is also not indicated if both open and closed bugs were used in the analysis. Even if researchers use the same data (e.g., Eclipse 2008 data set from [10]), if they were downloaded at different times the data may be different because some open bugs may close between downloads. Other duplicate bug studies [1, 10] reported results on subsets of the Eclipse, Open Office and Mozilla datasets. The largest dataset of 213,000 bugs, extracted from Eclipse was available online for the MSR (Mining Software Repositories) 2008 challenge and it was used for the bug duplication problem by Sureka et al. [11]. For the same dataset, Lerch et al. [5] reports findings based on only 211,843 bugs and Sun et al. [10] reports findings based on 209,058 bug reports. In addition, no research, so far uses the entire set of bugs from any of the above mentioned systems to address a problem with detecting duplicate bug reports.

In this data paper we address the above problems by presenting datasets for duplicate bug identification in open-source systems to help with the duplicate bug report identification problem. We extracted and processed *all* the bug reports from the following systems: Eclipse, Open Office, Mozilla and NetBeans. In addition to the bugs, we also generate a set of all pairs of duplicates. This data can be used to compute textual and categorical features to be used by classification algorithms. Given this dataset, different researchers can compare different duplicate bug detection algorithms with respect to a common base which supports reliable and sound comparisons between replications.

2. DATA DESCRIPTION
The datasets presented in this paper contain bug information downloaded from the Bugzilla websites of the following open source products: Eclipse, Open Office, Mozilla and NetBeans. When a user submits a bug to the Bugzilla system, some attributes are automatically created, while others are entered by the submitter. An example of a bug from Eclipse in XML is shown in Figure 1.

In Bugzilla, the data is stored in a MySQL dataset. The main table where the bugs are stored contains around twenty fields. Not all these fields are necessarily used for duplicate bug identification. The following attributes where selected to be downloaded for this problem: bug_id, short_desc, long_desc, product, component, status, resolution, priority, version, open date and dup_id. The *dup_id* contains a reference to a bug that the current bug is considered to be a duplicate of. The attributes have the standard meaning as defined by Bugzilla (*http://www.bugzilla.org/docs/2.16/html/how.html*).

```
<bugzilla version="4.4.1">
  <bug>
    <bug_id>119285</bug_id>
    <creation_ts>2005-12-05 13:38:00
    </creation_ts>
    <short_desc>Replace with uncommitted
      changes dialog shows same class twice
    </short desc>
    <product>Platform</product>
    <component>Team</component>
    <version>3.2</version>
    <rep_platform>PC</rep_platform>
    <op_sys>Windows 2000</op_sys>
    <bug_status>RESOLVED</bug_status>
    <resolution>DUPLICATE</resolution>
    <dup_id>119056</dup_id>
    <long_desc> … </long_desc>
    ...
  </bug>
</bugzilla>
```

Figure 1. Eclipse bug information in XML format.

The short description and the long description included in a bug report are the two fields containing natural language text. The assumption is that a pair of duplicate bugs will have some similarities between the two text fields. It is also necessary that they belong to the same product and component.

3. GENERATING THE DATASETS

The main steps involve web crawling, cleaning, generating groups, and generating pairs. We describe this next.

3.1 Web Crawling

The web crawling Python framework Scrapy [8] was used to collect the data. For each of the four systems we looped through all the bugs submitted before January 1 2014, using bug identification numbers. After a bug is downloaded, its information is sent into an item pipeline for initial processing. During this step we validate the bug data by checking if the bug id and the short title for the current bug are not null. If one of these two fields is empty we drop that bug, otherwise the process moves to the next step. Next, each bug is inserted into a mongoDB collection, using the python module mongoDB pipeline for Scrapy.

3.2 Cleaning and Preprocessing

After the data was downloaded, we clean and preprocess it specifically to the problem of detecting duplicate bug reports. Let's assume *bugA* is a duplicate bug (with a non-empty dup_id), having resolution set to "DUPLICATE". For each such duplicate bug, the *bug_id* listed in the *dup_id* field is checked against the entire dataset. Usually, for most bugs there is a *bugB* such that *bugB.bug_id* equals to *bugA.dup_id*.

We identify all bugs that do not have a bug in the dataset referred by their *dup_id* field to be non-duplicates and so the resolution of such bugs is changed from "DUPLICATE" to "NONDUPLICATE". There were other bugs with resolution set to "DUPLICATE", but with an empty *dup_id* field. In such cases, we also change the resolution from "DUPLICATE" to "NONDUPLICATE".

