A public unified bug dataset for java and its assessment regarding metrics and bug prediction

Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket & Tibor Gyimóthy

Software Quality Journal

ISSN 0963-9314
Volume 28
Number 4

Software Qual J (2020) 28:1447-1506
DOI 10.1007/s11219-020-09515-0

Springer

Software Quality Journal (2020) 28:1447–1506
https://doi.org/10.1007/s11219-020-09515-0

A public unified bug dataset for java and its assessment regarding metrics and bug prediction

Rudolf Ferenc[1] · Zoltán Tóth[1] · Gergely Ladányi[1] · István Siket[1] · Tibor Gyimóthy[2]

Published online: 3 June 2020

Abstract
Bug datasets have been created and used by many researchers to build and validate novel bug prediction models. In this work, our aim is to collect existing public source code metric-based bug datasets and unify their contents. Furthermore, we wish to assess the plethora of collected metrics and the capabilities of the unified bug dataset in bug prediction. We considered 5 public datasets and we downloaded the corresponding source code for each system in the datasets and performed source code analysis to obtain a common set of source code metrics. This way, we produced a unified bug dataset at class and file level as well. We investigated the diversion of metric definitions and values of the different bug datasets. Finally, we used a decision tree algorithm to show the capabilities of the dataset in bug prediction. We found that there are statistically significant differences in the values of the original and the newly calculated metrics; furthermore, notations and definitions can severely differ. We compared the bug prediction capabilities of the original and the extended metric suites (within-project learning). Afterwards, we merged all classes (and files) into one large dataset which consists of 47,618 elements (43,744 for files) and we evaluated the bug prediction model build on this large dataset as well. Finally, we also investigated cross-project capabilities of the bug prediction models and datasets. We made the unified dataset publicly available for everyone. By using a public unified dataset as an input for different bug prediction related investigations, researchers can make their studies reproducible, thus able to be validated and verified.

Keywords Bug dataset · Code metrics · Static code analysis · Bug prediction

1 Introduction

Finding and eliminating bugs in software systems has always been one of the most important issues in software engineering. Bug or defect prediction is a process by which we try to learn

✉ Rudolf Ferenc
ferenc@inf.u-szeged.hu

Extended author information available on the last page of the article.

from mistakes committed in the past and build a prediction model to leverage the location and amount of future bugs. Many research papers were published on bug prediction, which introduced new approaches that aimed to achieve better precision values (Zimmermann et al. 2009; Xu et al. 2000; Hall et al. 2012; Weyuker et al. 2010). Unfortunately, a reported bug is rarely associated with the source code lines that caused it or with the corresponding source code elements (e.g., classes, methods). Therefore, to carry out such experiments, bugs have to be associated with source code which in and of itself is a difficult task. It is necessary to use a version control system and a bug tracking system properly during the development process, and even in this case, it is still challenging to associate bugs with the problematic source code locations.

Although several algorithms were published on how to associate a reported bug with the relevant, corresponding defective source code (Dallmeier and Zimmermann 2007; Wong et al. 2012; Cellier et al. 2011), only few such bug association experiments were carried out. Furthermore, not all of these studies published the bug dataset or even if they did, closed source systems were used which limits the verifiability and reusability of the bug dataset. In spite of these facts, several bug datasets (containing information about open-source software systems) were published and made publicly available for further investigations or to replicate previous approaches (Weyuker et al. 2011; Robles 2010). These datasets are very popular; for instance, the NASA and the Eclipse Bug Dataset (Zimmermann et al. 2007) were used in numerous experiments (Gray et al. 2012; Jureczko and Madeyski 2010; Shepperd et al. 2013).

The main advantage of these bug datasets is that if someone wants to create a new bug prediction model or validate an existing one, it is enough to use a previously created bug dataset instead of building a new one, which would be very resource consuming. It is common in these bug datasets that all of them store some specific information about the bugs, such as the containing source code element(s) with their source code metrics or any additional bug-related information. Since different bug prediction approaches use various sources of information as predictors (independent variables), different bug datasets were constructed. Defect prediction approaches and hereby bug datasets can be categorized into larger groups based on the captured characteristics (D'Ambros et al. 2012):

- Datasets using process metrics (Moser et al. 2008; Nagappan and Ball 2005).
- Datasets using source code metrics (Basili et al. 1996; Briand et al. 1999; Subramanyam and Krishnan 2003).
- Datasets using previous defects (Kim et al. 2007; Ostrand et al. 2005).

Different bug prediction approaches use various public or private bug datasets. Although these datasets seem very similar, they are often very different in some aspects that is also true within the categories mentioned above. In this study, we gather datasets that can be found, but we will focus on datasets that use static source code metrics. Since this category itself has grown so immense, it is worth studying it as a separate unit. This category has also many dissimilarities between the existing datasets including the granularity of the data (source code elements can be files, classes, or methods, depending on the purpose of the given research or on the capabilities of the tool used to extract data) and the representation of element names (different tools may use different notations). For the same reason, the set of metrics can be different as well. Even if the names or the abbreviations of a metric calculated by different tools are the same, it can have different meanings because it can be

defined or calculated in a slightly different way. The bug-related information given for a source code element can also be contrasting. An element can be labeled whether it contains a bug, sometimes it shows how many bugs are related to that given source code element. From the information content perspective, it is less important, but not negligible that the format of the files containing the data can be CSV (Comma Separated Values), XML, or ARFF (which is the input format of Weka (Hall et al. 2009)), and these datasets can be found on different places on the Internet.

In this paper, we collected 5 publicly available datasets and we downloaded the corresponding source code for each system in the datasets and performed source code analysis to obtain a common set of source code metrics. This way, we produced a unified bug dataset at class and file level as well. Appendix explains the structure of the unified bug dataset which is available online.

To make it easier to imagine how a dataset looks like, Table 1 shows an excerpt of an example table where each row contains a Java class with its basic properties like Name or Path, which are followed by the source code metrics (e.g., WMC, CBO), and the most important property, the number of bugs.

After constructing the unified bug dataset, we examined the diversity of the metric suites. We calculated Pearson correlation and Cohen's *d* effect size, and applied the *Wilcoxon signed-rank test* to reveal these possible differences. Finally, we used a decision tree algorithm to show the usefulness of the dataset in bug prediction.

We found that there are statistically significant differences in the values of the original and the newly calculated metrics; furthermore, notations and definitions can severely differ. We compared the bug prediction capabilities of the original and the extended metric suites (within-project learning). Afterwards, we merged all classes (and files) into one large dataset which consists of 47,618 elements (43,744 for files) and we evaluated the bug prediction model build on this large dataset as well. Finally, we also investigated cross-project capabilities of the bug prediction models and datasets. We made the unified dataset publicly available for everyone. By using a public unified dataset as an input for different bug prediction-related investigations, researchers can make their studies reproducible, thus able to be validated and verified.

Our contributions can be listed as follows:

- Collection of the public bug datasets and source code.
- Unification of the contents of the collected bug datasets.
- Calculation of a common set of source code metrics.
- Comparison of the metrics suites.
- Assessment of the meta data of the datasets.
- Assessment of bug prediction capabilities of the datasets.
- Making the results publicly available.

The first version of this work was published in our earlier conference paper (Ferenc et al. 2018). In this extended version, we collected recent systematic literature review papers in the field and considered their references as well. Moreover, some further related work was used in the paper. As another major extension, we investigated whether the differences in metric values are statistically significant. Finally, we built and evaluated within-project, merged, and cross-project bug prediction models on the unified dataset.

Table 1 Example dataset table (excerpt)

Type	Name	Path	Line	Col.	...	WMC	CBO	...	LOC	...	bug
Class	ASTParser	...	83	1	...	96	55	...	1077	...	1
Class	ASTRecoveryPropagator	...	28	1	...	131	57	...	422	...	0
Class	ASTRequestor	...	34	1	...	6	4	...	85	...	0
Class	ASTSyntaxErrorPropagator	...	20	1	...	44	13	...	129	...	0
Class	ASTVisitor	...	104	1	...	170	84	...	2470	...	0
Class	AbstractTypeDeclaration	...	27	1	...	20	11	...	230	...	0
Class	Annotation	...	25	1	...	16	12	...	157	...	0
Class	AnnotationBinding	...	27	1	...	63	31	...	217	...	2
Class	AnnotationTypeDeclaration	...	46	1	...	26	14	...	226	...	0
Class	AnonymousClassDeclaration	...	32	1	...	14	8	...	159	...	0
Class	ArrayAccess	...	28	1	...	27	7	...	243	...	0
Class	ArrayCreation	...	49	1	...	27	11	...	271	...	0
Class	ArrayInitializer	...	28	1	...	13	7	...	133	...	0
Class	ArrayType	...	30	1	...	23	7	...	211	...	0
Class	AssertStatement	...	28	1	...	24	8	...	234	...	0
Class	Assignment	...	30	1	...	33	9	...	312	...	0
Class	BindingComparator	...	33	1	...	91	15	...	275	...	0
Class	BindingResolver	...	31	1	...	53	45	...	971	...	0
...	...	...	...	...	...	...	...	...	...	...	...

The paper is organized as follows. First, in Section 2, we give a brief overview about how we collected the datasets. We also introduce the collected public datasets and present the characteristics they have. Next, Section 3 presents the steps needed to be done for the sake of unification. We show the original metrics for each dataset and propose the extended metrics suite in Section 4, where we also compare the original and extended metric suites with a statistical method. In Section 5, we summarize the metadata of the datasets and we empirically assess the constructed unified bug dataset by showing its bug prediction capabilities. Afterwards, we list the threats to validity in Section 6. We conclude the paper and discuss future work in Section 7. Finally, Appendix describes the structure of the unified bug dataset and shows its download location.

2 Data collection

In this section, we give a detailed overview about how we collected and analyzed the datasets. We applied a snowballing-like technique (Wohlin 2014) as our data collection process. In the following, we will describe how our start set was defined, what were the inclusion criteria, and how we iterated over the relevant papers.

2.1 Start set

Starting from the early 70's (Randell 1975; Horning et al. 1974), a large number of studies was introduced in connection with software faults. According to Yu et al. (2016), 729 studies were published until 2005 and 1564 until 2015 on bug prediction (the number of studies has doubled in 10 years). From time to time, the enormous number of new publications in the topic of software faults made it unavoidable to collect the most important advances in literature review papers (Hosseini et al. 2019; Herbold et al. 2017; Wahono 2015).

Since these survey or literature review papers could serve as strong start set candidates, we used Scopus and Google Scholar to look for these papers. We used these two search sites to fulfill the diversity rule and cover as many different publishers, years, and authors as possible. We considered only peer-reviewed papers. Our search string was the following: '`(defect OR fault OR bug) AND prediction AND (literature OR review OR survey)`'. Based on the title and the abstract, we ended up with 32 candidates. We examined these papers and based on their content, we narrowed the start set to 12 (Catal and Diri 2009; Hall et al. 2012; Radjenović et al. 2013; Herbold et al. 2017; Strate and Laplante 2013; Catal 2011; Malhotra and Jain 2011; Jureczko and Madeyski 2011b; Malhotra 2015; Wahono 2015; Adewumi et al. 2016; Li et al. 2018). Other papers were excluded since they were out of scope, lacked peer review, or were not literature reviews. The included literature papers cover a time interval from 1990 to 2017.

2.2 Collecting bug datasets

Now we have the starting set of literature review papers, next, we applied backward snowballing to gather all the possible candidates which refer to a bug dataset. In other words, we considered all the references of the review papers to form the final set of candidates. Only one iteration of backward snowballing was used, since the survey papers have already included the most relevant studies in the field and sometimes they have also included reviews about the used datasets.

After having the final set of candidates (687), we filtered out irrelevant papers based on keywords, title, and abstract and we also searched for the string 'dataset' or 'data set' or 'used projects'. Investigating the remained set of papers, we took into consideration the following properties:

- Basic information (authors, title, date, publisher).
- Accessibility of the bug dataset (public, non public, partially public).
- Availability of the source code.

The latter two are extremely important when investigating the datasets, since we cannot construct a unified dataset without obtaining the appropriate underlying data.

From the final set of papers, we extracted all relevant datasets. We considered the following list to check whether a dataset meets our requirements:

- the dataset is publicly available,
- source code is accessible for the included systems,
- bug information is provided,
- bugs are associated with the relevant source code elements,
- included projects were written in Java,
- the dataset provides bug information at file/class level, and
- the source code element names are provided and unambiguous (the referenced source code is clearly identifiable).

If any condition is missing, then we had to exclude the subject system or the whole dataset from the study, because they cannot be included in the unified bug dataset.

Initially, we did not insist on examining Java systems; however, relevant research papers mostly focus on Java language projects (Sayyad Shirabad and Menzies 2005; Catal 2011; Radjenović et al. 2013). Consequently, we narrowed our research topic to datasets capturing information about systems written in Java. This way, we could use one static analysis tool to extract the characteristics from all the systems; furthermore, including heterogeneous systems would have added a bias to the unified dataset, since the interpretation of the metrics, even more the interpretable set of metrics themselves, can differ from language to language.

The list of found public datasets we could use for our purposes is the following (references are pointing to the original studies in which the datasets were first presented):

- PROMISE – Jureczko and Madeyski (2010)
- Eclipse Bug Dataset (Zimmermann et al. 2007)
- Bug Prediction Dataset (D'Ambros et al. 2010)
- Bugcatchers Bug Dataset (Hall et al. 2014)
- GitHub Bug Dataset (Tóth et al. 2016)

It is important to note that we will refer the Jureczko dataset as the PROMISE dataset throughout the study; however, the PROMISE repository itself contains more datasets such as the NASA MDP (Sayyad Shirabad and Menzies 2005) dataset (had to be excluded, since the source code is not accessible).

2.3 Public Datasets

In the following subsections, we will describe the chosen datasets in more details and investigate each dataset's peculiarities and we will also look for common characteristics. Before introducing each dataset, we show some basic size statistics about the chosen datasets,

which is presented in Table 2. We used the cloc (https://www.npmjs.com/package/cloc) program to measure the Lines of Code. We only took Java source files into consideration and we neglected blank lines.

2.3.1 PROMISE

PROMISE (Sayyad Shirabad and Menzies 2005) is one of the largest research data repositories in software engineering. It is a collection of many different datasets, including the NASA MDP (Metric Data Program) dataset, which was used by numerous studies in the past. However, one should always mistrust the data that comes from an external source (Petrić et al. 2016; Gray et al. 2011, 2012; Shepperd et al. 2013). The repository is created to encourage repeatable, verifiable, refutable, and improvable predictive models of software engineering. This is essential for maturation of any research discipline. One main goal is to extend the repository to other research areas as well. The repository is community based; thus, anybody can donate a new dataset or public tools, which can help other researchers in building state-of-the-art predictive models. PROMISE provides the datasets under categories like code analysis, testing, software maintenance, and it also has a category for defects.

One main dataset in the repository is the one from Jureczko and Madeyski (2010) which we use in our study. The dataset uses the classic Chidamber and Kemerer (C&K) metrics (Chidamber and Kemerer 1994) to characterize the bugs in the systems.

2.3.2 Eclipse Bug Dataset

Zimmermann et al. (2007) mapped defects from the bug database of Eclipse 2.0, 2.1, and 3.0. The resulting dataset lists the number of pre- and post-release defects on the granularity of files and packages that were collected from the BUGZILLA bug tracking system. They collected static code features using the built-in Java parser of Eclipse. They calculated some features at a finer granularity; these were aggregated by taking the average, total, and maximum values of the metrics. Data is publicly available and was used in many studies since then. Last modification on the dataset was submitted on March 25, 2010.

2.3.3 Bug Prediction Dataset

The *Bug prediction dataset* (D'Ambros et al. 2010) 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. D'Ambros et al. also extended the source

Table 2 Basic properties of the public bug datasets

Dataset	Systems	Versions	Lines of code
PROMISE	14	45	2,805,253
Eclipse Bug Dataset	1	3	3,087,826
Bug Prediction Dataset	5	5	1,171,220
Bugcatchers Bug Dataset	3	3	1,833,876
GitHub Bug Dataset	15	105	1,707,446

code metrics with change metrics, which, according to their findings, could improve the performance of the fault prediction methods.

2.3.4 Bugcatchers Bug Dataset

Hall et al. presented the *Bugcatchers* (Hall et al. 2014) Bug Dataset, which solely operates with bad smells, and found that coding rule violations have a small but significant effect on the occurrence of faults at file level. The Bugcatchers Bug Dataset contains bad smell information about Eclipse, ArgoUML, and some Apache software systems for which the authors used Bugzilla and Jira as the sources of the data.

2.3.5 GitHub Bug Dataset

Tóth et al. selected 15 Java systems from GitHub and constructed a bug dataset at class and file level (Tóth et al. 2016). This dataset was employed as an input for 13 different machine learning algorithms to investigate which algorithm family performs the best in bug prediction. They included many static source code metrics in the dataset and used these measurements as independent variables in the machine learning process.

2.4 Additional Bug Datasets

In this section, we show additional datasets which could not be included in the chosen set of datasets. Since this study focuses on datasets that fulfilled our selection criteria and could be used in the unification, we only briefly describe the most important but excluded datasets here.

