

Bug Prediction Capability of Primitive Enthusiasm Metrics

Péter Gál(✉)

Department of Software Engineering, University of Szeged,
Dugonics ter 13, Szeged 6720, Hungary
galpeter@inf.u-szeged.hu

Abstract. Bugs in software development life cycle are unavoidable. Manually finding these bugs is not always the most effective way. To aid this, various bug prediction approaches which are using code metrics are developed and are also still in active development.

In a previous work, the Primitive Enthusiasm code metrics were introduced to add detection capabilities for Primitive Obsession code smells. This paper explores the usability of the Primitive Enthusiasm metrics in a bug prediction scenario. To evaluate the new metrics, an existing source code bug data set was used. The correlation between existing metrics and the new PE metrics was investigated. Furthermore the effectiveness of bug prediction is investigated by building training models with and without the new metrics. Using a cross-project and a project version-based evaluation the results show that adding PE metrics can be beneficial for bug prediction.

Keywords: Bug prediction · Code metrics · Primitive obsession · Primitive enthusiasm · Static analysis

1 Introduction

Bugs during software development are bound to be introduced unwillingly. Each bug can have a different effect, one can cripple the whole system or it will just provide a minor annoyance, for example the layout or colouring is incorrect. Finding various bugs in applications could take from several minutes to hours or even more. As a project development continues, finding and fixing a bug costs more and more [1], therefore, researchers are also working on creating and improving various automated tools that can help a developer to quickly find a bug or even just provide hints on where the bug could be in the system. In pursuit of automated tools to provide bug predictions there are already several papers present [10,18]. Some of these tools use various source-level metrics to train and build models which will be used for bug predictions.

In this paper, we discuss the addition of Primitive Obsession-related metrics to improve bug prediction capabilities. To do this, an existing bug dataset [3] was extended with the Primitive Enthusiasm source-level metrics [15] and the impact of the new metrics was investigated.

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12955, pp. 246–262, 2021.
https://doi.org/10.1007/978-3-030-87007-2_18

The rest of the paper is organised as follows. In Sect. 2, the background and related works are discussed. Section 3 is an overview of the Primitive Enthusiasm metrics. Evaluation of the newly added metrics for bug prediction is presented in Sect. 4. In Sect. 5, the threats to validity are described. Finally, the paper is concluded in Sect. 6.

2 Background

Before investigating bug prediction capabilities we need to have a bit of background on related datasets and other bug prediction research. In addition, we need to discuss the Primitive Obsession code smell and related problems. This section provides an insight into these topics.

2.1 Bug Prediction and Datasets

Using a bug predictor tool could help developers to quickly triage an issue in a given system in a cost-effective manner. In one research [8] the authors used source code metrics in conjunction with machine learning methods to provide fault predictions. They observed that on the Mozilla suite there are source code metrics that can help to predict fault-prone classes. A different study [13] concluded that defect predictors that learned from source code metrics was useful and easy to use, in addition they are widely-used. Another conclusion of the paper that the goal of the learning should be used to select a preferred machine learning algorithm.

In order to evaluate a method that provides bug prediction capabilities, a dataset is required. Such dataset should provide additional data, such as various source code metrics and a correct classification of classes or methods that have bugs in them. In addition, it is favourable to have real-world software in a dataset, as ultimately a bug prediction tool should be used on real applications not on small benchmarks. To achieve this, a unified Java dataset was used in this paper, which was introduced by Ferenc et al. [3] This dataset is a collection of five publicly available bug datasets in a unified format. These are the following: the PROMISE dataset [11], the Eclipse bug dataset [22], the Bug Prediction dataset [2], the Bugcatchers dataset [9], and the GitHub bug dataset [17]. These datasets contain class and/or file level code metrics including information on defects. From these bug datasets the PROMISE, GitHub, and the Bug Prediction datasets were used in this paper, as these provide class-level metrics. In total, there are 66 systems in these three datasets and they range from small applications with few classes to large systems with thousands of classes.

2.2 Definition of Primitive Obsession

Usually data types can be categorized into two groups: primitive and complex. Primitive types are simple data types provided by a language, e.g. *char*, *integer*, etc. Complex types are constructed from these primitive types or by using other

complex types, and are usually named *class* or *struct*. Additionally, such complex types can encapsulate operations on their data, providing semantic knowledge on how the data can or should be used. For example, it would be convenient to place a *string* and two *integer*s that represent a task name, a start, and end time into a class instead of using them separately. By doing such data coupling and providing a name for the class will allow the developers to understand the connections between the data. However, it is also common to neglect such small, but useful code compositions. Similarly adding additional method parameters to an already long parameter list seems to be a quick solution to achieve a programming goal, but in the long run, this will make the relation between data harder to understand and will also decrease the maintainability of the source code. Excessive usage of primitive data types instead of creating small objects is the core of Primitive Obsession. Mäntylä et al. classify Primitive Obsession as a type of *bloater* smell [12], in addition it is symptom of the existence of overgrown, chaotic code parts. A simple example for Primitive Obsession is depicted in Fig. 1. The `NORMAL` and `URGENT` class constants on Lines 2 and 3, can be considered type codes. Such type codes are usually integers or strings that have a name, and are used to simulate types. In this case these are used to indicate task priorities. Type codes are widely used in various projects, and they can be considered a version of Primitive Obsession. They break the object-orient paradigm and can cause hidden dependencies [20]. There are various ways to remove such type codes from a project, for example, with a *Strategy* or *State* pattern [6].

```
1  class Task {
2    static const int NORMAL = 1;
3    static const int URGENT = 2;
4
5    public void work(int from, int to, int breakCount, String
         worker) {
6      if(worker == null || worker.length() == 0) {
7      /*...*/
8      }
9    }
10 }
```

Fig. 1. Sample code containing primitive obsessions

At Line 5, the `work` method is presented. This method has three integer parameters and a single string argument. As all of these arguments are primitive types, the method can be considered having the Primitive Obsession code smell. Even more if these arguments are repetitively used throughout the project's codebase. Please note, that in Java string are classes, thus they are not really primitive types, but logically it makes sense to categorize it as a kind of primitive type. Line 6 is also a good target for refactoring, because the checks present in this line should be encapsulated into a class. Especially, if these checks are used

multiple places. For more that contains Primitive Obsession examples, Steven A. Lowe's GitHub project[1] is a good start. This project provides a set of steps on how to clean up this kind of code smell.

