A Public Bug Database of GitHub Projects and Its Application in Bug Prediction

Conference Paper · July 2016

DOI: 10.1007/978-3-319-42089-9_44

3 authors:

Zoltan Toth
University of Szeged
11 PUBLICATIONS **48** CITATIONS
SEE PROFILE

Péter Gyimesi
University of Szeged
10 PUBLICATIONS **41** CITATIONS
SEE PROFILE

Rudolf Ferenc
University of Szeged
119 PUBLICATIONS **2,758** CITATIONS
SEE PROFILE

A Public Bug Database of GitHub Projects and its Application in Bug Prediction

Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc

Department of Software Engineering, University of Szeged, Hungary
zizo@inf.u-szeged.hu, pgyimesi@inf.u-szeged.hu, ferenc@inf.u-szeged.hu

Abstract. Detecting defects in software systems is an evergreen topic, since there is no real world software without bugs. Many different bug locating algorithms have been presented recently that can help to detect hidden and newly occurred bugs in software. Papers trying to predict the faulty source code elements or code segments in the system always use experience from the past. In most of the cases these studies construct a database for their own purposes and do not make the gathered data publicly available. Public datasets are rare; however, a well constructed dataset could serve as a benchmark test input. Furthermore, open-source software development is rapidly increasing that also gives an opportunity to work with public data.
In this study we selected 15 Java projects from GitHub to construct a public bug database from. We matched the already known and fixed bugs with the corresponding source code elements (classes and files) and calculated a wide set of product metrics on these elements. After creating the desired bug database, we investigated whether the built database is usable for bug prediction. We used 13 machine learning algorithms to address this research question and finally we achieved F-measure values between 0.7 and 0.8. Beside the F-measure values we calculated the bug coverage ratio on every project for every machine learning algorithm. We obtained very high and promising bug coverage values (up to 100%).

Keywords: bug prediction, bug database

1 Introduction

Software systems are likely to fail occasionally that is obviously unwanted both for the end users and for the software developers. Keeping the software quality at high-level is more important than ever, since customers define the reputation of the used subject system. Open-source software development paved its way, and has become a corner stone in the domain of evaluating research ideas and techniques dealing with computer science [19]. These publicly available systems gather a huge amount of historical data stored for example in version control systems or bug tracking systems. Researches have been using the opportunity given by these public information sets for a long time to prove the power of their approaches [1], [14], [24]. In spite of this fact, only a few publicly available

bug databases are presented to take role as a basis for further investigations. Many authors do not make the corpus used in their studies public, thus the experiments are not repeatable [12].

Our study tries to endorse the use of public databases for addressing different research questions such as bug prediction related ones by showing the power of our automatically generated bug database in bug prediction domain. We have developed a toolchain that automatically gathers different information about publicly available projects to build a bug database. We selected 15 Java projects from different domains to ensure the generality of the constructed database. The characteristics of these open-source projects were extracted from **GitHub**[1] that hosts millions of projects and using a static source code analyzer tool called **SourceMeter**.[2] We analyzed 15 projects with more than 3.5 million lines of code, and more than 114 thousand of commits in total. From the analyzed commit set we detected almost 6 thousand commits that referenced at least one bug (inducing a bug fix intention) according to the SZZ algorithm [22]. We used release versions of the systems and created bug databases for six-months-long intervals approximately.

To show the usefulness of the contained information we experimented with 13 machine learning algorithms and achieved quiet good results. For class level, the best algorithms resulted in higher than 0.7 F-measure values. For file level we achieved similar values, however a little lower ones. Almost full bug coverage can be reached by using these models by tagging only 30% of the source code elements as buggy. We defined two research questions, which are the following:

RQ 1: Is the constructed database usable for bug prediction? Which algorithms or algorithm families perform the best in bug prediction?
RQ 2: Which machine learning algorithms or algorithm families perform the best in bug coverage?

The remainder of the paper is organized as follows. Section 2 enumerates the most important research papers dealing with public and private bug databases, and bug prediction techniques. In Section 3, we propose our approach and show how the database is constructed, and what kind of data entries are stored in the dataset. Section 5 presents the power of the constructed database by evaluating different results of the applied machine learning algorithms. Finally, we summarize and conclude the paper.