2.4.1 Defects4J

Defects4J is a bug dataset which was first presented at the ISSTA conference in 2014 (Just et al. 2014). It focuses on bugs from software testing perspective. Defects4J encapsulates reproducible real world software bugs. Its repository[1] includes software bugs with their manually cleaned patch files (irrelevant code parts were removed manually) and most importantly it includes a test suite from which at least one test case fails before the patch was applied and none fails after the patch was applied. Initially, the repository contained 357 software bugs from 5 software systems, but it reached 436 bugs from 6 systems owing to active maintenance.

2.4.2 IntroClassJava

IntroClassJava (Durieux and Monperrus 2016) dataset is a collection of software programs each with several revisions[2]. The revisions were submitted by students and each revision is a maven project. This benchmark is interesting since it contains C programs transformed into Java. Test cases are also transformed into standard JUnit test cases. The benchmark consists of 297 Java programs each having at least one failing test case. The IntroClassJava dataset is very similar to Defects4J; however, it does not provide the manually cleaned fixing patches.

[1] https://github.com/rjust/defects4j
[2] https://github.com/Spirals-Team/IntroClassJava

2.4.3 QuixBugs

QuixBugs is a benchmark for supporting automatic program repair research studies (Lin et al. 2017). QuixBugs consists of 40 programs written in both Python and Java[3]. It also contains the failing test cases for the one-line bugs located in each program. Defects are categorized and each defect falls in exactly one category. The benchmark also includes the corrected versions of the programs.

2.4.4 Bugs.jar

Bugs.jar (Saha et al. 2018) is a large scale, diverse dataset for automatic bug repair[4]. Bugs.jar falls into the same dataset category as the previously mentioned ones. It consists of 1,158 bugs with their fixing patches from 8 large open-source software systems. This dataset also includes the bug reports and the test suite in order to support reproducibility.

2.4.5 Bears

Bears dataset (Madeiral et al. 2019) is also present to support automatic program repair studies[5]. This dataset makes use of the continuous integration tool named Travis to generate new entries in the dataset. It includes the buggy state of the source code, the test suite, and the fixing patch as well.

2.4.6 Summary

All the above described datasets are focusing on bugs from the software testing perspective and also support future automatic program repair studies. These datasets can be good candidates to be used in fault localization research studies as well. These datasets capture buggy states of programs and provide the test suite and the patch. Our dataset is fundamentally different from these datasets. The datasets we collected gather information from a wider time interval and provide information for each source code element by characterizing them with static source code metrics.

3 Data Processing

Although the found public datasets have similarities (e.g., containing source code metrics and bug information), they are very inhomogeneous. For example, they contain different metrics, which were calculated with different tools and for different kinds of code elements. The file formats are different as well; therefore, it is very difficult to use these datasets together. Consequently, our aim was to transform them into a unified format and to extend them with source code metrics that are calculated with the same tool for each system. In this section, we will describe the steps we performed to produce the unified bug dataset.

First, we transformed the existing datasets to a common format. This means that if a bug dataset for a system consists of separate files, we conflated them into one file. Next, we changed the *CSV* separator in each file to *comma (,)* and renamed the number of bug

[3] https://github.com/jkoppel/QuixBugs
[4] https://github.com/bugs-dot-jar/bugs-dot-jar
[5] https://github.com/bears-bugs/bears-benchmark

column in each dataset to '*bug*' and the source code element column name to '*filepath*' or '*classname*' depending on the granularity of the dataset. Finally, we transformed the source code element identifier into the standard form (e.g. *org.apache.tools.ant.AntClassLoader*).

3.1 Metrics calculation

The bug datasets contain different kinds of metric sets, which were calculated with different tools; therefore, even if the same metric name appears in two or more different datasets, we cannot be sure they mean exactly the same metric. To eliminate this deficiency, we analyzed all the systems with the same tool. For this purpose, we used the free and open-source *OpenStaticAnalyzer* 1.0 (OSA)[6] tool that is able to analyze Java systems (among other languages). It calculates more than 50 different kinds (size, complexity, coupling, cohesion, inheritance, and documentation) of source code metrics for packages and class-level elements, about 30 metrics for methods, and a few ones for files. OSA can detect code duplications (Type-1 and Type-2 clones) as well, and calculates code duplication metrics for packages, classes, and methods. OpenStaticAnalyzer has two different kinds of textual outputs: the first one is an XML file that contains, among others, the whole structure of the source code (files, packages, classes, methods), their relationships, and the metric values for each element (e.g. file, class, method). The other output format is CSV. Since different elements have different metrics, there is one CSV file for each kind of element (one for packages, one for classes, and so on).

The metrics in the bug datasets were calculated with 5 different tools (inFusion Moose, ckjm, Visitors written for Java parser of Eclipse, Bad Smell Detector, SourceMeter – which is a commercial product based on OSA; see Table 16). From these 5 tools, only ckjm and SourceMeter/OSA are still available on the internet, but the last version of ckjm is from 2012 and the Java language evolved a lot since then. Additionally, ckjm works on the bytecode representation of the code, which makes it necessary to compile the source code before analysis. Consequently, we selected OSA because it is a state-of-the-art analyzer, which works on the source code that besides being easier to use, enables also more precise analysis.

Of course, further tools are also available, but it was not the aim of this work to find the best available tool.

For calculating the new metric values, we needed the source code itself. Since all datasets belonged to a release version of a given software, therefore, if the software was open-source and the given release version was still available, we could manage to download and analyze it. This way, we obtained two results for each system: one from the downloaded bug datasets and one from the OSA analysis.

3.2 Dataset unification

We merged the original datasets with the results of OSA by using the "unique identifiers" of the elements (Java standard names at class level and paths at file level). More precisely, the basis of the unified dataset was our source code analysis result and it was extended with the data of the given bug dataset. This means that we went through all elements of the bug dataset and if the "unique identifier" of an element was found in our analysis result, then these two elements were conjugated (paired the original dataset entry with the one found in the result of *OSA*); otherwise, it was left out from the unified dataset. Tables 3 and 4 show

[6] https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer

Table 3 Merging results (number of elements) – Class level datasets

Dataset	Name	OSA	Orig.	Dropped
PROMISE	Ant 1.3	530	125	0
	Ant 1.4	602	178	0
	Ant 1.5	945	293	0
	Ant 1.6	1,262	351	0
	Ant 1.7	1,576	745	0
	Camel 1.0	734	339	0
	Camel 1.2	1,348	608	13 (+5)
	Camel 1.4	2,339	872	0 (+31)
	Camel 1.6	3,174	965	0 (+38)
	Ckjm 1.8	9	10	1
	Forrest 0.6	159	6	0
	Forrest 0.7	76	29	0
	Forrest 0.8	53	32	0
	Ivy 1.4	421	241	0
	Ivy 2.0	637	352	0
	JEdit 3.2	552	272	0
	JEdit 4.0	647	306	0
	JEdit 4.1	722	312	0
	JEdit 4.2	888	367	0
	JEdit 4.3	1,181	492	0
	Log4J 1.0	180	135	0
	Log4J 1.1	217	109	0
	Log4J 1.2	410	205	0
	Lucene 2.0	758	195	1
	Lucene 2.2	1,394	247	1
	Lucene 2.4	1,522	340	1
	Pbeans 1	38	26	0
	Pbeans 2	77	51	0
	Poi 1.5	472	237	0
	Poi 2.0	667	314	0
	Poi 2.5	780	385	0
	Poi 3.0	1,508	442	0
	Synapse 1.0	319	157	0
	Synapse 1.1	491	222	0
	Synapse 1.2	618	256	0
	Velocity 1.4	275	196	0
	Velocity 1.5	377	214	1
	Velocity 1.6	458	229	1
	Xalan 2.4	906	723	0
	Xalan 2.5	992	803	0
	Xalan 2.6	1,217	885	0
	Xalan 2.7	1,249	909	0
	Xerces 1.2	564	440	0

Table 3 (continued)

Dataset	Name	OSA	Orig.	Dropped
	Xerces 1.3	596	453	0
	Xerces 1.4	782	588	42
Bug Prediction Dataset	Eclipse JDT Core 3.4	2,486	997	0
	Eclipse PDE UI 3.4.1	3,382	1,497	6
	Equinox 3.4	742	324	5
	Lucene 2.4	1,522	691	21
	Mylyn 3.1	3,238	1,862	457
GitHub Bug Dataset	Android U. I. L. 1.7.0	84	73	0
	ANTLR v4 4.2	525	479	0
	Elasticsearch 0.90.11	6,480	5,908	0
	jUnit 4.9	770	731	0
	MapDB 0.9.6	348	331	0
	mcMMO 1.4.06	329	301	0
	MCT 1.7b1	2,050	1,887	0
	Neo4j 1.9.7	6,705	5,899	0
	Netty 3.6.3	1,300	1,143	0
	OrientDB 1.6.2	2,098	1,847	0
	Oryx	562	533	0
	Titan 0.5.1	1,770	1,468	0
	Eclipse p. for Ceylon 1.1.0	1,651	1,610	0
	Hazelcast 3.3	3,765	3,412	0
	Broadleaf C. 3.0.10	2,094	1,593	0
Sum	All	**76,623**	**48,242**	**624**

the results of this merging process at class and file level, respectively: column *OSA* shows how many elements OSA found in the analyzed systems, column *Orig.* presents the number of elements originally in the datasets, and column *Dropped* tells us how many elements of the bug datasets could not be paired, and so they were left out from the unified dataset. The numbers in parentheses show the amount of dropped elements where the drop was caused because of the original sources were not real Java sources, such as *package-info.java* and *Scala files* (which are also compiled to byte code and hence included in the original dataset).

Although these numbers are very promising, we had to "modify" a few systems to achieve this, but there were cases where we simply could not solve the inconsistencies. The details of the source code modifications and main reasons for the dropped elements were the following:

Camel 1.2: In the *org.apache.commons.logging*, there were 13 classes in the original dataset that we did not find in the source code. There were 5 *package-info.java*[7] files in the system, but these files never contain any Java classes, since they are used for package level Javadoc purposes; therefore, OSA did not find such classes.

[7]Scala (see Camel 1.4) and *package-info.java* files are "not real" Java source files therefore they should not be included in the original results so their quantities are presented in parenthesis in Table 3.

Table 4 Merging results (number of elements) – File level datasets

Dataset	Name	OSA	Orig.	Dropped
Eclipse Bug Dataset	Eclipse 2.0	6,751	6,729	0
	Eclipse 2.1	7,909	7,888	0
	Eclipse 3.0	10,635	10,593	0
Bugcatchers Bug Dataset	Apache Commons	491	191	0
	ArgoUML 0.26 Beta	1,752	1,582	3
	Eclipse JDT Core 3.1	12,300	560	25
GitHub Bug Dataset	Android U. I. L. 1.7.0	63	63	0
	ANTLR v4 4.2	411	411	0
	Elasticsearch 0.90.11	3,540	3,035	0
	jUnit 4.9	308	308	0
	MapDB 0.9.6	137	137	0
	mcMMO 1.4.06	267	267	0
	MCT 1.7b1	1,064	413	0
	Neo4j 1.9.7	3,291	3,278	0
	Netty 3.6.3	914	913	0
	OrientDB 1.6.2	1,503	1,503	0
	Oryx	443	280	0
	Titan 0.5.1	981	975	0
	Ceylon for Eclipse 1.1.0	699	699	0
	Hazelcast 3.3	2,228	2,228	0
	Broadleaf C. 3.0.10	1,843	1,719	0
Sum	All	**57,530**	**43,772**	**28**

Camel 1.4: Besides the 7 *package-info.java* files, the original dataset contained information about 24 Scala files (they are also compiled to byte code); therefore, OSA did not analyze them.

Camel 1.6: There were 8 *package-info.java* and 30 Scala files.

Ckjm 1.8: There was a class in the original dataset that did not exist in version 1.8.

Forrest-0.8: There were two different classes that appeared twice in the source code; therefore, we deleted the 2 copies from the *etc/test-whitespace* subdirectory.

Log4j: There was a *contribs* directory which contained the source code of different contributors. These files were put into the appropriate sub-directories as well (where they belonged according to their packages), which means that they occurred twice in the analysis and this prevented their merging. Therefore, in these cases, we analyzed only those files that were in their appropriate subdirectories and excluded the files found in the *contribs* directory.

Lucene: In all three versions, there was an *org.apache.lucene.search.RemoteSearchable_Stub* class in the original dataset that did not exist in the source code.

Velocity: In versions 1.5 and 1.6 there were two *org.apache.velocity.app.event. implement.EscapeReference* classes in the source code; therefore, it was impossible to conjugate them by using their "unique identifiers" only.

Xerces 1.4.4: Although the name of the original dataset and the corresponding publication state that this is the result of Xerces 1.4.4 analysis, we found that 256 out of the 588 elements did not exist in that version. We examined a few previous and following versions as well, and it turned out that the dataset is much closer to 2.0.0 than to 1.4.4, because only 42 elements could not be conjugated with the analysis result of 2.0.0. Although version 2.0.0 was still not matched perfectly, we did not find a "closer version"; therefore, we used Xerces 2.0.0 in this case.

Eclipse JDT Core 3.4: There were a lot of classes which appeared twice in the source code: once in the "code" and once in the "test" directory; therefore, we deleted the test directory.

Eclipse PDE UI 3.4.1: The missing 6 classes were not found in its source code.

Equinox 3.4: Three classes could not be conjugated, because they did not have a unique name (there are more classes with the same name in the system) while two classes were not found in the system.

Lucene 2.4 (BPD): 21 classes from the original dataset were not present in the source code of the analyzed system.

Mylyn 3.1: 457 classes were missing from our analysis that were in the original dataset; therefore, we downloaded different versions of Mylyn, but still could not find the matching source code. We could not achieve better result without knowing the proper version.

ArgoUML 0.26 Beta: There were 3 classes in the original dataset that did not exist in the source code.

Eclipse JDT Core 3.1: There were 25 classes that did not exist in the analyzed system.

GitHub Bug Dataset Since OSA is the open-source version of SourceMeter, the tool used to construct the GitHub Bug Dataset, we could easily merge the results. However, the class level bug datasets contained elements having the same "unique identifier" (since class names are not the standard Java names in that case), so this information was not enough to conjugate them. Luckily, the paths of the elements were also available and we used them as well; therefore, all elements could be conjugated. Since they performed a machine learning step on the versions that contain the most bugs, we decided to select these release versions and present the characteristics of these release versions. We also used these versions of the systems to include in the unified bug dataset.

As a result of this process, we obtained a unified bug dataset which contains all of the public datasets in a unified format; furthermore, they were extended with the same set of metrics provided by the OSA tool. The last lines of Tables 3 and 4 show that only 1.29% (624 out of 48,242) of the classes and 0.06% (28 out of 43,772) of the files could not be conjugated, which means that only 0.71% (652 out of 92,014) of the elements were left out from the unified dataset in total.

Table 5 Metrics used in the PROMISE dataset

Name	Abbr.
Weighted methods per class	WMC
Depth of Inheritance Tree	DIT
Number of Children	NOC
Coupling between object classes	CBO
Response for a Class	RFC
Lack of cohesion in methods	LCOM
Afferent couplings	Ca
Efferent couplings	Ce
Number of Public Methods	NPM
Lack of cohesion in methods (by Henderson-Sellers)	LCOM3
Lines of Code	LOC
Data Access Metric	DAM
Measure of Aggregation	MOA
Measure of Functional Abstraction	MFA
Cohesion Among Methods of Class	CAM
Inheritance Coupling	IC
Coupling Between Methods	CBM
Average Method Complexity	AMC
McCabe's cyclomatic complexity	CC
Maximum McCabe's cyclomatic complexity	MAX_CC
Average McCabe's cyclomatic complexity	AVG_CC
Number of files (compilation units)	NOCU

In many cases, the analysis results of OSA contained more elements than the original datasets. Since we did not know how the bug datasets were produced, we could not give an exact explanation for the differences, but we list the two main possible causes:

– In some cases, we could not find the proper source code for the given system (e.g., Xerces 1.4.4 or Mylyn), so two different but close versions of the same system might be conjugated.
– OSA takes into account nested, local, and anonymous classes while some datasets simply associated Java classes with files.

4 Original and extended metrics suites

In this section, we present the metrics proposed by each dataset. Additionally, we will show a metrics suite that is used by the newly constructed unified dataset.

4.1 PROMISE

The authors Promiserepo (2018) calculated the metrics of the PROMISE dataset with the tool called *ckjm*. All metrics, except McCabe's Cyclomatic Complexity (CC), are *class* level

Table 6 Metrics used in the Eclipse Bug Dataset

Name	Abbr.
Number of method calls	FOUT
Method lines of code	MLOC
Nested block depth	NBD
Number of parameters	PAR
McCabe cyclomatic complexity	VG
Number of field	NOF
Number of method	NOM
Number of static fields	NSF
Number of static methods	NSM
Number of anonymous type declarations	ACD
Number of interfaces	NOI
Number of classes	NOT
Total lines of code	TLOC
Number of files (compilation units)	NOCU

metrics. Besides the C&K metrics, they also calculated some additional metrics shown in Table 5.