2.3 Challenges Using Primitive Obsession

Previously, the Primitive Obsession was not in the main focus of researchers. A team of researchers conducted a literature overview on code smells [21]. During the review, they gathered 319 papers from year 2000 till 2009. From these they examined 39 in more detail. One of the investigation point was to determine which code smells attracted the most attention from researchers. The most discussed topic was the Duplicated Code smell – from 39 papers 21 elaborated on this –, at the same time other code bad smells received very little attention. Among the unpopular smells was the Primitive Obsession that was only present in 5 papers. However, these papers also discussed Fowler's other code bad smells.

Another literature overview [7] by Gupta et al. collected 60 papers from 1999 to 2016, and concluded that four code bad smells did not have any detection method, including Primitive Obsession. An investigation on five tools that could detect code smells reported that none of them were able to find the Primitive Obsession code smells.

In a thesis, Roperia introduced JSmell for Java applications that could detect Primitive Obsession smells [16]. The detection technique for this smell was based on the number of primitive data types declared in a class. JSmell reports Primitive obsession when the number of primitively typed data is above than the average of a project, and the class was not instantiated. As the tool was not accessible, a comparison was not done between the JSmell's approach and the Primitive Enthusiasm metrics solution.

Other papers elaborated on the usefulness of code smells for maintenance indicators [14,19]. However, these cannot be considered as an all-in-one solution for defect predictions, but they still give insights into important maintainability aspects for a project, even more if they are combined.

3 Primitive Enthusiasm

The Primitive Enthusiasm (PE) metrics were previously introduced in various papers [5,15]. However, for the sake of completeness and to make the paper self-contained the metrics are shortly described here.

The base of the metrics iss the Formula 1 which describes how the primitive-typed parameters are collected for a given M_i method.

$$Primitives(M_i) := \langle P_{M_i,j} | 1 \leq j \leq |P_{M_i}| \wedge P_{M_i,j} \in PrimitiveTypes \rangle \quad (1)$$

For this formula the definitions of the parameters are the following:

[1] https://github.com/stevenalowe/kata-2-tinytypes.

- ***PrimitiveTypes*** is the set of types that are handled as primitive ones. For Java this contains the following types: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`, and `String`.
- $\boldsymbol{N}$ represents the number of methods in the current class.
- $\boldsymbol{M_i}$ denotes the ith method of the current class.
- $\boldsymbol{M_c}$ denotes the current method under investigation in the current class.
- $\boldsymbol{P_{M_i}}$ denotes the list of types used for parameters in the M_i method.
- $\boldsymbol{P_{M_i,j}}$ defines the type of the jth parameter in the M_i method.

Then, using this *Primitives* function three metrics were constructed. These are the LPE, GPE, and HPE metrics. These metrics are calculated for methods that have at least a single parameter.

Local Primitive Enthusiasm. Formula 2 depicts the calculation of the LPE metric.

$$LPE(M_c) := \frac{\sum_{i=1}^{N} |Primitives(M_i)|}{\sum_{i=1}^{N} \mid P_{M_i} \mid} < \frac{|Primitives(M_c)|}{\mid P_{M_c} \mid} \tag{2}$$

The left-hand side of the inequality calculates the percentage of how many parameters of the current class are of primitive types. While the right-hand side denotes how many parameters of the M_c method are of primitive types. This can be considered as the method's primitiveness ratio. If the right-hand side is greater than the left-hand side, that is, there are more primitive types in the M_c method's parameter list than in the given class on average, then the method is considered to be an LPE true method.

The formula is calculated method-by-method in a given class context, therefore, it is considered as a local value.

Global Primitive Enthusiasm. GPE is shown in Formula 3. In this formula, the G is the list of methods in the analysed system, and G_i is the ith method in this list. The main difference from LPE is that the left-hand side now describes the average number of primitive-typed arguments in the whole system and this is compared to the right-hand, side which is the current method's primitiveness ratio.

$$GPE(M_c) := \frac{\sum_{i=1}^{|G|} |Primitives(G_i)|}{\sum_{i=1}^{|G|} \mid P_{G_i} \mid} < \frac{|Primitives(M_c)|}{\mid P_{M_c} \mid} \tag{3}$$

The idea behind GPE was that this compares the current method to the methods in the whole project, which in turn can highlight methods that are above the system's average.

Hot Primitive Enthusiasm. By combining the LPE and GPE formulae Formula 4 was constructed. The purpose of this combination is to direct the attention of the developer to more suspicious code parts.

$$HPE(M_c) := LPE(M_c) \wedge GPE(M_c) \tag{4}$$

If a method is HPE true then it indicates that in terms of primitive method arguments it is an outstanding method in both a given class and in the whole system.

4 Evaluation

To investigate the Bug prediction capabilities of the PE metrics, an already existing Java-based bug dataset was used [3]. There are 66 systems in this bug data set. For each system, 63 commonly used source code metrics are already calculated and in addition, classes with bugs are marked. This data set was extended with the previously mentioned PE metrics and were calculated by the OpenStaticAnalyzer[2] (OSA), which is an open-source, multi-language, static code analyser framework developed at the Department of Software Engineering, University of Szeged.

In this section, we will describe the evaluation process: how the metrics were calculated, how the systems were selected from the bug data set, the correlation between metrics, a cross-project validation for bug prediction, and a project version-based bug prediction.