2 Related Work

Publishing databases as public resources for the scientific community is not a new idea [24], [13]. Many papers have dealt with bug databases and used some kind of bug prediction approaches as demonstrations [9]. Notwithstanding the numerous studies dealing with bug prediction, the number of publicly available

[1] https://github.com
[2] https://www.sourcemeter.com

bug databases are incredibly low and neglected. Researchers often use a database created for their own purposes but these datasets are not published for the community.

Many research studies deal with bug prediction and they use a database created for specific purposes. We tried to create a database that is publicly available and general enough to test different bug prediction methods [3],[20],[17],[15]. We gathered a wide range of software product metrics to characterize the known bugs, amongst others the classic object-oriented metrics [4],[23].

In our research work, we only found four publicly available bug databases. These four datasets mainly operate with classic C&K [4] metrics and contain accumulated information about bugs at a pre-release or post-release time. Granularity is usually file level or class level that means the database contains bug characteristics for files or classes, consequently bug prediction is limited to this granularity. None of these databases consist data obtained from GitHub, they mostly gathered them from Bugzilla and Jira. We conducted an experiment using GitHub as the source of information (both for version control and for bug tracking).

Out of these databases *Terapromise* is the most up to date and has also a coding rule violation [18] database. Based on the capability of the tool we used for static source code analysis, we gathered C&K metrics, rule violations, and software code clone related metrics such as number of clone instances located in the given source code elements.

The *Bug prediction dataset* [6] contains data extracted from 5 Java projects by using inFusion and Moose to calculate the classic C&K metrics for class level. The source of information was mainly CVS, SVN, Bugzilla and Jira from which the number of pre- and post-release defects were calculated.

The *Zimmermann Eclipse* [24] database is still publicly available, however the last extension/modification was applied on March 25, 2010. Zimmermann et al. gathered complexity metrics and metrics describing the structure of the built AST for file level to detect pre- and post-release defects. The dataset is created by using the public information stored in Bugzilla.

Bugcatchers [10] operates with bad smells (solely), and found that coding rule violations have a small but significant effect on the occurrence of faults at file level. Bugcatchers used Bugzilla and Jira as the sources of information.

Many other papers used bug databases to extract some additional data, but these databases have never been published. Such databases amongst others are the following: IBugs [5], Mozilla [9], and Eclipse [2].

In this paper, we present an approach that uses GitHub, and collects a wide set of metrics for approximately six-months-long time intervals. This database is suitable for bug prediction purposes, and can be easily extended to involve more open-source projects.

3 Approach

In this section we briefly summarize our previous work [8] that also dealt with bug databases, however in the approach some major differences are present. We will highlight the hot spots where the two approaches differ from each other. In our previous work, first we downloaded the data from GitHub, then we processed the raw data to obtain statistical measurements on the projects. At this point we selected the relevant software versions to be analyzed by the static source code analyzer. After the source code analysis, we performed the database building that results in a dataset that stores entries in pairs such that a source code element that used to have at least one bug in it is present with the source code metrics calculated before the bug(s) was/were fixed (buggy state) and with the state when the bugs were already fixed. In this process, for each issue, we determined the following important source code versions:

- the last version that contains the untouched bug (the version before the first commit that references the issue),
- the first version that contains the fixed source code (the version after the last commit that references the issue),
- the versions that also contain the bug (versions after the issue reported and before the first fix was made).

We detected the references between the commits and the bugs by using the SZZ algorithm [22]. GitHub also provides the linkage between issues and commits. These links are determined from the message of the commits. With the use of these links, we accumulated the bug related source code elements (faulty classes) on issue level. A source code element is bug related, if it was modified in a commit that references the issue. Then we marked the buggy source code elements in the versions listed above. The database was constructed from the last version that contains the untouched bug and from the first version that contains the fixed source code as mentioned above.

In this current study, we followed another approach that rather follows the usual methods described in Section 2 [13],[6],[24]. Let us consider a few bugs that were later fixed (consider Figure 1). There are 3 versions of the system: A, B, and C, and we have 3 bugs in the software. We fixed bug A before version B that means bug A is present in the system in version A. The same is true for bug B, however bug B was finally fixed after version B, thus bug B appears also in the output of version B. At this point bug A is already fixed that causes it not to appear in version B. Bug C is similar to bug A.