Table 1. Datasets and number of bugs

Dataset	Period	Last Bug ID	Number of Bugs
Eclipse	10/10/01 -12/31/14	424,786	423,559
Open Office	10/16/00 -12/31/14	123,941	123,865
Mozilla	04/07/98 -12/31/14	955,895	890,419
NetBeans	08/21/98 -12/31/14	239,895	237,142

We also needed to address the *cycle* problem [3]. This happens when for a pair of duplicate bugs (*bug1, bug2*), *bug1* contains a *dup_id* reference to *bug2* and *bug2* has a *dup_id* reference to *bug1*. In such cases we identified the earliest submitted bug and set that bug as the master bug, by removing its *dup_id*. A master bug is a bug that all other duplicate bugs refer to.

Next, we removed all the "OPEN" bugs from each dataset. To fully comprehend this step, it is necessary to study the lifecycle of a bug report. A bug is considered "OPEN" from the time of its submission until a resolution is decided. An open bug's *bug_status* field can take several values as listed in Table 2, but the resolution field is empty. Once a bug is "CLOSED" its *bug_status* can take the two values listed in Table 2 and its resolution can become one of the six values in Table 2.

Table 2. Bug status and resolution possible values

OPEN Bugs – status	UNCONFIRMED, NEW, ASSIGNED, REOPENED, READY
CLOSED Bugs – status	RESOLVED, VERIFIED
CLOSED Bugs - resolution	FIXED, INVALID, WONTFIX, DUPLICATE, WORKSFORME, INCOMPLETE

To be able to successfully discriminate between duplicate and non-duplicate bugs, we need to know the resolution value of the bug. That means only "CLOSED" bugs can be used for training and testing. "OPEN" bugs may be used later on for testing. One option could be the integration of detecting duplicates into Bugzilla to automatically report "The bug report you just submitted is a duplicate of bug X". After removing the "OPEN" bugs from the dataset, the number of remaining bugs in each of the datasets is reported in Figure 2. In a real life setting, when a developer or triager identifies a duplicate bug, they change the status of the bug to "RESOLVED" and set the resolution to "DUPLICATE". At the same time, a *bug_id* reference is added in the *dup_id* field of the bug. The resolution field helps us identify all the duplicate bugs.

3.3 Identifying Groups and Pairs of Bugs

It can be seen from Figure 2 that the duplicate bugs represent between 12.67% and 23% out of the total bugs for a system. There are definitely less duplicate bugs compared to non-duplicates. Therefore, the next goal is to identify all possible pairs of duplicate bugs for the training stage in a classification approach to duplicate bug identification.

First, we organized the bugs in groups. All bugs representing the same defect (bug) were included into the same group, based

on their *dup_id*. One bug per group is designated as the master bug (the bug that all other duplicate bugs refer to). As previously reported [3], the master bug is not always the earliest bug report, but the report which is closest to getting fixed.

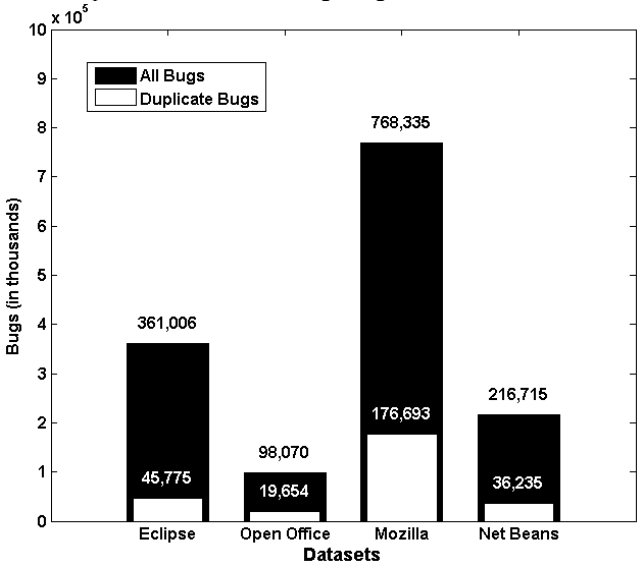

Figure 2. Number of bugs and duplicates

An example is shown in Table 3; there are four bugs that are duplicates in the group for master bug 2518. In this particular problem, it is not important which bug is master because they all fall into the same group of duplicate bugs. In a group, the master bug is the only one with no *dup_id*.