4.1 Calculating the Metrics

The selected bug data already contains mainly class-based metrics, and for each of the classes in this dataset, the PE metrics were calculated. However, the PE metrics are inherently method-based metrics. To resolve this minor incompatibility, the PE metrics were aggregated by class. That is, for a given class the number of LPE, GPE, and HPE true functions are counted and these values are used as class-level metrics. This aggregation inherently connects the PE metrics with the number of methods in a class. If there are more methods in a class, the class has a higher probability to have more PE true methods. After the aggregation, the class level PE metric value could have a value between 0 and the number of methods in that class.

In the following sections, the PE metrics will refer to this kind of aggregated value, if not mentioned otherwise.

Information on Selected Systems. In the used bug dataset, there are multiple versions present for some projects. To make the tables smaller in the paper, we have only selected the latest version of each project if there were multiple variants. Based on this criteria, 33 systems were selected from the original dataset.

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

Table 1. Class count, bug count, and PE metric count information on selected systems

	Class	Bugs	Bugs/Class %	LPE	GPE	HPE
Equinox	319	126	39.5	78	114	74
Lucene	670	63	9.4	115	175	85
Myly	1405	209	14.9	240	421	206
Eclipse PDE UI	1491	208	14.0	357	510	321
Android U. I. L.	73	20	27.4	7	23	7
ANTLR	479	21	4.4	60	119	56
Broadleaf	1593	292	18.3	173	271	164
Eclipse p. for Ceylon	1610	68	4.2	187	284	176
Elasticsearch	5908	678	11.5	560	1063	480
Hazelcast	3412	377	11.0	204	391	185
JUnit	731	35	4.8	17	29	12
MapDB	331	40	12.1	64	89	62
mcMMO	301	57	18.9	41	35	26
MCT	1887	9	0.5	146	219	106
Neo4j	5899	58	1.0	582	1025	498
Netty	1143	271	23.7	128	191	122
OrientDB	1847	280	15.2	288	402	253
Oryx	533	74	13.9	36	97	32
Titan	1468	96	6.5	135	263	126
Ant 1.7	681	165	24.2	100	190	84
Camel 1.6	795	170	21.4	88	173	87
Ckjm 1.8	8	5	62.5	1	1	1
Forrest 0.8	31	2	6.5	7	14	7
Ivy 2.0	294	37	12.6	81	97	68
JEdit 4.3	439	11	2.5	150	162	119
Log4J 1.2	191	175	91.6	30	71	29
Lucene 2.4	320	193	60.3	65	92	46
Pbeans 2	37	8	21.6	13	18	12
Poi 3.0	414	276	66.7	62	274	50
Synapse 1.2	228	84	36.8	47	83	46
Velocity 1.6	188	66	35.1	51	76	46
Xalan 2.7	848	837	98.7	191	208	147
Xerces 2.0	342	303	88.6	78	102	64

For these systems, we first checked how many classes there are and how many bugs were reported. In addition the aggregated PE metric values were calculated for each system. These values can be seen in Table 1.

The first column, Class describes the number of classes in the given system. The second, "Bugs" column represents the number of classes which have at least a single reported bugs in them, and the third "Bugs/Class" column gives a view on how many of the system classes have a bug in them. The LPE, GPE, and HPE columns show how many classes have at least a single LPE, GPE, or HPE method. There are various-sized projects ranging from a small 8 class system to a more than 5500 class system. What is interesting to see is that there are usually more GPE true classes than LPE true ones. For example, in the case of the Elasticsearch system, there are 503 more GPE true classes than LPE ones. In addition, the HPE value is usually lower than LPE value. This indicates that the combination of the LPE and GPE metrics can help focus the developers' attention.

There are two very small projects on this list: the Ckjm 1.8 and Forrest 0.8. For these two, as there are already a small number of classes present, the PE metric values are also small. These two systems will be interesting in Sect. 4.3.

4.2 Correlation Between Metrics

As there are various metrics already present in the bug dataset it is worthwhile to consider how the PE metrics are behaving compared to other metrics. The main question is this: Are there any metrics that have strong connections with these new PE metrics?

To answer this question, a correlation matrix was calculated using Pearson's correlation for each metric, merging all of the selected systems into a single input dataset. This matrix is shown in Table 2. A colour map was applied to help show the correlation values between the metrics. The greater the correlation between the metrics the darker the cell color.

Based on the correlation matrix, the LPE, GPE, and HPE metrics have a strong correlation. The LPE and GPE metrics have a 0.86 correlation value which is caused by the fact that there is only a minor difference between the calculation of these two metrics. In addition, the fact that the HPE has a high correlation with both LPE and GPE is not a surprise as the HPE metric by definition is calculated from these two metrics.

The highest correlation between the PE metrics and traditional metrics can be found for the NLM and LPE metrics that have the value of 0.58. All other correlation values are less than this value. By investigating some of the higher correlation values we could have an insight into the connection between the metrics.

Table 2. Correlation between PE and other metrics.