Since the faulty elements are determined from the viewpoint of reported issues, and the issues are independent from the selected release versions, this means that the bug information is scattered in time. If a bug was reported after a specific release version and fixed before the subsequent selected version then the bug does not appear in either of the databases. To solve this issue, a common solution is to aggregate the bug information to the selected release versions. For every issue, we determined the preceding release version and marked the buggy source code elements.

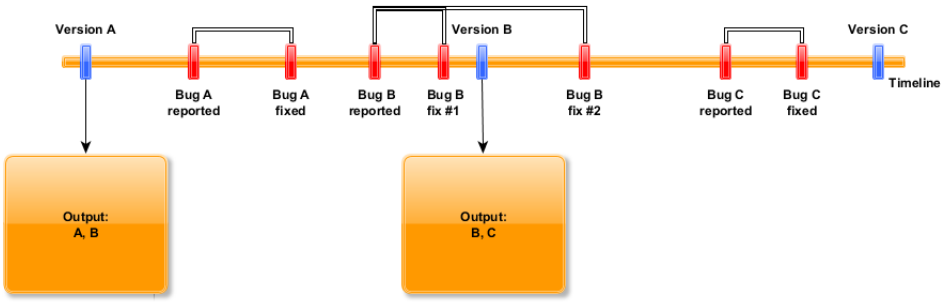

Fig. 1: The relationship between the bugs and release versions

In addition to the previous list, we determined

– the versions that partially contain the bug (versions after the first fix and before the last fix).

For the construction of the database we used the so-called traditional approach that means we collected release versions with approximately six-months-long time intervals for every project. We used six-month-long intervals since enough bugs and versions are present for such long time interval. Based on the age of a project, the number of selected release versions could differ for each project. We selected the release versions manually from the list of releases located on the projects GitHub pages. It is a common practice that projects use the release tag on a newly branched (from master) version of the source code. Since we use only the master branch as the main source of information, we had to perform a mapping when the hash id of the selected release is not representing a commit located in the master. Developers usually branch from master and then tag the branched version as release version, so our mapping algorithm detects when (time stamp) the release tag was applied on a version and searches for the last commit in the master branch that was made right before this time stamp.

We created a database for each of the release versions. Since bug tracking was not always used from the beginning of the projects, we could not assign any bug information to some of these earlier release versions. Also, the changing developer activity could result in lack of bug reports and consequently bug fixing commits are rare. All of these factors play roles in that the created databases vary in the number of bugs.

Similarly as in our earlier study, we computed some process metrics on file level from the data gathered from GitHub. This extra information is based on the actions performed on files by the developers. This means that if a file was not modified since it was uploaded with the initial commit then these extra metrics are zero. To avoid the misleading rows, we removed these files from the final database.

4 Chosen projects and the created databases

To select projects for the database construction, we examined many projects on GitHub. The main aspects were similar as in our previous paper. We chose 15 projects as data source. The selected software systems are listed in Table 1, together with some statistics. The first column contains the name of the projects with links to the GitHub repository in the footnote. The next column is the main domain of these systems. We can see that there is a large variance between the projects regarding the domain that strengthens the generality of the constructed database. The last three columns is the thousand Lines of Code, the Number of Commits and the Number of Bug Reports, respectively, on the master branch measured in May of 2015.

We constructed separate databases for class and file level. These databases are in CSV form (comma separated values). The first row in the CSV files contains header information such as unique identifier, source code position, source name, metric names, rule violation groups and number of bugs. The data in the rest of the lines follows this order. Each line represents a source code element (class, file).

In total we selected 105 release versions for the 15 projects and created 210 database files for six-months-long intervals. Table 2 presents the number of entries constructed for each project.