Table 3. Duplicate bugs for bug_id 2518

Bug ID	Summary	dup_id
2518	should be able to rename by clicking twice (1GFFEKT)	
2250	Selecting same resource again should rename (1GEAVD1)	2518
9605	renaming files	2518
156530	rename files and folders in navigator on slow double click	2518

The number of total bug groups for each dataset is listed in Table 4. Most groups just contain one other duplicate bug, as can be seen in Figure 3, represented by the first bar. The group size goes up to 51 for the Eclipse dataset and up to 592 for Mozilla. That means Mozilla has 592 bug reports describing the same defect. Once the groups were found, they are used to generate all possible combinations of duplicate bug pairs. The number of duplicate bugs makes just up to 23% of the total number of bugs. To improve the chances of getting good results when running a machine learning algorithm (during classification), we generate all possible pairs of duplicate bugs, not only the pairs containing the master bug. For the duplicate group shown in Table 3, six unique pairs of bugs are generated: (2518, 2250), (2518, 9605), (2518, 156530), (2250, 9605), (2250, 156530), (9605, 156530).

We wrote several tools to achieve the above steps. First, a Scrapy script was used to download the data into mongoDB. Next, a Python script was run to perform all the cleaning steps described. Finally, another Python script generated all the groups of duplicates and saved them in a list. The duplicate pairs were then computed as the combination of the elements in each group.

Table 4. Number of groups and pairs

Dataset	Groups	Max Size	Duplicate Pairs
Eclipse	25,455	51	110,137
Open Office	7,257	90	111,121
Mozilla	66,564	592	4,970,385
NetBeans	19,779	56	148,343

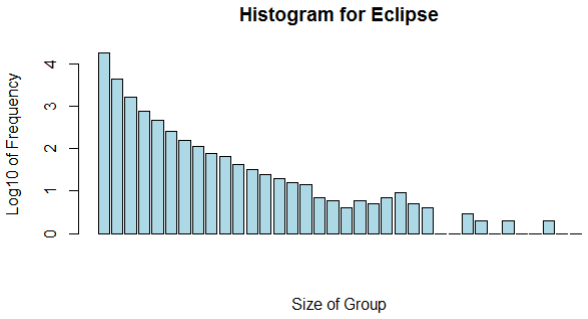

Figure 3. Histogram of group size in Eclipse

In addition to the duplicate pairs, four times as many non-duplicate pairs were randomly generated for each dataset. In our case, a non-duplicate pair refers to two bugs where the *dup_id* is empty. We could also pair bugs between groups (where two non-duplicates would have a non-empty different dup_id) to generate non-duplicates. We chose to use the former approach to generate a random sample of non-duplicates.

4. RELEVANCE OF THE DATASET

There are several research directions to solve the duplicate bug identification problem. Initial methods borrowed from information retrieval (IR) were based strictly on textual similarities. Only the textual fields such as summary and long description were used. Often the problem was modeled as a search problem, based on the top-k retrieval problem. Given a bug from the test set, the IR algorithm computes a similarity score between this bug and all the other bugs from the training dataset, orders the scores, and presents the top-k similar bugs to the user. The system is evaluated based on the percentage of actual duplicate bugs that were retrieved in the top-k list. The idea behind this approach was to restrict the search space for the human performing the bug triage from thousand bugs to less than one hundred. To perform this type of search a similarity matrix needs to be computed and stored. For open-source systems with large amount of bugs, it would not be possible to store this matrix in memory for further processing and saving parts of the matrix to the disk would be necessary. That is the main reason why most papers just report results on smaller datasets.

To deal with the above described problem, Sun et al. [10] proposed a new way to build the feature dataset. Instead of calculating the entire similarity matrix, they just calculated the similarities between all the duplicate pairs of bugs and randomly selected a subset of non-duplicate bugs. They also used a ranking machine learning approach based on Support Vector Machine (SVM) to identify the top-k possible duplicates for a given bug.

Once the dataset that includes pairs of bugs for duplicates and non-duplicates was built, the problem of identifying duplicate bugs can be reformulated as a simple binary classification problem instead of a supervised ranking problem. The datasets we

are proposing in this paper can be successfully used to run binary classification in order to identify duplicate bugs. We ran this method on a subset of the data presented here and the results outperformed previous approaches [4]. The classification algorithms were able to identify all duplicate pairs, providing a 100% recall, while predicting few non-duplicate bugs as duplicates.

We provide the datasets as a mongoDB dump. Several programming languages provide API interfaces to mongoDB, such as C, C++, Java or Python. Using the mongoimport utility the data can be accessed in chunks as needed to first generate features and then run classification algorithms such as *libSVM*. The data follows the same schema as Bugzilla and is available online[1].