	LPE	GPE	HPE		LPE	GPE	HPE
LPE	1.00	0.86	0.96	NM	0.28	0.27	0.26
GPE	0.86	1.00	0.91	NOA	0.03	0.01	0.03
HPE	0.96	0.91	1.00	NOC	0.04	0.04	0.03
AD	0.10	0.09	0.08	NOD	0.04	0.03	0.03
CBO	0.31	0.24	0.27	NOI	0.39	0.33	0.36
CBOI	0.19	0.20	0.19	NOP	0.06	0.04	0.05
CC	0.00	-0.01	0.00	NOS	0.40	0.35	0.38
CCL	0.11	0.09	0.10	NPA	0.02	0.02	0.02
CCO	0.09	0.08	0.09	NPM	0.26	0.27	0.25
CD	0.08	0.08	0.07	NS	0.11	0.12	0.11
CI	0.08	0.07	0.08	PDA	0.37	0.34	0.36
CLC	-0.01	-0.02	-0.01	PUA	0.36	0.41	0.36
CLLC	0.00	-0.01	0.00	RFC	0.53	0.49	0.50
CLOC	0.41	0.35	0.39	TCD	0.07	0.08	0.07
DIT	0.01	0.00	0.01	TCLOC	0.41	0.35	0.39
DLOC	0.38	0.33	0.36	TLLOC	0.49	0.44	0.47
LCOM5	0.26	0.28	0.24	TLOC	0.51	0.45	0.49
LDC	0.15	0.13	0.15	TNA	0.11	0.10	0.11
LLDC	0.16	0.14	0.16	TNG	0.17	0.15	0.17
LLOC	0.49	0.44	0.46	TNLA	0.13	0.12	0.13
LOC	0.51	0.45	0.48	TNLG	0.35	0.33	0.34
NA	0.09	0.07	0.08	TNLM	0.54	0.52	0.51
NG	0.18	0.17	0.18	TNLPA	0.02	0.02	0.02
NII	0.26	0.28	0.26	TNLPM	0.48	0.49	0.46
NL	0.28	0.25	0.26	TNLS	0.20	0.22	0.21
NLA	0.10	0.09	0.10	TNM	0.23	0.21	0.22
NLE	0.30	0.27	0.28	TNOS	0.42	0.36	0.39
NLG	0.36	0.34	0.35	TNPA	0.03	0.02	0.03
NLM	0.58	0.57	0.55	TNPM	0.23	0.22	0.22
NLPA	0.01	0.01	0.01	TNS	0.13	0.12	0.13
NLPM	0.50	0.52	0.49	WMC	0.51	0.45	0.49
NLS	0.20	0.22	0.20	bugs	0.15	0.14	0.14

The PE metrics have greater correlations with metrics that are calculated from the number of methods. These metrics are the NLM[3], TNLM[4], NLPM[5], TNLPM[6], RFC[7], and WMC[8] and their highest correlation values for PE metrics are 0.58, 0.54, 0.52, 0.48, 0.53, and 0.51 in the same order. These correlations are understandable, as the PE metrics have connections to the number of methods since the aggregation was done by class (as was described in Sect. 4.1). If there are more methods there is a higher chance to have a PE true method(s). In addition, the PE metrics are calculated from method argument lists, thus, if there are more

[3] NLM: Number of Local Methods.
[4] TNLM: Total Number of Local Methods.
[5] NLPM: Number of Local Public Methods.
[6] TNLPM: Total Number of Local Public Methods.
[7] RFC: Response set For Class.
[8] WMC: Weighted Methods per Class.

methods there are usually more arguments. Another interesting insight can be given if the correlation between these common metrics is investigated. Table 3 elaborates on the correlation between these commonly used metrics.

Table 3. Correlation between a selected set of method count related metrics.

	NLM	NLPM	RFC	TNLM	TNLPM	WMC
NLM	1	0.92	0.84	0.91	0.86	0.75
NLPM	0.92	1	0.75	0.82	0.89	0.55
RFC	0.84	0.75	1	0.79	0.72	0.68
TNLM	0.91	0.82	0.79	1	0.94	0.69
TNLPM	0.86	0.89	0.72	0.94	1	0.53
WMC	0.75	0.55	0.68	0.69	0.53	1

A strong correlation can be seen in most cases between these method-based metrics, and these values are not a surprise as the metrics are also calculated mainly or partially from the number of methods in a class. As shown in Table 2 and Table 3, the PE metrics have less correlation than these method-related metrics have between each other. This shows that the PE metrics can give an extra dimension – aside from the already existing method related metrics – as they are not tightly coupled even if there is a connection between PE and traditional method-based metrics.

The other set of interesting correlations with PE metrics are the ones related to lines of code: LOC[9], LLOC[10], CLOC[11], DLOC[12], TLOC[13], TLLOC[14], TCLOC[15], TNOS[16], NOS[17], The connection between these and the PE metrics can be attributed to the fact that if there are more lines of code then there are usually more methods in an application. Just like previously it is also worth checking the correlations between these common metrics. This matrix is depicted in Table 4.

As shown in Table 4, most of these metrics have high correlation values between each other. Just like before, these values are understandable since all of them are size-related metrics and calculated from various types of code lines.

Based on these data, it can be concluded that although the PE metrics have some kind of connection between already existing metrics they can still give extra information for a developer.

[9] LOC: Lines of Code.
[10] LLOC: Logical Lines of Code.
[11] CLOC: Comment Lines of Code.
[12] DLOC: Documentation Lines of Code.
[13] TLOC: Total Lines of Code.
[14] TLLOC: Total Logical Lines of Code.
[15] TCLOC: Total Comment Lines of Code.
[16] TNOS: Total Number of Statements.
[17] NOS: Number of Statements.

Table 4. Correlation between a selected set of line count related metrics.

	CLOC	DLOC	LLOC	LOC	NOS	TCLOC	TLLOC	TLOC	TNOS
CLOC	1.00	0.95	0.53	0.69	0.46	0.99	0.53	0.68	0.48
DLOC	0.95	1.00	0.39	0.56	0.32	0.94	0.40	0.56	0.34
LLOC	0.53	0.39	1.00	0.97	0.97	0.52	0.97	0.95	0.96
LOC	0.69	0.56	0.97	1.00	0.93	0.68	0.95	0.97	0.92
NOS	0.46	0.32	0.97	0.93	1.00	0.46	0.93	0.90	0.98
TCLOC	0.99	0.94	0.52	0.68	0.46	1.00	0.54	0.69	0.48
TLLOC	0.53	0.40	0.97	0.95	0.93	0.54	1.00	0.98	0.97
TLOC	0.68	0.56	0.95	0.97	0.90	0.69	0.98	1.00	0.93
TNOS	0.48	0.34	0.96	0.92	0.98	0.48	0.97	0.93	1.00

4.3 Cross-Project Bug Prediction

To evaluate the bug prediction capabilities of the PE metrics, a cross-project validation was performed. The cross-project validation was chosen as a method to see how the newly added metric changes the bug prediction, and classification.