Table 1: The selected projects

Project	**Domain**	**kLOC**	**NC**	**NBR**
Android Universal I. L.[3]	Android library	13	996	89
ANTLR v4[4]	Language processing	85	3,276	111
Elasticsearch[5]	Search engine	677	13,778	2,108
jUnit[6]	Test framework	36	2,053	74
MapDB[7]	Database engine	83	1,345	175
mcMMO[8]	Game	42	4,552	657
Mission Control T.[9]	Monitoring platform	204	975	37
Neo4j[10]	Database engine	648	32,883	439
Netty[11]	Networking framework	282	6,780	1,039
OrientDB[12]	Database engine	380	10,197	174
Oryx[13]	Machine learning	47	363	36
Titan[14]	Database engine	119	3,830	121
Eclipse p. for Ceylon[15]	IDE	165	6,847	666
Hazelcast[16]	Computing platform	515	16,854	2,354
Broadleaf Commerce[17]	E-commerce framework	283	9,292	652

Table 2: Number of database entries for the selected projects

Project	Class	File	#DB Files
Android Universal I. L.	639	478	12
ANTLR v4	2,353	2,029	10
Elasticsearch	54,562	23,252	24
jUnit	5,432	2,266	16
MapDB	2,740	962	12
mcMMO	1,393	1,348	12
Mission Control T.	6,091	1,904	6
Neo4j	32,156	18,306	18
Netty	11,528	8,349	18
OrientDB	11,643	9,475	12
Oryx	2,157	1,400	8
Titan	5,312	3,713	12
Eclipse p. for Ceylon	4,512	2,129	10
Hazelcast	25,130	14,791	18
Broadleaf Commerce	17,433	14,703	22
Total	183,078	105,105	210

Figure 2 depicts the above mentioned entry numbers on a bar chart. Some projects have an outstanding number of class and file entries, however we are going to present results on every project one-by-one by evaluating the best machine learning algorithms for different release versions. Out of the total 183,078 class level entries, Elasticsearch has 54,562 in 12 databases that is not surprising if we consider the size of the project (677 kLOC). However, Neo4J has the most commits (twice as much as the second project which is Hazelcast), it has considerably less bug reports that results in a smaller database. In general, the bigger the project and the more bug reports a project has the bigger database it results in.

[3] https://github.com/nostra13/Android-Universal-Image-Loader
[4] https://github.com/antlr/antlr4
[5] https://github.com/elasticsearch/elasticsearch
[6] https://github.com/junit-team/junit
[7] https://github.com/jankotek/MapDB
[8] https://github.com/mcMMO-Dev/mcMMO
[9] https://github.com/nasa/mct
[10] https://github.com/neo4j/neo4j
[11] https://github.com/netty/netty
[12] https://github.com/orientechnologies/orientdb
[13] https://github.com/cloudera/oryx
[14] https://github.com/thinkaurelius/titan
[15] https://github.com/ceylon/ceylon-ide-eclipse
[16] https://github.com/hazelcast/hazelcast
[17] https://github.com/BroadleafCommerce/BroadleafCommerce

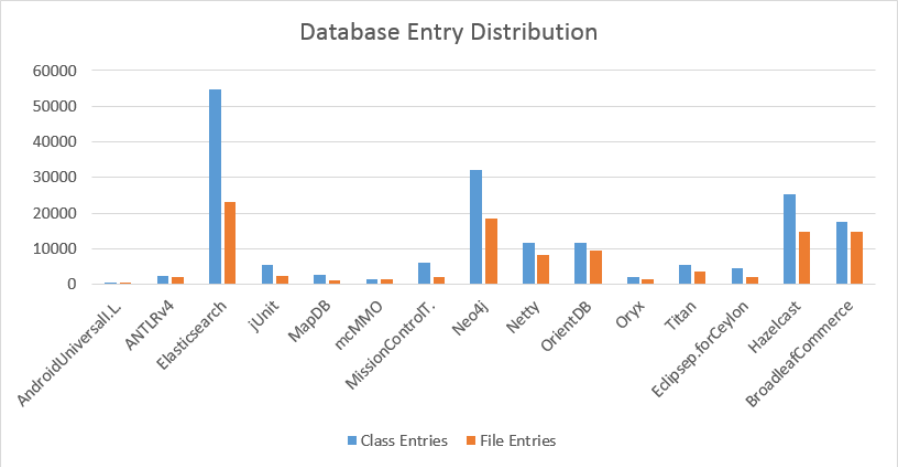

Fig. 2: Number of entries distribution

5 Evaluation

In this section we give exhaustive answers for the research questions by presenting our final results and achievements we made.