4.2 Eclipse Bug Dataset

In the Eclipse Bug Dataset, there are two types of predictors. By parsing the structure of the obtained abstract syntax tree, they Zimmermann et al. (2007) calculated the number of nodes for each type in a package and in a *file* (e.g., the number of return statements in a file). By implementing visitors to the Java parser of Eclipse, they also calculated various complexity metrics at method, class, file, and package level. Then they used avg, max, total avg, and total max aggregation techniques to accumulate to file and package level. The complexity metrics used in the Eclipse dataset are listed in Table 6.

4.3 Bug Prediction Dataset

The Bug Prediction Dataset collects product and change (process) metrics. The authors (D'Ambros et al. 2010) produced the corresponding product and process metrics at *class* level. Besides the classic C&K metrics, they calculated some additional object-oriented metrics that are listed in Table 7.

4.4 Bugcatchers Bug Dataset

The Bugcatchers Bug Dataset is a bit different from the previous datasets, since it does not contain traditional software metrics, but the number of bad smells for files. They used five bad smells which are presented in Table 8. Besides, in the CSV file, there are four source code metrics (blank, comment, code, codeLines), which are not explained in the corresponding publication (Hall et al. 2014).

Table 7 Metrics used in the Bug Prediction Dataset

Name	Abbr.
Number of other classes that reference the class	FanIn
Number of other classes referenced by the class	FanOut
Number of attributes	NOA
Number of public attributes	NOPA
Number of private attributes	NOPRA
Number of attributes inherited	NOAI
Number of lines of code	LOC
Number of methods	NOM
Number of public methods	NOPM
Number of private methods	NOPRM
Number of methods inherited	NOMI

4.5 GitHub Bug Dataset

The GitHub Bug Dataset (Tóth et al. 2016) used the free version of SourceMeter (2019) static analysis tool to calculate the static source code metrics including software product metrics, code clone metrics, and rule violation metrics. The rule violation metrics were not used in our research; therefore, Table 9 shows only the list of the software product and code clone metrics at class level. At file level, only a narrowed set of metrics is calculated, but there are 4 additional process metrics included as Table 10 shows.

4.6 Unified Bug Dataset

The unified dataset contains all the datasets with their original metrics and with further metrics that we calculated with OSA. The set of metrics calculated by OSA concurs with the metric set of the GitHub Bug Dataset, because SourceMeter is a product based on the free and open-source OSA tool. Therefore, all datasets in the Unified Bug Dataset are extended with the metrics listed in Table 9 except the GitHub Bug Dataset, because it contains the same metrics originally.

In spite of the fact that several of the original metrics can be matched with the metrics calculated by OSA, we decided to keep all the original metrics for every system included in the unified dataset, because they can differ in their definitions or in the ways of their calculation. One can simply use the unified dataset and discard the metrics that were calculated by OSA if s/he wants to work only with the original metrics. Furthermore, this provides an opportunity to confront the original and the OSA metrics.

Table 8 Bad smells used in the Bugcatchers Bug Dataset

Name
Data Clumps
Message Chains
Middle Man
Speculative Generality
Switch Statements

Table 9 Class Level Metrics used in the GitHub Bug Dataset

Name	Abbr.	Name	Abbr.
API Documentation	AD	Number of Local Public Methods	NLPM
Clone Classes	CCL	Number of Local Setters	NLS
Clone Complexity	CCO	Number of Methods	NM
Clone Coverage	CC	Number of Outgoing Invocations	NOI
Clone Instances	CI	Number of Parents	NOP
Clone Line Coverage	CLC	Number of Public Attributes	NPA
Clone Logical Line Coverage	CLLC	Number of Public Methods	NPM
Comment Density	CD	Number of Setters	NS
Comment Lines of Code	CLOC	Number of Statements	NOS
Coupling Between Object classes	CBO	Public Documented API	PDA
Coupling Between Obj. classes Inv.	CBOI	Public Undocumented API	PUA
Depth of Inheritance Tree	DIT	Response set For Class	RFC
Documentation Lines of Code	DLOC	Total Comment Density	TCD
Lack of Cohesion in Methods 5	LCOM5	Total Comment Lines of Code	TCLOC
Lines of Code	LOC	Total Lines of Code	TLOC
Lines of Duplicated Code	LDC	Total Logical Lines of Code	TLLOC
Logical Lines of Code	LLOC	Total Number of Attributes	TNA
Logical Lines of Duplicated Code	LLDC	Total Number of Getters	TNG
Nesting Level	NL	Total Number of Local Attributes	TNLA
Nesting Level Else-If	NLE	Total Number of Local Getters	TNLG
Number of Ancestors	NOA	Total Number of Local Methods	TNLM
Number of Attributes	NA	Total Number of Local Public Attr.	TNLPA
Number of Children	NOC	Total Number of Local Public Meth.	TNLPM
Number of Descendants	NOD	Total Number of Local Setters	TNLS
Number of Getters	NG	Total Number of Methods	TNM
Number of Incoming Invocations	NII	Total Number of Public Attributes	TNPA
Number of Local Attributes	NLA	Total Number of Public Methods	TNPM
Number of Local Getters	NLG	Total Number of Setters	TNS
Number of Local Methods	NLM	Total Number of Statements	TNOS
Number of Local Public Attributes	NLPA	Weighted Methods per Class	WMC

Table 10 File Level Metrics used in the GitHub Bug Dataset

Name	Abbr.
McCabe's Cyclomatic Complexity	McCC
Comment Lines of Code	CLOC
Logical Lines of Code	LLOC
Number of Committers	–
Number of developer commits	–
Number of previous modifications	–
Number of previous fixes	–

Instead of presenting all the definitions of metrics here, we give an external resource to show metric definitions because of the lack of space. All the metrics and their definitions can be found in the Unified Bug Dataset file reachable as an online appendix (see Appendix).

4.7 Comparison of the Metrics

In the unified dataset, each element has numerous metrics, but these values were calculated by different tools; therefore, we assessed them in more detail to get answers to questions like the following ones:

- Do two metrics with the same name have the same meaning?
- Do metrics with different names have the same definition?
- Can two metrics with the same definition be different?
- What are the root causes of the differences if the metrics share the definition?

Three out of the five datasets contain class level elements, but unfortunately, for each dataset, a different analyzer tool was used to calculate the metrics (see Table 16). To be able to compare class level metrics calculated by all the tools used, we needed at least one dataset for which all metrics of all three tools are available. We were already familiar with the usage of the ckjm tool, so we chose to calculate the ckjm metrics for the Bug Prediction dataset. This way, we could assess all metrics of all tools, because the Bug Prediction dataset was originally created with Moose, so we have extended it with the OSA metrics, and also – for the sake of this comparison – with ckjm metrics.

In the case of the three file level datasets, the used analyzer tools were unavailable; therefore, we could only compare the file level metrics of OpenStaticAnalyzer with the results of the other two tools separately on Eclipse and Bugcatchers Bug datasets.

In each comparison, we merged the different result files of each dataset into one, which contained the results of all systems in the given dataset and deleted those elements that did not contain all metric values. For example, in case of the Bug Prediction Dataset, we calculated the OSA and ckjm metrics, then we removed the entries which were not identified by all three tools. Because we could not find the analyzers used in the file level datasets, we used the merging results seen in Section 3.2. For instance, in case of the Bugcatchers Bug Dataset, the new merged (unified) dataset has 14,543 entries (491 + 1,752 + 12,300) out of which 2,305 were in the original dataset and not dropped (191 + 1,582 + 560 - 28).

The resulting spreadsheet files can be found in the Appendix. Table 11 shows how many classes or files were in the given dataset and how many of them remained. We calculated the basic statistics (minimum, maximum, average, median, and standard deviation) of the examined metrics and compared them (see Table 12). Besides, we calculated the pairwise differences of the metrics for each element and examined its basic statistics as well. In addition, the *Equal* column shows the percentage of the classes for which the two examined tools gave the same result (for example, at class level OSA and Moose calculated the same WMC value for 2,635 out of the 4,167 elements, which is 63.2%, see Table 12).

Since the basic statistic values give only some impression about the similarity of the metric sets, we performed a Wilcoxon signed-rank test (Myles et al. 2014), which determines whether two dependent samples were selected from populations having the same distribution. In our test, the H_0 hypothesis is that the difference between the pairs follows a symmetric distribution around zero, while the alternative H_1 hypothesis is that the difference between the pairs does not follow a symmetric distribution around zero. We used 95% confidence level in the tests to calculate the *p-values*. This means that if a *p-value* is higher

Table 11 Number of elements in the merged systems

Name	Merged	Remained elements
Bug Prediction Dataset	11,370	4,167
Eclipse Bug Dataset	25,295	25,210
Bugcatchers bug Dataset	14,543	2,305

than *0.05*, we accept the H_0 hypothesis; otherwise, we reject it and accept the H_1 alternative hypothesis instead. In all cases, the p-values were less than 0.001; therefore, we had to reject the H_0 and accept that the difference between the pairs does not follow a symmetric distribution around zero.

Although from statistical point of view the metric sets are different, we see that in many cases, there are lot of equal metric values. For example, in case of the file level dataset of Eclipse (see Table 14), only 11 out of 25,199 metrics are different, but 10 out of these 11 are larger only by 1 than their pairs so the test recognizes well that it is not symmetric. On the other hand, in this case, we can say that the two groups can be considered identical because less than 0.1% of the elements differ, and the difference is really neglectable. Therefore, we calculated *Cohen's d* as well which indicates the standardized difference between two means (Cohen 1988). If this difference, namely Cohen's d value, is less than 0.2, we say that the effect size is small and more than 90% of the two groups overlap. If the Cohen's d value is between 0.2 and 0.5, the effect size is medium and if the value is larger than 0.8, the effect size is large. Besides, to see how strong or weak the correlations between the metric values are, we calculated the *Pearson correlation coefficient*. The correlation coefficient ranges from -1 to 1, where 1 (or -1) means that there is linear equation between the two sets, while 0 means that there is no linear correlation at all. In general, the larger the absolute correlation coefficient is, the stronger the relationship is. In the literature, there are different recommendations for the threshold above the correlation is considered strong (Gyimothy et al. 2005a; Jureczko 2011a), in this research, we used 0.8.

4.7.1 Class Level Metrics

The unified bug dataset contains the class level metrics of OSA and Moose on the Bug Prediction dataset. We downloaded the Java binaries of the systems in this dataset and used ckjm version 2.2 to calculate the metrics. The first difference is that while OSA and Moose calculate metrics on source code, ckjm uses Java bytecode and takes "external dependencies" into account; therefore, we expected differences, for instance, in the coupling metric values.

We compared the metric sets of the three tools and found that, for example, CBO and WMC have different definitions. On the other hand, efferent coupling metric is a good example for a metric which is calculated by all three tools, but with different names (see Table 12, CBO row). In the following paragraphs, we only examine those metrics whose definitions coincide in all three tools even if their names differ. Table 12 shows these metrics where the *Metric* column contains the abbreviation of the most widely used name of the metric. The *Tool* column presents the analyzer tools, in the *Metric name* column, the metric names are given using the notations of the different datasets. The "$tool_1 - tool_2$" means the pairwise difference where, for each element, we extracted the value of tool_2 from the value of tool_1 and the name of this "new metric" is diff. The following columns present the basic statistics of the metrics. The Equal column denotes the percentage of the elements having

Table 12 Comparison of the common class level metrics in the Bug Prediction dataset

Metric	Tool	Metric n.	Min	Max	Avg	Med	Dev	Equal	Cohen	Pearson
WMC	OSA	NLM	0	426	11.04	7	18.12	–	–	–
	Moose	Methods	0	403	9.96	6	14.38	–	–	–
	ckjm	WMC	1	426	11.96	7	18.49	–	–	–
	OSA−Moose	diff	−4	**420**	1.08	0	9.40	**63.2%**	**0.066**	**0.857**
	OSA−ckjm	diff	−48	0	−0.91	0	1.94	**51.9%**	**0.050**	**0.995**
	Moose−ckjm	diff	**−421**	4	−1.99	−1	9.57	41.1%	**0.120**	**0.859**
CBO	OSA	CBO	0	214	8.86	5	12.25	–	–	–
	Moose	fanOut	0	93	6.22	4	7.79	–	–	–
	ckjm	Ce	0	213	13.78	8	16.88	–	–	–
	OSA−Moose	diff	−32	161	2.65	2	7.61	16.0%	0.258	**0.801**
	OSA−ckjm	diff	−120	83	−4.91	−1	9.72	26.2%	0.333	**0.823**
	Moose−ckjm	diff	−160	32	−7.56	−4	11.84	7.4%	0.575	0.780
CBOI	OSA	CBOI	0	607	9.38	3	26.14	–	–	–
	Moose	fanIn	0	355	4.69	1	14.30	-	–	–
	ckjm	Ca	0	611	7.64	2	22.13	–	–	–
	OSA−Moose	diff	−18	**607**	4.69	1	16.55	43.4%	0.222	**0.821**
	OSA−ckjm	diff	−100	189	1.74	0	11.02	**59.6%**	**0.072**	**0.909**
	Moose−ckjm	diff	**−611**	146	−2.95	−1	15.30	39.6%	**0.158**	0.727

Table 12 (continued)

Metric	Tool	Metric n.	Min	Max	Avg	Med	Dev	Equal	Cohen	Pearson
RFC	OSA	RFC	0	600	22.82	12	34.53	–	–	–
	Moose	rfc	0	2,603	50.62	23	108.06	–	–	–
	ckjm	RFC	2	684	38.93	23	49.72	–	–	–
	OSA−Moose	diff	**−2,095**	600	−27.80	−8	83.70	8.4%	0.347	0.786
	OSA−ckjm	diff	−327	12	−16.11	−9	22.72	4.9%	0.376	**0.917**
	Moose−ckjm	diff	−673	**2,049**	11.69	−1	75.42	5.7%	**0.139**	0.787
DIT	OSA	DIT	0	8	1.31	1	1.63	–	–	–
	Moose	dit	**1**	9	2.08	2	1.44	–	–	–
	ckjm	DIT	0	5	0.38	0	0.60	–	–	–
	OSA−Moose	diff	−3	0	−0.76	−1	0.43	23.9%	0.496	**0.969**
	OSA−ckjm	diff	−5	8	0.94	1	1.96	22.1%	0.761	0.418
	Moose−ckjm	diff	−4	9	1.70	2	1.79	30.2%	1.540	0.453
NOC	OSA	NOC	0	107	0.73	0	3.27	–	–	–
	Moose	noc	0	49	0.64	0	2.55	–	–	–
	ckjm	NOC	0	107	0.64	0	2.95	–	–	–
	OSA−Moose	diff	−3	97	0.08	0	1.68	**96.8%**	**0.028**	**0.863**
	OSA−ckjm	diff	0	42	0.09	0	1.15	**97.1%**	**0.029**	**0.937**
	Moose−ckjm	diff	−97	34	0.01	0	1.81	**95.8%**	**0.003**	0.794

Table 12 (continued)

Metric	Tool	Metric n.	Min	Max	Avg	Med	Dev	Equal	Cohen	Pearson
LOC	OSA	LLOC	2	8,746	131.99	56	357.39	–	–	–
	Moose	LinesOfCode	0	7,341	124.01	51	306.54	–	–	–
	ckjm	LOC	4	26,576	399.42	147	1142.60	–	–	–
	OSA−Moose	diff	−1,068	7,824	7.98	3	157.69	3.1%	**0.024**	**0.898**
	OSA−ckjm	diff	−19,150	112	**−267.43**	−91	791.30	0.6%	0.316	**0.988**
	Moose−ckjm	diff	−26,541	198	**−275.41**	−93	879.89	0.1%	0.329	**0.893**
NPM	OSA	NLPM	0	404	7.23	4	13.67	–	–	–
	Moose	PublicMethods	0	387	6.42	4	11.28	–	–	–
	ckjm	NPM	0	404	7.48	5	13.64	–	–	–
	OSA−Moose	diff	−4	236	0.81	0	6.55	**68.0%**	**0.065**	**0.879**
	OSA−ckjm	diff	−8	0	−0.25	0	0.45	**75.8%**	**0.018**	**0.999**
	Moose−ckjm	diff	−237	3	−1.06	0	6.55	**62.2%**	**0.085**	**0.879**

the same metric value (i.e., the difference is 0), and the last two columns are the Cohen's d value and the Pearson correlation coefficient (where it is appropriate). We highlighted with bold face those values that suggest that the two metric set values are close to each other from a given aspect:

- if more than half of the element pairs are equal (i.e., Equal is above 50%),
- if the effect size is small (i.e., Cohen's d is less than 0.2),
- if there is strong linear correlation between the elements (i.e., the absolute value of the Pearson correlation coefficient is larger than 0.8).

Next, we will analyze the metrics one at a time.

WMC: This metric expresses the complexity of a class as the sum of the complexity of its methods. In its original definition, the method complexity is deliberately not defined exactly; and usually the uniform weight of 1 is used. In this case, this variant of WMC is calculated by all three tools. Its basic statistics are more or less the same and the pairwise values of OSA and ckjm seem to be closer to each other (see OSA−ckjm row) than to Moose and the extremely high Pearson correlation value (0.995) supports this. Among the Moose results, there are several very low values where the other tools found a great number of methods and that caused the extreme difference (e.g., the max. value of OSA−Moose is 420). In spite of this, the Pearson correlations between the results Moose and the other tools are high. On the other hand, OSA and Moose gave the same result for almost two third of the classes, which means that the difference probably comes from outliers. And finally, the Cohen's *d* values are small, so we can say that the three tools gave very similar results but with notable outliers.

