Agnieszka Patalas, Wojciech Cichowski, Michał Malinka, Wojciech Stępniak, Piotr Maćkowiak, and Lech Madeyski, "Software Metrics in Boa Large-Scale Software Mining Infrastructure: Challenges and Solutions" in Software Engineering: Improving Practice through Research (B. Hnatkowska and M. Śmiałek, eds.), pp. 131–146, 2016.

Chapter 8

Software Metrics in Boa Large-Scale Software Mining Infrastructure: Challenges and Solutions

1. Introduction

Boa is a tool that can be used for data mining repositories of open-source projects. It contains the full history of a repository—from every revision's date and author, data on added, deleted and modified files to the complete state of the repository at the moment of commit. All data can be obtained by using the dedicated language. Boa provides a set of functions, which can be used for advanced data filtering [1, 2].

Boa has already been used for a variety of studies, including developers' willingness to adapt new Java features [3] or the licenses used in open-source projects [4]. So far they have not been metrics-oriented, even though the tool is intended to be used this way, as implicated by the inclusion of appropriate examples in the documentation of Boa [5] (e.g., *What are the number of attributes (NOA), per-project and per-type?*, *What are the number of public methods (NPM), per-project and per-type?*).

In this paper we focus on using Boa infrastructure to answer three research questions:

1) Which of the classic, widely known, software engineering metrics can be implemented in Boa? The implementation of classic software engineering metrics in Boa and publication of calculation scripts will make it easier to extend existing small-scale empirical software engineering research using software metrics, performed usually on a small number of projects, to a large-scale research.

2) What new metrics, that take advantage of the Boa's unique infrastructure, can be proposed? This paper will serve as a guide, for other researchers and practitioners, who shows how to implement new software metrics taking into account the unique features, as well as limitations, of the Boa large-scale software repository mining platform.
3) What is the feasibility of defect prediction models based on large number of projects data obtained from Boa data sets? According to our knowledge, this is one of the first attempts (if not the first) to build large-scale software defect prediction models based on a very large number of projects. Existing software defect prediction models usually base on a very limited number of projects.

Presented study refers to state of Boa framework during October 2015 – January 2016 period – when the source material was gathered.

2. Research methodology

In this section we introduce briefly into the following topics: how we selected projects for further investigation (see Section 2.1), how we implemented software metric scripts using the Boa language (see Section 2.2), and how we built software defect prediction models using software metrics from Boa (see Section 2.3), including also how we obtained data from the Boa output files (see Section 2.4).

2.1. Projects selection

Boa source code described in this paper has been developed and tested on two Boa data sets: September 2015 GitHub, and September 2013 SourceForge. A special filtering has been applied to select projects passing some entry criteria. The software projects explored in our study had to pass the following criteria:

1) **They have to have a code repository with revisions.** The *2013 September*/*SourceForge* data set consists of 700k projects. Our analysis with Boa queries has shown that 30% of them have no code repository [6, Section 2.1]. Out of remaining 489k (amount close to this stated by

Boa developers – 494,158 [7]) 4,767 projects have two repositories. Repositories in those projects have common history of revisions [6, Section 1.1]. In case of projects with multiple repositories, only the first of them is considered during study to avoid data duplication. Out of 489k projects with one code repository, 423k of them had no code revisions (commits) [6, Section 1.2]. It is difficult to determine whatever or not Boa is missing some data—the data sets have been defined for a given month in a given year, and current state of the repository might be different. The *2015 September/GitHub* data set has 7.83 million projects. 95% of them have no code repository in the Boa framework, even though the majority of them is available from the GitHub website. They are active and public, but most of them have had no commits since 2013 [6, Section 1.3]. From 380k projects with repository, only 2486 of them had commits in 2015 [6, Section 1.4]. Out of the entire GitHub dataset, 4% of projects have code repositories with revisions [6, Section 1.5].