To do this prediction, the Weka [4] tool was used with the default Random Forest classificator. The classification label was the bug presence for a given class. For each project, a model was trained and all other projects were used as test data. This process was done for the original metrics in the dataset and for the PE extended metrics giving us two sets of classification results. After this, the weighted average F-measure[18] difference was calculated for each train-test system pair. This difference can show us how the classification results changed after adding the new metrics to the dataset. These differences can be examined in Table 5.

The first and second column describes the project on which the training was done. The IDs of the projects are used as column labels to reduce the size of the table a bit. The columns after these are the project IDs on which the trained model was evaluated.

In pursuit to improve understandability, a colour-coding was applied to the values in the table. To indicate improvement, cells that have a value of 0.05 or higher are coloured green and values less or equal to -0.05 are coloured red to indicate a decrease in weighted F-measure values. Due to rounding, it is possible that a value that is equal to one of these thresholds is not coloured. There are also cells that do not have any value, aside from the matrix main diagonal. In these cases, the classificator was unable to calculate the weighted F-measure, that usually means that there were only bugged or non-bugged classes reported by the classificator resulting in an invalid weighted F-measure value.

The highest decrease in F-measure value was in the case of the Xerces 2.0 system. Based on Table 1, most of the classes in the system have a bugged class. Due to this, an over-fitting can be observed by the classificator marking

[18] Weighted average F-measure is calculated with the following formula: $\frac{Fmeasure(a)*InstanceCount(a)+Fmeasure(b)*InstanceCount(b)}{InstanceCount(a)+InstanceCount(b)}$ Where a and b are the two classes for classification.