CBO: In this definition, CBO counts the number of classes the given class depends on. Although it is a coupling metric, it counts efferent (outer) couplings; therefore, the metric values should have been similar. On the other hand, based on the statistical values and the pairwise comparison including Equal and Cohen values, we can say that these metrics differ significantly. We can observe strong linear correlation among them (Pearson values are close or above 0.8) but since there are few equal values we can suspect that they differ by a constant in most cases. The reasons can be, for example, that ckjm takes into account "external" dependencies (e.g., classes from *java.util*) or it counts coupling based on generated elements too (e.g., generated default constructor), but further investigation would be required to determine all causes.

CBOI: It counts those classes that use the given class. Although the basic statistics of *OSA* and *ckjm* are close to each other, its pairwise comparison suggests that they are different because there are large outliers and the averages of the diffs are commensurable with the averages of the tools. Based on Equal, Cohen, and Pearson values, it seems that the metric values of OSA and ckjm are close to each other but the metric values of *Moose* are different. The main reason can be, for example, that *OSA* found two times more classes; therefore, it is explicable that more classes use the given class or *ckjm* takes into account the generated classes and connections as well that exist in the bytecode, but not in the source code.

RFC: All three tools defined this metric in the same way but the comparison shows that the metric values are very different. The tools are able to calculate the same metric value for less than 10% of the classes (Equal value) but the high or almost high Pearson correlation

values indicate that there is some connection between the values. The reasons for this are mainly the same as in case of the CBO metric.

DIT: Although the statistical values "hardly" differ compared to the previous ones, these values are usually small (as the max. values show); therefore, these differences are quite large. From the minimal values, we can see that Moose probably counts `Object` too as the base class of all Java classes, while the other two tools neglect this. The only significant connection among them is the very large Paerson correlation (0.969) between OSA and Moose but its proper explanation would require a deeper investigation of the tools.

NOC: Regarding this metric, the three tools calculate very similar values. More than 95% of the metric values are the same for each paring which is very good. On the other hand, the remaining almost 5% of the values may differ significantly because the minimum and maximum values of the differences are large compared with the absolute maximum metric values and, at the same time, the Pearson correlation values are not extremely large, which means that the 5% impairs it a lot.

LOC: Lines of code should be the most unambiguous metric, but it also differs a lot. Although this metric has several variants and it is not defined exactly how Moose and ckjm counts it, we used the closest one from OSA based on the metric values. The very large value of ckjm is surprising, but it counts this value from the bytecode; therefore, it is not easy to validate it. Besides, OSA and Moose have different values, in spite of the fact that both of them calculate LOC from source code. The 0 minimal value of Moose is also interesting and suggests that either Moose used a different definition or the algorithm was not good enough. We found the fewest equal metric pairs for LOC metric (Equal values are 3.1%, 0.6%, 0.1%) but the large Pearson correlation values (close or above 0.9%) suggest that the metrics differ mainly by a constant value only.

NPM: Based on the statistical results, the number of public methods metrics seems to be the most unambiguous metric. Both the basic statistics and the Equal, Cohen, and Pearson triplet imply that the metrics are close to each other. OSA and ckjm are really close to each other (75% of the values are the same and the rest is also close to each other because the Pearson correlation coefficient is 0.999), while Moose has slightly different results. However, the average difference is around 1, which can be caused by counting implicit methods (constructors, static init blocks) or not.

The comparison of the three tools revealed that, even though they calculate the same metrics, in some cases, the results are very divergent. A few of the reasons can be that *ckjm* calculates metrics from bytecode while the other two tools work on source code, or *ckjm* takes into account external code as well while *OSA* does not. Besides, we could not compare the detailed and precise definitions of the metrics to be sure that they are really calculated in the same way; therefore, it is possible that they differ slightly which causes the differences.

4.7.2 File level metrics

Bugcatchers, Eclipse, and GitHub Bug Dataset are the ones that operate at file level (GitHub Bug Dataset contains class level too). Unfortunately, we could make only pairwise comparisons between file level metrics, since we could not replicate the measurements used in the Eclipse Bug Dataset (custom Eclipse JDT visitors were used) and in the Bugcatchers Bug Dataset (unknown bad smell detector was used).

In case of Bugcatchers Bug Dataset, we compared the results of OSA and the original metrics which were produced by a code smell detector. Since OSA only calculates a narrow set of file level metrics, Logical Lines of Code (LLOC) is the only metric we could use in this comparison. Table 13 presents the result of this comparison. Min, max, and median values are likely to be the same. Moreover, the average difference between LLOC values is less than 1 with a standard deviation of 6.05 which could be considered insignificant in case of LLOC at file level. Besides, more than 90% of the metric values are the same, and the remaining values are also close because the Cohen's d is almost 0 and the Pearson correlation coefficient is very close to 1. This means that the two tools calculate almost the same LLOC values.

There is an additional common metric (CLOC) which is not listed in Table 13 since OSA returned 0 values for all the files. This possible error in OSA makes it superfluous to examine CLOC in further detail.

In case of the Eclipse Bug Dataset, LLOC values are the same in most of the cases (see Table 14). OSA counted one extra line in 10 cases out of 25,210 and once it missed 7 lines which is a negligible difference. This is the cause of the "perfect" statistical values. Unfortunately, there is a serious sway in case of McCabe's Cyclomatic Complexity. There is a case where the difference is 299 in the calculated values, which is extremely high for this metric. We investigated these cases and found that OSA does not include the number of methods in the final value. There are many cases when OSA gives 1 as a result while the Eclipse Visitor calculates 0 as complexity. This is because OSA counts class definitions but not method definitions. There are cases where OSA provides higher complexity values. It turned out that OSA took the ternary operator (?:) into consideration, which is correct, since these statements also form conditions. Both calculation techniques seem to have some minor issues or at least we have to say that the metric definitions of cyclomatic complexity differ. This is why there are only so few matching values (4%) but the Cohen's d and the Pearson correlation coefficient suggest that these values are still related to each other.

The significant differences both at class and file levels show that tools interpret and implement the definitions of metrics differently. Our results further strengthen the conclusion reported by Lincke et al. (2008), who described similar findings.

5 Evaluation

In this section, we first evaluate the unified bug dataset's basic properties like the number of source code elements and the number of bugs to gain a rough overview of its contents. Next, we show the metadata of the dataset like the used code analyzer or the calculated metric set. Finally, we present an experiment in which we used the unified bug dataset for its main purpose, namely for bug prediction. Our aim was not to create the best possible bug prediction model, but to show that the dataset is indeed a usable source of learning data for researchers who want to experiment with bug prediction models.

5.1 Datasets and Bug Distribution

Table 15 shows the basic properties about each dataset. In the *SCE* column, the number of source code elements is presented. Based on the granularity, it means the number of classes

Table 13 Comparison of file level metrics in the Bugcatchers dataset

Metric	Tool	Met. name	Min	Max	Avg	Med	Dev	Equal	Cohen	Pearson
LLOC	OSA	LLOC	3	5,774	93.33	41	221.16	–	–	–
	Smell Detector	code	3	5,774	92.34	40	219.06	-	-	-
	OSA−Smell Detector	diff	-11	130	**0.98**	0	**6.05**	**90.8%**	**0.004**	**1.000**

Table 14 Comparison of file level metrics in the Eclipse dataset

Metric	Tool	Met. name	Min	Max	Avg	Med	Dev	Equal	Cohen	Pearson
LLOC	OSA	LLOC	3	5,228	122.59	52	230.02	–	–	–
	Visitor	TLOC	3	5,228	122.59	52	230.02	–	–	–
	OSA−Visitor	diff	-7	1	**0.0001**	0	**0.048**	**100.0%**	**0.000**	**1.000**
McCC	OSA	McCC	1	1,198	19.55	5	48.27	–	–	–
	Visitor	VG_sum	0	1,479	28.06	10	**60.35**	–	–	–
	OSA−Visitor	diff	**-299**	123	−8.50	-4	15.83	4.0%	**0.156**	**0.982**

Table 15 Basic properties of each dataset

Name	Granularity	SCE	SCEwBug	SCEwBug%	#Bug	kLLOC	Bug/kLine
Ant 1.3	class	125	20	16.00%	33	33	1.00
Ant 1.4	class	178	40	22.47%	47	43	1.09
Ant 1.5	class	293	32	10.92%	35	72	0.49
Ant 1.6	class	351	92	26.21%	184	98	1.88
Ant 1.7	class	745	166	22.28%	338	116	2.91
Camel 1.0	class	339	13	3.83%	14	26	0.54
Camel 1.2	class	590	216	36.61%	522	47	11.11
Camel 1.4	class	841	145	17.24%	335	76	4.41
Camel 1.6	class	928	188	20.26%	500	99	5.05
Ckjm 1.8	class	9	5	55.56%	23	8	2.88
Forrest 0.6	class	6	1	16.67%	1	19	0.05
Forrest 0.7	class	29	5	17.24%	15	4	3.75
Forrest 0.8	class	32	2	6.25%	6	3	2.00
Ivy 1.4	class	241	16	6.64%	18	31	0.58
Ivy 2.0	class	352	40	11.36%	56	54	1.04
JEdit 3.2	class	272	90	33.09%	382	55	6.95
JEdit 4.0	class	306	75	24.51%	226	63	3.59
JEdit 4.1	class	312	79	25.32%	217	72	3.01
JEdit 4.2	class	367	48	13.08%	106	88	1.20
JEdit 4.3	class	492	11	2.24%	12	109	0.11
Log4J 1.0	class	135	34	25.19%	61	10	6.10
Log4J 1.1	class	109	37	33.94%	86	14	6.14
Log4J 1.2	class	205	189	92.20%	498	23	21.65
Lucene 2.0	class	194	91	46.91%	268	68	3.94
Lucene 2.2	class	246	144	58.54%	414	111	3.73
Lucene 2.4	class	340	203	59.71%	632	126	5.02
Pbeans 1	class	26	20	76.92%	36	3	12.00
Pbeans 2	class	51	10	19.61%	19	6	3.17
Poi 1.5	class	237	141	59.49%	342	63	5.43
Poi 2.0	class	314	37	11.78%	39	82	0.48
Poi 2.5	class	385	248	64.42%	496	94	5.28
Poi 3.0	class	442	281	63.57%	500	140	3.57
Synapse 1.0	class	157	16	10.19%	21	20	1.05
Synapse 1.1	class	222	60	27.03%	99	33	3.00
Synapse 1.2	class	256	86	33.59%	145	46	3.15
Velocity 1.4	class	196	147	75.00%	210	26	8.08
Velocity 1.5	class	213	141	66.20%	330	33	10.00
Velocity 1.6	class	228	78	34.21%	190	37	5.14
Xalan 2.4	class	723	110	15.21%	156	104	1.50
Xalan 2.5	class	803	387	48.19%	531	126	4.21
Xalan 2.6	class	885	411	46.44%	625	154	4.06
Xalan 2.7	class	909	898	98.79%	1,213	160	7.58
Xerces 1.2	class	440	71	16.14%	115	65	1.77

Table 15 (continued)

Name	Granularity	SCE	SCEwBug	SCEwBug%	#Bug	kLLOC	Bug/kLine
Xerces 1.3	class	453	69	15.23%	193	69	2.80
Xerces 1.4	class	547	396	72.39%	1,554	74	21.00
Lucene 2.4	class	670	63	9.40%	96	126	0.76
Mylyn 3.1	class	1,405	209	14.88%	296	166	1.78
PDE UI 3.4.1	class	1,492	208	13.94%	340	186	1.83
Equinox 3.4	class	319	126	39.50%	239	64	3.73
Eclipse JDT Core 3.4	class	997	206	20.66%	374	630	0.59
Apache Com-mons	file	191	84	43.98%	223	53	4.21
ArgoUML 0.26 Beta	file	1,579	192	12.16%	285	186	1.53
Eclipse JDT Core 3.1	file	535	310	57.94%	567	1,594	0.36
Eclipse 2.0	file	6,729	2,610	38.79%	7,635	792	9.64
Eclipse 2.1	file	7,888	2,139	27.12%	4,975	988	5.04
Eclipse 3.0	file	10,593	2,913	27.50%	7,422	1,307	5.68
Android U. I. L. 1.7.0	class	73	20	27.40%	35	4	8.75
ANTLR v4 4.2	class	479	21	4.38%	27	40	0.68
Broadleaf C. 3.0.10	class	1,593	292	18.33%	353	125	2.82
Eclipse p. for Ceylon 1.1.0	class	1,610	68	4.22%	104	112	0.93
Elasticsearch 0.90.11	class	5,908	678	11.48%	1,182	362	3.27
Hazelcast 3.3	class	3,412	380	11.14%	1,053	179	5.88
JUnit 4.9	class	731	35	4.79%	41	16	2.56
MapDB 0.9.6	class	331	40	12.08%	96	47	2.04
mcMMO 1.4.06	class	301	57	18.94%	114	23	4.96
MCT 1.7b1	class	1,887	9	0.48%	9	104	0.09
Neo4j 1.9.7	class	5,899	58	0.98%	60	328	0.18
Netty 3.6.3	class	1,143	271	23.71%	423	76	5.57
OrientDB 1.6.2	class	1,847	280	15.16%	494	184	2.68
Oryx	class	533	74	13.88%	80	29	2.76
Titan 0.5.1	class	1,468	96	6.54%	106	80	1.33
Android U. I. L. 1.7.0	file	63	18	28.57%	33	4	8.25

Table 15 (continued)

Name	Granularity	SCE	SCEwBug	SCEwBug%	#Bug	kLLOC	Bug/kLine
ANTLR v4 4.2	file	411	41	9.98%	59	40	1.48
Broadleaf C. 3.0.10	file	1,719	286	16.64%	350	125	2.80
Eclipse p. for Ceylon 1.1.0	file	699	57	8.15%	94	112	0.84
Elasticsearch 0.90.11	file	3,035	487	16.05%	899	362	2.48
Hazelcast 3.3	file	2,228	317	14.23%	911	179	5.09
JUnit 4.9	file	308	42	13.64%	50	16	3.13
MapDB 0.9.6	file	137	22	16.06%	59	47	1.26
mcMMO 1.4.06	file	267	57	21.35%	114	23	4.96
MCT 1.7b1	file	413	6	1.45%	6	104	0.06
Neo4j 1.9.7	file	3,278	32	0.98%	33	328	0.10
Netty 3.6.3	file	913	243	26.62%	381	76	5.01
OrientDB 1.6.2	file	1,503	270	17.96%	493	184	2.68
Oryx	file	280	44	15.71%	48	29	1.66
Titan 0.5.1	file	975	70	7.18%	80	80	1.00

or files in the system. There are systems in the datasets with a wide variety of size from 2,636 *Logical Lines of Code (LLOC)*[8] up to 1,594,471.

SCEwBug means the number of source code elements which have at least one bug; *SCEwBug%* is the percentage of the source code elements with bugs in the dataset. The *SCEwBug%* as the percentage of buggy classes or files describes how well-balanced the datasets are. Since it is difficult to overview the numbers, Figs. 1 and 2 show the distribution of the percentages of faulty source code elements (SCEwBug%) for classes and files, respectively. The percentages are shown horizontally, and, for example, the first column means the number of systems (part of a dataset) that have between 0 and 10 percentages of their source code elements buggy (0 included, 10 excluded). In case of the systems that give bug information at class level, this number is 11 (5 at file level). We can see that there are systems where the percentages of the buggy classes are very high, for example, 98.79% of the classes are buggy for Xalan 2.7 or 92.20% for Log4J 1.2. Although the upper limit of SCEwBug% is 100%, the reader may feel that these values are extremely high and it is very difficult to believe that a release version of a system can contain so many bugs. The other end is when a system hardly contains any bug, for example, in case of MCT and Neo4j, less than 1% of the classes is buggy. From the project point of view, it is very good but on the other hand, these systems are probably less usable when we want to build bug prediction models. This phenomenon further strengthens the motivation to have a common unified bug dataset which can blur these extreme outliers. Although there are many systems with extremely high or low SCEwBug% values, we used them "as is" later in this research because their validation was not the aim of this work.

[8]Lines of code not counting comments and whitespace.

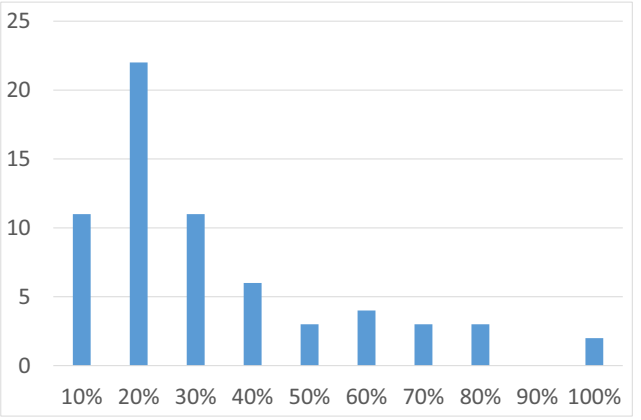


Fig. 1 Fault distribution (classes)

5.2 Meta data of the Datasets

Table 16 lists some properties of the datasets, which show the circumstances of the dataset, rather than the data content. Our focus is on how the datasets were created, how reliable the used tools and the applied methods were. Since most of the information in the table was already described in previous sections (Analyzer, Granularity, Metrics, and Release), in this section, we will describe only the Bug information row.