2) **They have to have over 100 commits.** The projects picked should be mature enough for metrics calculation. A larger number of commits usually means a larger number of *fixing revisions*, which are in turn used for development of software defect prediction models.
3) **They have to be written in Java.** Java has been picked for this research due to being a mature, object-oriented language, popular among developers. It is also worth mentioning that Boa is written in Java, as well as provides extra Java-specific options, such as recognizing Java source files with and without parsing errors.

The Boa language implementation of filters to select projects fulfilling the above mentioned criteria is presented in Listing 1.

```
before node: Project -> {
  # They have to be written in Java.
  ifall (i: int; !match(`^java$`,
    lowercase(node.programming_languages[i]))) stop;

  # They have to have a code repository with revisions.
  if(len(node.code_repositories) > 0) {
    visit(node.code_repositories[0]);
```

```
  }
  stop;
}
before node: CodeRepository -> {
  # They have to have over 100 commits.
  if(len(node.revisions) < 100) stop;
  ...
}
```

Listing 1. Implementation of filters

The final number of projects that passed our entry criteria is presented in Table 1.

Table 1. Data sets

Dataset	All projects	Accepted projects
GH small	7,988	29
GH medium	783,982	2485
GH large	7,830,023	25307
SF small	7,029	50
SF medium	69,735	666
SF large	699,331	7407

2.2. Implementation of SE metrics

All of the metrics are calculated for classes. Each of the metric is implemented as a different Boa query, and is run on all Boa data sets mentioned in Section 2.

Due to long execution time, only data from GH small and SF small data sets are used for creating prediction models later on.

The output file of a query has to have the following data:

- the ID of the project
- the ID of the class
- the value of the calculated metric or the expected value.

This approach makes it possible to effortlessly merge all values gathered as the outputs of Boa queries, so they can be used as an input data set for a prediction model.

2.3. Defect prediction model

Software defect prediction model is aiming to find the classes that cause the most defects. A simple strategy to find them is searching for the classes that had been fixed most frequently.

2.3.1. Expected value – NCFIX

The expected value in our defect prediction model is Number of Class Fixes. Based on Boa's abilities, it is assumed the class has been fixed, if the two following conditions have been met:

- the file containing the class has been modified in a revision;
- the revision is marked as a fixing revision by the Boa's function *isfixingrevision* [1].

The list of classes and their fixes is obtained by the following algorithm:

1) Create an empty key-value collection for storing respectively: files in projects, number of fixing revisions for each file.
2) Visit a project's repository revision.
3) Check if it's a fixing revision.
4) Investigate the files changed in this revision.
   (a) If a file is marked as deleted, remove it from the collection.
   (b) If a file is added to the project in the current revision, add it to the collection:
      i. with a value of 1 if the revision is a fixing one;
      ii. with a value of 0 otherwise.
   (c) If a file is modified in the current revision, update it in the collection
      i. increment the number of fixes by one, if the revision is a fixing one;
      ii. leave it otherwise.
5) 5. Repeat steps 2-4 until you reach the most recent revision and there is no more revisions to check.
6) For all files stored in the collection, select only the ones that declare classes. Return the identifiers of the classes, and numbers of fixes corresponding to their files as the output.

The algorithm is inspired by the *getsnapshot* function implemented by Boa [1], which returns the state of the repository at given time stamp.

2.4. The use of Boa API and Weka

To allow easy management of Boa jobs and connecting job outputs with development of defect prediction models, a simple Java program [8] has been written. The software uses Boa Java API [9] release 0.1.0 to run jobs. Data from Boa is transformed into *.arff* file of following format:

```
@RELATION classes
@ATTRIBUTE class ID string
@ATTRIBUTE M_1 NUMERIC
. . .
@ATTRIBUTE M_N NUMERIC
@ATTRIBUTE fixingRevisions NUMERIC
```

where *classID* is an identifier of a studied class; *M_1 ... M_N* is a vector of calculated metrics for a class from latest repository SNAPSHOT; *fixingRevisions* attribute is the expected value described in Section 2.3.1.