RQ 1: Is the constructed database usable for bug prediction? Which algorithms or algorithm families perform the best in bug prediction?

We evaluated our database by applying machine learning algorithms for all of the constructed data sets. The bug information in our database is present as number of bugs. To apply machine learning (classification), first we grouped the source code elements into two classes based on the occurrence of bugs in them. Instances with non-zero bug cardinality form a class (defective elements) and instances with zero bug number constitute the second separate class (non-defective elements).

If we look at the ratio between the number of defective and the number of non-defective elements, we may notice that there are way more non-defective elements in a software version than defective. Considering that we are planning to apply machine learning algorithms, it could distort the results, because the non-buggy instances get more emphasis. To deal with this issue, we applied random under sampling method to equalize the learning corpus [11],[21]. We randomly selected elements from the non-buggy class to match the size of the buggy category. This way we got a training set with the same number of positive and negative instances. We repeated this kind of learning 10 times and calculated an average. For the training, we used 10-fold cross validation and compared the results based on precision, recall, and F-measure metrics where these metrics are defined in the following way:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

where TP (True Positive) is the number of classes/files that were predicted as faulty and observed as faulty, FP (False Positive) is the number of classes/files that were predicted as faulty but observed as not faulty, FN (False Negative) is the number of classes/files that were predicted as non-faulty but observed as faulty. We carried out the training with the popular machine learning library called Weka.[18] It contains algorithms from different categories, for instance Bayesian methods, support vector machines, and decision trees.
We used the following algorithms:

- NaiveBayes
- NaiveBayesMultinomial
- Logistic
- SGD
- SimpleLogistic
- SMO
- VotedPerceptron [7]
- DecisionTable
- OneR
- PART
- J48 (C4.5) [16]
- RandomForest
- RandomTree

We analyzed software versions with six-month intervals from 15 projects. In total, we selected 105 release versions. 80 of these versions contain bug information due to the reasons mentioned in Section 3. 5 of the 80 versions contain too few buggy elements to apply machine learning. We ended up with 75 suitable versions for the training on class level. On file level, we got only 72, because in one buggy file there could be more than one buggy class, thus the size of the training set for a specific version could differ based on the granularity of the database.

[18] http://www.cs.waikato.ac.nz/ml/weka/

Class level First we investigated whether the class level databases are suitable for bug prediction purposes. Presenting the results for 15 projects (105 release versions) using all 13 machine learning algorithms would end up in a giant table that human eyes could not process, or at least can not focus on the most relevant parts. Consequently, we only present the best algorithms here to make it more easy to overview and find the best ones. Furthermore, for each project we selected the interval which has the most database entries to ensure the suitable size of the training corpus. Then, we used 10-fold cross-validation for that interval as described earlier. We chose the algorithms simply by calculating the averages on F-measure values and considered the best 5 algorithms. Table 3 presents the F-measure values for these 5 algorithms at class level.

Table 3: F-measures at class level

Project	SGD	SimpleLogistic	SMO	PART	RandomForest
Android Universal I. L.	0.6258	0.5794	0.5435	0.6188	0.7474
ANTLR v4	0.7586	0.7234	0.7379	0.7104	0.8066
Elasticsearch	0.7197	0.7304	0.7070	0.7171	0.7755
jUnit	0.7506	0.7649	0.7560	0.7262	0.7939
MapDB	0.7352	0.7667	0.7332	0.7421	0.7773
mcMMO	0.7192	0.6987	0.7203	0.6958	0.7418
Mission Control T.	0.7819	0.7355	0.7863	0.6862	0.8161
Neo4j	0.6911	0.7156	0.6835	0.6731	0.6767
Netty	0.7295	0.7437	0.7066	0.7521	0.7937
OrientDB	0.7485	0.7359	0.7310	0.7194	0.7823
Oryx	0.8012	0.7842	0.8109	0.7754	0.8059
Titan	0.7540	0.7558	0.7632	0.7301	0.7830
Eclipse p. for Ceylon	0.6891	0.7078	0.6876	0.7283	0.7503
Hazelcast	0.7128	0.7189	0.6965	0.7267	0.7659
Broadleaf Commerce	0.8019	0.8084	0.8081	0.7813	0.8210