The Bug Prediction Dataset used the commit logs of SVN and the modification time of each file in CVS to collect co-changed files, authors, and comments. Then, they D'Ambros et al. (2010) linked the files with bugs from Bugzilla and Jira using the bug id from the commit messages. Finally, they verified the consistency of timestamps. They filtered out inner classes and test classes.

The PROMISE dataset used Buginfo to collect whether an SVN or CVS commit is a bugfix or not. Buginfo uses regular expressions to detect commit messages which contain bug information.

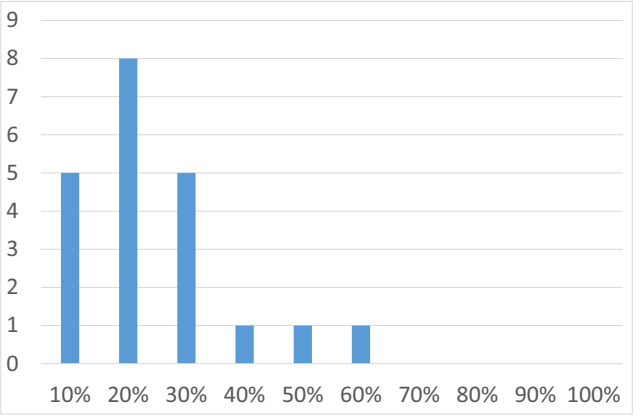


Fig. 2 Fault distribution (files)

Table 16 Meta data of the datasets

	Bug Prediction Dataset	PROMISE	Eclipse Bug Dataset	Bugcatchers Bug Dataset	GitHub Bug Dataset
Analyzer	inFusion Moose	ckjm	Visitors written for Java parser of Eclipse	Bad Smell Detector	SourceMeter
Granularity	Class	Class	File	File	Class, File
Bug information	CVS, SVN, Bugzilla, Jira	SVN, CVS	CVS, Bugzilla	CVS, SVN, Bugzilla, Jira	GitHub
Metrics	C&K, process metrics	C&K	Complexity, Structure of abstract syntax tree	Bad Smell	C&K, Complexity, Clone, Rule violation
Release	post	pre	pre & post	pre	post

The bug information of the Eclipse Bug Dataset was extracted from the CVS repository and Bugzilla. In the first step, they identified the corrections or fixes in the version history by looking for patterns which are possible references to bug entries in Bugzilla. In the second step, they mapped the bug reports to versions using the version field of the bug report. Since the version of a bug report can change during the life cycle of a bug, they used the first version.

The Bugcatchers Bug Dataset followed the methodology of Zimmermann et al. (Eclipse Bug Dataset). They developed an Ant script using the SVN and CVS plugins of Ant to checkout the source code and associate each fault with a file.

The authors of the GitHub bug dataset gathered the relevant versions to be analyzed from GitHub. Since GitHub can handle references between commits and issues, it was quite handy to use this information to match commits with bugs. They collected the number of bugs located in each file/class for the selected release versions (about 6-month-long time intervals).

5.3 Bug Prediction

We evaluated the strength of bug prediction models built with the Weka (Hall et al. 2009) machine learning software. For each subject software system in the Unified Bug Dataset, we created 3 versions of ARFF files (which is the default input format of Weka) for the experiments (containing only the original, only OSA, and both set of metrics as predictors). In these files, we transformed the original bug occurrence values into two classes as follows: 0 bug → non buggy class, at least 1 bug occurrence → buggy class. Using these ARFF files, we could run several tests about the effectiveness of fault prediction models built based on the dataset.

5.3.1 Within-project Bug Prediction

As described in Section 4, we extended the original datasets with the source code metrics of the OSA tool and we created a unified bug dataset. We compared the bug prediction capabilities of the original metrics, the OSA metrics, and the extended metric suite (both metric suites together). First, we handled each system individually, so we trained and tested on the same system data using ten-fold cross-validation. To build the bug prediction models, we used the J48 (C4.5 decision tree) algorithm with default parameters. We used only J48 since we did not focus on finding the best machine learning method, but we rather wanted to show a comparison of the different predictors' capabilities with this one algorithm. We chose J48, because it has shown its power in case of the GitHub Bug Dataset (Tóth et al. 2016) and because it is relatively easy to interpret the resulting decision trees and to identify the subset of metrics which are the most important predictors of bugs. Different machine learning algorithms (e.g., neural networks) might provide different results. It is also important to note that we did not use any sampling technique to balance the datasets, we used the datasets 'as is'. We will outline some connections between the machine learning results and the characteristics of the datasets (e.g., SCEwBug%).

The F-measure[9] results can be seen in Table 17 for classes and in Table 18 for files. We also included the *SCEwBug%* characteristic of each dataset to gain a more detailed view

[9]F-measure is the harmonic mean of precision and recall.

Table 17 F-Measure and AUC values in independent training at class level

	Original		OSA		Merged		
Dataset	F-Measure	AUC	F-Measure	AUC	F-Measure	AUC	SCEwBug%
Eclipse JDT Core 3.4	0.665	0.687	0.819	0.699	0.817	0.637	9.40
Equinox 3.4	0.727	0.711	0.764	0.793	0.790	0.798	14.88
Lucene 2.4 BPD	0.891	0.635	0.878	0.611	0.879	0.443	13.94
Mylyn 3.1	0.932	0.635	0.806	0.600	0.813	0.610	39.50
PDE UI 3.4.1	0.856	0.638	0.820	0.608	0.816	0.592	20.66
Ant 1.3	0.799	0.603	0.846	0.747	0.827	0.707	16.00
Ant 1.4	0.724	0.681	0.717	0.603	0.726	0.616	22.47
Ant 1.5	0.887	0.579	0.854	0.562	0.871	0.571	10.92
Ant 1.6	0.775	0.665	0.752	0.704	0.745	0.663	26.21
Ant 1.7	0.794	0.727	0.788	0.652	0.788	0.654	22.28
Camel 1.0	0.943	0.433	0.946	0.494	0.932	0.391	3.83
Camel 1.2	0.626	0.624	0.673	0.627	0.686	0.647	36.61
Camel 1.4	0.805	0.637	0.806	0.594	0.814	0.612	17.24
Camel 1.6	0.744	0.597	0.767	0.628	0.776	0.661	20.26
Ckjm	-	-	-	-	-	-	55.56
Forrest 0.6	-	-	-	-	-	-	16.67
Forrest 0.7	0.793	0.737	0.676	0.279	0.676	0.320	17.24
Forrest 0.8	0.891	0.900	0.907	0.100	0.907	0.700	6.25
Ivy 1.4	0.897	0.493	0.915	0.547	0.899	0.511	6.64
Ivy 2.0	0.855	0.624	0.835	0.517	0.847	0.585	11.36
Jedit 3.2	0.727	0.695	0.703	0.710	0.724	0.689	33.09
Jedit 4.0	0.774	0.662	0.766	0.655	0.780	0.654	24.51
Jedit 4.1	0.747	0.640	0.778	0.698	0.781	0.753	25.32
Jedit 4.2	0.866	0.634	0.847	0.650	0.849	0.631	13.08
Jedit 4.3	0.967	0.469	0.965	0.620	0.780	0.608	2.24
Log4J 1.0	0.776	0.637	0.835	0.759	0.781	0.687	25.19
Log4J 1.1	0.739	0.681	0.743	0.718	0.849	0.696	33.94
Log4J 1.2	0.892	0.621	0.885	0.421	0.963	0.495	92.20
Lucene 2.0	0.610	0.571	0.659	0.635	0.634	0.626	46.91
Lucene 2.2	0.584	0.631	0.655	0.657	0.661	0.638	58.54
Lucene 2.4	0.698	0.689	0.656	0.688	0.663	0.676	59.71
Pbeans 1	0.813	0.633	0.769	0.741	0.813	0.779	76.92
Pbeans 2	0.727	0.565	0.686	0.503	0.740	0.553	19.61
Poi 1.5	0.743	0.744	0.753	0.747	0.782	0.778	59.49
Poi 2.0	0.851	0.465	0.841	0.729	0.831	0.576	11.78
Poi 2.5	0.787	0.792	0.809	0.788	0.789	0.791	64.42
Poi 3.0	0.768	0.763	0.768	0.768	0.764	0.759	63.57
Synapse 1.0	0.830	0.524	0.843	0.531	0.839	0.576	10.19
Synapse 1.1	0.758	0.683	0.767	0.717	0.749	0.672	27.03

Table 17 (continued)

	Original		OSA		Merged		
Dataset	F-Measure	AUC	F-Measure	AUC	F-Measure	AUC	SCEwBug%
Synapse 1.2	0.738	0.670	0.734	0.677	0.733	0.663	33.59
Velocity 1.4	0.827	0.744	0.824	0.733	0.856	0.796	75.00
Velocity 1.5	0.706	0.660	0.728	0.696	0.778	0.742	66.20
Velocity 1.6	0.703	0.649	0.774	0.783	0.748	0.730	34.21
Xalan 2.4	0.802	0.621	0.807	0.645	0.794	0.562	15.21
Xalan 2.5	0.654	0.648	0.693	0.686	0.699	0.718	48.19
Xalan 2.6	0.751	0.747	0.730	0.707	0.747	0.750	46.44
Xalan 2.7	0.994	0.654	0.992	0.820	0.992	0.745	98.79
Xerces 1.2	0.815	0.716	0.824	0.683	0.823	0.716	16.14
Xerces 1.3	0.860	0.629	0.837	0.625	0.836	0.743	15.23
Xerces 1.4	0.938	0.904	0.804	0.815	0.932	0.935	72.39
Average	**0.793**	**0.653**	**0.793**	**0.646**	**0.798**	**0.656**	-
Android U. I. L. 1.7.0	0.749	0.657	0.742	0.677	-	-	27.40
ANTLR v4 4.2	0.944	0.618	0.922	0.555	-	-	4.38
Broadleaf C. 3.0.10	0.898	0.826	0.881	0.796	-	-	18.33
Eclipse p. for Ceylon 1.1.0	0.942	0.611	0.939	0.586	-	-	4.22
Elasticsearch 0.90.11	0.871	0.645	0.874	0.683	-	-	11.48
Hazelcast 3.3	0.876	0.693	0.878	0.658	-	-	11.14
jUnit 4.9	0.927	0.579	0.928	0.624	-	-	4.79
MapDB 0.9.6	0.911	0.659	0.886	0.673	-	-	12.08
mcMMO 1.4.06	0.791	0.593	0.776	0.568	-	-	18.94
MCT 1.7b1	0.993	0.480	0.993	0.478	-	-	0.48
Neo4j 1.9.7	0.985	0.494	0.985	0.486	-	-	0.98
Netty 3.6.3	0.809	0.699	0.802	0.699	-	-	23.71
OrientDB 1.6.2	0.868	0.687	0.862	0.715	-	-	15.16
Oryx	0.889	0.764	0.898	0.725	-	-	13.88
Titan 0.5.1	0.922	0.696	0.926	0.706	-	-	6.54
Average	**0.892**	**0.647**	**0.886**	**0.642**	**-**		-

of the results. The tables contain two average values since the GitHub Bug Dataset used SourceMeter, which is based on OSA to calculate the metrics.

The results of OSA and the Merged metrics would be the same. This could distort the averages, so we decided to detach this dataset from others when calculating the averages.

Table 18 F-Measure and AUC values in independent training at file level

	Original		OSA		Merged		
Dataset	F-Measure	AUC	F-Measure	AUC	F-Measure	AUC	SCEwBug%
Apache Commons	0.779	0.733	0.454	0.536	0.712	0.778	43.98
Argo UML 0.26 Beta	0.899	0.730	0.832	0.677	0.880	0.719	12.16
Eclipse JDT Core 3.1	0.827	0.657	0.528	0.567	0.638	0.658	57.94
Eclipse 2.0	0.682	0.654	0.670	0.713	0.694	0.660	38.79
Eclipse 2.1	0.749	0.623	0.750	0.711	0.745	0.639	27.12
Eclipse 3.0	0.728	0.597	0.735	0.694	0.727	0.603	27.50
Average	**0.777**	**0.666**	**0.662**	**0.650**	**0.733**	**0.676**	**-**
Android U. I. L. 1.7.0	0.611	0.484	0.624	0.654	-	-	28.57
ANTLR v4 4.2	0.900	0.795	0.849	0.753	-	-	9.98
Broadleaf C. 3.0.10	0.862	0.759	0.823	0.795	-	-	16.64
Eclipse p. for Ceylon 1.1.0	0.879	0.639	0.874	0.574	-	-	8.15
Elasticsearch 0.90.11	0.775	0.669	0.778	0.678	-	-	16.05
Hazelcast 3.3	0.829	0.688	0.796	0.626	-	-	14.23
jUnit 4.9	0.800	0.585	0.801	0.592	-	-	13.64
MapDB 0.9.6	0.784	0.686	0.766	0.654	-	-	16.06
mcMMO 1.4.06	0.731	0.669	0.794	0.614	-	-	21.35
MCT 1.7b1	0.977	0.291	0.978	0.551	-	-	1.45
Neo4j 1.9.7	0.985	0.474	0.985	0.474	-	-	0.98
Netty 3.6.3	0.677	0.700	0.732	0.762	-	-	26.62
OrientDB 1.6.2	0.739	0.759	0.751	0.722	-	-	17.96
Oryx	0.771	0.467	0.861	0.735	-	-	15.71
Titan 0.5.1	0.894	0.498	0.894	0.498	-	-	7.18
Average	**0.814**	**0.611**	**0.820**	**0.646**	**-**	**-**	**-**

Results for classes need some deeper explanation for clear understanding. There are a few missing values, since there were less than 10 data entries; not enough to do the ten-fold cross-validation on (Ckjm and Forrest-0.6).

There are some outstanding numbers in the tables. Considering Xalan 2.7, for example, we can see F-measure values with 0.992 and above. This is the consequence of the distribution of the bugs in that dataset which is extremely high as well (98.79%). The reader could

argue that resampling the dataset would help to overcome this deficiency; however, the corresponding AUC value[10] for the Original dataset is not outstanding (only 0.654). Besides, later in this section, we will investigate the bug prediction capabilities of the datasets in a cross-project manner and it turns out that these extreme outliers generally performed poor in cross-project learning as we will describe it later in detail. Let us consider Forrest 0.8 as an example where the *SCEwBug%* is low (6.25); however, the F-measure values are still high (0.891-0.907). In this case, the high values came from the fact that there are 32 classes out of which only 2 are buggy, so the decision tree tends to mark all classes as non buggy. On the other hand, the AUC values range from very poor (0.100) to very high (0.900) which suggest that for small examples and small number of bugs, it is difficult build "reliable" models.

The averages of F-measure and AUC slightly change in case of class level datasets and the average of the merged dataset is only slightly better than the Original or the OSA. If we consider F-measure, the GitHub Bug Dataset performed 10% better generally, while the averages of UAC decreased by only 1% which is neglectable. One possible explanation can be that the PROMISE dataset includes all the smaller projects, while the GitHub Bug Dataset and the Bug Prediction Dataset rather contain larger projects.

Small differences in case of the GitHub Bug Dataset come from the difference of the versions of the static analyzers. The SourceMeter version used in creating the GitHub Bug Dataset is older than the OSA version used here, in which some metric calculation enhancements took place. (SourceMeter is based on OSA.)

Results at file level are quite similar to class level results in terms of F-measure and AUC. The only main difference is that the F-measure averages of OSA for Bugcatchers and Eclipse bug datasets are notably smaller than the other values. The reason for this might be that there are only a few file level metrics provided by OSA, and a possible contradiction in metrics can decrease the training capabilities (we saw that even LLOC values are very different). On the other hand, the corresponding AUC value is close to the others so deeper investigation is required to find out the causes. Besides, the average F-measure of the Merged model is slightly worse while the average AUC is slightly better than the Original which means that in this case the OSA metrics were not able to improve the bug prediction capability of the model. The small difference between the Original and the OSA results in case of the GitHub Bug Dataset comes from the slightly different metric suites, since OSA calculates the Public Documented API (PDA) and Public Undocumented API (PUA) metrics as well. Furthermore, the aforementioned static analyzer version differences also caused this small change in the average F-measure and AUC values.

5.3.2 Merged Dataset Bug Prediction

So far, we compared the bug prediction capabilities of the old and new metric suites on the systems separately, but we have not used its main advantage, namely, there are metrics that were calculated in the same way for all systems (OSA metric suite). Seeing the results of the within-project bug prediction, we can state that creating a larger dataset, which includes projects varying in size and domain, could lead to a more general dataset with increased usability and reliability. Therefore, we merged all classes (and files) into one large dataset which consists of 47,618 elements (43,744 for files) and by using 10-fold cross validation,

[10] AUC is the area under the ROC curve, which is a measure of how well a parameter can distinguish between two diagnostic groups.