3. Results

In this section three kinds of contribution are discussed, related to implementation of classic and new software metrics in Boa, as well as development of software defect prediction models on a basis of very large number of software projects. The latter can be seen as a way to address external validity threats common for most of the empirical studies focused on software defect prediction. All metrics' implementations are available to download via links provided in appendix [6, Section 3].

3.1. Implementation of classic software engineering metrics

This section presents how to implement scripts to collect some of the well-known, classic software metrics [10] in Boa. The metrics were chosen based on their popularity and Boa's limitations.

3.1.1. Obtaining classes

Using *getsnapshot* function implemented in Boa, all files available in the most recent revision of the project are gathered. Then, they are filtered so that only the files containing classes are taken into consideration. The data stored in the *Declaration* [1] and its attributes are used for calculating the value of a metric.

3.1.2. Inheritance issue

Each declaration (class or interface) node in Boa has its array of parents [1]. However, those parents are presented only as *Types*, meaning, they only have *TypeKind* (determining if it's a class, interface, or something else) and name, without its full package path or any other identifier. If two classes or interfaces in a project have the same name, but they are in different packages, it is impossible to determine which one is the ancestor of a given declaration. Therefore, all metrics using inheritance (such as all of the MOOD metrics [11], Depth of Inheritance Tree, Number of Children and Coupling between Object Classes [10]) had to be unfortunately, excluded from the study.

3.1.3. Metrics obtained directly from the Declaration node

Weighted Methods per Class (WMC) in its base version—the sum of methods in a class, Number of Fields (NoF) and Number of Nested Declarations (NoND), presented in Table 2, have been successfully implemented using the structure of the *Declaration* node alone.

Table 2. Declaration attributes and associated metrics

Attribute	**Metric**
methods	WMC
fields	NoF
nested_declarations	NoND

For each of those metrics, the value is a length of the attribute array. The execution time for those metrics is relatively small, up to 10 minutes for the

biggest data sets, which clearly shows the advantages of using Boa and the approach to calculate metrics using the structure of the *Declaration* node, presented in this paper.

3.1.4. Response For a Class (RFC)

The RFC metric was implemented as a number of methods in the class, added to number of remote methods directly called by methods of the class. The issue with the implementation of this metric is that Boa makes it difficult to recognize the difference between class' inner method and method of the external classes of the same identifier. For example: the method *getId*() of class *A*, called in class *B*, is seen as the same as method *getId*() in class *B*. If class *A* called two methods of the same name from different classes (class *B* and class *C*), those would be indistinguishable as well. There is no direct method that would allow to instantly determine the types of called methods' arguments [1] as well as the type of instance of variable from which the method was called [6, Section 1.6]. Such information can be obtained only by deeper analysis of Boa's AST tree, to the level of single *Statements*.

The simplified version of the metric, that ignores this nuance, has been successfully implemented and ran for both Boa's data sets.

3.2. Implementation of new software metrics

The metrics presented below have been developed by us upon learning more about the Boa architecture and its tree structure.

3.2.1. Number of Statements in Methods

The NoSiM metric is calculated as a sum of all statements in class methods. The nodes calculated are of the Boa type *Statement*. For studied Java classes, those nodes are either blocks of code marked by '{}' or single code expressions. The implementation of this metric is a starting point for implementation of a Lines of Code (LoC) metric. To achieve the LoC metric, all class' fields, number of methods, and such, would have to be added.

3.2.2. Maximum Depth of Declaration Nesting

MDoDN is the maximum level of class nesting in a class. For the following code:

```
class A {
  class B {
    class C {}
  }
  class D {}
}
```

the result for class *A* would be 3 (the depth of *C* class). The metric is not calculated for nested classes (in the example: *B*, *C*, and *D*). For implementation of this metric, Boa's stack functions are used. Every time the node of a nested *Declaration* is entered, it is pushed onto the stack. The metric value is the stack's element count.