Table 5. Weighted fmeasure changes in case of cross-project validation

ID	Name	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
1	Equinox		-0.00	-0.02	-0.03	0.06	0.01	0.00	-0.01	-0.01	0.01	0.02	0.03	-0.03	0.01	-0.00	0.04	0.00	0.00	0.01	-0.05	-0.03	0.26	0.25	-0.01	-0.02	0.00	0.04	0.00	-0.11	0.00	0.04	0.01	0.08
2	Lucene	-0.04		-0.00	0.01	-0.04	0.01	0.01	-0.01	0.01	-0.00	-0.00	-0.02	0.03	-0.00	-0.01	-0.00	0.01	-0.02	0.00	0.03	0.03			-0.01	0.01	-0.02	0.01	0.01	0.03	-0.00	0.07	-0.03	-0.03
3	Myly	0.01	0.03		0.01	-0.05	-0.01	-0.03	0.03	-0.02	0.02	-0.01	0.04	-0.07	0.01	-0.00	-0.01	-0.01	0.00	-0.01	-0.03	0.01	0.28	0.02	0.04	0.03	0.04	-0.01	0.03	-0.02	0.09	-0.04	0.03	0.03
4	Eclipse PDE UI	-0.02	0.00	-0.00		0.09	-0.02	-0.03	0.01	-0.00	-0.00	-0.01	-0.03	-0.00	-0.00	-0.00	-0.02	-0.01	-0.01	0.01	0.02	0.02	0.46	-0.09	-0.04	0.04	0.02	0.02	-0.06	0.04	-0.07	-0.03	-0.08	0.00
5	Android U. I. L.	0.02	0.00	0.02	-0.04		-0.03	-0.07	-0.04	-0.02	-0.02	-0.12	-0.01	-0.04	-0.02	-0.04	-0.06	0.03	-0.01	-0.03	0.10	-0.04	0.12	-0.06	-0.10	0.05	-0.02	0.03	0.08	-0.10	-0.04	0.00	-0.05	-0.04
6	ANTLR	0.03	0.00	-0.00	0.00	0.03		-0.03	0.00	-0.00	-0.01	-0.00	-0.02	-0.01	0.00	0.00	0.02	0.00	-0.00	0.01	-0.03	0.01		-0.05	0.00	-0.02	-0.00	0.00	-0.02	-0.02	0.03	0.04	-0.01	-0.10
7	Broadleaf	-0.04	0.00	0.01	-0.00	-0.07	-0.03		-0.02	-0.01	-0.00	-0.01	-0.03	-0.04	0.01	0.00	0.01	0.00	0.02	-0.01	0.03	-0.02	0.28	0.02	-0.02	0.00	0.00	0.10	0.02	-0.00	-0.09	-0.05	0.04	0.03
8	Ceylon	-0.04	-0.00	0.02	-0.01	0.03	0.02	0.01		0.01	-0.01	-0.00	-0.03	0.02	0.00	0.01	0.03	-0.01	-0.01	0.01	-0.00	0.00			-0.04	-0.01	-0.07	0.05	-0.02	-0.00	0.02	0.02	0.01	-0.04
9	Elasticsearch	-0.02	-0.01	-0.02	-0.01	0.03	-0.01	0.02	-0.01		0.02	-0.01	-0.03	-0.01	0.00	-0.01	-0.01	-0.00	0.01	-0.01	-0.01	-0.00	-0.09	-0.07	-0.01	-0.02	-0.03	0.02	-0.02	-0.02	-0.01	-0.05	0.04	-0.12
10	Hazelcast	0.02	0.00	-0.00	0.00	0.06	-0.02	-0.03	0.01	-0.01		0.01	-0.02	-0.03	-0.00	-0.00	0.02	0.03	-0.02	0.01	0.01	-0.00		-0.07	0.01	-0.02	0.11	0.01	-0.14	0.00	-0.06	-0.00	0.12	-0.02
11	JUnit	0.00	-0.02	0.00	-0.04	-0.02	0.01	0.05	-0.03	-0.01	-0.02		0.00	0.03	-0.01	-0.01	0.02	0.00	0.05	0.00	0.03	0.04	0.25	0.07	-0.02	-0.09	0.07	0.06	0.13	0.11	0.03	0.02	0.01	0.24
12	MapDB	-0.02	-0.01	0.01	0.02	0.04	-0.00	-0.03	0.01	0.02	-0.01	-0.02		0.05	0.00	-0.01	0.01	-0.01	-0.02	-0.01	-0.02	-0.01		0.04	0.03	0.00	-0.02	-0.03	-0.02	-0.09	-0.07	0.13	-0.12	-0.06
13	mcMMO	0.05	0.00	-0.01	0.02	0.01	0.04	0.02	-0.01	-0.01	-0.05	0.01	0.00		-0.02	-0.01	-0.03	-0.03	-0.01	0.00	-0.01	0.03		0.07	0.01	0.02	0.08	0.05	-0.13	-0.09	-0.09	0.02	-0.13	-0.10
14	MCT	-0.01	0.01	-0.01	0.00		-0.02	0.00	-0.01	-0.00	-0.00	-0.00	0.02	-0.00		-0.00	-0.02	0.00		-0.00	-0.01	-0.00				-0.01	0.00	0.01		-0.01	-0.00	-0.05	-0.03	-0.07
15	Neo4j	0.02	-0.00	0.00	0.01	-0.02	0.00	-0.01	0.00	-0.00	-0.01	0.00	0.01	0.02	-0.00		0.00	0.00		-0.00	-0.01	-0.01			-0.01	-0.01		0.02		-0.01			0.02	0.01
16	Netty	-0.00	-0.02	-0.00	0.03	-0.05	-0.02	0.02	0.01	-0.02	0.02	-0.00	-0.00	-0.01	-0.02	-0.00		-0.04	0.04	0.00	-0.05	0.03		-0.05	0.01	0.00	-0.05	0.05	-0.04	-0.01	0.08	0.01	0.04	0.08
17	OrientDB	0.00	0.00	0.01	0.03	0.00	0.02	-0.03	-0.01	-0.00	-0.02	-0.01	-0.06	0.01	-0.00	-0.01	0.02		0.01	0.00	0.01	0.02		-0.03	-0.02	-0.01	-0.00	-0.01	-0.05	0.03	0.02	-0.04	0.08	-0.10
18	Oryx	-0.06	0.01	0.01	0.01	0.08	0.01	-0.00	-0.02	-0.00	-0.01	0.01	-0.02	-0.05	0.02	0.01	-0.03	-0.01		0.01	0.03	-0.02	0.28	-0.01	0.05	0.01	-0.01	-0.01	0.08	-0.05	0.05	-0.06	-0.06	-0.08
19	Titan	-0.03	0.02	0.00	-0.01	-0.01	-0.01	-0.00	-0.01	-0.00	-0.01	-0.00	-0.02	0.02	-0.01	0.00	-0.00	-0.00	-0.02		-0.02	-0.00		0.05	-0.01	0.02	0.04	0.05	0.00	-0.02	-0.02	0.07	0.04	0.02
20	Ant 1.7	0.08	-0.00	-0.01	-0.02	0.01	0.05	0.03	0.09	0.07	0.06	0.15	0.07	0.01	0.06	0.11	0.06	0.01	-0.02	0.06		-0.01		-0.05	-0.01	-0.01	0.00	-0.05	0.00	0.00	-0.05	0.09	0.02	0.00
21	Camel 1.6	0.01	0.02	-0.00	-0.00	-0.01	-0.01	0.00	-0.00	-0.01	0.00	0.00	-0.00	-0.05	-0.02	-0.02	0.02	-0.00	-0.04	0.01	-0.05		0.35	-0.07	-0.03	-0.04	-0.00	0.06	0.07	0.09	0.06	-0.15	0.05	0.14
22	Ckjm 1.8	-0.00	0.09	0.00	0.08	-0.04	0.04	-0.02	0.05	0.04	0.12	0.08	0.05	-0.05	0.11	0.09	0.03	0.07	-0.01	0.02	0.12	-0.04		-0.04	-0.03	-0.01	0.04	-0.03	0.00	-0.09	0.17	-0.01	-0.03	-0.07
23	Forrest 0.8	-0.02	0.06	-0.01	0.04	-0.03	0.05	0.01	0.03	0.01	0.05	0.03	0.01	0.04	0.05	0.04	-0.02	0.01	0.00	0.07	0.05	-0.02			0.01	0.04	-0.09	-0.00	0.02	-0.15	-0.05	0.02	-0.11	0.05
24	Ivy 2.0	-0.02	-0.02	0.01	0.02		0.02	0.01	-0.00	0.01	-0.00	-0.00	0.03	-0.01	-0.00	0.00	-0.01	0.00	-0.01	0.00	0.01	0.01		-0.05		0.05	-0.09	0.01	-0.08	-0.06	-0.04	0.03	-0.00	0.02
25	JEdit 4.3	-0.01	-0.00	-0.00	0.00		0.01	-0.00	-0.00	-0.01	-0.00	0.01	-0.01		-0.01	-0.00	-0.01	-0.01		0.00	0.01	-0.01			-0.00		0.02	0.00	0.00	0.03	-0.01	0.00	0.11	-0.02
26	Log4J 1.2	-0.06	0.03	0.01	0.04	0.04	0.19	0.01	-0.01	0.08	-0.04	0.05	0.03	0.08	0.07	0.03	0.04	-0.03	-0.05	0.01	0.01	0.07		-0.11	0.06	0.11		0.00	0.10	0.05	-0.02	-0.00	-0.01	-0.03
27	Lucene 2.4	-0.08	0.00	0.04	-0.00	-0.05	-0.02	-0.04	-0.03	0.01	-0.01	-0.05	-0.04	-0.00	-0.02	-0.02	-0.02	-0.01	0.00	-0.03	-0.00	-0.03	0.00	0.00	-0.08	-0.05	0.06		-0.12	0.02	0.02	-0.01	0.05	0.11
28	Pbeans 2	-0.06	-0.03	0.04	-0.02	0.05	0.04	0.01	0.01	-0.00	0.03	0.01	-0.01	0.01	0.01	-0.02	0.00	0.00	0.04	0.01	-0.01	0.00	0.13	-0.04	0.03	-0.09	0.12	0.06		-0.05	0.05	-0.03	0.12	0.05
29	Poi 3.0	0.08	0.01	-0.03	0.01	0.04	-0.02	0.07	0.07	0.01	0.05	0.08	0.07	0.15	0.03	0.03	0.04	0.01	0.05	0.05	0.01	-0.01	-0.16	-0.12	0.03	0.01	-0.02	-0.01	0.01		0.07	0.00	0.02	0.07
30	Synapse 1.2	0.12	-0.03	0.05	0.01	0.03	-0.06	0.03	0.04	-0.00	0.00	0.02	0.04	0.05	0.02	0.00	0.06	0.04	-0.01	-0.03	-0.05	-0.06	0.00	0.02	-0.00	-0.00	0.03	-0.02	0.12	0.03		0.02	0.00	-0.09
31	Velocity 1.6	-0.03	0.03	0.05	0.00	-0.01	0.03	0.04	-0.03	0.02	0.02	-0.01	-0.05	-0.03	0.00	0.02	0.01	-0.04	0.04	0.01	0.04	-0.00	0.00	0.01	0.04	-0.03	0.04	-0.02	0.10	-0.13	0.04		0.01	0.12
32	Xalan 2.7	-0.05	-0.04	0.02	-0.02	0.02	-0.02	0.01	-0.02	-0.00	-0.01	-0.05	0.01	0.02	0.03	0.00	0.03	0.03	-0.02	-0.01	0.01	-0.01		-0.06	-0.01	0.01	-0.02	0.01		0.01	0.01	-0.02		0.00
33	Xerces 2.0	-0.11	-0.06	-0.02	-0.10	-0.05	0.04	-0.05	-0.08	-0.06	-0.08	-0.13	-0.08	0.08	-0.11	-0.10	-0.10	-0.09	0.01	-0.16	-0.04	-0.04		-0.06	-0.08	0.03	-0.04	0.02	-0.17	-0.01	0.07	0.04	-0.00	