As one can observe, values can be highly different by projects which can be caused by various reasons (size of the constructed dataset). For instance, let us consider the Android Universal Image Loader and Broadleaf Commerce projects. The Android project is the smallest one in size, Broadleaf is one of the middle-sized projects. Android has 639 class level entries in total (6 DB files), however Broadleaf has 17,433 entries (11 DB files) that is more suitable for being a training corpus. Nevertheless, if we take a closer look on the results we can see that the best F-measure values occurred also in small projects such as in Oryx or MCT, ergo we cannot generalize this conjecture to be true; however, further investigations should to be done to prove that. Tree-, function- and rule-based models performed the best in this scenario. F-measure values are up to 0.8210

that is a promising result. Before answering the first research question let us investigate the results at file level as well.

File level File level is different in some aspects from class level. For example, a completely distinct set of metrics (and also fewer) are calculated for file level entries. The best file level machine learning results are shown in Table 4. At first sight one can see that the results are in a wider range than in the case of class level. RandomForest has the highest F-measure values in case of files too. Furthermore, another tree based algorithm (J48) also performs nicely in this case. Two funtion-based (Logistic and SimpleLogistic) and one rule-based algorithm are in the top. Considering these results we can answer our research question.

Table 4: F-measures at file level

Project	Logistic	SimpleLogistic	PART	J48	RandomForest
Android Universal I. L.	0.5983	0.6230	0.6632	0.6215	0.6214
ANTLR v4	0.7638	0.7941	0.7443	0.8267	0.7645
Elasticsearch	0.6303	0.6280	0.6718	0.7025	0.7169
jUnit	0.6950	0.6530	0.6142	0.6613	0.6591
MapDB	0.7466	0.7337	0.7702	0.7790	0.8158
mcMMO	0.6864	0.6717	0.6583	0.6509	0.6951
Mission Control T.	0.7039	0.6700	0.6287	0.6573	0.7049
Neo4j	0.6621	0.7154	0.6766	0.6504	0.7150
Netty	0.6483	0.6549	0.6646	0.6823	0.7120
OrientDB	0.6868	0.6772	0.7157	0.7182	0.7234
Oryx	0.5537	0.5687	0.6500	0.6569	0.7331
Titan	0.6590	0.6813	0.6595	0.6407	0.6919
Eclipse p. for Ceylon	0.6664	0.6403	0.7141	0.7026	0.6837
Hazelcast	0.6883	0.6980	0.6742	0.6790	0.6946
Broadleaf Commerce	0.7244	0.7206	0.7736	0.7797	0.7875

Answering RQ 1: *Considering F-measure values for the chosen releases we can state that such databases are suitable for bug prediction by using machine learning algorithms to build prediction models. In bug prediction domain the RandomForest performed best in addition to function and rule based machine learning algorithms thus one should consider these first to build prediction models using our databases.*

After having insight in the bug prediction results, another question is put in words since the algorithms could perform better if they mark more classes/files buggy. It is an important aspect to see how many bugs are covered by the marked classes/files and what proportion of classes/files were marked as buggy.

RQ 2: Which machine learning algorithms or algorithm families perform the best in bug coverage?

Contrary to the investigation for the previous research question, in this context we cannot perform the same evaluation since we used random under sampling to equalize the number of buggy and non-buggy source code elements for the learning corpus, thus not all entries are included in the evaluation. For bug coverage we use the previously built 10 models (for the equalized training sets - with random under sampling) and evaluate it on the whole training set (without random under sampling). During the evaluation we use majority voting for an element (if more than five models predict the element as faulty then we tag it as faulty otherwise we tag it as non-faulty).