3.2.3. Number of Anonymous Declarations

NoAD for Java is a sum of all anonymous children classes in the parent class. To calculate this metric, the Expression Boa node is tested for having a Declaration with a parameter of *ANONYMOUS* type.

3.2.4. Cumulative metrics

Metrics NoM, NoF, NoSiM, NoAD and NoND have been also successfully implemented in cumulative versions (CNoM [6, Section 1.7], CNoF [6, Section 1.8], CNoSiM [6, Section 1.9], CNoAD [6, Section 1.10], CNoND [6, Section 1.11]), where calculated value is a sum of metric for not only a class, but also all its nested and local classes.

3.3. Defect prediction model

The defect prediction model presented below is a single defect prediction model calculated for a high number of Boa projects. This is different from

a traditional approach, with a single, or several projects used to develop defect prediction models.

Data obtained from the Boa output files (described in Section 2.4) is randomly separated into training set and testing set (in 9:1 proportion). The *fixingRevisions* attribute in the testing set is nulled out, so it can be calculated using prediction model.

We used Random Forest to build defect prediction model. Random Forest generates a lot of random samples which are the subsets of training data set. A decision tree is generated for each of the samples [12]. The parameters listed below have been determined experimentally:

- number of trees: 200,
- max depth: 12,
- number of features: 12,
- cross-validation folds: 10,
- random seed: 1

The results of 10-fold cross-validation are presented in Table 3. Pearson product-moment correlation coefficient *r* shows a low correlation between the results from defect prediction model and real values, with high error ratio. Those results are further analyzed in Section 4.

Table 3. Results of evaluation of the prediction model

Evaluation attribute	**GH 2015 (small)**	**SF 2013 (small)**
Correlation coefficient (R)	0.215	0.244
Mean absolute error (MAE)	2.16	0.603
Root mean squared error (RMSE)	9.96	1.32
Relative absolute error (RAE)	102%	93.3%
Root relative squared error (RRSE)	100%	97.8%

3.4. Reference values of software metrics

The subsequent goal was to characterize a large number of open source projects available from Boa by means of software metrics in order to create reference values of software metrics. Table 4 presents descriptive statistics for each of calculated metrics among the data sets.

4. Discussion

The presented prediction model was tested on small data sets, but with correct resources it can be easily scaled to use full data sets with up to 25k subjects. This use case would be, to the best of our knowledge, the first attempt to create a large scale defect prediction model, as other examples from literature show prediction models developed using less than 200 projects [13, 14, 15].

The performance of the prediction model is poor due to the fact that a majority of classes studied has zero fixing revisions and therefore input data is highly unbalanced, see Table 5. However, the quality of prediction model and employing methods to deal with the class imbalance problem are not the main objectives of the study. Our aim was to show that it is possible to collect all the data necessary to build a large-scale software defect prediction model using the Boa platform.

Results from Table 4 show that not for all metrics standard deviation is lower for filtered datasets. This can be caused by the nature of metrics (such as NoND, NoAD, MDoDN), which are unlikely to have a high mean value in majority of projects.

4.1. Further research

It is worth to look at the way the fix in the revision is identified. Boa-provided function *isfixingrevision* is based only on the commit message text analysis. We assume this function is not ideal and integrating Boa API with outside software, such as bug tracking systems, can be a better solution to determine existing bugs in code revisions.

The data used for building prediction models in our study has big disproportions. Applying different filters and criteria (more mature projects, different languages and so on) could provide better data set for analysis, with more fixing revisions.

An interesting path of further research are process metrics [15, 16], which reflect changes over time and are becoming the crucial ingredients of software defect prediction models.

Table 4. Mean, median and standard deviation for metrics calculated in the study.