5. CHALLENGES AND LIMITATIONS

5.1 Challenges

Initially, after downloading the bug reports in Scrapy we used the item pipeline and saved the datasets into JSON files with the goal of importing these files into mongoDB. The size of the downloaded JSON files was between 125 MB and 234 MB. However, for *mongoimport*, the limit on document size is 16 MB. This means additional steps were needed to be performed. First, the datasets must be divided into less than 16 MB chunks and second, each chunk would be uploaded using the *mongoimport* command. This procedure was greatly simplified by using the mongoDB pipeline for Scrapy. This is a Python module that allows saving Scrapy data directly into mongoDB collections. Once we switched to the mongoDB pipeline in Scrapy, the download process went much quicker and smoothly with only the Mozilla set running overnight to download all the data.

It took approximately 60 work hours to write the Scrapy programs and download the data and 40 hours to clean, preprocess and generate the bug pairs. Testing and documentation is included in these times. To derive new datasets using the same procedure described in this paper, one will require half the time stated above.

5.2 Limitations

Each dataset was initially divided based on the *bug_status* attribute into two sets, one containing the closed bugs and one containing the open bugs. We used the values of the *bug_status* at the time we downloaded the data, in January 2014. Since the *bug_status* and the *resolution* of each bug changes frequently, the corresponding commit data associated with each bug is needed to determine which bugs were open and which bugs were closed before 1/1/2014. The datasets we present do not contain any commit data.

Our datasets include only attributes most often used to predict duplicate bugs. Other bug attributes (besides the ones shown in Figure 1) available in Bugzilla were not included. We also did not include comments and attachments in the datasets. This is being done as part of our ongoing work. In spite of these limitations, we believe that the most important and interesting attributes were selected. Researchers planning to use this dataset to detect duplicate bugs may want to choose a subset or all of the attributes for their duplicate bug detection task. Our dataset is also only limited to bugs that were CLOSED since these have completed their bug life cycle.

[1] *http://www.csis.ysu.edu/~alazar/msr14data*

6. SUMMARY

The ability to correctly identify bug duplicates greatly helps with the bug triage process. Several researchers address this problem but most of the time use non-overlapping datasets to compare their results. Even if overlapping datasets were used, the time of download might affect the number of bugs downloaded because some bugs might have been resolved between any two downloads. We have made four datasets available to researchers specifically for the duplicate bug detection problem. These datasets contain all bugs ever submitted since the project's inception until the end of 2013. In addition to the bugs, all pairs of duplicate bugs are also provided along with a larger randomized set of non-duplicate bugs to be able to discriminate between duplicates and non-duplicates. We believe this dataset will serve as a benchmark for large-scale duplicate bug detection. In addition, this data could also be used to predict bug severity and bug priorities.

7. REFERENCES

[1] Alipour, A., Hindle, A. and Stroulia, E. 2013. A contextual approach towards more accurate duplicate bug report detection. *Proceedings of the Tenth International Workshop on Mining Software Repositories* (2013), 183–192.

[2] Amoui, M., Kaushik, N., Al-Dabbagh, A., Tahvildari, L., Li, S. and Liu, W. 2013. Search-based duplicate defect detection: an industrial experience. *Proceedings of the Tenth International Workshop on Mining Software Repositories* (2013), 173–182.

[3] Bettenburg, N., Premraj, R., Zimmermann, T. and Kim, S. 2008. Duplicate bug reports considered harmful ... really? *IEEE International Conference on Software Maintenance, 2008. ICSM 2008* (2008), 337–345.

[4] Lazar, A., Ritchey, S. and Sharif, B. 2014. Improving the Accuracy of Duplicate Bug Report Detection Using Textual Similarity Measures. *11th Working Conference on Mining Software Repositories* (2014), 4 pages to appear.

[5] Lerch, J. and Mezini, M. 2013. Finding Duplicates of Your Yet Unwritten Bug Report. *2013 17th European CSMR* (2013), 69–78.

[6] Nguyen, A.T., Nguyen, T.T., Nguyen, T.N., Lo, D. and Sun, C. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2012), 70–79.

[7] Runeson, P., Alexandersson, M. and Nyholm, O. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. *29th, ICSE* (2007), 499–510.

[8] Scrapy: An open source web scraping framework for Python: *http://scrapy.org/*. Accessed: 2014-02-21.

[9] Serrano Zanetti, M., Scholtes, I., Tessone, C.J. and Schweitzer, F. 2013. Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities. *eprint arXiv:1302.6764*. (Feb. 2013).

[10] Sun, C., Lo, D., Khoo, S.-C. and Jiang, J. 2011. Towards more accurate retrieval of duplicate bug reports. *Proceedings of the 2011 26th ASE* (Washington, DC, 2011), 253–262.

[11] Sureka, A. and Jalote, P. 2010. Detecting Duplicate Bug Report Using Character N-Gram-Based Features. *Software Engineering Conference (APSEC), 2010 17th Asia Pacific* (2010), 366–374.