

An unsupervised learning-based methodology for uncovering behavioural patterns for specific types of software defects

**Gabriela Czibula, Ioana-Gabriela Chelaru, Istvan Gergely Czibula,
and Arthur-Jozsef Molnar**

**Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca,
Romania**

{gabriela.czibula, ioana.chelaru, istvan.czibula, arthur.molnar}@ubbcluj.ro

September 21, 2023

- 1 Problem Statement
- 2 Relevance
- 3 Unsupervised learning for SDP
- 4 Apache Ivy dataset
- 5 Our Contribution

What is Software Defect Prediction (SDP)?

- consists of detecting software modules prone to be defective in the future
- important during all stages of the software lifecycle
- problem of major relevance in SBSE

Search-based software engineering seeks to rephrase software engineering tasks into optimization problems that are naturally approached using artificial intelligence algorithms.

- enhances software quality
- can guide code review and testing-related activities
- AI can perform regular checks to ensure software reliability
- SDP plug-ins for IDEs could be developed to boost productivity
- useful for large legacy software

Types of Defects

- compilation errors (should not pass QA)
- copy-pasted problems
- concurrency issues
- connectivity errors to DB/microservices
- platform limitations or hardware issues
- client requirement misunderstanding
- coding standards or other style conventions
- etc.



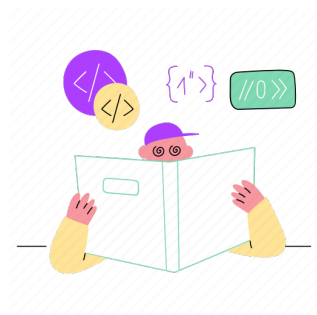
Overall Difficulty

- not enough relevant data to outperform previous ML models
- severe class imbalance
- code correctness may vary along the product's lifetime
- many types of defects that have different degrees of severity
- regular supervised learning ML models underperform

Solutions from the literature

The literature is dominated by supervised learning solutions, such as:

- conventional ML predictors [1]
- fuzzy models
- evolutionary computation to recurrent neural networks
- deep learning models [2]



Defect taxonomies:

- the *Orthogonal Defect Classification* proposed by IBM
- *Defect Origins, Types and Modes* proposed by HP
- the *IEEE Standard Classification for Software Anomalies*

The datasets do not contain labels for types of defects \implies supervised learning models for specific classes of bugs remain impossible to build

- **RQ1:** *How to develop an UL-based methodology which would allow uncovering specific classes of software defects and the relevant set of features characterizing them?*
- **RQ2:** *How to determine the set of software features (attributes) that characterize specific types of defects and would be useful for detecting those particular classes?*
- **RQ3:** *To what extent are the obtained software defect classes correlated with actual findings in the source code (related to the defective software entities and the performed modifications in the source code for fixing these issues)?*

Apache Ivy Dataset

Ivy_{all} : dataset containing all 2237 instances

$Ivy_{defects}$: dataset containing only the 177 defective instances

\mathcal{F} : the set of the 2225 used software metrics (4189 total, of which 1964 were zero for both defective and non-defective classes

Version	Instances	Defective instances	Defective rate
Apache Ivy 1.4.1	240	11	0.046
Apache Ivy 2.0.0	352	50	0.142
Apache Ivy 2.1.0	357	49	0.137
Apache Ivy 2.2.0	363	37	0.102
Apache Ivy 2.3.0	451	13	0.029
Apache Ivy 2.4.0	474	17	0.036

Table: Total number of classes, number of defective classes and defective rate for Apache Ivy releases in our study.

Source Code Feature Extractors

- based on static code
- warnings produced by the PMD analysis tool [3]
- extracted from the AST representation
- based on code churn [4, 5, 6]



Algorithm 1: The stages of the proposed methodology.

Procedure *Partitioning*(D, p) is

Input: D the current set of defective application classes from the software system (in our case study 2225-dimensional real-valued feature vectors); p is a positive predefined threshold used as a stopping criterion of the recursive process.