Metric	GH2015 (all)			GH2015 (fixes > 0)			SF2013 (all)			SF2013 (fixes > 0)		
	μ	$\tilde{x}$	σ	μ	$\tilde{x}$	σ	μ	$\tilde{x}$	σ	μ	$\tilde{x}$	σ
NOAD	0.12	0	0.86	0.15	0	1.05	0.17	0	1.31	0.34	0	2.40
CNOAD	0.13	0	0.93	0.17	0	1.13	0.19	0	1.39	0.36	0	2.51
NOND	0.32	0	1.20	0.32	0	1.28	0.14	0	0.83	0.22	0	1.00
NOF	2.98	1	15.64	3.10	1	9.29	3.75	2	8.51	4.36	2	11.43
CNOM	7.10	3	18.69	7.50	4	13.84	8.97	5	14.21	11.48	6	19.27
MDODN	0.20	0	0.44	0.22	0	0.47	0.14	0	0.38	0.20	0	0.45
CNOSIM	40.33	13	104.15	48.51	16	126.33	65.84	26	177.32	90.83	33	221.83
NOSIM	36.80	13	92.57	44.43	15	116.30	61.58	24	170.26	83.28	31	208.54
NOM	6.40	3	16.85	6.67	3	11.64	8.15	5	12.41	10.12	5	17.09
CNOF	7.10	3	18.69	7.50	4	13.84	8.97	5	14.21	11.48	6	19.27
CNOND	0.33	0	1.26	0.34	0	1.37	0.14	0	0.87	0.23	0	1.05
RFC	13.03	7	23.27	15.19	8	22.55	19.06	12	24.10	24.81	15	32.75

Table 5. Number of classes with zero and more than zero fixes in datasets

Amount of class fixes	GH 2015(small)	SF 2013 (small)
0	13296 (58.9%)	30244 (80.1%)
>0	9260 (41.1%)	7504 (19.9%)

5. Conclusions

Overall, the goal of the research, as described with research questions – implementation of software metrics in Boa and collecting data sets from a large number of projects, e.g., for the sake of prediction models – has been achieved.

We were able to implement some of the classic software engineering metrics using Boa, we presented some Boa-specific metrics, and we made an attempt to create a defect prediction model with the data we gathered. This proves that Boa can be a useful tool for data mining analysis in this particular field, as well as for creating sophisticated queries regarding its data sets. However, Boa is still a new framework that comes with a few disadvantages, and some of the metrics and operations were impossible to implement at the moment. In the following sections, the challenges met and our solutions are presented.

5.1. Challenges

Boa uses visitor pattern – one of Boa's greatest strengths – which sometimes might provide unexpected results if queries are not written properly.

5.1.1. Local and nested classes

One of the first issues we encountered creating Boa queries was a different size of output jobs. For our metrics, we gathered all classes from all projects. Therefore, for the same data set, all queries should return the same number of rows. As it turned out, the difference was caused by the behaviour of the visitor pattern, used by Boa. When source code contains a local class (class defined inside one of the methods) or a nested class (a class declared inside of another class), this class is visited by the visitor pattern before the analysis of the class containing it ends. Upon returning to the class-container, some of its metrics and calculations had been assigned to the local or nested class.

Solution: Boa offers implementation of stacks, which we started using while visiting local and nested classes. We took advantage of this solution implementing the Maximum Depth of Declaration Nesting metric described in Section 3.2.2.

5.1.2. Boa code compilers

Boa uses two different code compilers for SourceForge and GitHub data sets. As the framework is still in early development, sometimes the same query acts differently depending on the data set used.

Example: One of Boa sample queries "How many committers are there for each project?" [17] works fine in SF [6, Section 1.12], but causes compilation error in GH [6, Section 1.13]. In that case, a small change in the code notation solved the issue [6, Section 1.14]:

- Code resulting with error:

```
committers [p. code_repositories[i]. revisions[j].
  committer.username] = true;
```

- Code resulting with success:

```
username : string = p. code_repositories[i].
   revisions[j].committer.username ;
   committers[username] = true ;
```

This example shows that a person creating queries with Boa might run into different issues depending on the data set picked.