Table 19 Most dominant predictors per dataset

Dataset	Level	Dominant predictors
Bug Prediction Dataset	class	**wmc**, **TNOS**, cvsEntropy
GitHub Bug Dataset	class	**WMC**, NOA, **TNOS**
PROMISE	class	LOC, DIT, TNM
Unified Bug Dataset	class	CLOC, TCLOC, CBO, NOI, DIT
Bugcatchers Bug Dataset	file	**code**, PDA, SpeculativeGenerality
GitHub Bug Dataset	file	**McCC**, NumOfPrevMods, NumOfDevCommits
Eclipse Bug Dataset	file	TypeLiteral, NSF_max, **MLOC_sum**
Unified Bug Dataset	file	**LOC**, **McCC**

we evaluated the bug prediction model built by J48 on this large dataset as well. The F-measure is 0.818 for the classes and 0.755 for the files while the AUC is 0.721 for classes and 0.699 for files. Although the F-measure values are, for example, a little worse than the average of GitHub results (0.892 for classes and 0.820 for files), the AUC values are better than the best averages (0.656 for class and 0.676 for files) even if they were measured on a much larger and very heterogeneous dataset.

After training the models at class and file levels, we dug deeper to see which predictors are the most dominant ones. Weka gives us pruned trees as results both at class and file levels. A pruned tree is a transformed one obtained by removing nodes and branches without affecting the performance too much. The main goal of the pruning is to reduce the risk of overfitting. We not only constructed a Unified Bug Dataset for all of the systems together with the unified set of metrics, but we also built one for each collected dataset, one for the Bug Prediction Dataset, one for the PROMISE dataset, etc. in which we could use a wider set of metrics (including the original ones).

Table 19 presents the most dominant metrics for the datasets extracted from the constructed decision trees. We marked those metrics with bold face that occur in multiple datasets. At class level, WMC (Weighted Methods per Class) and TNOS (Total Number of Statements) are the most important ones; however, they happened not to be the most dominant ones in the Unified Bug Dataset.

Regarding the Unified Bug Dataset, we found CLOC, TCLOC (comment lines of code metrics), DIT (Depth of Inheritance), CBO (Coupling Between Objects) and NOI (Number of Outgoing Invocations) as the most dominant metrics at class level. In other words, these metrics have the largest entropy. This set of metrics being the most dominant is reasonable, since it is easier to modify and extend classes that have better documentation. Furthermore, coupling metrics such as CBO and NOI have already demonstrated their power in bug prediction (Gyimóthy et al. 2005b; Briand et al. 1999). DIT is also an expressive metric for fault prediction (El Emam et al. 2001).

At file level, the most dominant predictors in the different datasets show overlapping with the ones being dominant in the Unified Bug Dataset. LOC (Lines of Code) and McCC (McCabe's Cyclomatic Complexity) were the upmost variables to branch on in the decision tree. These two metrics are reasonable as well, since the larger the file, the more it tends to be faulty. Complexity metrics are also important factors to include in fault prediction (Zimmermann et al. 2007).

The diversity of the most dominant metrics shows the diversity of the different datasets themselves.

Table 20 Cross training (PROMISE - Class level) - F-Measure values

Project	SCEwBug%	Avg. F-m.	Ant 1.3	Camel 1.0	Ckjm 1.8	Forrest 0.6	Ivy 1.4	Jedit 3.2	Log4J 1.0	Log4J 1.2	Lucene 2.0	Lucene 2.2
Ant 1.3	16.0	0.71		0.830	0.776	0.776	0.785	0.784	0.794	0.762	0.754	0.739
Camel 1.0	3.8	0.66	0.755		0.719	0.719	0.732	0.731	0.747	0.711	0.701	0.684
Ckjm 1.8	*55.6*	*0.49*	0.307	0.442		0.488	0.481	0.482	0.496	0.495	0.496	0.497
Forrest 0.6	16.7	0.63	0.767	0.741	0.697		0.712	0.710	0.728	0.691	0.681	0.663
Ivy 1.4	6.6	0.67	0.807	0.756	0.714	0.714		0.735	0.748	0.715	0.707	0.692
Jedit 3.2	33.2	0.69	0.733	0.746	0.719	0.718	0.728		0.750	0.725	0.721	0.711
Log4J 1.0	25.2	0.70	0.753	0.731	0.722	0.722	0.733	0.734		0.723	0.718	0.713
Log4J 1.2	**92.2**	**0.20**	0.060	0.118	0.180	0.179	0.177	0.179	0.163		0.183	0.189
Lucene 2.0	46.9	0.67	0.750	0.718	0.696	0.696	0.697	0.698	0.700	0.680		0.680
Lucene 2.2	*58.5*	*0.47*	0.311	0.369	0.428	0.428	0.446	0.463	0.476	0.477	0.480	
Pbeans 1	**76.9**	**0.36**	0.320	0.319	0.352	0.353	0.350	0.345	0.329	0.336	0.341	0.346
Pbeans 2	19.6	0.71	0.737	0.769	0.747	0.747	0.759	0.755	0.767	0.743	0.737	0.728
Poi 1.5	*59.5*	*0.55*	0.394	0.538	0.555	0.555	0.550	0.554	0.578	0.571	0.569	0.567
Poi 2.0	11.8	0.66	0.734	0.742	0.705	0.705	0.714	0.712	0.726	0.699	0.691	0.678
Synapse 1.0	10.2	0.65	0.772	0.745	0.706	0.706	0.720	0.720	0.734	0.699	0.691	0.675
Velocity 1.4	**75.0**	**0.27**	0.223	0.209	0.232	0.233	0.229	0.229	0.252	0.265	0.267	0.271
Velocity 1.6	34.2	0.67	0.700	0.693	0.697	0.698	0.707	0.707	0.697	0.681	0.677	0.669
Xalan 2.4	15.2	0.69	0.792	0.767	0.718	0.718	0.732	0.741	0.760	0.728	0.721	0.705
Xalan 2.7	**98.8**	**0.10**	0.044	0.054	0.073	0.073	0.066	0.068	0.062	0.078	0.083	0.092
Xerces 1.2	16.1	0.64	0.734	0.723	0.677	0.677	0.692	0.693	0.711	0.681	0.671	0.655
Xerces 1.4	**72.4**	**0.29**	0.353	0.280	0.278	0.279	0.277	0.276	0.261	0.271	0.273	0.278

Table 20 (continued)

Pbeans 1	Pbeans 2	Poi 1.5	Poi 2.0	Synapse 1.0	Velocity 1.4	Velocity 1.6	Xalan 2.4	Xalan 2.7	Xerces 1.2	Xerces 1.4
0.718	0.718	0.706	0.711	0.674	0.660	0.648	0.659	0.590	0.596	0.588
0.660	0.661	0.648	0.655	0.618	0.604	0.593	0.608	0.524	0.532	0.524
0.500	0.500	0.503	0.493	0.507	0.514	0.513	0.511	0.510	0.505	0.501
0.637	0.638	0.623	0.631	0.587	0.573	0.562	0.576	0.487	0.496	0.487
0.671	0.672	0.658	0.665	0.622	0.610	0.601	0.613	0.533	0.540	0.532
0.696	0.697	0.691	0.691	0.678	0.665	0.655	0.664	0.584	0.589	0.579
0.703	0.704	0.696	0.697	0.677	0.672	0.668	0.671	0.624	0.627	0.621
0.200	0.199	0.205	0.198	0.220	0.230	0.236	0.224	0.271	0.268	0.273
0.672	0.673	0.671	0.664	0.665	0.655	0.651	0.656	0.619	0.620	0.615
0.496	0.496	0.497	0.496	0.495	0.495	0.494	0.483	0.497	0.488	0.485
	0.358	0.363	0.358	0.375	0.378	0.386	0.378	0.401	0.393	0.395
0.714		0.709	0.709	0.687	0.676	0.668	0.674	0.620	0.623	0.614
0.564	0.564		0.569	0.575	0.576	0.574	0.572	0.562	0.557	0.554
0.663	0.664	0.652		0.630	0.621	0.614	0.622	0.560	0.565	0.557
0.652	0.652	0.638	0.645		0.594	0.582	0.596	0.509	0.516	0.509
0.277	0.277	0.280	0.283	0.298		0.323	0.316	0.347	0.343	0.343
0.660	0.661	0.660	0.661	0.645	0.641		0.642	0.624	0.620	0.614
0.684	0.685	0.677	0.683	0.655	0.642	0.632		0.599	0.604	0.594
0.105	0.105	0.113	0.109	0.134	0.142	0.149	0.142		0.199	0.205
0.632	0.632	0.618	0.626	0.587	0.576	0.567	0.579	0.513		0.519
0.286	0.287	0.292	0.288	0.302	0.306	0.312	0.314	0.343	0.343	

5.3.3 Cross-project Bug Prediction

As a third experiment, we trained a model using only one system from a dataset and tested it on all systems in that dataset. This experiment could not have been done without the unification of the datasets, since a common metric suite is needed to perform such machine learning tasks. The result of the cross training is an NxN matrix where the rows and the columns of the matrix are the systems of the dataset and the value in the i^{th} row and j^{th} column shows how well the prediction model performed, which was trained on the i^{th} system and was evaluated on the j^{th} one.

We used the OSA metrics to test this criterion, but the bug occurrences are derived from the original datasets which are transformed into buggy and non buggy labels. The matrix for the whole Unified Bug Dataset would be too large to show here; thus, we will only present submatrices (the full matrices for file and class levels are available in the Appendix). A submatrix for the PROMISE dataset can be seen in Tables 20 and 21. Only the first version is presented for each project except for the ones where an other version has significantly different prediction capability (for example, Xalan 2.7 is much worse than 2.4 or Pbeans 2 is much better than 1). The values of the matrix are F-measure and AUC values, provided by the J48 algorithm. Avg. F-m. and Avg. AUC columns[11] present the average of the F-measures and AUC values the given model achieved on the other systems. For a better overview, we repeat SCEwBug% value here as well (see Table 15).

We can observe (see Table 20) that models built on systems (highlighted bold in the table) having lots of buggy classes (more than 70%) performed very poor in the cross validation if we consider the average F-measure (the values are under 0.4). Even more, these models do not work well on other very buggy systems either, for example, model trained on Xalan 2.7 (the most buggy system) achieved only 0.078 F-measure on Log4J 1.2 (the second most buggy system). Besides, in case of Ckjm 1.8, Lucene 2.2, and Poi 1.5, the rate of buggy classes is between 50 and 70 percent, and their models are still weak, the average F-measure is between 0.4 and 0.6 only. And finally, the other systems can be used to build such models that achieve better result, namely higher F-measures, on other systems, even the ones having lots of bugs. On the other hand, this trend cannot be observed on the AUC values (see Table 21), because they range between 0.483 and 0.629 but there is no "white line" which means that there is no model whose performance is very poor for all other systems. However, there are 0.000 and 0.900 AUC values in the table that suggest that the bug prediction capabilities heavily depend on the training and testing dataset. For example, models evaluated on Forrest 0.6 have extreme values, and it is probably only a matter of luck whether the model takes into account the appropriate metrics or not.

Table 22 shows the cross training F-measure values for the GitHub Bug Dataset. Testing on Android Universal Image Loader is the weakest point in the matrix, as it is clearly visible. However, the values are not critical, the lowest value is still 0.611. Based on F-measure, Elasticsearch performed slightly better than the others in the role of a training set. This might be because of the size of the system, the average amount of bugs, and the adequate number of entries in the dataset. On the other hand, the AUC values in the column of Android Universal Image Loader are not significantly worse than any other value (see Table 23), but rather the values in its row seems a little bit lower. From AUC point of view, Mission Control T. is the most critical test system because it has the lowest (0.181) and the highest

[11] We present the average values only for PROMISE dataset because it contains extremely low values, namely almost white lines.

Table 21 Cross training (PROMISE - Class level) - AUC values

Project	SCEwBug%	Avg. AUC	Ant 1.3	Camel 1.0	Ckjm 1.8	Forrest 0.6	Ivy 1.4	Jedit 3.2	Log4J 1.0	Log4J 1.2	Lucene 2.0	Lucene 2.2
ant-1.3	16.0	0.524		0.395	0.500	0.500	0.812	0.578	0.456	0.476	0.510	0.468
camel-1.0	3.8	0.537	0.516		0.500	0.500	0.658	0.506	0.657	0.556	0.544	0.565
ckjm-1.8	*55.6*	0.585	0.520	0.656		0.200	0.558	0.606	0.699	0.703	0.610	0.583
forrest-0.6	16.7	0.500	0.500	0.500	0.500		0.500	0.500	0.500	0.500	0.500	0.500
ivy-1.4	6.6	0.629	0.767	0.576	0.600	0.500		0.690	0.692	0.587	0.661	0.607
jedit-3.2	33.2	0.580	0.707	0.593	0.650	0.400	0.719		0.762	0.584	0.634	0.545
log4j-1.0	25.2	0.574	0.695	0.475	0.400	0.400	0.523	0.568		0.575	0.623	0.636
log4j-1.2	**92.2**	0.483	0.531	0.446	0.475	0.600	0.510	0.493	0.533		0.552	0.562
lucene-2.0	46.9	0.592	0.781	0.594	0.500	0.500	0.814	0.684	0.546	0.390		0.609
lucene-2.2	*58.5*	0.614	0.538	0.659	0.900	0.600	0.637	0.532	0.680	0.527	0.648	
pbeans-1	**76.9**	0.558	0.595	0.564	0.625	0.300	0.589	0.582	0.585	0.469	0.598	0.580
pbeans-2	19.6	0.543	0.511	0.454	0.500	0.600	0.561	0.565	0.533	0.443	0.578	0.580
poi-1.5	*59.5*	0.551	0.551	0.598	0.875	0.400	0.426	0.553	0.646	0.583	0.517	0.528
poi-2.0	11.8	0.532	0.546	0.665	0.500	0.400	0.722	0.444	0.487	0.388	0.495	0.542
synapse-1.0	10.2	0.509	0.526	0.556	0.700	0.400	0.326	0.489	0.624	0.537	0.487	0.567
velocity-1.4	**75.0**	0.485	0.444	0.332	0.600	0.900	0.317	0.442	0.642	0.596	0.453	0.474
velocity-1.6	34.2	0.592	0.499	0.630	0.825	0.000	0.670	0.573	0.681	0.559	0.628	0.617
xalan-2.4	15.2	0.605	0.728	0.541	0.600	0.500	0.501	0.697	0.743	0.571	0.649	0.560
xalan-2.7	**98.8**	0.511	0.514	0.522	0.500	0.500	0.507	0.544	0.535	0.487	0.469	0.516
xerces-1.2	16.1	0.493	0.471	0.570	0.250	0.500	0.683	0.518	0.543	0.476	0.451	0.479
xerces-1.3	**72.4**	0.529	0.595	0.511	0.575	0.000	0.629	0.594	0.518	0.436	0.550	0.506

Table 21 (continued)

Pbeans 1	Pbeans 2	Poi 1.5	Poi 2.0	Synapse 1.0	Velocity 1.4	Velocity 1.6	Xalan 2.4	Xalan 2.7	Xerces 1.2	Xerces 1.4
0.554	0.444	0.540	0.535	0.513	0.538	0.509	0.552	0.532	0.505	0.560
0.421	0.638	0.476	0.497	0.511	0.504	0.515	0.580	0.550	0.538	0.512
0.575	0.633	0.642	0.543	0.620	0.561	0.573	0.620	0.783	0.495	0.522
0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
0.575	0.676	0.626	0.648	0.690	0.523	0.666	0.728	0.676	0.523	0.571
0.525	0.621	0.606	0.549	0.650	0.499	0.489	0.550	0.455	0.578	0.489
0.550	0.674	0.510	0.668	0.516	0.563	0.604	0.694	0.594	0.641	0.574
0.467	0.344	0.499	0.470	0.504	0.398	0.454	0.490	0.408	0.495	0.422
0.575	0.534	0.641	0.578	0.564	0.451	0.569	0.699	0.689	0.571	0.556
0.850	0.463	0.626	0.584	0.622	0.630	0.539	0.597	0.712	0.398	0.541
	0.705	0.600	0.572	0.526	0.514	0.619	0.571	0.559	0.465	0.550
0.625		0.579	0.565	0.521	0.488	0.589	0.587	0.565	0.527	0.485
0.288	0.570		0.447	0.620	0.488	0.574	0.566	0.722	0.430	0.635
0.538	0.761	0.544		0.459	0.683	0.592	0.454	0.492	0.486	0.436
0.475	0.513	0.534	0.540		0.525	0.474	0.509	0.491	0.426	0.488
0.625	0.393	0.431	0.442	0.473		0.533	0.333	0.427	0.411	0.429
0.775	0.533	0.652	0.540	0.709	0.470		0.619	0.820	0.461	0.580
0.600	0.677	0.644	0.646	0.687	0.468	0.618		0.619	0.527	0.516
0.475	0.561	0.535	0.518	0.514	0.524	0.494	0.511		0.515	0.484
0.521	0.570	0.410	0.562	0.480	0.512	0.541	0.508	0.348		0.464
0.721	0.644	0.581	0.527	0.582	0.498	0.525	0.533	0.399	0.547	0.636