more classes as bugged. A similar over-fitting can be observed in the case of the Xalan 2.7 project. However, the F-measure change cannot be observed in Table 5, and this is due to the fact that in most cases by using the original dataset for classification the calculated weighted F-measure was already low as it was unable to correctly predict if the class is bugged or not.

Most of the increase can be seen when the classifier was trained on the JUnit, Ant 1.7, Ckjm 1.8, Log4J 1.2, or Poi 3.0 projects. For these systems, there are multiple instances where the F-measure change is greater or equal to 0.05 and there are a only few instances where a decrease can be observed.

The two small systems, Ckjm 1.8 and Forrest 0.8, have an overall improvement. Additionally, using these systems as test data could make the classificator over-fit in some cases as shown by the empty cells in the row ID 22 and 23. However, in other cases there are quite big improvements when using bigger systems for training and these two as test data. For example, in the case of the equinox training data the F-measure improvements were 0.26 and 0.25.

Overall, there are 123 instances where the weighted F-measure changes are greater than or equal to 0.05. In 107 cases the changes of F-measure are less than -0.05. These cases clearly indicate that adding the PE metrics can be beneficial for bug prediction. Furthermore, the increase of the weighted F-measure is greater than the decrease present in the change matrix. Based on these data, adding the PE metrics for further bug prediction classifications can be worthwhile.

4.4 Bug Prediction Across Versions

Training and testing the bug prediction across projects is good and can be useful. However, during a software development process usually a single application is developed from which multiple releases are created. As the system, evolves an earlier version of the project could be used to predict bugs. In this case, an earlier version can be used to train a bug prediction model and used on later versions. Fortunately, the selected bug data set contains multiple versions for some systems. These systems are the Ant, Camel, Ivy, JEdit, Log4J, Lucene, Pbeans, Poi, Synapse, Velocity, Xalan, and xerces. Due to size constraints, only a small subset of these projects will be presented here for this bug prediction evaluation across versions.

In the case of the Xerces 2.0 system in Table 5 the weighted F-measure values were usually in the negative. That is, the model trained with Xerces 2.0 was unable to improve the prediction for other projects. In this case, it is worth investigating what would happen if the various Xerces versions were compared by weighted F-measure changes between the original data set and the PE extended data set. This comparison can be observed in Table 6.

Same as before, the first column describes the systems on which the training was performed and the other columns are the systems where the evaluation was performed. This table describes that using an older Xerces version for training data set with the added PE metrics can improve the bug prediction capabilities for future Xerces versions. The most notable change is when Xerces 1.3 was used as a train data set and the 2.0 version was used to evaluate the predictions.

Table 6. Weighted f-measure changes in case of the Xerces project across versions

	Xerces 1.2	Xerces 1.3	Xerces 2.0
Xerces 1.2		0.02	-0.00
Xerces 1.3	0.02		0.06
Xerces 2.0	-0.01	-0.01	

During the comparison of the other projects with multiple versions, the most negative changes can be observed in case of the Ant system and all other projects have better values. The Ant system's comparison is depicted in Table 7.

Table 7. Weighted f-measure changes in the case of the ant project across versions

	Ant 1.3	Ant 1.4	Ant 1.5	Ant 1.6	Ant 1.7
Ant 1.3		-0.01	0.01	-0.07	-0.01
Ant 1.4	-0.01		0.05	-0.02	-0.02
Ant 1.5	-0.05	-0.03		0.02	0.01
Ant 1.6	-0.08	-0.01	0.03		-0.02
Ant 1.7	0.00	-0.02	-0.02	0.03	

In this case, there are a bit more than 10 cases where the addition of the PE metrics resulted in slightly negative values. Still, most of the changes are less than 0.05 indicating that in most cases the change is very small and can be ignored.

The Velocity project gave one of the best results when the PE metrics were added. The improvement in the weighted F-measure value can be observed in Table 8.

Table 8. Weighted f-measure changes in the case of the Velocity project across versions

	Velocity 1.4	Velocity 1.5	Velocity 1.6
Velocity 1.4		0.03	0.09
Velocity 1.5	0.13		-0.00
Velocity 1.6	0.03	0.03	

In almost every version combination, the addition of PE metrics improved the F-measure value. If the 1.4 version was used as the training data, the evaluation of the bug prediction improved on newer versions of the project.