During our research, we often used Boa dictionaries. Dictionaries are defined by Boa as *map*[*key_type*] of [*value_type*]. Boa returns an error, if *int* is used as a *value_type*. We must have stored our integer values as strings, which resulted in converting value to integer each time it was used in calculations, and then back to string to update the map.

5.1.3. Debugging process

The errors reported by Boa are often lacking any sort of description. The debugging process comes down to commenting out parts of queries to check which fragments are causing errors. Each code test takes about a minute (and then some follow-up time to check if the output data is correct), and sometimes multiple tests are required to find the source of an error. There is no way of tracking the execution of the queries.

Solution: All variables used during the debugging process have to be initiated, by defining its type and aggregation method, and then returned in the output file.

5.2. Contribution

The paper describes our experience with using Boa platform for implementing software engineering metrics and defect prediction models. Our findings can be useful for both researchers – with solutions presented in Section 5.1 and provided source codes for metrics we implemented – as well as developer teams and project managers, providing an example for obtaining large-scale SE metrics for projects of particular profile (i.e. number of commits, used programming language and so on). The metric implementations proposed

by us are scalable – calculated for classes, but could be as well implemented for packages or projects.

Based on our findings, we confirm that Boa can be a powerful data mining tool, which can be used for a variety of research, alone and with usage of other software, like Weka, as demonstrated in Section 2.4.

References

[1] Iowa State University of Science and Technology: The Boa Programming Guide. http://boa.cs.iastate.edu/docs/, 2015, accessed: October 18, 2015.

[2] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 422–431, IEEE Press, 2013.

[3] R. Dyer, H. Rajan, H. A. Nguyen, T. N. Nguyen. Mining billions of fast nodes to study actual and potential usage of java language features. In: Proceedings of the 36th International Conference on Software Engineering. pp. 779–790, ACM, 2014.

[4] C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. German, D. Poshyvanyk. License usage and changes: A largescale study of java projects on github. In: The 23rd IEEE International Conference on Program Comprehension, ICPC, 2015.

[5] Iowa State University of Science and Technology: Example Boa Programs, http://boa.cs.iastate.edu/examples/, 2015, accessed: October 11, 2015.

[6] A. Patalas, W. Cichowski, M. Malinka, W. Stepniak, P. Mackowiak, L. Madeyski. Appendix to Software Metrics in Boa Large-Scale Software Mining Infrastructure: Challenges and Solutions, 2016, http://madeyski.e-informatyka.pl/download/PatalasEtAl16Appendix.pdf

[7] Iowa State University of Science and Technology: Boa. Mining Ultra-Large-Scale Software Repositories. Dataset Statistics, http://boa.cs.iastate.edu/stats/, 2015, accessed: October 18, 2015.

[8] Java research software, source code for metrics and statistical tests, https://github.com/Aknilam/metrics-research-software

[9] Iowa State University of Science and Technology: Boa. Mining Ultra-Large-Scale Software Repositories. Client API, http://boa.cs.iastate.edu/api/, 2015, accessed: October 18, 2015.

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

[11] F.B. e Abreu. Design quality metrics for object-oriented software systems. ERCIM News 23, 1995.

[12] L. Breiman. Random forests. Machine Learning 45(1), pp. 5–32, 2001.

[13] M. Jureczko, L. Madeyski. Towards Identifying Software Project Clusters with Regard to Defect Prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. pp. 9:1–9:10. PROMISE '10, ACM, New York, USA, 2010.
[14] M. Jureczko, L. Madeyski. Cross–project defect prediction with respect to code ownership model: An empirical study. e-Informatica Software Engineering Journal 9(1), pp. 21–35, 2015.
[15] L. Madeyski, M. Jureczko. Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study. Software Quality Journal 23(3), pp. 393–422, 2015.
[16] M. Jureczko, L. Madeyski. A review of process metrics in defect prediction studies. Metody Informatyki Stosowanej 30(5), pp. 133–145, 2011.
[17] Iowa State University of Science and Technology: Example Boa Programs, http://boa.cs.iastate.edu/examples/, 2015, accessed: October 18, 2015.