Table 22 Cross training (GitHub – Class level) - F-Measure values

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Commerce	Eclipse p. for Ceylon	Elastic search	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.		0.885	0.772	0.822	0.813	0.821	0.827	0.827	0.825	0.836	0.868	0.859	0.852	0.850	0.853
ANTLR v4	0.611		0.785	0.852	0.843	0.842	0.847	0.847	0.845	0.862	0.893	0.882	0.876	0.874	0.876
Broadleaf Commerce	0.635	0.877		0.928	0.870	0.859	0.863	0.862	0.860	0.873	0.894	0.886	0.880	0.879	0.879
Eclipse p. for Ceylon	0.611	0.894	0.781		0.847	0.847	0.851	0.851	0.848	0.864	0.895	0.884	0.879	0.877	0.879
Elasticsearch	0.642	0.868	0.835	0.875		0.900	0.902	0.900	0.897	0.904	0.914	0.904	0.897	0.895	0.895
Hazelcast	0.635	0.876	0.792	0.831	0.828		0.864	0.863	0.861	0.871	0.888	0.879	0.872	0.871	0.872
jUnit	0.644	0.849	0.774	0.837	0.819	0.813		0.822	0.821	0.835	0.867	0.858	0.854	0.853	0.854
MapDB	0.670	0.852	0.799	0.855	0.852	0.853	0.857		0.858	0.871	0.894	0.884	0.879	0.877	0.878
mcMMO	0.642	0.879	0.793	0.855	0.845	0.849	0.853	0.853		0.866	0.888	0.880	0.874	0.873	0.875
Mission Control T.	0.611	0.890	0.773	0.843	0.836	0.836	0.840	0.840	0.838		0.890	0.879	0.872	0.870	0.872
Neo4j	0.611	0.890	0.774	0.843	0.836	0.836	0.841	0.840	0.838	0.856		0.878	0.871	0.869	0.871
Netty	0.644	0.873	0.787	0.848	0.834	0.831	0.837	0.836	0.834	0.847	0.874		0.869	0.867	0.868
OrientDB	0.611	0.845	0.800	0.857	0.835	0.841	0.846	0.846	0.845	0.859	0.888	0.878		0.882	0.882
Oryx	0.712	0.874	0.787	0.845	0.837	0.840	0.845	0.845	0.843	0.857	0.879	0.870	0.865		0.868
Titan	0.611	0.895	0.787	0.849	0.840	0.843	0.847	0.847	0.845	0.859	0.890	0.880	0.874	0.872	

Table 23 Cross training (GitHub – Class level) - AUC values

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Com-merce	Eclipse p. for Ceylon	Elastic search	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.		0.257	0.282	0.335	0.415	0.390	0.559	0.361	0.428	0.706	0.464	0.504	0.382	0.664	0.515
ANTLR v4	0.578		0.548	0.699	0.641	0.624	0.548	0.713	0.672	0.792	0.541	0.610	0.664	0.548	0.641
Broadleaf Com-merce	0.541	0.748		0.635	0.633	0.649	0.572	0.667	0.604	0.753	0.564	0.624	0.622	0.589	0.644
Eclipse p. for Ceylon	0.537	0.532	0.439		0.503	0.553	0.538	0.558	0.518	0.486	0.520	0.480	0.584	0.502	0.490
Elastic search	0.637	0.504	0.556	0.484		0.523	0.505	0.509	0.511	0.656	0.488	0.563	0.571	0.525	0.531
Hazelcast	0.341	0.662	0.412	0.615	0.547		0.486	0.531	0.493	0.494	0.479	0.463	0.564	0.671	0.349
jUnit	0.571	0.730	0.466	0.443	0.526	0.514		0.645	0.668	0.821	0.502	0.577	0.494	0.428	0.538
MapDB	0.542	0.685	0.619	0.654	0.589	0.597	0.713		0.626	0.694	0.608	0.541	0.592	0.577	0.529
mcMMO	0.576	0.523	0.532	0.584	0.637	0.673	0.694	0.611		0.814	0.608	0.657	0.589	0.666	0.584
Mission Control T.	0.491	0.536	0.530	0.554	0.515	0.527	0.513	0.541	0.524		0.514	0.524	0.537	0.506	0.517
Neo4j	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500		0.500	0.500	0.500	0.500
Netty	0.539	0.609	0.451	0.522	0.512	0.523	0.506	0.487	0.537	0.603	0.553		0.535	0.522	0.501
OrientDB	0.485	0.650	0.555	0.577	0.485	0.508	0.511	0.495	0.552	0.643	0.511	0.494		0.441	0.490
Oryx	0.637	0.675	0.534	0.668	0.607	0.573	0.596	0.628	0.558	0.420	0.612	0.528	0.506		0.581
Titan	0.617	0.673	0.510	0.440	0.492	0.547	0.572	0.367	0.404	0.181	0.530	0.519	0.464	0.495	

Table 24 Cross training (Bug prediction dataset - Class level) - F-Measure values

Train/Test	Eclipse JDT Core 3.4	Equinox 3.4	Lucene 2.4 BPD	Mylyn 3.1	PDE UI 3.4.1
Eclipse JDT Core 3.4		0.878	0.874	0.848	0.839
Equinox 3.4	0.451		0.614	0.664	0.641
Lucene 2.4 BPD	0.754	0.706		0.808	0.809
Mylyn 3.1	0.730	0.702	0.753		0.815
PDE UI 3.4.1	0.733	0.695	0.748	0.769	

Table 25 Cross training (Bug prediction dataset - Class level) - AUC values

Train/Test	Eclipse JDT Core 3.4	Equinox 3.4	Lucene 2.4 BPD	Mylyn 3.1	PDE UI 3.4.1
Eclipse JDT Core 3.4		0.489	0.467	0.555	0.601
Equinox 3.4	0.593		0.638	0.590	0.535
Lucene 2.4 BPD	0.561	0.487		0.560	0.506
Mylyn 3.1	0.685	0.642	0.628		0.569
PDE UI 3.4.1	0.383	0.435	0.548	0.626	

Table 26 Cross training (GitHub – File level) - F-Measure values

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Commerce	Eclipse p. for Ceylon	Elastic search	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.		0.791	0.736	0.724	0.707	0.724	0.727	0.726	0.724	0.725	0.753	0.746	0.743	0.743	0.749
ANTLR v4	0.595		0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820
Broadleaf Commerce	0.631	0.832		0.842	0.803	0.803	0.803	0.803	0.800	0.806	0.848	0.835	0.826	0.825	0.829
Eclipse p. for Ceylon	0.595	0.841	0.797		0.804	0.805	0.805	0.806	0.803	0.809	0.850	0.835	0.829	0.828	0.832
Elastic search	0.595	0.817	0.771	0.797		0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820
Hazelcast	0.595	0.819	0.775	0.799	0.783		0.791	0.790	0.788	0.796	0.843	0.828	0.819	0.818	0.822
jUnit	0.722	0.813	0.759	0.767	0.759	0.766		0.768	0.766	0.771	0.802	0.793	0.788	0.788	0.793
MapDB	0.622	0.847	0.808	0.822	0.813	0.814	0.815		0.814	0.819	0.855	0.843	0.837	0.836	0.839
mcMMO	0.710	0.806	0.800	0.786	0.770	0.780	0.782	0.781		0.780	0.791	0.784	0.779	0.778	0.782
Mission Control T.	0.595	0.821	0.773	0.797	0.782	0.785	0.785	0.785	0.782		0.841	0.826	0.818	0.817	0.821
Neo4j	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790		0.825	0.816	0.815	0.820
Netty	0.682	0.818	0.773	0.795	0.774	0.783	0.784	0.784	0.784	0.789	0.827		0.818	0.817	0.819
OrientDB	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825		0.815	0.820
Oryx	0.637	0.830	0.768	0.795	0.787	0.789	0.791	0.790	0.787	0.794	0.839	0.825	0.816		0.822
Titan	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	

(0.821) AUC values. The reason can be the same as for Forrest 0.6. In general, the AUC values are very diverse ranging from 0.181 to 0.821, and the average is 0.548.

Tables 24 and 25 show the F-measure and AUC values of cross training for the Bug Prediction Dataset. Based on F-measure, Eclipse JDT Core passed the other systems in terms of training but its AUC values, or at least the first two, seem worse than the others. Equinox performed the worst in the role of being a training set, i.e., having the lowest F-values, but from AUC point of view, Equinox is the most difficult test set because 3 out of 4 of its AUC values are much worse than the average. From testing and training, and from F-measure and AUC value point of view, Mylyn is the best system.

The above described results were calculated for class level datasets. Let us now consider the file level results. The three file level datasets are the GitHub, the Bugcatchers, and the Eclipse bug datasets.

The GitHub Bug Dataset results can be seen in Tables 26 and 27. As in the case of class level, the Android Universal Image Loader project performed the worst in the role of being the test set based on F-measure and the worst system for building models (low AUC values). It would be difficult to select the best system, but the Mission Control T. system is the most critical test system again because it has the lowest and highest AUC values.

The cross-training results of the Bugcatchers Bug Dataset are listed in Tables 28 and 29. The table contains 3 high and 3 low F-measure values and they range in a wide scale from 0.234 to 0.800 while the AUC values are much closer to each other (0.500−0.606). Since only three systems are used and there is no significantly best or worst system, we cannot state any conclusion based on these results.

In the Eclipse Bug Dataset, there are only three systems as well but they are three different versions of the same system, namely Eclipse; therefore, we expected better results. As we can see in Tables 30 and 31, in this case, the F-measure and AUC values coincide which means that either both of them are high or both of them are low. The model built on version 2.0 performed better (0.705 and 0.700 F-measures and 0.723 and 0.708 AUC values) on the other two systems than the models built on the other two systems and evaluated on version 2.0 (0.638 and 0.604 F-measures and 0.639 and 0.633 AUC values). This is perhaps caused by the fact that this version contains the least number of bug entries. The other two systems are "symmetrical", the results are almost the same when one is the train and the other one is the test system.

We also performed a full cross-system experiment involving all systems from all datasets. This matrix is, however, too large to present here; consequently, it can be found in the Appendix and Tables 32 and 33 show only the average of the F-measures and AUC values of the models performed on other datasets. More precisely, we trained a model using each system separately and tested this model on the other systems, and we calculated the averages of the F-measures and AUC values to see how they perform on other datasets. For example, we can see that the average F-measures of the models trained and tested on PROMISE is only 0.607 while if these models are validated on GitHub dataset the average is 0.729. Examining F-measures, we can see that in general, models trained on PROMISE dataset perform the worst, and what is surprising is that they gave the worst result (0.607) on themselves. On the other hand, if we consider AUC values, the model trained on PROMISE achieved the best result on PROMISE (0.554 vs 0.544). This means that in this case AUC values do not help us to select the best or worst predictor/test dataset or even compare the results. Another interesting observation is that the models perform better on GitHub dataset no matter which dataset was used for training (GitHub column). On the other hand, GitHub models achieve only slightly worse F-measures on the other datasets than the best one on

Table 27 Cross training (GitHub – File level) - AUC values

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Commerce	Eclipse p. for Ceylon	Elastic search	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.		0.468	0.271	0.355	0.426	0.433	0.637	0.341	0.465	0.271	0.391	0.534	0.374	0.665	0.339
ANTLR v4	0.649		0.682	0.710	0.667	0.631	0.634	0.732	0.727	0.777	0.687	0.621	0.627	0.538	0.648
Broadleaf Commerce	0.537	0.598		0.729	0.675	0.641	0.549	0.684	0.752	0.667	0.570	0.604	0.644	0.598	0.632
Eclipse p. for Ceylon	0.517	0.593	0.528		0.565	0.549	0.500	0.680	0.592	0.725	0.541	0.538	0.574	0.512	0.543
Elastic search	0.598	0.582	0.679	0.626		0.637	0.559	0.752	0.699	0.295	0.616	0.607	0.622	0.553	0.623
Hazelcast	0.481	0.460	0.579	0.652	0.545		0.500	0.597	0.534	0.209	0.523	0.496	0.564	0.537	0.498
jUnit	0.707	0.723	0.732	0.658	0.665	0.647		0.598	0.681	0.653	0.721	0.669	0.642	0.634	0.630
MapDB	0.327	0.462	0.493	0.643	0.576	0.547	0.368		0.574	0.633	0.463	0.511	0.529	0.486	0.481
mcMMO	0.608	0.799	0.655	0.685	0.684	0.640	0.563	0.806		0.814	0.728	0.611	0.622	0.519	0.600
Mission Control T.	0.450	0.534	0.492	0.506	0.485	0.497	0.463	0.472	0.535		0.496	0.474	0.499	0.535	0.511
Neo4j	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500		0.500	0.500	0.500	0.500
Netty	0.656	0.686	0.733	0.678	0.673	0.652	0.698	0.746	0.698	0.856	0.753		0.693	0.784	0.733
OrientDB	0.573	0.686	0.718	0.559	0.661	0.603	0.600	0.794	0.640	0.786	0.737	0.736		0.611	0.750
Oryx	0.682	0.724	0.725	0.663	0.644	0.610	0.658	0.595	0.664	0.578	0.689	0.703	0.613		0.648
Titan	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	

Table 28 Cross training (Bugcatchers - File level) - F-Measure values

Train/Test	Apache	ArgoUML	Eclipse JDT Core 3.1
Apache		0.800	0.676
ArgoUML	0.436		0.665
Eclipse JDT Core 3.1	0.388	0.234	

Table 29 Cross training (Bugcatchers - File level) - AUC values

Train/Test	Apache	ArgoUML	Eclipse JDT Core 3.1
Apache		0.541	0.519
ArgoUML	0.563		0.606
Eclipse JDT Core 3.1	0.500	0.588	

Table 30 Cross training (Eclipse - File level) - F-Measure values

Train/Test	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Eclipse 2.0		0.705	0.700
Eclipse 2.1	0.638		0.725
Eclipse 3.0	0.604	0.676	

Table 31 Cross training (Eclipse - File level) - AUC values

Train/Test	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Eclipse 2.0		0.723	0.708
Eclipse 2.1	0.639		0.700
Eclipse 3.0	0.633	0.702	

Table 32 Average F-measures of the full cross-system experiment (class level)

Train/Test	GitHub	PROMISE	Bug prediction
GitHub	0.842	0.678	0.727
PROMISE	0.729	0.607	0.617
Bug prediction	0.815	0.680	0.742

Table 33 Average AUC values of the full cross-system experiment (class level)

Train/Test	GitHub	PROMISE	Bug prediction
GitHub	0.548	0.544	0.562
PROMISE	0.552	0.554	0.557
Bug Prediction	0.555	0.541	0.555

Table 34 Average F-measures of the full cross-system experiment (file level)

Train/Test	GitHub	Bugcatchers	Eclipse
GitHub	0.786	0.613	0.583
Bugcatchers	0.654	0.533	0.529
Eclipse	0.769	0.660	0.675

the given dataset (0.678 vs 0.680 on PROMISE and 0.727 vs 0.742 on Bug pred.), which suggests that the good testing results are not a consequence of some unique feature of the dataset because it can also be used to train "portable" bug prediction models.

Table 34 shows the average F-measures for the file level datasets. We can observe a similar trend, namely, all models performed the best on GitHub dataset. Besides, the results of Bugcatchers on itself is poor and the other two bug prediction models sets performed better on it. On the other hand, the AUC values do not support this observation (see Table 35). Compared with class level, the AUC values range on a wider scale, from 0.525 to 0.684, and suggest that the Eclipse is the best dataset for model building and this observation is supported by the F-measures as well.

To sum up the previous findings based on the full cross-system experiment, it may seem like file level models performed slightly better than class level predictors because the average F-measure is 0.717 and the average AUC value is 0.594 for files while they are only 0.661 and 0.552 for classes. On the other hand, this might be the result of the fact that we saw that several systems of PROMISE contain too many bugs; therefore, they cannot be used alone to build sophisticated prediction models. Another remarkable result is that all models performed notably better on GitHub dataset, which requires further investigation in the future.

6 Threats to validity

First of all, we accepted the collected datasets "as is", which means that we did not validate the data, we just used them to create the unified dataset and to examine the bug prediction capabilities of the different bug datasets. Since the bug datasets did not contain the source code, neither a step-by-step instruction on how to reproduce the bug datasets, we had to accept them, even if there were a few notable anomalies in them. For example, Camel 1.4 contains classes with LOC metrics of 0; in the Bugcatchers dataset, there are two MessageChains metrics and in several cases, the two metric values are different; or there are more datasets with extreme SCEwBug% (buggy classes percentage) values (more than 90% high or less than 1% low).

Table 35 Average AUC values of the full cross-system experiment (file level)

Train/Test	GitHub	Bugcatchers	Eclipse
GitHub	0.587	0.525	0.585
Bugcatchers	0.579	0.553	0.612
Eclipse	0.661	0.564	0.684

Although the version information was available for each system, in some cases, there were notable differences between the result of OSA and the original result in the corresponding bug dataset. Even if the analyzers would parse the classes in different ways, the number of files should have been equal. If the analysis result of OSA contains the same number of elements or more, and (almost) all elements from the corresponding bug dataset could be paired, we can say that the unification is acceptable, because all elements of the bug dataset were put into the unified dataset. On the other hand, for a few systems, we could not find the proper source code version and we had to leave out a negligible number of elements from the unified dataset.