Table 5: Bug coverage at class level

Project	NaiveBayes	PART	J48	RandomForest	RandomTree
Android Universal I. L.	0.71 (0.21)	1.00 (0.39)	1.00 (0.47)	1.00 (0.42)	1.00 (0.42)
ANTLR v4	0.93 (0.20)	1.00 (0.35)	1.00 (0.26)	1.00 (0.27)	1.00 (0.27)
Elasticsearch	0.86 (0.14)	1.00 (0.33)	1.00 (0.32)	1.00 (0.32)	1.00 (0.32)
jUnit	0.82 (0.15)	1.00 (0.26)	1.00 (0.29)	1.00 (0.27)	1.00 (0.24)
MapDB	1.00 (0.25)	1.00 (0.29)	1.00 (0.21)	1.00 (0.26)	1.00 (0.26)
mcMMO	0.72 (0.18)	1.00 (0.40)	1.00 (0.39)	1.00 (0.41)	1.00 (0.36)
Mission Control T.	0.80 (0.21)	1.00 (0.22)	1.00 (0.32)	1.00 (0.18)	1.00 (0.17)
Neo4j	1.00 (0.14)	1.00 (0.34)	1.00 (0.27)	1.00 (0.36)	1.00 (0.39)
Netty	0.82 (0.18)	0.98 (0.34)	0.98 (0.32)	0.98 (0.35)	0.98 (0.33)
OrientDB	0.83 (0.18)	1.00 (0.31)	1.00 (0.32)	1.00 (0.31)	1.00 (0.33)
Oryx	0.92 (0.26)	1.00 (0.30)	0.93 (0.25)	1.00 (0.28)	1.00 (0.30)
Titan	0.66 (0.11)	0.94 (0.29)	0.94 (0.29)	0.94 (0.29)	0.94 (0.32)
Eclipse p. for Ceylon	0.79 (0.14)	1.00 (0.34)	0.98 (0.27)	1.00 (0.32)	1.00 (0.36)
Hazelcast	0.85 (0.14)	0.99 (0.32)	0.99 (0.31)	1.00 (0.31)	1.00 (0.32)
Broadleaf Commerce	0.60 (0.19)	1.00 (0.30)	0.94 (0.29)	1.00 (0.28)	1.00 (0.31)
Average	0.82 (0.18)	0.99 (0.32)	0.99 (0.31)	1.00 (0.31)	1.00 (0.31)

Table 5 and Table 6 show bug coverage values (ratio of covered bugs) and the ratio of how many classes or files have been tagged as faulty to obtain the bug coverage. Trees are performing best if considering only the bug coverage, however they tagged more than 31% of the source code elements as buggy in average. NaiveBayes is the other end of the story, since it has the lowest average of bug coverage, but tags the smallest amount of entries as buggy. Same results occurred at file level but here we present some other algorithms (not the best five) to show the differences in machine learning algorithms. We can state that our database is useful for finding bugs in software with high bug coverage.

Table 6: Bug coverage at file level

Project	RandomForest	DecisionTable	SGD	Logistic	NaiveBayes
Android Universal I. L.	1.00 (0.46)	1.00 (0.68)	0.46 (0.10)	0.81 (0.27)	0.81 (0.33)
ANTLR v4	1.00 (0.32)	0.91 (0.30)	0.91 (0.20)	0.91 (0.24)	0.82 (0.18)
Elasticsearch	1.00 (0.39)	0.94 (0.35)	0.82 (0.19)	0.83 (0.24)	0.73 (0.16)
jUnit	1.00 (0.44)	0.94 (0.30)	0.83 (0.25)	0.83 (0.31)	0.89 (0.20)
MapDB	1.00 (0.33)	1.00 (0.36)	0.93 (0.19)	0.97 (0.28)	0.90 (0.25)
mcMMO	1.00 (0.42)	0.93 (0.44)	0.81 (0.27)	0.82 (0.29)	0.75 (0.21)
Mission Control T.	1.00 (0.25)	1.00 (0.38)	1.00 (0.24)	1.00 (0.24)	1.00 (0.19)
Neo4j	1.00 (0.30)	1.00 (0.38)	1.00 (0.24)	1.00 (0.25)	0.80 (0.12)
Netty	1.00 (0.44)	0.99 (0.60)	0.85 (0.34)	0.88 (0.36)	0.73 (0.14)
OrientDB	1.00 (0.41)	0.97 (0.49)	0.95 (0.42)	0.92 (0.38)	0.79 (0.14)
Oryx	1.00 (0.43)	1.00 (0.66)	0.64 (0.17)	0.73 (0.32)	0.36 (0.09)
Titan	1.00 (0.37)	1.00 (0.45)	1.00 (0.64)	0.89 (0.38)	0.72 (0.11)
Eclipse p. for Ceylon	1.00 (0.39)	1.00 (0.42)	0.76 (0.21)	0.83 (0.25)	0.65 (0.11)
Hazelcast	1.00 (0.38)	0.95 (0.37)	0.87 (0.21)	0.89 (0.28)	0.80 (0.12)
Broadleaf Commerce	1.00 (0.33)	0.88 (0.34)	0.78 (0.21)	0.80 (0.24)	0.69 (0.14)
Average	1.00 (0.38)	0.97 (0.43)	0.84 (0.26)	0.87 (0.29)	0.76 (0.17)