/* Stage 1. Determine a partition $\mathcal{G} = \{G_1, G_2, \dots, G_n\}$ of D using a SOM */

$\mathcal{G} \leftarrow \text{SOMClustering}(D)$

/* Stage 2. For each cluster G_i previously identified determine the most relevant subset of features F_i characterizing it. */

for $i \leftarrow 1, n$ **do**

 | $F_i \leftarrow \text{mostRelevantFeatures}(G_i)$

end

/* the identified clusters are processed and analysed */

for $i \leftarrow 1, n$ **do**

 /* Stage 3. If the cluster contains a small number of instances (less than or equal to the threshold p) */

if $|G_i| \leq p$ **then**

 | @ the cluster and its relevant features are analysed from a software engineering perspective

else

 /* Stage 4. The procedure is applied again, recursively, on that cluster */

 | $\text{Methodology}(G_i, p)$

end

end

EndProcedure

Stage 1 - *Soft Organizing Maps* : unsupervised learning model connected to artificial neural network literature used for dimensionality reduction and for visualizing high-dimensional data; they are trained to produce an output map which is a low-dimensional (usually two-dimensional) representation of a high-dimensional input space.

Stage 2 - *Univariate feature selection* : works by assigning scores to features based on univariate statistical tests and then selecting the best features (with the highest scores); it uses *F-test* for feature scoring and selects the features according to the *K highest scores*

Stage 3 - *Software engineering perspective* : manually examined the commit history of the determined clusters of defective application

Experimental Results I

Table: Clusters obtained during the first three recursive calls of the procedure depicted in the described algorithm.

Recursive call	Set of defects (D)	# of defects	SOM	Detected clusters
1	$Ivy_{defects}$	177	Figure 1(a)	$C1 = \{0-188, 0-82, 0-70\}$ $C2 = \{3-0, 2-2\}$ $C3 = \{1-9, 1-25, 1-21, 2-27, 2-23, 2-11, 2-34, 3-21, 4-44, 5-17, 5-32\}$ $C4 = Ivy_{defects} \setminus C1 \setminus C2 \setminus C3$
2	$C3$	11	Figure 1(b)	$C3.1 = \{2-34\}$ $C3.2 = \{5-32, 5-17, 4-44\}$ $C3.3 = \{1-9, 1-21, 1-25, 2-11, 2-23, 2-27, 3-21\}$
3	$C3.3$	7	Figure 1(c)	$C3.3.1 = \{1-21, 1-25, 2-27\}$ $C3.3.2 = \{3-21\}$ $C3.3.3 = \{2-11\}$ $C3.3.4 = \{2-23\}$ $C3.3.5 = \{1-9\}$

Experimental Results II

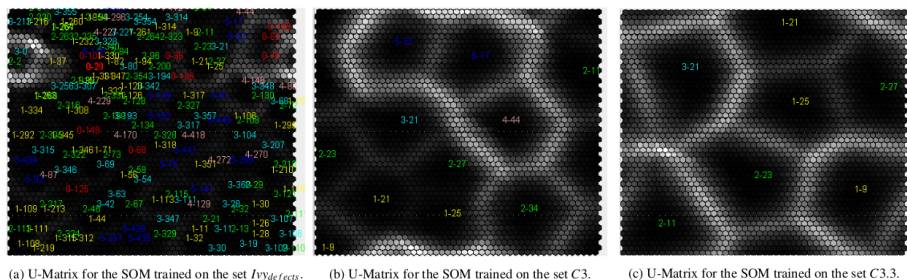


Fig. 1: SOMs trained on the sets of defects D depicted in Table 2.

Experimental Results III



Fig. 2: The most important 15 features in separating a cluster of defects (*Ivy*_{defects} - left; C3 - middle; C4 - right) from the rest of instances.