Many systems were more than 10 years old when the actual Java version was 1.4 and these systems were analyzed according to that standard. The Java language has evolved a lot since then and we analyzed all systems according to the latest standard, which might have caused minor but negligible mistakes in the analysis results.

In Section 4.7, we used ckjm 2.2 to analyze the projects included in the Bug Prediction Dataset. We chose version 2.2 since the original paper did not mark the exact version of ckjm (D'Ambros et al. 2010); consequently, we experimented with different ckjm versions (1.9, 2.0, 2.1, 2.2) and we experienced version 2.2 to be the best candidate, since it produced the smallest differences in metric values compared with the original metric values in the Bug Prediction Dataset.

We used a heuristic method based on name matching to conjugate the elements of the datasets. Although there were cases where the conjugation was unsuccessful, we examined these cases manually and it turned out that the heuristics worked well and the cause of the problem originated from the differences of the two datasets (all cases are listed in Section 3). We examined the successful conjugations as well and all of them were correct. Even though the heuristics could not handle elements having the same name during the conjugation, only a negligible amount of such cases happened.

Even when the matching heuristics worked well, the same class name could have different meanings in different datasets. For example, OSA handles nested, local, and anonymous classes as different elements, while other datasets did not take into account such elements. Even more, the whole file was associated with its public class. This way, a bug found in a nested or inner class is associated with the public class in the bug datasets, but during the matching, this bug will be associated with the wrong element of the more detailed analysis result of OSA.

7 Conclusion and future work

There are several public bug datasets available in the literature, which characterize the bugs with static source code metrics. Our aim was to create one public unified bug dataset, which contains all the publicly available ones in a unified format. This dataset can provide researchers real value by offering a rich bug dataset for their new bug prediction experiments.

We considered five different public bug datasets: the PROMISE dataset, the Eclipse Bug Dataset, the Bug Prediction Dataset, the Bugcatchers Bug Dataset, and the GitHub Bug Dataset. We gave detailed information about each dataset, which contains, among others, their size, enumeration of the included software systems, used version control, and bug tracking systems.

We developed and executed a method on how to create the unified set of bug data, which encapsulates all the information that is available in the datasets. Different datasets use different metric suites; hence, we collected the Java source code for all software systems of each dataset, and analyzed them with one particular static source code analyzer (OpenStaticAnalyzer) in order to have a common and uniform set of code metrics (bug predictors) for every system. We constructed the unified bug dataset from the gathered public datasets at file and class level and made this unified bug dataset publicly available to anyone for future use.

We investigated the possible differences between the calculated metric values. For this purpose, we ran the ckjm analyzer tool on the Bug Prediction Dataset in order to have all the metrics from different metric suites. Then, we selected the source code elements which were identified by all three static analyzers. For this subset of source code elements, we calculated the pairwise differences and provided basic statistics for these differences as well. We performed the same steps for the Bugcatchers and the Eclipse bug datasets at file level. To see if there is any statistically significant difference, we applied a pairwise *Wilcoxon signed-rank test*. We experienced statistically significant differences in all cases except in the case of NOC (at class level between Moose and ckjm) and LLOC (at file level between OSA and Eclipse).

We evaluated the datasets according to their meta data and functional criteria. Metadata analysis includes the investigation of the used static analyzer, granularity, bug tracking and version control system, and the set of used metrics. As functional criteria, we compared the bug prediction capabilities of the original metrics, the unified ones, and both together. We used the J48 decision tree algorithm from Weka to build and evaluate bug prediction models per project (within-project learning) in the unified bug dataset. Next, we united the contents of the datasets both at file (43,744 elements) and class level (47,618 elements), and evaluated the bug prediction capabilities of this united dataset. This large dataset blurs the deficiencies of the smaller datasets (for example, datasets with more than 90% of buggy source elements). As an additional functional criterion, we used different software systems for training and for testing the models, also known as cross-project training. We performed this step on all the systems of the various datasets. Our experiments showed that the unified bug dataset can be used effectively in bug prediction.

We encourage researchers to use this large and public unified bug dataset in their experiments and we also welcome new public bug datasets. As a future work, we would like to keep this unified dataset up-to-date and extend it with newer public datasets (i.e., Had-oops dataset (Harman et al. 2014), Mutation bug dataset Bowes et al. 2016) and others, which will be published in the future.

Acknowledgements This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things" and by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary.

Funding information Open access funding provided by University of Szeged.

Appendix

The Unified Bug Dataset 1.2 which was created during this work is available as an online appendix at: https://doi.org/10.5281/zenodo.3693685 and at http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet

The *UnifiedBugDataset-1.2.zip* file contains

- the original bug datasets in their original form,
- the list of projects contained in each dataset,
- the source code of the systems that was used to develop the datasets,
- the unified dataset in CSV and ARFF format at file/class level,
- the dataset containing only the results of OpenStaticAnalyzer in ARFF format at file/class level,
- description of the OpenStaticAnalyzer metrics,
- metrics comparisons in spreadsheet format of the PROMISE (2018), Eclipse (Zimmermann et al. 2007), Bug Prediction (D'Ambros et al. 2010), Bugcatchers (Hall et al. 2014), and GitHub (Tóth et al. 2016) bug datasets,
- the results of the within-project training at file/class level in spreadsheet format,
- the results of the cross training at file/class level in spreadsheet format.

For a more exhaustive description of the exact contents of the files and usage information, one should refer to the '`README.txt`' file which is located in the root folder of the Unified Bug Dataset package.

References

(2018). The promise repository of empirical software engineering data. http://openscience.us/repo.

(2019). SourceMeter static code analyzer. https://www.sourcemeter.com.

Adewumi, A., Misra, S., Omoregbe, N., Crawford, B., Soto, R. (2016). A systematic literature review of open source software quality assessment models. *SpringerPlus*, *5*(1), 1936.

Basili, V.R., Briand, L.C., Melo, W.L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, *22*(10), 751–761.

Bowes, D., Hall, T., Harman, M., Jia, Y., Sarro, F., Wu, F. (2016). Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (pp. 330–341): ACM.

Briand, L.C., Daly, J.W., Wust, J.K. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, *25*(1), 91–121.

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications*, *36*(4), 7346–7354.

Catal, C. (2011). Software fault prediction: a literature review and current trends. *Expert Systems with Applications*, *38*(4), 4626–4636.

Cellier, P., Ducassé, M., Ferré, S., Ridoux, O. (2011). Multiple fault localization with data mining. In *SEKE* (pp. 238–243).

Chidamber, S.R., & Kemerer, C.F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493.

Cohen, J. (1988). *Statistical Power Analysis for the Behavioral Sciences*: Routledge.

Dallmeier, V., & Zimmermann, T. (2007). Automatic extraction of bug localization benchmarks from history. In *Proceedings of Int'l conference on Automated Software Engineering* (pp. 433–436): Citeseer.

D'Ambros, M., Lanza, M., Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *7Th working conference on mining software repositories (MSR)* (pp. 31–41): IEEE.

D'Ambros, M., Lanza, M., Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, *17*(4-5), 531–577.

Durieux, T., & Monperrus, M. (2016). *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*. Technical report, Universite Lille 1, https://hal.archives-ouvertes.fr/hal-01272126/document.

El Emam, K., Melo, W., Machado, J.C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, *56*(1), 63–75.

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T. (2018). A public unified bug dataset for java. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'18* (pp. 12–21). New York: ACM. https://doi.org/10.1145/3273934.3273936.

Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B. (2011). The misuse of the nasa metrics data program data sets for automated software defect prediction. In *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011), IET* (pp. 96–103).

Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B. (2012). Reflections on the nasa mdp data sets. *IET Software*, *6*(6), 549–558.

Gyimothy, T., Ferenc, R., Siket, I. (2005a). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, *31*(10), 897–910.

Gyimóthy, T., Ferenc, R., Siket, I. (2005b). Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. In *IEEE Transactions on Software Engineering*, (Vol. 31 pp. 897–910): IEEE Computer Society.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H. (2009). The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, *11*(1), 10–18.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, *38*(6), 1276–1304.

Hall, T., Zhang, M., Bowes, D., Sun, Y. (2014). Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *23*(4), 33.

Harman, M., Islam, S., Jia, Y., Minku, L.L., Sarro, F., Srivisut, K. (2014). Less is more: Temporal fault predictive performance over multiple hadoop releases. In Le Goues, C., & Yoo, S. (Eds.) *Search-based software engineering* (pp. 240–246). Cham: Springer International Publishing.

Herbold, S., Trautsch, A., Grabowski, J. (2017). A comparative study to benchmark cross-project defect prediction approaches. *IEEE Transactions on Software Engineering*, *44*(9), 811–833.

Horning, J., Lauer, H., Melliar-Smith, P., Randell, B. (1974). A program structure for error detection and recovery. *Operating Systems*, 171–187.

Hosseini, S., Turhan, B., Gunarathna, D. (2019). A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, *45*(2), 111–147. https://doi.org/10.1109/TSE.2017.2770124.

Jureczko, M., & Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE '10* (pp. 9:1–9:10). New York: ACM. https://doi.org/10.1145/1868328.1868342.

Jureczko, M. (2011a). Significance of different software metrics in defect prediction. *Software Engineering: An International Journal*, *1*(1), 86–95.

Jureczko, M., & Madeyski, L. (2011b). A review of process metrics in defect prediction studies. *Metody Informatyki Stosowanej*, *5*, 133–145.

Just, R., Jalali, D., Ernst, M.D. (2014). Defects4j: A Database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA* (pp. 437–440). New York: ACM. https://doi.org/10.1145/2610384.2628055.

Kim, S., Zimmermann, T., Whitehead, Jr.E.J., Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering* (pp. 489–498): IEEE Computer Society.

Li, Z., Jing, X., Zhu, X. (2018). Progress on approaches to software defect prediction. *IET Software*, *12*(3), 161–175. https://doi.org/10.1049/iet-sen.2017.0148.

Lin, D., Koppel, J., Chen, A., Solar-Lezama, A. (2017). Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH Companion 2017* (pp. 55–56). New York: ACM. https://doi.org/10.1145/3135932.3135941.

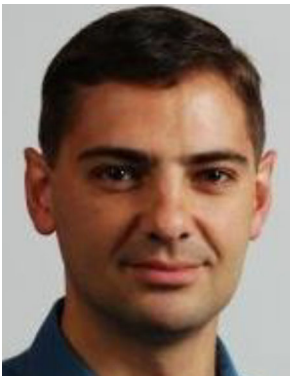
Lincke, R., Lundberg, J., Löwe, W. (2008). Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08* (pp. 131–142). New York: ACM. https://doi.org/10.1145/1390630.1390648.

Madeiral, F., Urli, S., de Almeida Maia, M., Monperrus, M. (2019). Bears: An extensible java bug benchmark for automatic program repair studies. arXiv:1901.06024.

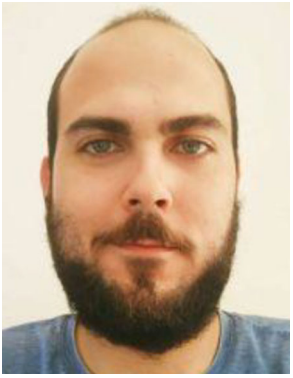
Malhotra, R., & Jain, A. (2011). Software fault prediction for object oriented systems: a literature review. *ACM SIGSOFT Software Engineering Notes*, *36*(5), 1–6.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, *27*, 504–518.

Moser, R., Pedrycz, W., Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, ACM.
Myles, H., Douglas, A.W., Eric, C. (2014). *Nonparametric Statistical Methods*, 3rd edn. New York: Wiley.
Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *2005. ICSE 2005. Proceedings. 27th International Conference on Software Engineering* (pp. 284–292): IEEE.
Ostrand, T.J., Weyuker, E.J., Bell, R.M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, *31*(4), 340–355.
Petrić, J., Bowes, D., Hall, T., Christianson, B., Baddoo, N. (2016). The jinx on the nasa software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (p. 13): ACM.
Radjenović, D., Heričko, M., Torkar, R., Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, *55*(8), 1397–1418.
Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, *10*(2), 220–232.
Robles, G. (2010). Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)* (pp. 171–180): IEEE.
Saha, R., Lyu, Y., Lam, W., Yoshida, H., Prasad, M. (2018). Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)* (pp. 10–13).
Sayyad Shirabad, J., & Menzies, T. (2005). *The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering*. Canada: University of Ottawa. http://promise.site.uottawa.ca/SERepository.
Shepperd, M., Song, Q., Sun, Z., Mair, C. (2013). Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering*, *39*(9), 1208–1215.
Strate, J.D., & Laplante, P.A. (2013). A literature review of research in software defect reporting. *IEEE Transactions on Reliability*, *62*(2), 444–454. https://doi.org/10.1109/TR.2013.2259204.
Subramanyam, R., & Krishnan, M.S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, *29*(4), 297–310.
Tóth, Z., Gyimesi, P., Ferenc, R. (2016). A public bug database of github projects and its application in bug prediction. In *International Conference on Computational Science and Its Applications* (pp. 625–638): Springer.
Wahono, R.S. (2015). A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, *1*(1), 1–16.
Weyuker, E.J., Ostrand, T.J., Bell, R.M. (2010). Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, *15*(3), 277–295.
Weyuker, E.J., Bell, R.M., Ostrand, T.J. (2011). Replicate, replicate, replicate. In *2011 Second International Workshop on Replication in empirical software engineering research (RESER)* (pp. 71–77). IEEE.
Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering* (p. 38): Citeseer.
Wong, W.E., Debroy, V., Xu, D. (2012). Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *42*(3), 378–396.
Xu, Z., Khoshgoftaar, T.M., Allen, E.B. (2000). Prediction of software faults using fuzzy nonlinear regression modeling. In *2000, Fifth IEEE International Symposim on High Assurance Systems Engineering. HASE 2000* (pp. 281–290): IEEE.
Yu, Z., Kraft, N.A., Menzies, T. (2016). How to read less: Better machine assisted reading methods for systematic literature reviews. arXiv:161203224.
Zimmermann, T., Premraj, R., Zeller, A. (2007). Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering* (p. 9): IEEE Computer Society.
Zimmermann, T., Nagappan, N., Gall, H., Giger, E., Murphy, B. (2009). Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 91–100): ACM.



Rudolf Ferenc is the head of the Department of Software Engineering and an associate professor at the University of Szeged, Hungary. His research interests include static code analysis, metrics, quality assurance, design pattern and antipattern mining, and bug detection. He leads the Static Code Analysis group, which develops tools for analyzing the source code of various languages. These tools calculate code metrics, and detect coding issues and duplications. He has more than 100 publications in these fields with over 2000 citations. He is leading several R&D projects, which are related to quality assessment, improvement and architecture reconstruction of software systems for major banks and software development companies in Hungary. He has been serving as Program Co-Chair and Program Committee member at the major conferences in this field (ICSE, ICSME, ESEC/FSE, SANER, CSMR, WCRE, ICPC, SCAM, FASE, etc.) since 2005.



Zoltán Tóth is assistant lecturer at the University of Szeged. He is a PhD candidate, who has submitted his thesis to the doctoral committee for review. He received his MSc degree in Computer Science from the University of Szeged in 2014. His main research interests include static code analysis, software metrics, bug prediction, quality assurance. He has participated in several R&D projects related to software quality assurance. He took part in the development of static source code analyzers for RPG and JavaScript software systems.

Gergely Ladányi is a pre-doctoral fellow at the University of Szeged. His main interests are software quality, quality monitoring, maintainability, and quality models. His papers were published in acknowledged conferences and journals such as ICSME, CSMR, EASST, SANER, ASEA, and PROMISE. He participated in numerous R&D projects related to software maintainability.

István Siket received the PhD degree in computer science from the University of Szeged in 2011. He was a member of the program committee of the International Conference on Software Technologies (ICSOFT). His research interests include source code analysis, measurement, quality assurance, and bug detection. He has participated in several R&D projects related to source code analysis and quality assurance.

Tibor Gyimóthy is professor at the Software Engineering Department at the University of Szeged in Hungary and the head of the MTA-SZTE Research Group on Artificial Intelligence. His research interests are concentrated on high quality software development and maintenance. His main contributions are methods and tools for identifying and predicting harmful program elements in the source code. He published over 210 research papers with more than 2600 independent citations. The software maintenance tools developed in his team are used by numerous software companies in the world. His scientific contributions were awarded with the Széchenyi Professor Grant, László Kalmár Award, HAS Academic Prize, Gábor Dénes Award and Szent-Györgyi Albert Prize. He has been the member of over 40 international scientific program committees and co-chaired 5 conferences.

Affiliations

Rudolf Ferenc[1] · Zoltán Tóth[1] · Gergely Ladányi[1] · István Siket[1] · Tibor Gyimóthy[2]

Zoltán Tóth
zizo@inf.u-szeged.hu

Gergely Ladányi
lgergely@inf.u-szeged.hu

István Siket
siket@inf.u-szeged.hu

Tibor Gyimóthy
gyimothy@inf.u-szeged.hu

[1] Department of Software Engineering, University of Szeged, Szeged, Hungary

[2] MTA-SZTE Research Group on Artificial Intelligence, Szeged, Hungary