Since we are in lack of space to introduce wide tables here we present our whole set of results as online appendix together with the full bug database at the following URL:
http://www.inf.u-szeged.hu/~ferenc/papers/GitHubBugDataSet/

We can now answer our second research question.

Answering RQ 2: *Tree based machine learning algorithms performed best in this scenario, with the highest bug coverage ratio. At class level circa 31% of the elements were tagged as buggy, but the F-measure values are still high (higher than 0.71). For file level the values are lower but in total the results are very similar to class level.*

6 Conclusion and Future Work

In this paper we proposed an approach for creating bug databases for selected release versions in an automatic way using the popular source code hosting system named GitHub. We gathered 15 Java projects from different domains to fulfill the need of generality. After constructing six-months-long release intervals we gathered bugs and the corresponding source code elements and organized them into databases.

We applied 13 machine learning algorithms on them to investigate whether the database is usable for bug prediction purposes. We experienced quite good

results for tree based algorithms (Random Forest, J48, Random Tree) with respect of F-measure values and bug coverage ratios.

In the future, we are planning to make our tool open-source, thus anybody can use or even improve our method. We plan to do more experiments with our models on other projects. We will try to identify (with statistical methods) connection between the usefulness of the database and other descriptors such as size of the projects or amount of the reported bugs.

Acknowledgment

This work was partially supported by the European Union project "REPARA – Reengineering and Enabling Performance And poweR of Applications", project number: 609666.

References

1. Erik Arisholm and Lionel C Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17. ACM, 2006.
2. P. Bangcharoensap, A. Ihara, Y. Kamei, and K. Matsumoto. Locating source code to be fixed based on initial bug reports - a case study on the eclipse project. In *Empirical Software Engineering in Practice (IWESEP), 2012 Fourth International Workshop on*, pages 10–15, Oct 2012.
3. Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.
4. Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.
5. Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.
6. Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.
7. Y. Freund and R. E. Schapire. Large margin classification using the perceptron algorithm. In *11th Annual Conference on Computational Learning Theory*, pages 209–217, New York, NY, 1998. ACM Press.
8. Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc. Characterization of source code defects by data mining conducted on github. In *Computational Science and Its Applications–ICCSA 2015*, pages 47–62. Springer, 2015.
9. Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, 2005.
10. Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33, 2014.
11. Haibo He, Edwardo Garcia, et al. Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on*, 21(9):1263–1284, 2009.

12. Yasutaka Kamei and Emad Shihab. Defect prediction: Accomplishments and future challenges.
13. Tim Menzies, Bora Caglayan, Zhimin He, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, June 2012.
14. Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.
15. Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Automating algorithms for the identification of fault-prone files. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 219–227. ACM, 2007.
16. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.
17. Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.
18. Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 403–414, 2015.
19. Georg Von Krogh and Eric Von Hippel. The promise of research on open source software. *Management science*, 52(7):975–983, 2006.
20. Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63. ACM, 2014.
21. Shuo Wang and Xin Yao. Using class imbalance learning for software defect prediction. *Reliability, IEEE Transactions on*, 62(2):434–443, 2013.
22. Chadd Williams and Jaime Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.
23. Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *Software Engineering, IEEE Transactions on*, 32(10):771–789, 2006.
24. Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.