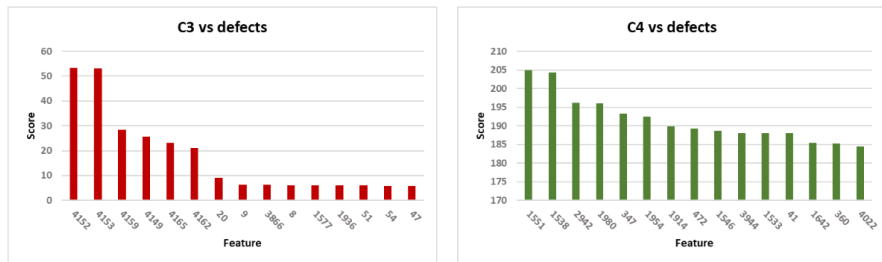


Fig. 3: The most important 15 features in separating a cluster of defects (C3 - left; C4 - right) from the rest of software defects.

Module name	Cluster	ID	Defect ID (affected release(s))
org/apache/ivy/core/module/descriptor/DefaultDependencyDescriptor.java	C1	0-70	IVY-374, IVY-442 (1.4.1), IVY-982 (2.0.0 - 2.4.0)
org/apache/ivy/core/module/id/ModuleId.java	C1	0-82	IVY-422 (1.4.1), IVY-1362 (2.2.0)
org/apache/ivy/Main.java	C1	0-188	IVY-457 (1.4.1), IVY-1009 (2.0.0), IVY-1321 (2.2.0 - 2.4.0), IVY-1355, IVY-1483 (2.3.0, 2.4.0)
org/apache/ivy/Ivy.java	C2	2-2, 3-0	IVY-336, IVY-346, IVY-509 (1.4.1)
org/apache/ivy/ant/IvyTask.java	C3.1 C3.2	2-34 4-44	IVY-65, IVY-453, IVY-497 (1.4.1), IVY-1412 (2.3.0)
org/apache/ivy/ant/IvyCacheFileset.java	C3.2	5-17	IVY-497 (1.4.1), IVY-1272 (2.2.0), IVY-1475, IVY-1631 (2.4.0)
org/apache/ivy/ant/IvyDependencyUpdateChecker.java	C3.2	5-32	IVY-1549 (2.4.0)
org/apache/ivy/ant/IvyFindRevision.java	C3.3.1	1-21	IVY-497 (1.4.1), IVY-1344 (2.3.0)
	C3.3.2	3-21	
	C3.3.4	2-23	
org/apache/ivy/ant/IvyMakePom.java	C3.3.1	1-25, 2-27	no defects in JIRA
org/apache/ivy/ant/IvyArtifactReport.java	C3.3.3	2-11	IVY-453, IVY-497 (1.4.1), IVY-1150 (2.1.0), IVY-1212 (2.1.0)
	C3.3.5	1-9	

Conclusions:

- the results correlate with the human approach when categorizing defects
- models tailored for specific types of defects can improve accuracy
- by creating tailored models, we could accelerate the adoption of software defect prediction approaches by the industry





Future Enhancements:



- extend experiments on other Apache projects
- consider the impact of feature selection and alternative feature-based representations of the software entities
- build supervised defect predictors for the specific types of defects detected here

Thank you!

Q&A

Bibliography I

-  R. Malhotra, “Comparative analysis of statistical and machine learning methods for predicting faulty modules,” *Applied Soft Computing*, vol. 21, pp. 286–297, 2014.
-  I. Batool and T. A. Khan, “Software fault prediction using data mining, machine learning and deep learning techniques: A systematic literature review,” *Computers and Electrical Engineering*, vol. 100, p. 107886, 2022.
-  GitHub, “PMD - An extensible cross-language static code analyzer,” 2023.
<https://pmd.github.io/>.
-  R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *ICSE '08*, (New York, NY, USA), p. 181–190, ACM, 2008.

-  M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, p. 531–577, aug 2012.
-  A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 78–88, 2009.