Overall, the investigation of the bug prediction capabilities across multiple versions can be concluded as a viable option with the PE metrics.

5 Threats to Validity

The results shown in this paper are only using a small selected set of bug datasets. Although these datasets are widely known and used, they are relatively old, and there is no system among them that would be using a newer version than Java 8. Additional databases should be included in further research.

Calculations of the PE metrics were aggregated into class-level metrics, which could also skew results a bit. In addition, the metric calculation is currently only available for the Java language, but in order to fully evaluate their usefulness it would be good to adapt the metrics for other languages, e.g. C++.

The machine learning results are based only on a single algorithm. In further studies other classifiers should be investigated to see how the metrics perform.

6 Conclusion

In this paper, the Primitive Enthusiasm metrics were investigated to see how they perform in the case of bug prediction. The three PE metrics are the Local Primitive Enthusiasm, the Global Primitive Enthusiasm, and the Hot Primitive Enthusiasm metrics which capture the usages of primitively typed arguments in methods. The correlation between already existing metrics and the PE metrics is presented. Based on this correlation data the PE metrics did provide additional aspects. Although they have a bit of a connection with other metrics, the PE metrics still have a clear differentiation from traditional metrics.

In a cross-project validation, the bug prediction capabilities of the PE metrics were tested. Although there were cases where the trained models performed less adequately overall, there were improvements in classification of bugged classes.

Investigation of the project version-based bug prediction lead to a result where the PE metrics were able to add improvements in terms of weighted F-measure.

Based on these information, the addition of PE metrics for already existing metrics data sets can indeed improve bug prediction capabilities. In the future, we would like to continue to investigate the PE metric capabilities with other machine learning algorithms to see how they perform. Additionally, we would like to add other bug datasets into this research.

Acknowledgements. This research was supported by the EU-supported Hungarian national grant GINOP-2.3.2-15-2016-00037 and by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary.

References

1. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. Computer **34**(1), 135–137 (2001). https://doi.org/10.1109/2.962984
2. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 31–41 (2010). https://doi.org/10.1109/MSR.2010.5463279
3. Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: A public unified bug dataset for java and its assessment regarding metrics and bug prediction. Softw. Q. J. 1–60 (2020). https://doi.org/10.1007/s11219-020-09515-0
4. Frank, E., Hall, M.A., Witten, I.H.: Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques. Fourth Edition, Morgan Kaufmann, Burlington (2016)
5. Gál, P., Pengő, E.: Primitive enthusiasm: a road to primitive obsession. In: The 11h Conference of PhD Students in Computer Science, pp. 134–137. University of Szeged (2018)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co. Inc, USA (1995)
7. Gupta, A., Suri, B., Misra, S.: A systematic literature review: code bad smells in java source code. In: Gervasi, O., et al. (eds.) ICCSA 2017. LNCS, vol. 10408, pp. 665–682. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62404-4_49
8. Gyimóthy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. **31**(10), 897–910 (2005). https://doi.org/10.1109/TSE.2005.112
9. Hall, T., Zhang, M., Bowes, D., Sun, Y.: Some code smells have a significant but small effect on faults. ACM Trans. Softw. Eng. Method. **23(**4) (2014). https://doi.org/10.1145/2629648
10. Jayanthi, R., Florence, L.: Software defect prediction techniques using metrics based on neural network classifier. Cluster Comput. **22**(1), 77–88 (2018). https://doi.org/10.1007/s10586-018-1730-1
11. Jureczko, M., Madeyski, L.: Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE 2010, ACM (2010). https://doi.org/10.1145/1868328.1868342
12. Mäntylä, M.V., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: Proceedings of the International Conference on Software Maintenance. ICSM. pp. 381–384. IEEE (2003). https://doi.org/10.1109/ICSM.2003.1235447
13. Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.: Defect prediction from static code features: current results, limitations, new approaches. Autom. Softw. Eng. **17**, 375–407 (2010). https://doi.org/10.1007/s10515-010-0069-5
14. Moonen, L., Yamashita, A.: Do code smells reflect important maintainability aspects? In: Proceedings of the 2012 IEEE International Conference on Software Maintenance. ICSM, pp. 306–315. IEEE (2012). https://doi.org/10.1109/ICSM.2012.6405287
15. Pengő., E., Gál., P.: Grasping primitive enthusiasm - approaching primitive obsession in steps. In: Proceedings of the 13th International Conference on Software Technologies, ICSOFT, pp. 389–396. INSTICC, SciTePress (2018). https://doi.org/10.5220/0006918804230430

16. Roperia, N.: JSmell: A Bad Smell detection tool for Java systems. Master's thesis, Maharishi Dayanand University (2009)
17. Tóth, Z., Gyimesi, P., Ferenc, R.: A public bug database of Github projects and its application in bug prediction. In: Gervasi, O., et al. (eds.) ICCSA 2016. LNCS, vol. 9789, pp. 625–638. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42089-9_44
18. Wahono, R.S.: A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks. J. Softw. Eng. **1**, 1–16 (2015)
19. Yamashita, A., Moonen, L.: To what extent can maintenance problems be predicted by code smell detection? - an empirical study. Inf. Softw. Technol. **55**(12), 2223–2242 (2013). https://doi.org/10.1016/j.infsof.2013.08.002
20. Yu, Z., Rajlich, V.: Hidden dependencies in program comprehension and change propagation. In: Proceedings 9th International Workshop on Program Comprehension, IWPC 2001, pp. 293–299 (2001). https://doi.org/10.1109/WPC.2001.921739
21. Zhang, M., Hall, T., Baddoo, N.: Code bad smells: a review of current knowledge. J. Softw. Maintenance Evol. **23**(3), 179–202 (2011). https://doi.org/10.1002/smr.521
22. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, PROMISE 2007, p. 9. IEEE (2007). https://doi.org/10.1109/PROMISE.2007.10